Resource Management in Distributed Stream Processing: From Fine-Grain Elasticity to Proactive Planning

Donatien Schmitz

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

February 2026

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Prof. Etienne **Rivière** (Advisor)	UCLouvain, Belgium
Prof. Peter **Van Roy** (Chair)	UCLouvain, Belgium
Prof. Ramin **Sadre** (Secretary)	UCLouvain, Belgium
Dr. Anne-Cécile **Orgerie**	CNRS, Rennes, France
Prof. Romain **Rouvoy**	Inria, Univ. Lille, France
Mr. Sabri **Skhiri**	Eura Nova, Belgium

Resource Management in Distributed Stream Processing: From Fine-Grain Elasticity to Proactive Planning

by Donatien Schmitz

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the Région Wallonne under the project GEPICIAD, by a gift from Eura Nova, and by Grid5000.

And I knew exactly what to do.
But in a much more real sense,
I had no idea what to do.
MICHAEL G. SCOTT

Abstract

Distributed Stream Processing (DSP) systems have become essential for processing large-scale, real-time data streams in various applications, ranging from social media analytics to IoT data processing. DSP applications are represented as directed graphs of operators that continuously process incoming data tuples. These operators have diverse and dynamic resource requirements, making resource management a challenging task. Efficient resource management in DSP is crucial to ensure optimal performance, cost-effectiveness, and adaptability to dynamic workloads. Current approaches treat resource management at a coarse granularity, leading to suboptimal resource utilization, and rely on reactive mechanisms that may not adequately capture the individual requirements of operators.

This thesis addresses the challenges of resource management in DSP by proposing novel approaches that span from fine-grain elasticity to proactive planning. First, we conduct an in-depth analysis of real-world DSP workloads to understand their characteristics and resource requirements. Based on these insights, we develop a fine-grain elasticity mechanism that allows DSP systems to dynamically adjust CPU and memory resources at the operator level, enabling more efficient resource utilization and improved performance under varying workloads. Next, we introduce a multi-query auto scaling framework that leverages fine-grain elasticity combined with optimized placement strategies to manage resources across multiple concurrent DSP applications. This framework aims to minimize resource costs while ensuring that performance objectives are met for all applications. Finally, we propose a proactive resource planning approach that models the performance of individual DSP applications to predict their resource requirements for unseen workloads.

Acknowledgments

It would require a book to fully express my gratitude to all those who have supported me throughout this journey. However, I would like to take this opportunity to acknowledge a few key individuals and groups who have been instrumental in my success.

First and foremost, I would like to thank my advisor, Prof. Etienne Rivière, for his unwavering support, guidance, and encouragement. Since the very beginning of my PhD, and even before during my master's thesis, he has been a constant source of motivation. This journey was not "un long fleuve tranquille", but his mentorship made it manageable and rewarding. I would like to also extend my gratitude to Dr. Guillaume Rosinosky with whom I had the opportunity to collaborate during the first two years of my PhD. His knowledge of the field and underlying technologies greatly helped me get on-board and make significant progress in my research.

I am also grateful to my PhD committee members, Anne-Cécile, Peter, Ramin, Romain, and Sabri, for their valuable feedback and insights during my thesis defense.

Four and a half years have passed since I first set foot in the INGI department as a PhD student. Looking back, I am filled with fond memories of the time spent with my colleagues and friends. The first two years were met with the incredible support of a special someone. Although our paths have diverged since then, I will always cherish the moments we shared. Thank you Clara for this.

This thesis allowed me to stay a "student" for a little longer, which I greatly appreciated. Without this opportunity, I would not have met so many wonderful people. I would like to thank all the members of the *Academicus Victoria Athena Ordo* for the unforgettable experiences and friendships formed during our time together, including the *Calotte's Choir*.

I would also like to express my gratitude to my friends outside of academia, who have been a constant source of joy and support throughout this journey. My deepest thanks go to my close friends: Alex, Nicolas, Simon, and Sophie.

What would a PhD be without coffee breaks and social gatherings? I would like to thank my colleagues at INGI, both past and present, for the stimulating discussions, collaborations, and camaraderie. First, the belotteurs and belotteuses of INGI, for the countless games that provided a much-needed break from work. Then, the members of INGI who I can say have become

good friends instead of just colleagues: Achille, Alice, Augustin D., Benoit R., Charles P., Emma, François FDK, François M., Louis, Mathias, Matthieu, Olivier G., Vany, Yinan, and many others.

A special mention goes to my current officemates with whom I shared many laughs and memorable moments: Amaury F., Benoit D., Damien, Edouard C., François R., Juliette, Lucile, Melanie, and Théo. Thank you for making the office a lively and enjoyable place to work.

Of course I cannot forget to mention the core member of this department, the person with whom I shared laughs, frustrations, and countless cups of coffee: Vanessa. Thank you for being such a wonderful person and for always being there to lend a helping hand.

Last year I had the pleasure of being an actor in "La Revue des Ingénieurs 2025". I would like to thank all the members of the revue team for their hard work and dedication in putting together such a fantastic show. It was an unforgettable experience that I will always cherish. My special thanks to Lola, Marion, and Simon for their support and friendship throughout the process.

Finally, I would like to thank my family for their unwavering love and support. To my parents, siblings, extended family, and my cat Gus, thank you for always believing in me and encouraging me to pursue my dreams. I will always be joyful knowing that you are proud of my accomplishments. This thesis is dedicated to all of you.

Contents

Introduction 1

The rapid growth of data generation in recent years has led to an increasing demand for large-scale data processing systems. Traditional database systems, designed for structured data and transactional workloads, are ill-suited for processing such large-scale data due to their limited scalability, and inability to handle the volume, velocity, and variety of modern data streams. Distributed data processing enables the handling of massive datasets by distributing the workload across multiple nodes in a cluster, allowing for parallel processing and improved fault tolerance. However, the complexity of managing distributed systems and ensuring efficient resource utilization is a significant challenge. This complexity has led to the emergence of distributed data processing paradigms. These paradigms offer the users high-level abstractions to process data at scale, while abstracting away the underlying complexities of distributed systems. First generation systems, such as MapReduce [DG08], were designed to process large datasets in batch mode, providing fault tolerance and scalability. However, the need for real-time data processing has driven the development of distributed stream processing (DSP) systems, which can process data as it arrives, enabling low-latency analytics and decision-making. The latest generation of DSP systems, such as Apache Flink [Car+15], unified the batch and stream processing paradigms, allowing users to process both historical and real-time data using a single framework. Systems like Flink are widely adopted in industry, included by companies such as Alibaba, Uber, and Netflix, to power their real-time data processing needs.

1.1 Thesis Context

DSP has gained significant traction in various domains, including finance, healthcare, and e-commerce, where real-time data processing is crucial for applications such as fraud detection [Car+18; Ban+23], personalized recommendations [Hua+15], and patient monitoring [Shu23]. DSP systems enable organizations to derive insights from data streams in real-time, allowing them to make informed decisions and respond to changing conditions promptly. They are often integrated with other big data technologies, such as Apache Kafka [KNR+11] for data ingestion, to create comprehensive data process-

ing pipelines. This is the case of Digazu[1], a data lake platform that leverages Apache Flink to provide real-time analytics and insights to its users.

Despite the advancements in DSP systems, many challenges remain, such as scalability, fault tolerance, and efficient resource utilization. Efficiently allocating resources to DSP applications is critical to ensure optimal performance and cost-effectiveness. Traditional approaches often rely on static allocation, which can lead to underutilization or over-provisioning of resources. This is due to the heterogeneity of DSP workloads, which can exhibit varying resource requirements based on factors such as data volume, query complexity, and stateful operations. This thesis addresses these challenges by proposing novel mechanisms for resource management and elasticity in DSP systems.

1.2 Thesis Contributions

The contributions of this thesis were motivated by the challenges faced in managing resources for DSP applications in a large-scale deployment, namely the predictability and reactivity aspects. During the course of this work, we participated in periodic discussions with the Digazu engineering team to identify pain points and areas for improvement in their DSP infrastructure. This collaboration provided valuable insights into the practical challenges of deploying and managing DSP applications in a production environment.

To further understand the workload characteristics and resource requirements of DSP applications in Digazu, we conducted an extensive analysis of real-world workloads collected from the platform. This analysis revealed several key insights into the behavior of DSP applications, including workload patterns, resource utilization, and query characteristics. The analysis of 129 DSP queries running in Digazu for long periods of time (from several days to weeks) showed that many queries exhibit complex stateful operations, leading to performance degradation under static resource allocation strategies. Moreover, we observed the presence of multi-query workloads, where multiple DSP applications run concurrently on shared resources, leading to resource contention and suboptimal performance. Furthermore, the majority of these queries were found to experiencing low loads for extended periods, punctuated occasionally by bursts of high activity.

Based on the previous observations, we identify the need for more adaptive and fine-grained resource management, as well as multi-query awareness approaches in DSP systems. Finally, we recognize the importance of accurate capacity planning tools to estimate the resource requirements of DSP applications under varying workload scenarios, priori to their deployment.

[1] `https://www.digazu.com/`

The main contributions of this thesis are as follows:

- We analyze real-world workloads from a production deployment, characterizing a hybrid batch-stream production workload and identifying key patterns and challenges in resource management for DSP applications [Sch+25]. For this, we instrumented the Digazu platform to collect detailed workloads of DSP application executions, including resource usage metrics, query execution plans, and workload patterns. This analysis provides a foundation for understanding the resource management challenges in DSP systems and informs the design of our proposed mechanisms.

- Following this analysis, we identify inefficiencies in the existing resource management strategy. We conduct experiments to motivate the need for more adaptive and fine-grained resource management approaches in DSP systems. More precisely, we show that static resource allocation can lead to suboptimal performance and resource utilization in the presence of complex stateful operations. To this effect, we propose **Justin** [SRR25], a hybrid CPU/memory elastic scaling mechanism for DSP. **Justin** dynamically adjusts the memory requirement of DSP applications based on workload characteristics, enabling more efficient resource utilization and improved performance.

  Our hybrid CPU/memory elastic scaling mechanism demonstrates significant improvements in resource utilization and performance for DSP applications. More precisely, it effectively identifies memory-intensive operations and dynamically adjusts memory allocation, leading to more efficient scaling decisions. In our experiments, it achieved up to 40% reduction in memory and CPU usage with no over-provisioning compared to the state-of-the-art scaling mechanisms.

- The analysis of multi-query workloads revealed the need for more fine-grained resource allocation and optimized task placement strategies in DSP systems. This is further motivated by the presence of long-lived queries with low loads that lead to overallocation inefficiencies. Building on these insights and on the mechanisms proposed in **Justin**, we propose **Charon** [SRR26], a fine-grain resource allocation and optimized task placement mechanism for multi-query DSP systems. **Charon** leverages workload characteristics and resource usage patterns to optimize resource allocation and task placement, reducing resource contention and improving overall system performance.

  Our fine-grain resource allocation and optimized task placement mechanism significantly improves system performance in multi-query DSP

scenarios. By leveraging useful runtime metrics, **Charon** efficiently reduces the required resources per query in a multi-query environment. Furthermore, **Charon** manages to improve the default task placement strategy of Apache Flink, leveraging a Bin-Packing algorithm, leading to better resource utilization. Our experiments show that **Charon** can reduce resource consumption, requiring only 43% to 60% of the resources needed by the default Flink scheduler to achieve similar performance.

- Finally, we address the challenge of accurately estimating the resource requirements of DSP applications under varying workload scenarios. More specifically, we identify the need for capacity planning tools that can predict resource needs prior to deployment. This is particularly important for large-scale deployments with limited resources or *pay-as-you-go* pricing models. To this effect, we present **StreamBed** [RSR24b], a capacity planner for DSP systems. In contrast to the previous contributions that focus on runtime mechanisms, StreamBed provides an offline capacity planning tool that helps system administrators and users to estimate the resource requirements of DSP applications. StreamBed builds a performance model based on small-scale benchmark executions and uses this model to predict the resource requirements of DSP applications under different workload scenarios.

  Our capacity planner, **StreamBed**, provides accurate resource requirement estimates for DSP applications. StreamBed's performance model, built from small-scale benchmark executions, effectively predicts resource needs under various workload scenarios. In our evaluation, we show that StreamBed can accurately predict the volume of necessary resources with a low cores.hours budget, for queries reaching more than 1000 cores and ingesting up to 190 millions events per second.

To evaluate the effectiveness of our proposed mechanisms, we conduct extensive experiments using well-established benchmarks on a cluster of up to 122 nodes, each equiped with 18 CPU cores and 96 GB of RAM. Instead of relying on simulations, we implement and evaluate our proposed solutions in Apache Flink, a widely adopted DSP system, although the principles and techniques presented in this thesis are applicable to other DSP systems as well.

1.3 Publications

The list of publications that resulted from this thesis is as follows:

- **Streambed: Capacity Planning for Stream Processing**. Guillaume Rosinosky, Donatien Schmitz, and Etienne Rivière. In Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems (DEBS), 2024. Lyon, France, 90-102.

- **A Tale of Many Streams: Characterizing a Hybrid Batch-Stream Production Workload in Digazu, a Data Lake supported by Apache Kafka and Flink**. Donatien Schmitz, Luc Berrewaerts, Guillaume Rosinosky, Sabri Skhiri, and Etienne Rivière. In Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems (DEBS), 2025. Göteborg, Sweden, 188-198.

- **Justin: Hybrid CPU/Memory Elastic Scaling for Distributed Stream Processing**. Donatien Schmitz, Guillaume Rosinosky, and Etienne Rivière. In IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2025. Lille, France, 102-118.

  This publication received the **IFIP Best Paper Award** at DAIS 2025 and the **IFIP DisCoTec-wide Best Paper Award** at DisCoTec 2025.

- **Charon: Fine-Grain Resource Allocation and Optimized Task Placement for Multi-Query Distributed Stream Processing**. Donatien Schmitz, Guillaume Rosinosky, and Etienne Rivière. In Proceedings of the 41st ACM/SIGAPP Symposium On Applied Computing (SAC), 2026. Thessaloniki, Greece.

1.4 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** provides the background necessary to understand the context of this thesis, including big data systems, distributed stream processing, and a focus on Apache Flink.

- **Chapter 3** presents our work on characterizing a hybrid batch-stream production workload in Digazu, a data lake supported by Apache Kafka and Flink.

- **Chapter 4** formulates the problems addressed in this thesis, identifying key challenges in distributed stream processing and the need for efficient resource management, elasticity, and prediction.

- **Chapter 5** presents our work on Justin, a hybrid CPU/memory elastic scaling mechanism for distributed stream processing.

- **Chapter 6** presents our work on Charon, a fine-grain resource allocation and optimized task placement mechanism for multi-query distributed stream processing.
- **Chapter 7** presents StreamBed, a capacity planner for distributed stream processing.
- **Chapter 8** concludes this thesis with a summary of our contributions and a discussion of future work.

Background

2

This chapter provides the necessary background to understand the context and contributions of this thesis. We first introduce cloud computing and big data processing in Section 2.1. This section provides a general and historical overview of data processing, leading us to focus on distributed stream processing in Section 2.2. We then present Apache Flink, the distributed stream processing system used in this thesis, in Section 2.3. Finally, we discuss container orchestration and the Kubernetes system in Section 2.4, and Apache Kafka in Section 2.5.

2.1 Big Data Processing

Growing data volumes and the need for large-scale data processing have driven the development of various big-data processing paradigms. The primary goal of big data processing paradigms is to hide the complexity of distributed execution from application developers. Instead of manually managing complex tasks such as task scheduling and failure recovery, programmers express computations using simple, high-level operators (e.g., map, reduce, join). The runtime system translates these declarative descriptions into distributed computations, orchestrating data movement, resource allocation, and fault-tolerance so developers can focus on application logic rather than on the low-level details of distributed systems. To better understand the focus on stream processing in this thesis, we first provide an historical overview of data processing paradigms.

Batch Processing. Abstracting the processing of large datasets in a distributed manner was popularized by the MapReduce [DG08] programming model. MapReduce has been widely adopted for various data processing tasks, including data mining [NY14; SV15], log analysis [LWW13; MK17], and machine learning [Lan+15; TAW19]. The core idea of MapReduce is to divide datasets into smaller chunks that can be processed in parallel across a cluster of machines. The model abstracts the computation into two main functions: *Map*, which processes input data and produces intermediate key-value pairs, and *Reduce*, which aggregates these intermediate results to produce the final output. An important step in the MapReduce model is the *shuffle* phase, which redistributes the intermediate key-value pairs based on their keys to

ensure that all values associated with the same key are sent to the same reducer. This shuffle phase is crucial for the correctness of the Reduce function, as it relies on receiving all relevant data for each key.

The MapReduce model offers a simple and powerful abstraction for distributed data processing, enabling developers to write parallel applications without dealing with the complexities of distributed systems. These complexities are handled by the underlying MapReduce framework, which manages task scheduling, data distribution, and fault tolerance. Apache Hadoop [25a] is one of the most popular implementations of the MapReduce model. Hadoop provides a distributed file system (HDFS [Bor+08]) for storing large datasets and a MapReduce engine for executing MapReduce jobs across a cluster of machines.

> **Example 2.1.1.** A typical example of a MapReduce query is counting the occurrences of words in a large collection of documents. The Map function processes each document, emitting a key-value pair for each word (e.g., "word", 1). The Reduce function then aggregates these pairs by summing the counts for each unique word, resulting in a final count for each word across the entire dataset. A shuffle phase, hidden from the user, ensures that all occurrences of the same word are sent to the same reducer for aggregation.

While systems such as Hadoop are highly effective for processing large volumes of static data, they exhibit certain limitations in terms of flexibility and efficiency. Specifically, the separation of computation into distinct Map and Reduce phases necessitates to store intermediate results on disk. This design choice, while facilitating fault tolerance and scalability, introduces significant I/O overhead. Furthermore, any modification to the Reduce logic requires the entire dataset to be re-read from disk.

To address these limitations, Apache Spark [Zah+10] emerged as a more flexible and efficient alternative for batch processing. Spark introduces the concept of Resilient Distributed Datasets (RDDs), which are immutable collections of objects that can be processed in parallel. RDDs support in-memory computation, allowing for faster data processing by minimizing disk I/O. Apache Spark also provides a rich set of high-level APIs for various data processing tasks, including SQL queries [Arm+15], machine learning [Men+16; Har+17; BEI18], and graph processing [SD15; Sal+16].

The principles behind Apache Spark remains the same as MapReduce: abstracting the complexity of distributed data processing and fault tolerance from the user while providing interfaces closer to traditional query languages, such as SQL.

2.2 Distributed Stream Processing

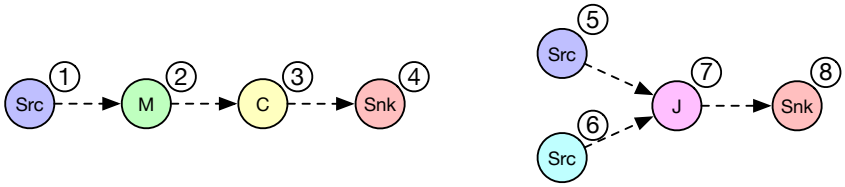
Despite their effectiveness for processing large static datasets, Batch Processing systems like Hadoop and Spark are not well-suited for applications that require real-time data processing. However, many modern applications, such as real-time analytics [Tön+14; Mil+16], monitoring systems [SSL13], and online recommendation engines [Hua+15], require the ability to process continuous streams of data with low latency. Thus, the third generation of big data processing systems, known as Distributed Stream Processing (DSP) systems, emerged to address the need for real-time data processing.

Apache Spark first introduced streaming capabilities with Spark Streaming [Zah+13], which processes data in micro-batches. It does so by dividing the incoming data stream into small batches, treating each batch as a RDD and applying the same transformations as in batch processing. The next evolution in Stream Processing came with systems like Apache Storm [14b] which provide true low-latency stream processing capabilities. They accomplish this by processing data on a per-event basis, thus by being *event-centric*, where each event is represented as a key-value pair. Apache Flink [15a] is of the latest generation of DSP system and allows mixing both batch and stream processing within the same framework, providing a unified approach to data processing. Other notable stream processing systems include Apache Samza [15b] and Apache Beam [16].

As for batch processing systems, DSP systems provide abstractions for expressing computations as high-level operators, whose execution is handled transparently by the underlying runtime system. A powerful feature of stream processing systems lies in their ability to support stateful computations, enabling state to be maintained across multiple events.

Stream Processing Model. Stream Processing queries are defined as a directed acyclic graph (DAG) where vertices represent *operators* and edges represent the flow of events between them. A complete graph is composed of one or more *sources*, a series of intermediate operators performing transformations or aggregations, and one or more *sinks*. A source is an operator that ingests events from an external system, such as a message queue or a database. A sink is an operator that outputs events to an external system, possibly to another stream processing system. In this thesis, we refer to this representation as the *logical plan* of a stream processing query.

> **Example 2.2.1.** Figure 2.1a illustrates a simple stream processing query that counts the occurrences of words in a stream of text data (Example 2.1.1). The query consists of a source operator that reads text data ①, a series of intermediate operators that split the text into words ② and count their occurrences ③, and a sink operator that outputs the



(a) A simple stream processing query that counts word occurrences in a text stream.

(b) A more complex stream processing query that joins two input streams.

Figure 2.1: Examples of stream processing queries: (a) word count, (b) join of two streams.

word counts ④. The source operator ingests text data from an external system and emits individual lines of text as events. The split operator processes each line of text, emitting a key-value pair for each word (e.g., ("word", 1)) to the count operator. The count operator aggregates these pairs by summing the counts for each unique word. Finally, the sink operator outputs the word counts to an external system, such as a database or a dashboard.

Example 2.2.2. Figure 2.1b illustrates a more complex stream processing query that joins two input streams based on a common key. The query consists of two source operators that read data from two different external systems ⑤, ⑥, a join operator that combines events from both streams based on a common key ⑦, and a sink operator that outputs the joined results ⑧.

2.2.1 Parallel Execution of DSP Queries

The primary goal of DSP systems is to process high-throughput data streams with low latency. The performance of a DSP system is often measured in terms of throughput (i.e., the number of events processed per second) and latency (i.e., the time taken to process an event from ingestion to output). However, operators in a DSP system have limited processing capacity and can become bottlenecks when the incoming data rate exceeds their processing capabilities. To handle high-throughput data streams, DSP systems support both horizontal and vertical scaling. Horizontal scaling refers to the ability to increase or decrease the number of instances of an application to handle varying workloads. This is typically achieved through the use of load balancers, which distribute incoming requests across multiple instances of the application. In this context, we say that the application is *scaled out (in)* if the number of instances is increased (decreased). In contrast, vertical scaling

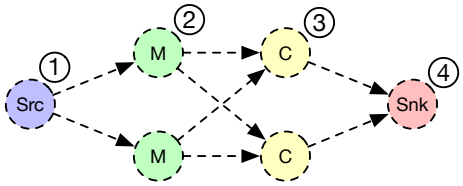


Figure 2.2: Execution plan for the word count query. The Source and Sink operators do not scaled out and keep a parallelism of 1, while the Split and Count operators are scaled out to a parallelism of 2.

refers to the ability to increase or decrease (*scale up* or *down*) the resources (e.g., CPU, memory) allocated to the existing instances of an application.

The state-of-the art approach to achieve better performance in stream processing systems is through horizontal scaling. Operators in a stream processing query can be executed in parallel to handle high-throughput data streams. This is typically achieved by partitioning the input data and distributing it across multiple instances of the operator, called *tasks*. Each task processes a subset of the data, allowing for parallel execution and improved performance. This representation of a query with parallelized operators is called the *execution plan* of a stream processing query.

> **Example 2.2.3.** Figure 2.2 illustrates the word count query from Figure 2.1a scaled out. The source and sink operators are not parallelized, while the split and count operators are each parallelized into two tasks, respectively. The source operator splits the input data and distributes it across its two downstream tasks, allowing each task to process a subset of the data in parallel.

2.2.2 Stateful Stream Processing

In the previous example, the count operator needs to maintain state to keep track of the counts for each word. Stateful stream processing allows operators to maintain state across multiple events, enabling complex computations such as aggregations. Stateful operators can also leverage windowing techniques to group events based on time or count, allowing for volume-based aggregations. For example, a windowed count operator could maintain separate counts for each word within a sliding time window, enabling real-time analytics on recent data.

In stream processing, each events is typically represented as a key-value pair, where the key is used to identify the state associated with the event. The key is often derived from the event's content, such as a primary key in the event's schema. To achieve state consistency in stateful operators, events with the same key must be routed to the same task. This is typically achieved

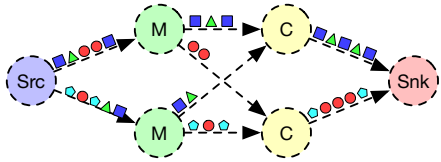


Figure 2.3: Key-based partitioning ensures that all events with the same key (represented by colored shapes) are routed to the same task.

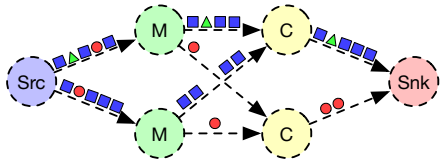


Figure 2.4: Example of a stream processing query experiencing data skew. The upper Count operator task is overloaded due to a popular key (blue square), while the other one is underutilized.

through key-based partitioning, where events are partitioned based on a specific key (e.g., the word in the word count example). This ensures that all events with the same key are processed by the same task, allowing the operator to maintain accurate state.

> **Example 2.2.4.** Figure 2.3 illustrates the key-based partitioning for the word count query. Each colored shape represents a different key (i.e. a different word), and all events with the same key are routed to the same task.

Stateful stream processing systems provide mechanisms for state management [Car+17], including checkpointing and state recovery, to ensure fault tolerance and consistency. These mechanisms allow operators to periodically save their state to durable storage, enabling recovery in the event of failures. Consistent snapshots of the operators state allows the system to restore the state to a known good point in time, ensuring that the processing can resume without data loss or inconsistency. The state is re-distributed to the operators' tasks based on the key-based partitioning scheme.

Data Skew. Data skew refers to the uneven popularity of certain keys in a data stream, leading to an imbalanced distribution of events across tasks. For example, in a word count application, some words may appear much more frequently than others, resulting in certain tasks receiving a disproportionate number of events. Data skew can lead to performance degradation, as tasks processing popular keys may become bottlenecks, while others remain underutilized.

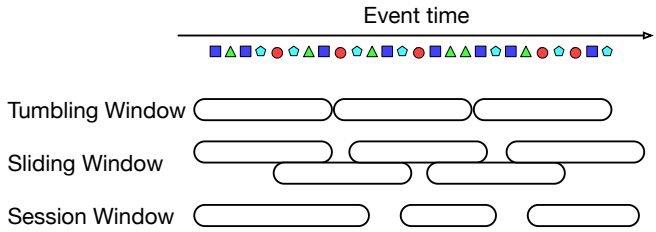


Figure 2.5: Windowing groups events based on time or count. Windows can be tumbling (non-overlapping), sliding (overlapping), or session-based (dynamic based on activity).

> **Example 2.2.5.** Figure 2.4 illustrates a stream processing query experiencing data skew. The upper Count operator task is overloaded due to a popular key (blue square), while the other one is underutilized. This imbalance can lead to increased latency and reduced throughput, as the overloaded task struggles to keep up with the incoming data rate.

2.2.3 Windowing

Windowing techniques are commonly used in stateful stream processing to perform aggregations over specific time intervals or event counts, enabling more granular analysis of the data. Common windowing strategies include tumbling windows, sliding windows, and session windows [Ver+23], as illustrated in Figure 2.5.

> **Example 2.2.6.** In the word count example, a sliding window of 1 minute with a slide interval of 30 seconds could be used to count the occurrences of each word within the last minute, updating the counts every 30 seconds. The windowed count operator would maintain separate counts for each word within each 1-minute interval.

> **Example 2.2.7.** A more complex example involves joining two streams based on a common key within a tumbling window. Performing the join operation on each event would require scanning the entire state of the other stream for matching keys, leading to a significant increase in the number of state operations compared to a simple aggregation. Instead, the two streams can be temporarily maintained in state until the window closes, at which point the join is performed for all events in the window.

The presence of windows in a stream processing query can significantly impact the performance of the system. A common phenomenon observed in windowed stream processing queries is the presence of *stragglers*, i.e., tasks

that take significantly longer to process their assigned data compared to others. Stragglers can arise due to various factors, including data skew, or late arriving events that need to be processed within a specific window. Stragglers can lead to increased latency and reduced throughput, as the overall progress of the query is often determined by the slowest task.

2.2.4 Summary

We summarize the main concepts of this section, used throughout this thesis, as follows:

- A stream processing **query** is represented as a directed acyclic graph (DAG) of operators connected by data flows, called the **logical plan**.
- An **operator** is a logical processing unit that performs transformations or aggregations on the events (e.g., filter, map, flatmap, join).
- An **event** is an individual data item processed by the operators in the query, represented as a key-value pair.
- The **throughput** is defined as the number of events effectively processed per second by the stream processing system. It contrasts with the **rate**, which is the number of events fed into the system per second.
- A **source** is an operator that ingests events from an external system.
- A **sink** is an operator that outputs events to an external system.
- **Parallelization** in stream processing refers to the action of operators to be executed concurrently to handle high-throughput data streams.
- A **task** is an instance of an operator that effectively processes a subset of the data.
- The **execution plan** of a stream processing query is the representation of the query with parallelized operators.
- **Stateful** stream processing allows operators to maintain state across multiple events, enabling complex computations and aggregations.
- **Key-based partitioning** ensures that all events with the same key are routed to the same task, allowing stateful operators to maintain accurate state.
- **Windowing** techniques group events based on time or count, allowing for more granular aggregations in stateful operators.

2.3 Apache Flink

Distributed stream processing execution and orchestration come with a lot of challenges, from state and resource management to fault tolerance and elasticity. To address these challenges, DSP systems, such as Apache Storm [14b], Apache Samza [15b], and Apache Flink [15a], have emerged over the last decade, with Apache Flink being one of the most prominent. It is widely used by many companies, including Alibaba, Twitter, Netflix, Uber, or Zalando [FA16; FS21; Clo21; Doo22; Tem22].

For the remainder of this thesis, we will use Apache Flink (or simply Flink) as the DSP system of choice. We provide an overview of Flink, its architecture, its state management, and its resource management and elasticity features.

2.3.1 Programming Models

Flink provides several interfaces for defining stream processing queries, including Java, SQL, and Python APIs. The most commonly-used API is the Java DataStream API, which provides a high-level abstraction for defining data processing pipelines. The user defines a query by specifying a series of transformations and aggregations on the input data streams. These transformations are provided as a set of built-in operators such as `map`, `filter`, `flatMap`, and `join`. The API also provides support for stateful operations, such as `keyBy` and `window`. The user can also define custom operators via the *Process Functions* API but is responsible for complex aspects, such as state management. Flink then converts the user-defined query into a logical plan, which is then optimized and transformed into an execution plan.

Listing 2.1: A simple word count query in Flink using the Java API.

```
1  StreamExecutionEnvironment env = StreamExecutionEnvironment.
       getExecutionEnvironment();
2  DataStream<String> text = env.fromElements("hello world", "hello
       flink");
3  DataStream<Tuple2<String, Integer>> counts = text
4      .flatMap(new Splitter())
5      .keyBy(value -> value.f0)
6      .sum(1);
7  counts.print();
8  env.execute("WordCount Example");
```

Example 2.3.1. Listing 2.1 shows a simple word count query in Flink using the DataStream API, which provides a high-level abstraction for defining stream processing queries. The query reads a stream of text lines (Line 2), splits each line into words (Line 4), counts the occurrences of each word (Line 6), and prints the results to the console (Line 7). Line 5 ensures that the same words are routed to their respective task. The

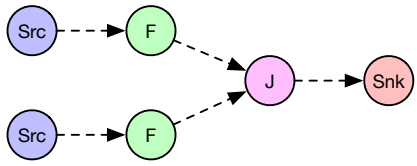


Figure 2.6: Example of the resulting logical plan of a Flink query expressed in SQL. The query from Listing 2.2 shows the jointure of two streams, `auction` and `person`, to retrieve the names and locations of sellers for auctions in a specific category.

> query is then submitted for execution using the `execute` method.

A popular alternative to the DataStream API is the Table API [Fli25], which provides a declarative way to express stream processing queries using SQL-like syntax. Users can use either the Java API, allowing them to express queries with a semantic similar to SQL, or directly write SQL queries. In the later case, Flink provides a SQL parser, using Apache Calcite [14a], that converts SQL queries into logical plans, which are then optimized and transformed into execution plans.

Listing 2.2: A more complex flink query expressed in SQL.

```
SELECT
    P.name, P.city, P.state, A.id
FROM
    auction AS A INNER JOIN person AS P on A.seller = P.id
WHERE
    A.category = 10 and (P.state = 'OR' OR P.state = 'ID' OR P.state =
        'CA');
```

> **Example 2.3.2.** Another example of a Flink query is shown in Listing 2.2, expressed in SQL. This query joins two streams, `auction` and `person`, to retrieve the names and locations of sellers for auctions in a specific category. Flink uses the Table API to convert SQL queries into logical plans, which are then optimized and transformed into execution plans. Figure 2.6 shows the resulting logical plan.

2.3.2 Query Optimization

Flink offers various optimizations to improve the performance of a query, including operator chaining, which combines multiple operators into a single operator to reduce communication overhead. This chaining can be performed on stateless operators (e.g., `map`, `filter`) but is prevented by key-based partitioning (e.g., `keyBy`) or stateful operations (e.g., `window`). For the rest of this thesis, we use the following terminology:

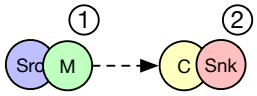


Figure 2.7: Example of physical operators after operator chaining. The logical operators `source` and `map` from Figure 2.1a have been chained into a single physical operator ①. The logical operators `count` and `sink` have also been chained into physical operator ②. Because of the key-based partitioning that prevents operator chaining, physical operator ① is connected to physical operator ② through a network channel.

- We call a **Logical Operator (LO)** an operator defined in the logical plan of a query, prior to any optimization.
- We call a **Physical Operator (PO)** an operator resulting from the optimization process, including operator chaining.[1]

> **Example 2.3.3.** Figure 2.7 shows an example of physical operators resulting from operator chaining. The logical operators `source` and `map` from Figure 2.1a have been chained into a single physical operator ①. The logical operators `count` and `sink` have also been chained into physical operator ②. Because of the key-based partitioning that prevents operator chaining, physical operator ① is connected to physical operator ② through a network channel.

2.3.3 Architecture

Flink follows a master-worker architecture, where a *Job Manager* is responsible for coordinating the execution of stream processing queries, and *Task Managers* are responsible for effectively executing the tasks that make up a query. Both Job Managers and Task Managers are processes running in Java Virtual Machines (JVMs) with their own respective resources.

Job Manager and Task Managers. The Job Manager is responsible for managing the lifecycle of a query, including its submission, scheduling, and monitoring. It also coordinates fault tolerance and recovery (Section 2.3.4), ensuring that the query continues to run in the event of failures. The Job Manager maintains the metadata of the query, including its logical and execution plan, state, and checkpoints. It also communicates with the Task Managers to coordinate the execution of the query.

Task Managers are responsible for executing the tasks that make up a query. A Task Manager is a JVM process with fixed resources (CPU, memory), which can execute multiple tasks concurrently. Each Task Manager has

[1] The graph of physical operators is also referred to as the *optimized plan*.

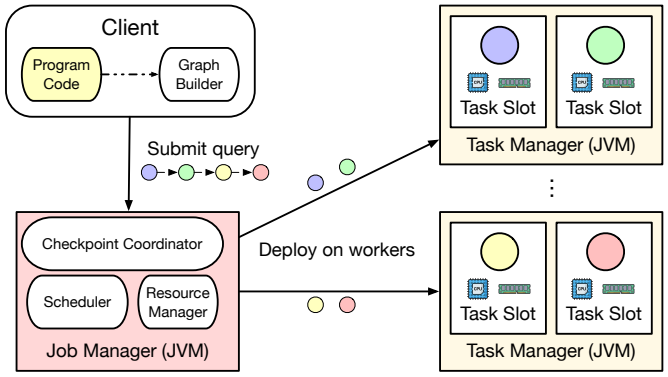


Figure 2.8: Architecture of Flink. The Job Manager is responsible for managing the lifecycle of a query, including its submission, scheduling, and monitoring. Task Managers are responsible for executing the tasks that make up a query.

a fixed number of small computation units, called *task slots*, which effectively receive events, apply the logic of a single task, and forward the results to another task slot. The exchange of data between tasks depends on the locality of the tasks. If two logically connected tasks are executed in the same Task Manager, data is exchanged through in-memory channels. If they are executed in different Task Managers, data is exchanged through network channels, where data is serialized and sent over TCP connections.

Figure 2.8 shows the architecture of Flink, with a single Job Manager and multiple Task Managers. The program submitted by the user, either in Java or SQL, is first parsed and converted into a logical plan. The logical plan is then optimized and transformed into an execution plan, which is then scheduled by the Job Manager across the available Task Managers.

2.3.4 State Management

The state of stateful operators' tasks must be persisted across the processing of different events. The storage must allow for reconfiguration and restoration after failures [Car+17]. Stream processing systems, including Flink, therefore require efficient state management mechanisms to ensure fault tolerance and consistency. Flink provides several state backends for managing state, including in-memory, disk-based, and hybrid state backends. The choice of a state backend depends on the requirements of the application, including the size of the state, latency requirements, and required fault tolerance guarantees.

While in-memory state backends provide low-latency access to state, they may not be suitable for large state sizes due to memory limitations. Disk-

based state backends, on the other hand, can handle larger state sizes but may introduce higher latencies. Flink uses RocksDB [25h] as its default state backend, which provides a balance between performance and fault-tolerance.

RocksDB. RocksDB is an embeddable, high-performance key-value store. It is designed to provide low-latency access to data while ensuring durability and fault tolerance. RocksDB achieves this by using a log-structured merge-tree (LSM tree) data structure, which allows for efficient writes. An LSM tree uses a combination of in-memory and on-disk storage to optimize both read and write performance. Data is first written to an in-memory table (memtable) and periodically flushed to disk as an immutable sorted file (SSTable). SSTables are periodically compacted to merge smaller files into larger ones, helping to reduce the number of files. This design minimizes random I/O and maximizes sequential I/O, resulting in high write throughput. Additionally, RocksDB uses a write-ahead log (WAL) to ensure durability. All writes are first recorded in the WAL before being applied to the memtable. In the event of a crash, RocksDB can recover the state by replaying the WAL. Finally, an LRU-cache is used to further improve read performance by storing frequently accessed data in memory. Read requests are first checked against the memtables and the LRU-cache before being accessed on disk. Bloom filters are also used to quickly determine if a key is present in an SSTable, reducing unnecessary disk reads.

LSM-trees have drawn significant attention. Numerous works have explored the optimization of LSM-trees through various techniques. These techniques include smart compaction scheduling [Bal+19], reduced write amplification [Raj+17], or improved cache management [Bal+17]. More recently, Mei *et. al.* [Mei+25] proposed a disaggregated LSM-tree design that separates the storage and compute layers. The motivation behind this design is to 1) remove the constraint of state size imposed by the local disk, 2) prevent resource contention for state operations, and 3) reduce the duration of checkpoining and reconfigurations. These work address challenges orthogonal to this thesis as they focus on systems used independently by stream processing systems.

2.3.5 Checkpointing and Reconfigurations

Checkpointing is a fundamental mechanism in Flink to ensure fault tolerance and support reconfigurations, such as scaling operations. Flink periodically takes consistent snapshots of the state of all stateful operators' tasks. This mechanism was first introduced by Carbone *et. al.* [Car+17] and is based on the Chandy-Lamport distributed snapshot algorithm [CL85]. The algorithm works by injecting special markers into the data stream, which propagate through the operators' tasks. When a task receives a marker, it takes a snap-

shot of its state and forwards the marker to its downstream tasks.

The Job Manager requests checkpoints at regular intervals, and each Task Manager participating in the query execution takes a snapshot of its state and stores it in a durable storage system, such as HDFS or S3. Checkpoints are taken asynchronously, allowing the query to continue processing events while the checkpoint is being taken. In the event of a failure, the Job Manager can restore the state of the query from the latest checkpoint to prevent re-processing of events. Checkpoints can also be used for other purposes, such as state migration [Din+15; Hof+19].

Reconfigurations. Reconfigurations in Flink can occur in response to various events, such as task failures or changes in the deployed queries' execution plans. After a fault, Flink can use the latest checkpoint to restore the state of the affected tasks and re-establish the data flow. Data sources such as Kafka are used to replay any data that may have been processed following the previous checkpoint. During re-scaling, Flink can adjust the number of task instances for a given operator, either by adding or removing tasks. To do this, Flink stops the sources and takes a specific checkpoint called a *savepoint*. It then re-deploys the query with the new parallelism and restores the state from the savepoint.

Flink uses a consistent hashing scheme to partition the state o f operators' tasks. When scaling out, the state of existing tasks is redistributed among the new set of tasks based on the consistent hashing scheme. When scaling in, the state of the tasks being removed is redistributed among the remaining tasks. This is done with *key groups*, which are tiny partitions of the key space. The number of key groups gives the maximum parallelism allowed for an operator.

> **Example 2.3.4.** Consider an operator with a maximum parallelism of 256 and a current parallelism of 2. The key space is divided into 256 key groups: $KG_{0,\ldots,255}$. With a parallelism of 2, the first task is assigned key groups $KG_{0,\ldots,127}$, while the second task is assigned $KG_{128,\ldots,255}$. If we scale out the operator to a parallelism of 4, task 1 will be assigned key groups $KG_{0,\ldots,63}$, task 2 will be assigned $KG_{64,\ldots,127}$, task 3 will be assigned $KG_{128,\ldots,191}$, and task 4 will be assigned $KG_{192,\ldots,255}$. The state of the original tasks is redistributed accordingly.

2.3.6 Resource Management

Both Job Managers and Task Managers in Flink are allocated a fixed amount of resources (CPU, memory) at startup. The Job Manager typically requires fewer resources than Task Managers, as it is primarily responsible for coordinating the execution of queries. We disregard the resource management of Job Managers in this thesis, as they are not involved in the actual processing

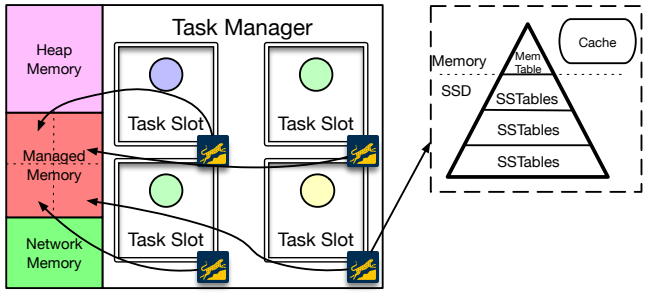


Figure 2.9: Resource management in Flink. Task Managers have a fixed amount of resources (CPU, memory) and a fixed number of task slots. Each task slot receives a fixed amount of resources, determined by dividing the total resources of the Task Manager by the number of task slots. RocksDB uses a Managed memory pool, equally shared among all task slots, to create its memtables and caches.

of data, and focus on the resource management of Task Managers.

Each Task Manager has a fixed amount of resources and a fixed number of task slots. The number of task slots is configurable at the startup of the Task Manager and is typically set to the number of CPU cores available to the Task Manager. Each task slot receives a fixed amount of resources, determined by dividing the total resources of the Task Manager by the number of task slots. This resource allocation strategy therefore allocates the same amount of resources to all task slots in a Task Manager.

CPU Management. Task slots are single-threaded but are not dedicated to a specific CPU core, meaning that multiple task slots can run on the same core. Background tasks, such as checkpointing or state compaction, are executed in separate threads and compete for CPU resources with the tasks.

Memory Management. Memory in a Task Manager is divided into two main categories: *JVM heap memory* and *Managed memory*. Managed memory is the memory pool used by RocksDB to store its data structures, including the memtables and the LRU-cache. The size of the Managed memory pool is configurable and typically set to a fraction of the total memory of the Task Manager. It is equally shared among all task slots in the Task Manager. The remaining memory is used as JVM heap memory, which is used by the Task Manager for either user code or internal data structures. JVM heap memory is also used for network buffers, which are used to exchange data between tasks.

Figure 2.9 illustrates resource management in Flink for a single Task Manager. The Task Manager has four task slots, each receiving a fixed amount of CPU and memory resources. The Managed memory pool is equally divided among all task slots, while the remaining memory is used as JVM heap and

network memory. Each RocksDB instance created by a task slot uses its share of the Managed memory to create its memtables and caches.

2.3.7 Elasticity

As depicted in Section 2.3.6, task slots in Flink receive a fixed amount of resources. Tasks executed in the tasks slots are therefore limited by their allocated resources and can only perform a limited amount of operations per second. When the workload increases, tasks may become overwhelmed and unable to keep up with the incoming data rate. Elasticity, or elastic scaling, is the ability of a system to automatically adapt to workload changes by provisioning and de-provisioning resources. In Flink, it is typically achieved by increasing or decreasing the parallelism of operators in a query.

Elastic scaling has been extensively studied in the context of cloud computing [LEB15; Cou+15; Al-+17; San+20; Ahm23], with various techniques proposed for scaling virtual machines and containers. The techniques often rely on monitoring resources usage and application-level metrics to make scaling decisions, e.g. using threshold-based approaches. Other approaches use model-based techniques to predict the resource requirements of applications and make scaling decisions accordingly [MK20; Rat+23].

Elastic scaling in DSP has also been widely studied [RM19; Car+22]. First generations of elastic scaling, such as Dhalion [Flo+17], relied on threshold-based approaches — such as *"if the CPU usage of an operator exceeds 80%, then scale this operator out"* — to make scaling decisions. Such systems scaled operators one at a time, requiring a significant number of scaling decisions to achieve the desired performance. More recently, Kalavri et.al. [Kal+18] proposed DS2, a model-based approach that scales multiple operators at a time, requiring fewer reconfiguration steps than with the previous approaches to converge to the desired performance. A variant of DS2 has been integrated into Flink as the default auto scaler and is considered to be the state-of-the-art in elastic scaling for DSP.

The DS2 Algorithm. DS2 builds on the principle that the performance of a stream processing query is determined by its bottleneck operators. A bottleneck operator is an operator that is unable to keep up with the incoming data rate, causing backpressure in its upstream operators. DS2 first introduces the notion of *busy time* (or *useful* time), which is the fraction of time an operator spends processing events over one second. An operator can be in one of three states:

- **Idle** if it is not processing any events, i.e. its upstream operators are not producing any events.
- **Busy** if it is effectively processing events. This includes both the time

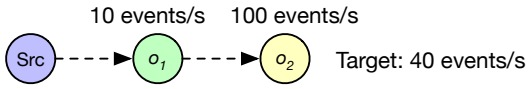


Figure 2.10: Example of an under-provisioned query. Operator o_1 can process up to 10 events per second, while operator o_2 can process up to 100 events per second. The target throughput of the query is 40 events per second but is limited to 10 as it is capped by operator o_1.

spent (de)serializing events, applying the operator logic, and forwarding the results to downstream operators.

- **Backpressured** if it is unable to forward its processed events to its downstream operator(s). This happens if the downstream operator(s) are busy or backpressured. Backpressured operators cascade the backpressure to their upstream operators, which may also become backpressured.

At any given time, the sum of the fractions of time an operator spends in each state is equal to 100%. A busy time of 100% means that the operator is continuously processing events and has reached its maximum processing capacity. A busy time below 100% means that the operator is not fully utilizing its allocated resources. Following this principle, DS2 introduces the notion of *true processing rate*, which corresponds to how many events an operator can process per unit of *useful* time. This contrasts with the *observed processing rate*, which corresponds to how many events an operator processes per unit of observed time.

> **Example 2.3.5.** Consider the query from Figure 2.10, where operator o_1 is a bottleneck. Operator o_1 can process up to 10 events per second, while operator o_2 can process up to 100 events per second. The target throughput of the query is 40 events per second, but the maximum throughput is limited to 10 events per second by operator o_1. Operator o_1 is therefore busy 100% of the time, while operator o_2 is busy 10% of the time and idle 90% of the time. The observed processing rate of operator o_1 is 10 events per second, while its true processing rate is also 10 events per second. The observed processing rate of operator o_2 is 10 events per second, while its true processing rate is 100 events per second. To increase the throughput of the query to 40 events per second, we can increase the parallelism of operator o_1 from 1 to 4, allowing it to process up to 40 events per second — assuming linear scalability.

Model Formulation. The DS2 first recursively computes the *aggregated true output rate* of each operator, defined as the rate at which an operator can

process events when itself and its upstream operators are deployed at their optimal parallelism to keep up with the incoming data rate. It is computed starting from the source operators and propagating downstream. The *optimal parallelism* of each operator is then computed using the ratio of aggregated true output rate of its upstream operators to the average true processing rate of the operator itself.

The DS2 algorithm makes several assumptions, including that operators scale linearly with their parallelism and that the true processing rate of an operator remains constant over time. Another strong assumption is that the target throughput of the query is known in advance. In practice, the data sources are dictated by external systems, such as message queues or databases, and the target throughput may not be known or show important fluctuations.

2.3.8 Summary of Concepts

In this section, we provided an overview of Flink, its architecture, its state management, and its resource management and elasticity features. We now summarize the main concepts of this section, used throughout this thesis, as follows:

- A **Job Manager** is a JVM process responsible for coordinating the execution of stream processing queries.
- A **Task Manager** is a JVM process responsible for executing the tasks that make up a query.
- A **task slot** is a fixed computation unit within a Task Manager that executes a single task.
- **RocksDB** is the default state backend in Flink, providing performant write operations and fault-tolerance.
- **Managed Memory** is a memory pool within a Task Manager that is used by RocksDB. It is equally shared among all task slots in the Task Manager.

2.4 Orchestration with Kubernetes

Flink applications can be deployed and managed on various platforms, including on-premises clusters, cloud platforms, and container orchestration systems. The more popular approach for deploying Flink applications is through container orchestration platforms, such as Kubernetes. Job Managers and Task Managers are deployed as containers with specific resource requirements (CPU, memory) and configurations. Kubernetes then manages the lifecycle of these containers.

Kubernetes [25g] is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a robust and scalable platform for running distributed applications, including DSP systems like Flink. Applications are deployed as *pods*, which are the smallest deployable units in Kubernetes. A pod can contain one or more containers, which share the same network namespace and storage volumes. Kubernetes provides various features for managing the lifecycle of pods, including rolling updates, self-healing, and scaling.

Flink Kubernetes Operator. Deploying and managing Flink clusters on Kubernetes can be complex and prone to errors when done manually. Numerous *Kubernetes Operators* have been developed to simplify the deployment and management of Flink clusters on Kubernetes. The general idea behind Kubernetes Operators is to extend the Kubernetes API with custom resources and controllers that automate the deployment and management of specific applications. These operators encapsulate the best practices and knowledge required to deploy and manage the application effectively, giving users a simplicist interface to interact with.

Listing 2.3: Example of a Flink cluster deployment on Kubernetes using the Flink Kubernetes Operator.

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkCluster
metadata:
  name: example-flink-cluster
spec:
  image: flink:latest
  jobManager:
    resource:
      cpu: 1
      memory: 2048m
  taskManager:
    resource:
      cpu: 4
      memory: 4096m
    numberOfTaskSlots: 4
```

Spotify initially developed the Flink Kubernetes Operator [Spo23] to manage Flink clusters on Kubernetes internally, altough it is no longer actively maintained. The Flink Kubernetes Operator [25c] is a collective effort from the Flink community to provide a native Kubernetes experience for deploying and managing Flink clusters. It provides a custom resource definition (CRD) for defining and managing the lifecycle of Flink clusters. Listing 2.3 shows an example of a Flink cluster deployment on Kubernetes using the Flink Kubernetes Operator. The deployment specifies the resource requirements for the Job Manager and Task Managers, as well as their own specific configurations (e.g., number of task slots per Task Manager). Initially, only the Job Manager is deployed. The Task Managers are then deployed by the Job Manager based

on the resource requirements specified in the deployment. The Flink Kubernetes Operator also takes care of removing excess Task Managers when they are no longer needed.

Listing 2.4: Example of a Flink job submission on Kubernetes using the Flink Kubernetes Operator.

```
apiVersion: flink.apache.org/v1beta1
kind: FlinkSessionJob
metadata:
  name: query1
spec:
  deploymentName: example-flink-cluster
  job:
    jarURI: https://forge.uclouvain.be/DonatienSchmitz/thesis/-/raw
        /main/nexmark/Query1.jar
    parallelism: 1
```

Jobs are submitted to the Flink cluster through the Job Manager, which is exposed as a Kubernetes service. The job submission is done using a CRD called `FlinkSessionJob`. Listing 2.4 shows an example of a Flink job submission on Kubernetes using the Flink Kubernetes Operator. The job submission specifies the JAR file containing the job code and the desired initial parallelism for the job.

DS2 on Flink Kubernetes Operator. The DS2 algorithm presented in Section 2.3.7 has recently been integrated in the Flink Kubernetes Operator, allowing for dynamic scaling of Flink applications running on Kubernetes. This implementation however, slightly differs from the original DS2 algorithm. Although the original DS2 paper made the assumption that the target throughput is known in advance, this is not the case in the context of the Flink Kubernetes Operator. The target throughput is not known beforehand, as it depends on the workload submitted to the Flink cluster. To address this, the DS2 implementation in the Flink Kubernetes Operator uses a threshold-based approach to determine the optimal parallelism of operators.

Instead of computing the optimal parallelism based on a target throughput, the user can specify a target average busy time in which the system aims to keep the operators busy. The parameters `scaleUpThreshold` and `scaleDownThreshold` define the upper and lower bounds for the average busy time. The target optimal parallelism is then computed based on the observed average busy time of the operators. With the lack of a known target throughput, this implementation often leads to more scaling decisions before stabilization than the original DS2 algorithm but remains effective in adapting the resources needed to the workload.

2.5 Apache Kafka

Apache Kafka [KNR+11] is a high-performance stream platform designed for fault tolerance and horizontal scalability. A Kafka deployment uses several storage servers, or *brokers* (each hosting multiple hard drives or SSDs). Datasets in Kafka are streams of *events* grouped as *topics*. Each topic is partitioned, typically using a hash function over event identifiers. Each partition is stored by several *brokers* for fault tolerance. A Kafka client connects to one or more partitions to receive new events or replay events starting from a specific index. Events in Kafka are structured according to a *schema*, i.e., a pre-determined number of fields with a given type. Kafka also supports basic processing, e.g., using KafkaStreams [Bej24].

Kafka topics are often used as data sources and sinks for DSP systems, including Flink. Topics use offsets to track the position of consumers in the stream. When a Flink job reads from a Kafka topic, it periodically commits the offsets of the events it has processed (i.e. during checkpoints). In case of a failure or reconfiguration, the Flink job can resume processing from the last committed offsets, ensuring exactly-once processing semantics.

2.6 Summary

In this chapter, we provided the necessary background to understand the context of this thesis. We introduced a historical overview of data processing leading to distributed stream processing (DSP). We then presented Apache Flink, the DSP system used in this thesis, discussing its architecture, state management, fault tolerance, and resource management. Finally, we discussed container orchestration with Kubernetes, focusing on the Flink Kubernetes Operator for deploying and managing Flink clusters and jobs.

This background sets the stage for the subsequent chapters. First, we characterize a DSP workload in the context of a production data lake in Chapter 3. This study allows us to identify the challenges and requirements for efficient stream processing in such environments, which we present in Chapter 4. These challenges motivate our research on improving resource management, elasticity, and resource usage predictability in DSP systems, which we address in the following chapters.

Characterizing a Real-World Stream Processing Workload

3

The previous chapter provided background on distributed stream processing (DSP) with a focus on Apache Flink. In this chapter, we analyze the characteristics of an actual data lake production workload, including 129 queries and 142 data sources. This analysis is motivated by our collaboration with Eura Nova, an industrial partner operating this data lake. During discussions with Eura Nova, we identified key challenges they face in managing their stream processing infrastructure. To better understand these challenges, we conducted a comprehensive analysis of the workload collected from their stream processing system.

Objectives. We study an industrial deployment of Digazu [Nov24], a modern data lake solution commercialized by Eura Nova. Digazu streamlines data management and interrogation workflows. It provides simple graphical interfaces for data management and the execution of SQL queries on streams of historical and new data. For its back-end, Digazu leverages scale-out deployments of Kafka and Flink. This backend is deployed on a private or public cloud using the Kubernetes container management system [25g].

The deployment we observe is at a large international company.[1] This company deploys complex business analytics on a combination of multiple IoT data streams and real-time data flows resulting from their customers' and employees' interactions with their IT infrastructure. Data scientists and business experts use this Digazu deployment to run a variety of queries. These queries target either historical data recorded in Kafka (*batch queries*), stream data ingested live by Kafka soon after its production (*streaming queries*), and very often a combination of both (*hybrid queries*). The latter ones first replay historical data from Kafka and then switch to processing newly-arriving data as it is ingested. The hybrid batch-stream query model relies on processing the entire data set, resulting in the absence of temporal windowed operations in queries.

[1] While this client company authorized this observation and the publication of our results, our confidentiality agreements do not allow us to reveal its name.

We instrumented the back-end of this Digazu instance (i.e., Kafka and Flink on Kubernetes). We collected application-level metrics about data sources and queries for a representative period of 24 hours, together with system-level metrics about the configuration of these systems (e.g., scale-out configurations used by Flink queries). Queries in Digazu run for long periods, often spanning days or weeks. Therefore, the workloads captured represent snapshot of the system's behavior during this 24-hour period.

The objective of this study is not to produce a benchmark that will be reused across this thesis to evaluate our contributions. Without access to the raw data and queries, it is impossible to faithfully reproduce these workloads. Rather, we aim to identify the key challenges of DSP by analyzing a real production workload. Our focus is on characterizing workload properties, resource-usage patterns, and query structures.

The sensitive nature of the data and queries processed in this deployment prevents us from using the raw workloads directly. The scripts and tools used to extract and analyze the workloads were approved by the client company but their execution remained under the control of Eura Nova. Therefore, findings will inform research directions instead of serving as a standardized benchmark.

Outline. The remainder of this chapter is organized as follows. Section 3.1 provides background information data lakes. In Section 3.2, we describe our methodology for collecting and analyzing the workload. Section 3.3 presents the results of our analysis, highlighting key workload, and query characteristics. Finally, Section 3.5 summarizes our findings and discusses their implications for the design and management of DSP systems.

3.1 Data Lakes

The type of production platform we have been able to observe is a *data lake*. A data lake integrates data ingestion, storage, and analysis and enables informed business decisions [Nar+19]. As a logically centralized platform, it allows business analysts to access in a single location historical and newly-arriving data, or *sources*, and execute processing queries to extract business intelligence (following the "Extract-Load-Transform" pattern [HCM22]). The results of queries are then stored in the data lake, forming new sources. A data lake facilitates the complete lifecycle of data management, from metadata handling and data governance to the orchestration of queries [RZ19].

Logically centralizing data and processing emerged via the data *warehouse* concept for structured, relational data [Gar98]. Data lakes extend this idea to unstructured data and large-scale settings. They rely on *scale-out* backends for storage and processing. Early data lakes focused on at-rest data, prominently using Apache HDFS [KC13] for storage and Hadoop [Whi12]

for processing. Apache Spark [Zah+16] added support for iterative and interactive queries. Latest-generation data lakes integrate support for stream data. The ability to quickly react to incoming data and update the output of *persistent* queries is, indeed, a key business asset [Car+20]. The couple formed by Apache Kafka [RP23] and Apache Flink [15a] is probably the most popular backend today for high-performance data lakes integrating fault-tolerant, high-performance storage with support for batch, streaming, and hybrid queries [Clo21; Doo22; 24a].

3.2 Methodology

We detail how we analyzed our workloads, including constraints and limitations due to the industrial production setting and confidentiality agreements.

Collection of workloads. The production deployment of Digazu was instrumented to collect information from Kafka and Flink operation and at the system/infrastructure level. We developed custom monitoring scripts and configurations to gather relevant metrics while minimizing the performance impact on the production system. The scripts were executed under the supervision of Eura Nova's IT team to ensure compliance with their operational policies. The resulting workloads were stored securely and analyzed from inside the Eura Nova premises to respect data privacy and confidentiality agreements. Finally, all collected statistics were anonymized before analysis to protect sensitive information about the company's operations.

The complete infrastructure runs as a collection of containers managed with Kubernetes. For this particular client, Digazu is deployed on a cluster of 10 nodes, each with 32 CPU cores and 64 GiB of RAM. The cluster hosts 100 Task Managers, each equipped with 2 CPU cores and 2 GiB of RAM. Each Task Manager can host up to 4 task slots. Three Kafka brokers are deployed, each with 4 CPUs, 4 GiB of RAM, and a 3 TiB storage. For both system- and application-level metrics, we leveraged Prometheus [24b] for collection and storage. Our workloads span one full day (24 hours) of data (from 10 am to 9:59 am the next day). This observation period was chosen to represent a typical business day for the company. The resulting workloads analysis contains both static information (e.g., the structure of submitted queries) and dynamic information collected as time-series aggregates by Prometheus (e.g., the rate, in events per second, processed by a Flink task). All time series have a periodicity of 10 seconds. We do not have access to system-level metrics from the underlying Kubernetes cluster (e.g., CPU and memory usage of nodes and pods).

Collected metrics and anonymization. We collect metrics from two main origins: (1) data sources from Kafka, and (2) queries running on Flink. This

data collection follows strict data access and anonymization rules agreed with the company. This impacts the completeness of the data compared to cleartext data collection, but it was a necessary compromise for the realization of our study.

3.2.1 Topics

The data lake host 466 Kafka topics, some of which are used as data sources and others as sinks. The 129 queries recorded in the workloads access 142 of these different topics as sources. These topics correspond to a large diversity of sources within the company, e.g., IoT streams, live events from user interactions, or the integration of existing data silos (e.g., relational databases). They contain a mix of at-rest and live data. Although we do not have direct access to the nature of the sources, i.e., statistics on the events, we were able to retrieve their total volume of at-rest data. We can observe when a hybrid or batch query replays complete topics (i.e., from index 0) and infer how many events were processed as part of such an initial batch phase. Following this phase, we can observe the topic's streaming rate of newly-arriving events. We do not have access to the cleartext schemas used for events stored in Kafka topics, but we can infer most of this information from queries. Topic names are made visible by the TableAPI, along with their field names. We record the *number* of fields (schema size) using source Logical Operators' (LO) readings. The *type* of each field is unknown to the source (LO), e.g., `int`, `string`, `date`, `boolean`, etc. However, we can infer this field type in 80% of the cases by analyzing the query graph and its LOs. Indeed, many LOs explicitly enforce a field type for their operation from which we can make this deduction.

3.2.2 Queries

The 129 queries we observe from 92 users are all expressed in SQL using Flink's Table API. For each query, we extract its logical structure, i.e., the set of LOs, their nature (e.g., source, sink, `filter`, `join`, ...), their connections in the query graph, and which ones need to maintain state. In addition, we record the physical query implementation, i.e., after Flink compiles it for execution: we record the number of Physical Operators (PO), which LOs they aggregate, and which key (field identifier) is used for keyed streams crossing PO boundaries.

All queries ran over the full, 24-hour observation period. We do not have access to the initial submission time or responsible user.

The query deployment maps each PO to several physical tasks. This degree of parallelism is chosen by platform administrators (i.e., there was no

elastic scaling during the day of observation). We record this degree of parallelism for each PO.

For each physical task of a running query, we collect several time series about its execution. The output rate and written Bytes give the average number of events and volume of data this task outputs per second. Aggregating over its tasks yields the same metric for a complete PO. Input rate and read Bytes function similarly.

Some POs contain one or more stateful LO(s), making them *de facto* stateful. Knowing the state size of a stateful PO's task(s) is interesting for memory provisioning and capacity planning [RSR24a; Agn+24]. Collecting the exact state size requires instrumenting the state backend of Flink (RocksDB), which we evaluated as having a too large impact to be deployed on this production system. Instead, we rely on checkpoint sizes as a proxy for state size (see Section 2.3.5). We build upon the fact that Flink queries' states are periodically checkpointed to disk [Car+17] for recovery after faults and reconfigurations. The size of incremental checkpoints is an indicator of the state size and its evolution over time. We collect it together with the checkpoint collection time.

Correlation analysis. We further perform correlation analyses using the Pearson correlation coefficient [Ben+09], which we denote P_c in this thesis. The Pearson correlation coefficient is widely used in statistics to measure the strength and direction of the linear relationship between two continuous variables. Given two variables X and Y, P_c is defined as:

$$P_c = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$$

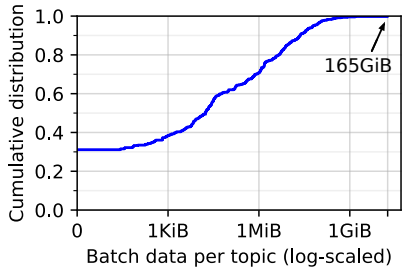
where $cov(X,Y)$ is the covariance of X and Y, σ_X and σ_Y are the standard deviations of X and Y, and μ_X and μ_Y are the means of X and Y. P_c ranges from -1 to 1, where 1 indicates a perfect positive linear correlation, -1 indicates a perfect negative linear correlation, and 0 indicates no linear correlation.

3.3 Workload analysis

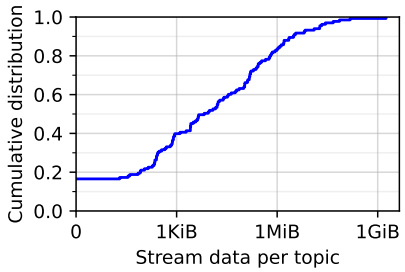
We analyze our workloads in three phases. First, we explore the nature of *data* stored in Kafka (§3.3.1). Then, we characterize the structure of *queries* (§3.3.2) and their execution (§3.3.3).

3.3.1 Data in and from Kafka

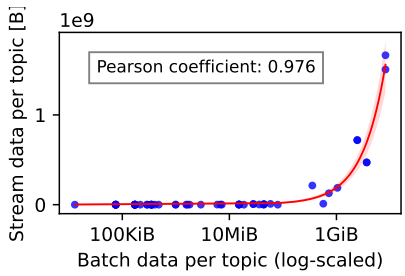
The Digazu data lake hosts 466 topics of varying sizes, with the largest topic containing 165 GiB of data. Among those topics, 285 (~61%) are direct data sources, while the remaining 181 (~39%) are not used as sources for other



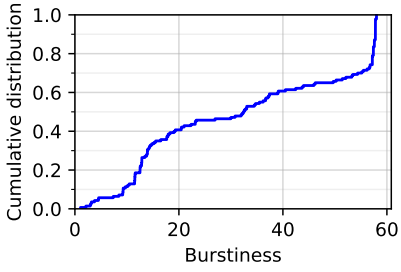
(a) Batch data per topic.



(b) Stream data per topic (during observation period).



(c) Correlation between batch and stream data per topic. ($P_c = 0.976$)



(d) Distribution of stream data burstiness.

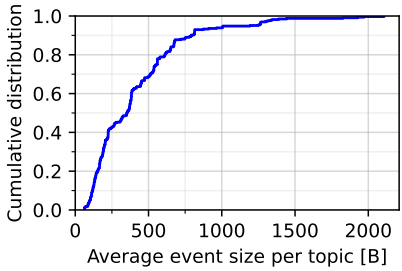
Figure 3.1: Data volumes across Kafka topics.

queries. A total of 142 Kafka topics were used by *at least* one query during our 24-hour observation period. During this period, these topics received a total of 19,607,011 events for 20 GiB of data.

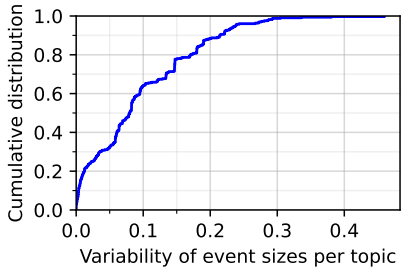
Figure 3.1 presents the volume of batch and stream data per topic. Figure 3.1a shows the volume of batch data per topic, which is the data that was already present in Kafka before our observation period. We see that the majority of topics have a small amount of batch data, with only a few topics accounting for the majority of the data. Figure 3.1b shows that ~18% of the topics did not ingest new events during the observation period, while a single topic has ingested ~16 GiB worth of events. Overall, we identify a significant disparity in the volume of stream data per topic with a median value of ~66 KiB.

We further evaluate a possible linear correlation between the volume of stream data generated by topics and their initial batch data size. Figure 3.1c shows the scatter plot of the two metrics (the X-axis is displayed in log scale for readability) for the 142 topics used during the observation period. A value of P_c of 0.976 indicates a strong linear correlation between the two metrics.

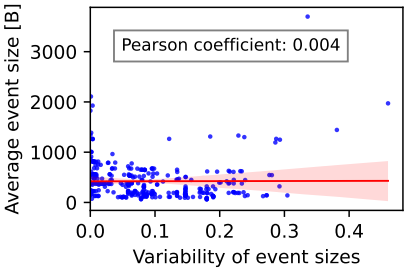
We then analyze the topics' "*burstiness*", i.e., their tendency to produce events in bursts rather than continuously. Figure 3.1d shows the CDF of a



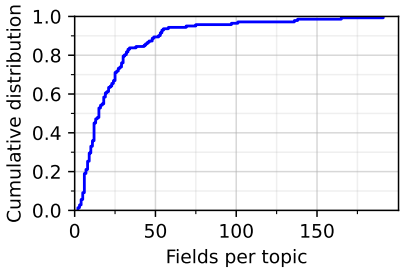
(a) Distribution of average event sizes.



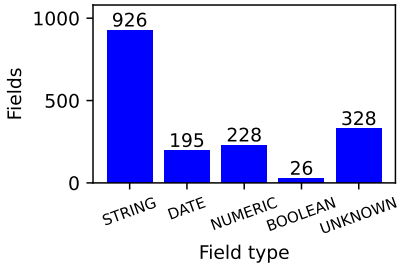
(b) Distribution of event sizes variability.



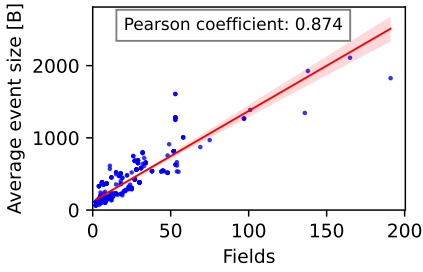
(c) Correlation between average event size and its variability. ($P_c = 0.004$)



(d) Distribution of schema size (#fields).



(e) Frequency of field types.



(f) Correlation between event schema size and average event size. ($P_c = 0.874$)

Figure 3.2: Events characteristics across Kafka topics. Red thin line indicates the linear regression, and the red-shaded zone represents the 95% confidence interval.

burstiness metric that we define as the peak-to-average ratio, i.e., the ratio between the maximum throughput observed in a 5-minute time window and the average throughput. A burstiness value of 1 indicates a perfectly steady flow of events. Bursty patterns have greater burstiness values. The results show that the large majority of topics produce events in bursts. We evaluate a possible linear correlation between the volume of data generated by topics and their burstiness. A value of $P_c = -0.13$ indicates the absence of a clear linear correlation.

In Figure 3.2, we further analyze the nature of events in these topics. Figure 3.2a shows the average event size distribution across topics, ranging from 60 B to 3,700 B, with a median of 353 B. To understand whether this small average event size does not "hide" an unbalanced size distribution mixing small and large events, Figure 3.2b plots *variability*, i.e., the ratio between standard deviation and average of event sizes. For 60% of the topics, event size has low variability (< 10%). For the remaining 40%, this variability remains very moderate (< 50%).

We evaluate the correlation between the average event size and variability (normalized as a fraction of this average size). Figure 3.2c is a scatter plot showing, for each topic, a point linking its average event size and its variability. In addition to P_c, we plot the linear regression using a red line along with a 95% confidence interval using a red-shaded zone. The coefficient between average event size and topic size variability is $P_c = 0.004$, indicating no linear relation between the two metrics.

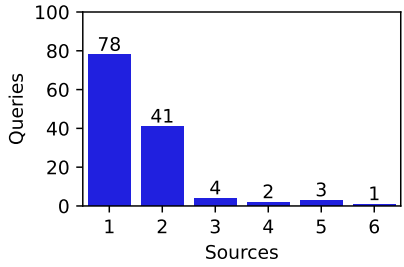
We now analyze the schema characteristics across the 142 topics. Figure 3.2d presents the distribution of the number of fields. Schema sizes range from 2 to 191 fields, with a median of 15 fields. The distribution of the 80% of types information we could infer from queries (as detailed in Section 3.2) is given by Figure 3.2e. Most identified field types are strings, followed by numerical values, dates, and booleans. The correlation between the average event size and the schema size is given by Figure 3.2f, with a strong positive linear correlation ($P_c = 0.874$). The primary factor contributing to event size appears to be the number of fields rather than some particularly large values over a subset of these fields.

Takeaway 1. Data received in the data lake is highly heterogeneous in volume but consists of relatively small events with limited size variability. The principal factor for event size is the size of the schema, possibly featuring many fields.

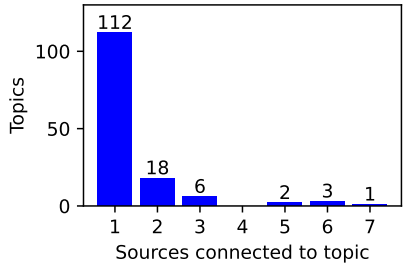
3.3.2 Queries' static properties and structures

We now analyze the nature of the 129 queries submitted to Flink. In this subsection, we first look at the structure of queries, both upon submission (i.e., the graph of LOs) and after compilation and optimization by Flink (i.e., the resulting set of POs).

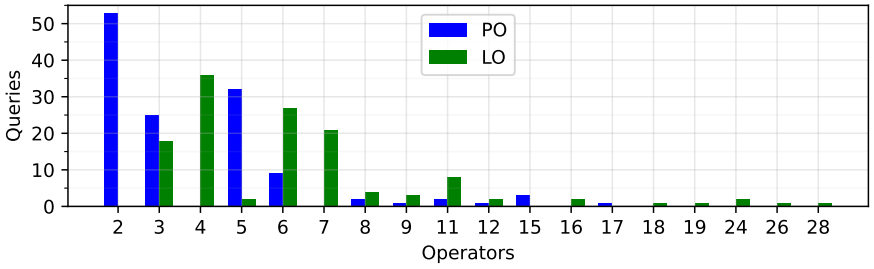
We first look at the source Kafka topics used by queries (hereafter denoted as "sources" for brevity). Figure 3.3a shows that 78/129 (~60%) of the queries use a single source. 41 (~32%) use two sources, and only 10 (~8%) use more, with up to 6 sources. Figure 3.3b shows the number of queries reading from each of the 142 Kafka topics (note that, as we do not have access to the full list of Kafka topics but only infer those used by queries, each topic is sourced



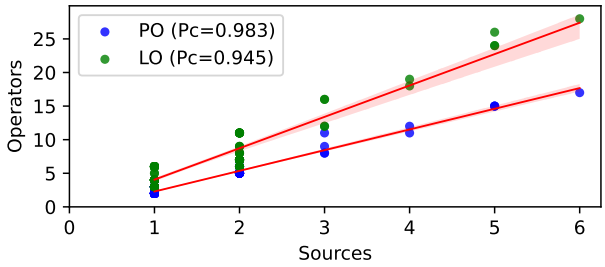
(a) Sources per query.



(b) Sources per topic.



(c) POs and LOs per query.

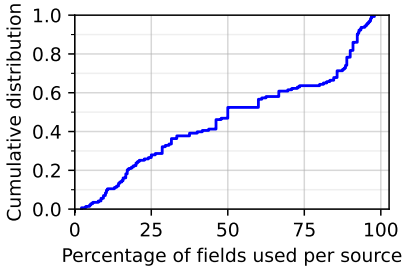


(d) Scatter plot of the number of POs/LOs and sources per query.

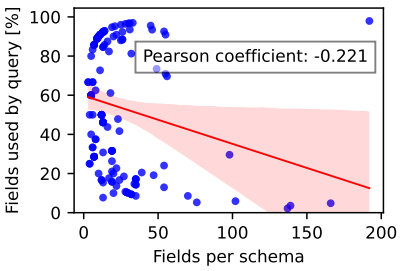
Figure 3.3: Queries sources and graph size.

by at least one query in the workload). 112/142 (~79%) of the topics are used by a *single* query. The other 30 topics are used by a small number of queries, with up to 7 queries for a single topic. This may be a surprising result: a classical expectation (the 80/20 rule) is that a large number of queries use a small number of topics. In fact, the largest topic by volume is accessed by a single query, and there is no clear correlation between the volume of data and the number of queries using a topic.

Figure 3.3c presents the distribution of the queries graph size before and after operator merge (i.e., number of LOs and POs). We observe that merging effectively reduces the number of physical operators that have to be deployed using physical tasks in the Flink cluster: the majority of queries have less than



(a) Distribution of useful fields per query. The X-axis represents the pourcentage of fields present in the source effectively used by the query.



(b) Correlation between percentage of *useful* fields and initial schema size. ($P_c = -0.221$)

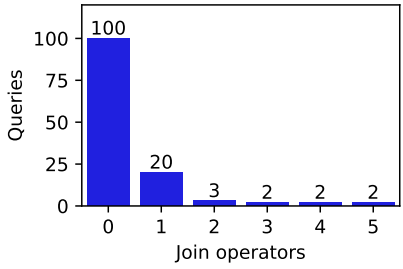
Figure 3.4: Statistic on topic schema usage.

4 POs, and the distribution is shifted leftwards. Queries have from 3 to 28 LOs but at most 17 POs.

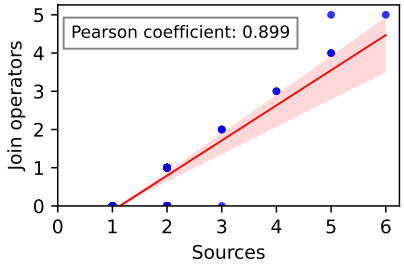
The main factor preventing the merge of LOs into POs is the presence of stateful operators in the query (see Section 2.3.2), particularly `join` LOs, as the two flows consumed by such operators are generally not from the same original source and/or not partitioned using the same key. Multi-source queries are typical of this case. Figure 3.3d relates the query graph sizes to the number of sources, establishing a strong linear correlation in both cases ($P_c = 0.945$ for LOs and $P_c = 0.983$ for POs). Most queries with 2 or 3 POs (53/79, or ~68%) are *stateless* ones with a single source, i.e., performing simple transformations on individual events. In fact, we identify 8 queries (~6%) that only filter events and 45 queries (~35%) either transforming or projecting events.

We then analyze what we call the percentage of *useful* fields of a source. While reading data from a Kafka topic, Flink fetches the whole schema of the records even if only a subset of the schema is necessary for the query. This results in unnecessary network traffic and larger deserializations. We define the ratio of *useful* fields as the ratio between the number of fields necessary to execute the query's logic and the number of fields in the initial schema. Figure 3.4a shows the distribution of ratios for each source. Note that some sources did not expose the schemas to Flink and were omitted from the figure. The results indicate that the majority of sources transfer unnecessary data with a median of 50% of fields being *useful*. We also note that none of the queries used all the fields of a source. Finally, we analyze a possible correlation between the percentage of *useful* fields and the initial schema size. Figure 3.4b shows a scatter plot of the two metrics. The correlation coefficient has a value of $P_c = -0.221$, indicating a weak negative linear correlation.

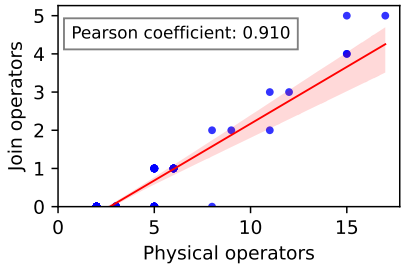
We further analyze the operators used in the queries, focusing on *state-*



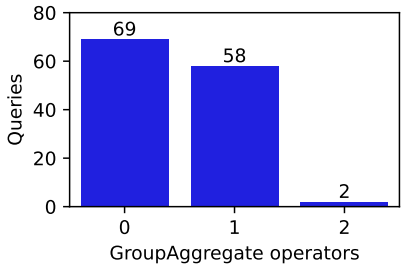
(a) `join` LOs per query.



(b) Correlation between number of `join` LOs and sources. ($P_c = 0.899$)



(c) Correlation between number of `join` LOs and number of POs. ($P_c = 0.910$)

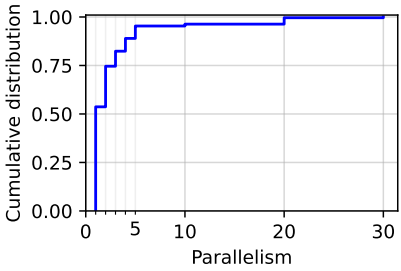


(d) `group-by/aggregation` LOs per query.

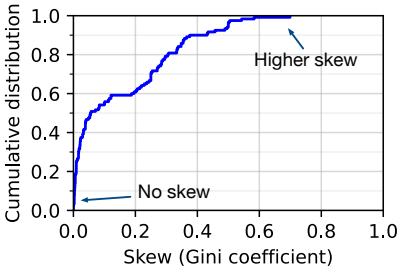
Figure 3.5: Statistics on queries' operators.

ful operators. As previously stated, the query model of hybrid batch-stream processing results in the absence of time-based operations, such as windows. As a result, none of the stateful operators uses windowed operations, i.e., all compute over the entire data set from their corresponding sources.

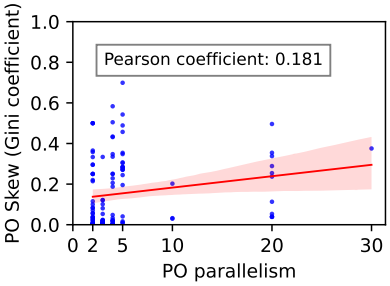
Our results are in Figure 3.5. 76/129 (~59%) of the queries have at least one stateful LO (and, therefore, at least one stateful PO). 29/129 (~22%) of the queries have at least one `join` LO (Figure 3.5a); only a few queries have 2 or more, and none has more than 5. Unsurprisingly, there is a strong linear correlation ($P_c = 0.910$) between this number and the number of sources as `join` LOs mostly consume data from distinct streams (Figure 3.5b). Note, however, that some queries have more `join` LOs than sources, as a single topic may be *split* by different `filter` LOs before joining the two resulting sub-streams. The strong linear correlation between the number of `join` LOs and the total number of POs in the optimized query graph (Figure 3.5c) concurs with our previous observation that such operators are the principal obstacle to operator merging and, as a result, a key factor in query complexity and deployment cost. Finally, we observe the distribution of the number of `group-by/aggregate` LOs (i.e., collecting data for individual keys in a partitioned stream and computing `sum, count, min, max, ...` ag-



(a) CDF of the degree of parallelism across all POs of all queries.



(b) CDF of the skew metric of all POs with parallelism > 1.



(c) Correlation between parallelism degree and skew for all POs with parallelism > 1. ($P_c = 0.181$)

Figure 3.6: Runtime statistics of queries.

gregates for each key). 69/129 queries (~53%) have no such LO, while the rest (60/129) have one or, for only two queries, two of them.

Takeaway 2. Queries tend to be relatively complex and include stateful operators. Operator merging is successful in many cases but limited by `join` operators, making queries with multiple sources intrinsically more complex to deploy and run. On the other hand, the distribution of sources' popularity is relatively balanced.

3.3.3 Dynamic query statistics

In this final phase, we analyze the runtime behavior of queries. We report some configuration choices made in the observed deployment, e.g., on the parallelism of POs. Then, we study the load distribution between parallel tasks of a given operator to measure event skew. Last, we study state size indirectly by analyzing checkpointing metrics.

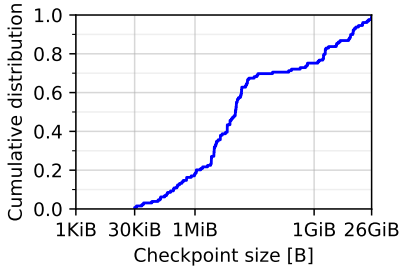
The POs of the 129 queries are deployed over 1,365 tasks. Figure 3.6a presents the distribution of tasks per PO. More than half of POs (~53%) are not scaled out, i.e., they have a parallelism of 1. Conversely, some POs run

with a parallelism of 10, 20, or even 30 tasks. The engineers provisioned tasks statically to cope with the bursts of data associated with the (re)start of queries and the system does not use elastic scaling. As a result, operators' tasks may be underutilized outside of these bursts. Unfortunately, we cannot access systems-level metrics to validate this intuition.

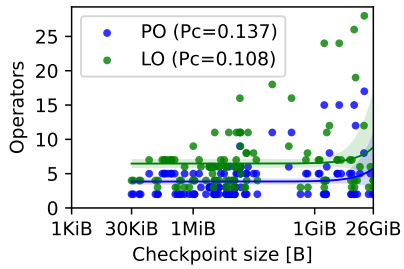
We continue by analyzing data skew, i.e., the unbalance in input data distribution across a PO's tasks. We measure this skew using the Gini coefficient [Mil97], a metric commonly used in economics and social sciences and also to measure the fairness of a load distribution [PT07; GC15]. The Gini coefficient, which we note G_c hereafter, is computed on the distribution of rates (in events) across all tasks of a PO. G_c ranges from 0 for a perfectly balanced distribution to 1 for a distribution in which one task gets all the input traffic. Figure 3.6b shows the skew metric G_c distribution for all POs with more than one task. A minority of POs have a highly skewed distribution of inputs over their tasks. Figure 3.6c shows the correlation between the parallelism of POs with more than one task, and their skew metric G_c. With $P_c = 0.181$, there is a modest linear correlation between the two variables: Larger POs are also subject to skew, and slightly more so than smaller ones.

We now analyze statistics about checkpoints in Figure 3.7. Checkpoints are taken at regular intervals, i.e., every 10 minutes in our workload. Snapshotting the state of a query requires waiting at each of its PO for specific *marker* events from all upstream flows and storing a local snapshot before resuming processing (see Section 2.3.4). We collect two metrics: the combined size of these local PO state snapshots and the downtime incurred for the query by collecting them.

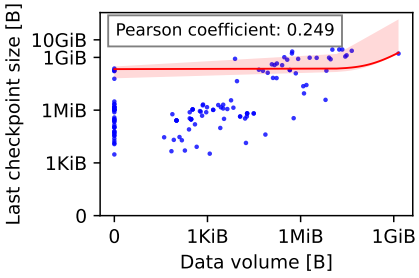
Figure 3.7a plots the CDF of the last observed checkpoint size across queries, ranging from ~30 KiB to ~26 GiB, and with a median value of ~10 MiB. The cumulated size of all snapshots is ~276 GiB. Even queries with only stateless LOs have a small snapshot of a few KiB, as their source PO must record the offset in their respective Kafka topic partition(s). We analyze the correlation between checkpoint size and the number of LOs and POs in queries. Figure 3.7b is a scatter plot for both POs (blue) and LOs (green). There is only a very small linear correlation between a query's graph complexity and its state size (P_c=0.108 for LOs and P_c=0.137 for POs). Figure 3.7c studies the correlation between the data volume processed by the query during our observation period and the size of the last checkpoint (note the log scale on both axes). Importantly, we point out that queries are long-lived and may have been deployed long before our observation period and accumulated state since then. We observe a slightly higher linear correlation (P_c=0.249) than for query complexity. Some low-volume (over our 24-hour observation period) queries have relatively large snapshots; these queries accumulate all events from source(s) in some `join` LOs.



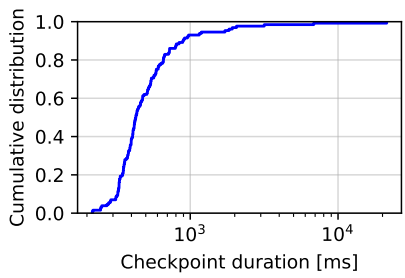
(a) CDF of last observed cumulative checkpoint size per query.



(b) Correlation between checkpoint size and number of POs/LOs.



(c) Correlation between last checkpoint size and data volume ingested during observation period (log scales). ($P_c = 0.249$)



(d) CDF of the average checkpoint duration per query.

Figure 3.7: Checkpointing and state statistics. All x-axes are in log-scale for readability.

Figure 3.7d presents the CDF of the average checkpointing duration across all queries. When a snapshot is taken, processing halts, resulting in downtime, possibly impacting queries with latency constraints (in this particular Digazu deployment, the expectation is that events be processed within one *minute* of their ingestion by the system, and there is no strict latency constraints that would require other forms of fault-tolerance [Ros+21]). The large majority (~92%) of the queries have an average checkpoint duration of less than 1 second. However, some queries with larger states require multiple seconds and as much as ~24 seconds average in the worst case. There is a moderate linear correlation ($P_c = 0.332$) between the checkpoint size and duration, but not a perfect one. Indeed, the checkpoint duration is primarily affected by the volume of new values in RocksDB, but these values may either replace (same key) or complement (new keys) those already stored, influencing how the checkpoint size evolves.

Takeaway 3. Runtime observations show that there is a significant amount of skew in data received by the queries' POs. The checkpoint size (accumulated state) of queries may be quite significant even when the volume of data in

the observation period is low, highlighting that very-long-lived queries with possibly large states but low instantaneous volumes are more common than short-lived, high-throughput queries.

3.4 Related Work

While the literature offers examples of workloads for data warehouses supporting static data [GRN20; Ren+13], or for relational workloads [HSY01; Phi+92] there is, to the best of our knowledge, no published analysis of a modern production data lake supporting hybrid batch-stream queries.

BM's Garcés-Erice *et al.* [GRN20] report statistics about a data lake built over Hadoop and running SQL queries over large, static data sets. The authors point out the difficulty of planning the resource consumption of such queries, given their complex structures and the heterogeneity of data sets. Two other notable works from IBM are Yu *et al.* [Phi+92]'s seminal paper on relational database workload characterization and the REDWAR workload analysis tool for DB2, and Hsui *et al.* [HSY01]'s comparison of a production database workload to a relational database benchmark.

TaoBao, an e-commerce company, reports a workload analysis of a 2000-node cluster supporting Apache Hadoop [Ren+13]. Researchers from Intel characterize the memory performance of large scale data processing workloads [Dim+13]. Chen *et al.* [CAK12] study the characteristics of MapReduce workloads across different e-commerce actors such as Facebook and Cloudera.

Halevy *et al.* [Hal+16] detail Goods, Google's data lake, but do not detail its workload. Microsoft researchers also report on the Azure Data Lake Store [Ram+17], a scale-out file system for Hadoop and MapReduce workloads. Performance results are based on synthetic benchmarks; the authors do not analyze production workloads.

3.5 Summary

We analyzed a production hybrid batch-stream workload of Digazu, a data lake building upon the popular Apache Kafka and Flink systems. Our findings show that this workload is characterized by many streams and queries that are highly heterogeneous in size and complexity. The existence of many low-volume, high-state persistent queries makes a case for re-designing stream processing engines and decoupling state and computation for such queries.

Publications. The work presented in this chapter contributed to the following publication:

A Tale of Many Streams: Characterizing a Hybrid Batch-Stream Production Workload in Digazu, a Data Lake supported by Apache Kafka and Flink. Donatien Schmitz, Luc Berrewaerts, Guillaume Rosinosky, Sabri Skhiri, and Etienne Rivière. In Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems (DEBS), 2025. Göteborg, Sweden, 188-198.

Identifying Shortcomings of Resource Management for Distributed Stream Processing

4

Our analysis of the workloads collected from a real-world stream processing deployment reveals several important insights into the characteristics of data sources and queries. This analysis was coupled with continuous discussions with data engineers and scientists working on this deployment, allowing us to better understand the practical challenges they face. In this chapter, we present a comprehensive review of the state of research on resource management in DSP. We identify key shortcomings and research opportunities in the area of resource management for DSP systems, and set the objectives of this thesis. Last, we present the methodology we followed to address these research objectives.

4.1 Resource Management in DSP

Resource management in DSP refers to the techniques and strategies used to efficiently allocate and utilize computational resources (e.g., CPU, memory, network bandwidth) in a DSP system. The problem can be broadly summarized as follows: given a set of logical operators, how to map and schedule their physical tasks onto a set of physical resources while optimizing certain objectives (e.g., throughput, latency, resource utilization, energy consumption).

In this section, we present a review of the state of research on resource management in DSP. We first discuss the main approaches to resource allocation and skew handling. We then review the closely related topics of task placement and operator/state migration. Last, we review existing research on predictive resource requirements in DSP.

4.1.1 Resource Allocation

The efficient allocation of computational resources is a critical aspect of resource management in DSP. It involves determining how much CPU, memory, and other resources should be allocated to each task in the DSP system to optimize performance and resource utilization. It is directly related to the concepts of horizontal and vertical scaling presented in Section 2.3.7.

As presented in Section 2.3.7, the state-of-the-art approach to reactive resource management in DSP is through horizontal scaling, where the number of task instances is adjusted based on the workload. While the work of Kalavri *et al.* [Kal+18] has been integrated into the Flink Kubernetes Operator auto-scaler, several other works have proposed alternative horizontal scaling strategies. Arkian *et al.* [Ark+21] propose a model-based auto-scaling approach for geo-distributed stream processing systems, which considers the network latency and bandwidth between different data centers when making scaling decisions. Similar to DS2, their system identifies overload using Flink's back-pressure signals, calibrates its model through runtime measurements, and applies scaling actions only when they are predicted to meaningfully improve throughput or reduce wasted resources.

Another line of work focuses on vertical scaling, where the resources allocated to containers (i.e. TM) executing the tasks (e.g., CPU, memory) are adjusted dynamically. For example, MEAD [Rus+21] is a vertical auto-scaling framework that adapts CPU resources to burstiness in the workload. Instead of relying on average data rates or simple utilization thresholds, MEAD models incoming streams using Markovian Arrival Processes (MAPs), which capture correlations and variability in bursty workloads, and combines these models with analytic queueing theory to predict operator response times accurately. MEAD continuously monitors input streams, fits workload models online, and chooses CPU resource levels for each operator to minimize cost while satisfying response-time constraints. Russo *et al.* [RCL23] propose a hierarchical auto-scaling framework, which decides which kind of computing nodes is required for each operator. In addition to horizontal scaling policies that adjust the number of task instances, they integrate vertical scaling by deciding which resources to allocate in a heterogeneous cluster context.

A lot of work has focused on CPU resources but vertically scaling memory has also received attention in other contexts, such as batch processing. Some works include MespaConfig [Zon+21], which optimizes the memory configuration of *multiple* Spark batch processing jobs co-located on a cluster. Their approach involves modelling how performance varies with both memory configuration and real-time system load. They learn load-sensitive performance models using neural networks then apply multi-objectives search to trade off throughput and memory usage, selecting Pareto-optimal config-

urations based in user-defined threshold policies. Iorgulescu *et al.* [Ior+17] define the *memory elasticity* principle and show how careful memory allocation below the working set size of batch processing, in-memory applications can yield close performance to larger allocations. They further explore this memory elasticity principle by applying it to cluster scheduling. Through better task scheduling using memory elasticity, they show that cluster-wide memory utilization can be improved significantly without impacting application performance.

We note however the lack of work on vertical memory scaling in DSP, which we argue is a critical aspect of resource management in this context, particularly for stateful stream processing.

Skew Handling. An orthogonal but closely related problem to resource allocation in DSP is data skew handling. Data skew refers to the uneven distribution of data across different partitions or operators in a DSP system. This can lead to performance bottlenecks, as some tasks may be overloaded while others remain underutilized. Effective skew handling strategies are essential for maintaining balanced workloads and optimizing resource utilization in DSP systems.

A number of techniques have been proposed to address data skew in DSP. While traditional key-based partitioning schemes use a single hash function to map keys to workers, leading to potential overload on certain workers, another technique called *shuffle grouping* randomly assigns incoming data to workers, which can help balance the load more evenly. This technique requires an additional aggregation operator downstream to combine the results from different workers. Nasir *et al.* [Nas+15] propose a *two choices* approach, allowing keys to be assigned to two selected workers. Based on commonly studied *power of two choices* load balancing technique [Mit02], their approach involves using two hash functions to select two candidate workers for each key. The source operator then assigns the key to the least-loaded of the two workers, helping to balance the load more effectively across the cluster. They further extend this work by considering more than two choices [Nas+16].

Cheng *et al.* [CZJ17] propose an altogether DSP system that natively handles skew through dynamic repartitioning of data based on observed key popularity. They leverage a light weight counting scheme to estimate the frequency of keys and identify heavy hitters. They then assign the popular keys using shuffle grouping while distributing the remaining keys using the default key grouping.

4.1.2 Task placement

The previous section discussed resource management from the perspective of allocating resources to tasks. Another important aspect of resource manage-

ment in DSP is the placement of tasks onto physical resources. The placement of tasks can have a significant impact on the performance and efficiency of the DSP system. Effective task placement strategies can help minimize data transfer costs, reduce latency, and improve resource utilization.

Early work on task placement includes SODA [Wol+08], by the IBM Systems team, which proposes heuristic strategies for placing multiple queries onto a cluster while optimizing for load balancing and consolidation. Unlike systems that only place tasks on nodes, SODA jointly performs job admission, job configuration selection, CPU allocation, and task placement to maximize a utility-based notion of "importance" derived from output stream value, quality, and priority. By combining mathematical optimization, dynamic programming, integer programming, and heuristics, SODA balances system load, reduces network traffic, and respects practical constraints such as security and licensing. Another relevant work includes Xing *et al.* [Xin+06] that propose algorithms for *resilient* placements, based on estimations of operators' capacities to handle short-term bursts and models of operators load scaling. The authors formalize the concept of a system's feasible set — the set of input rate combinations that do not exceed node capacities — and define the optimization goal as choosing an operator placement that maximizes the size of this feasible set. Because computing the exact optimal solution is intractable, they introduce greedy heuristics that balance individual input stream loads across nodes and reduce bottlenecks caused by combined stream impacts.

Several works consider the *shared resource contention* problem, where co-located tasks of DSP queries compete for the same types of resources on a deployment node (e.g., CPU or I/O). Among them, Q-Flink [Hos+20] is a quality-of-service (QoS) aware task placement framework for DSP. It monitors the workload patterns and uses model predictive control (MPC) to predict the load intensity of each query. It then uses this model to find near-optimal placement decisions that minimize resource contention and meet QoS requirements. CAPSys [Wan+25] is another contention-aware placement system for DSP that considers both CPU, I/O, and network contention when placing tasks. Its approach ensures that tasks with different resource usage patterns are co-located on the same node to minimize contention (i.e., stateless and stateful tasks). To solve the NP-hardness of the placement problem, CAPSys employs a Contention-Aware Placement Search (CAPS) algorithm that uses a combination of depth-first search and pruning techniques to efficiently explore the placement space. CAPSys is implemented in Apache Flink and shows significant improvements in throughput and latency compared to default placement strategies for single- and multi-query deployments.

Operator Migration. Operator migration in DSP involves moving a or several tasks of an operator from one physical resource to another, along with its

associated state. This is particularly important in scenarios where the workload is dynamic and can change over time, requiring the system to adapt to new resource demands and data distributions.

State and operator migration have been studied in various DSP environments including cloud computing, edge computing, and fog computing. Volnes *et al.* [VPG23] give a comprehensive survey of operator migration techniques in these contexts. Hoffmann *et al.* propose Megaphone [Hof+19], a latency-conscious state migration approach for DSP, focusing on minimizing the impact of state migration on overall system latency. Megaphone involves injecting special *migration* events into the data stream, which specifies when and how state migration should occur. These migration events are processed by the operator tasks, which then transfer the relevant state to the new tasks. Fang *et al.* [Fan+17] propose a framework for handling data skew in DSP through dynamic operator migration, aiming to balance workloads across the system. The authors formulate the migration decision as an optimization problem, seeking to minimize the overall processing time while considering the costs associated with migration.

These techniques are closely related to resource management and skew handling, as they often involve reconfiguring the placement of tasks and their associated state to optimize resource utilization and balance workloads.

4.1.3 Predictive Resource Provisioning

Predictive resource provisioning aims to anticipate the resource needs of DSP queries before they are deployed, allowing for proactive adjustments to resource allocations and placements.

A first line of work focuses on *zero-shot cost models* for DSP. Heinrich *et al.* [Hei+22] propose a zero-shot learned cost model for DSP, which predicts the performance (i.e. latency and throughput) of unseen queries based on their characteristics. They represent queries using a set of features (e.g., types of operators, type of window, data rates) and train a general model on a large set of queries. For their training, they generate 15,000 queries synthetically amongst which 5,000 are windowed aggregations, 5,000 are windowed joins with an aggregation, and 5,000 are a combination of two windowed joins with an aggregation. Their model leverages Graph Neural Networks (GNNs) to capture the structure of the query plan and predict its performance. Agnihotri *et al.* [Agn+23] propose to leverage this zero-shot cost model to predict the optimal parallelism settings for unseen queries. A first step to their approach is to generate a set of representative training data. Instead of randomly trying all parallel settings, they use rules to generate meaningful parallel query plans (e.g., operators with high input rate should have higher parallelism). The features of these plans are then extracted and used to train a prediction

model. Both approaches rely on a training process done over a large set of queries. As no dataset of *real* queries of this size exists, the authors have to resort to synthetic, randomly-generated queries that may not represent real workloads.

The need to plan the configuration or scale of computing infrastructure is obviously not limited to DSP. Higginson *et al.* [Hig+20] discuss the applicability of resource forecasting techniques based on machine learning for clustered database systems. They use historical resource usage data (CPU, memory, I/O) time series to train models that can predict future resource needs based on seasonal patterns and trends. URSA [Zha+20] is a capacity planning and scheduling system for relational database platforms, modeling the response of a database workload to provided resources and deriving just-sufficient resource specifications automatically.

4.2 Research Objectives

The previous section presented a comprehensive review of the state of research on DSP, highlighting key challenges in the area of resource management. Following this review, we identify several shortcomings that have not been fully addressed in the literature. In this section, we discuss three challenges and opportunities for research in DSP: 1) inefficient resource allocation strategy, 2) multi-query resource allocation and task placement optimization, and 3) predictive resource provisioning based on real queries. We discuss each of these challenges in detail, highlighting the limitations of existing approaches and proposing potential room for improvements.

While these challenges are the main focus of this thesis, we acknowledge that other important challenges exist in the area of DSP. Some complementary challenges include skew handling and operator migration, which are also critical for the efficient operation of DSP systems. These topics are highly relevant but complementary to the contributions presented in this thesis. We discuss in more detail our perspectives in Chapter 8.

Inefficient Resource Allocation Strategy. We find that the current resource allocation strategies in Flink and other systems [14b; 15b; 16] that give each task a fixed amount of CPU and memory resources are inefficient. This one-size-fits-all approach does not account for the diverse resource requirements of different operators, particularly in the context of stateful stream processing where stateless operators are allocated memory which is ultimately wasted. We argue that a more dynamic and operator-aware resource management strategy is needed to efficiently allocate resources based on the actual requirements of each operator. Similarly to vertical scaling of CPU resources, we advocate for vertical scaling of memory resources in DSP. This includes the ability to dynamically adjust memory allocation for stateful operators

based on their workload characteristics, such as the volume and pattern of state accesses.

Problem Statement 1

How to design an efficient resource management strategy for DSP that dynamically allocates CPU and memory resources based on the actual requirements of each operator, particularly in the context of stateful stream processing?

Proposed Solution

In Chapter 5, we propose a mechanism to dynamically adjust memory allocation for operators in Flink based on their requirements. We implement our solution as a hybrid vertical/horizontal auto-scaler, that adjusts both CPU and memory resources to efficiently manage resources in DSP.

Multi-Query Optimization. A large body of research on resource management in DSP focus on single-query deployments [Kal+18; Wan+25], where the decisions are made independently for each query in isolation. However, our analysis in Section 3.3 revealed that multiple queries are often deployed concurrently on the same cluster. It also showed that the amount of data processed by each query can vary significantly, leading a large amount of idle resources on some workers. This observation motivates the need for multi-query optimization techniques that can effectively share resources and optimize execution across multiple concurrent queries.

Problem Statement 2

How to design multi-query optimization techniques for DSP that effectively share resources and optimize execution across multiple concurrent queries, taking into account their diverse resource requirements and workload characteristics?

Proposed Solution

In Chapter 6, we propose an auto scaler that leverages vertical scaling of CPU and memory resources to efficiently manage resources in DSP, coupled with a task placement strategy that optimizes resource sharing across multiple concurrent queries and leveraging opportunities for consolidation.

Predictive Resource Provisioning. In complementary to reactive resource management techniques, predictive resource provisioning aims to anticipate

the resource needs of queries before they are deployed. The different approaches presented in the previous section regarding predictive resource management in DSP often rely on general performance models learned from a large set of synthetic queries. These models may not accurately capture the specific characteristics and resource requirements of individual queries, particularly in the context of complex queries with sub-linear scaling behavior. This limitation highlights the need for more accurate and query-specific predictive resource provisioning techniques. We argue that predictive resource provisioning should focus on accurately modeling the performance and resource usage of individual queries, taking into account their unique characteristics and workload patterns. This includes developing techniques to capture the non-linear scaling behavior of complex queries and accurately predicting their resource requirements based on historical data and workload characteristics. Furthermore, we advocate for lightweight predictive models that can be efficiently trained and deployed in real-world DSP environments, without requiring extensive training on large datasets of synthetic queries.

Problem Statement 3

How to design accurate and lightweight predictive resource provisioning techniques for DSP that effectively model the performance and resource usage of individual queries, taking into account their unique characteristics and workload patterns?

Proposed Solution

In Chapter 7, we propose a capacity planning framework for DSP that builds query-specific performance models based on small-scale deployments.

4.3 Methodology

In this thesis, we aim to address the challenges identified in Section 4.2 related to resource management in DSP systems. To achieve this, we employ a systematic methodology that involves defining evaluation metrics, setting up an evaluation stack, and using a state-of-the-art benchmarking framework to assess the effectiveness of our proposed solutions. Instead of relying on simulations, we conduct real-world experiments on a physical testbed to ensure the validity and applicability of our findings.

In this section, we present the methodology we use to study and address the challenges identified in Section 4.2. The methodology we present here is applied throughout the thesis, unless otherwise specified. We describe the

evaluation metrics we use to assess the effectiveness of our proposed solutions in Section 4.3.1. We then present the evaluation stack we use to implement and test our solutions in Section 4.3.2. Finally, we describe the benchmark used to evaluate our solutions in Section 4.3.3.

4.3.1 Evaluation Metrics

To evaluate the effectiveness of our proposed solutions, we consider several key metrics that capture the performance and efficiency of DSP systems.

Evaluating Resource Management. When comparing to state-of-the-art solutions, we want to answer the following questions:

- For a given target throughput, how much resources does our solution use compared to existing solutions?
- For a given resource quota, how much throughput does our solution achieve compared to existing solutions?
- How effectively does our solutions converge to optimal configurations compared to existing solutions?

Metrics used to evaluate automatic scaling (i.e., reactive resource management and consolidation) include:

- **Resource Utilization**: We measure the CPU and memory allocations of the running queries. This includes the number of CPU cores and amount of memory used by the stream processing system to achieve the desired rate for the queries.
- **Reconfigurations**: We measure the number of times the system reconfigures itself in response to workload changes. Fewer reconfigurations indicate a more stable and efficient system.

Evaluating Predictive Planning. When evaluating predictive resource provisioning solutions, we want to answer the following questions:

- Can we accurately predict resource requirements, prior to deployment, compared to actual resource usage during production deployments?
- How well does the predictive model generalize to different queries and workload patterns?
- How costly is the building of the predictive model in terms of time and resources?

To evaluate these aspects, we consider the following metrics:

- **Prediction Accuracy**: We measure the difference between the predicted resource requirements and the actual resource usage observed during production deployments.
- **Model Building Cost**: We measure the time and resources required to build the predictive model, including the number of TS and their respective resource allocation.

4.3.2 Evaluation Stack

One of the key aspects of our methodology is the use of a robust evaluation stack that allows us to implement and test our proposed solutions in a realistic environment. Our evaluation stack consists of a physical testbed, a container orchestration platform, and a set of monitoring tools to collect performance metrics. In addition, we leverage existing open-source tools and frameworks to be able to reproduce our experiments and facilitate comparisons with related work.

Experiments are all conducted using the Grid'5000 testbed [Cap+05], a large-scale and versatile experimental platform for distributed systems research. Grid'5000 provides a controlled environment where we can deploy and evaluate our solutions on a cluster of physical machines. For our experiments, we use the *Gros* cluster located in Nancy, France. The Gros cluster consists of 122 nodes, each equipped with an Intel Xeon Gold 5220 processor (18 cores at 2.2 GHz), 96 GiB of RAM, and connected via a 25 Gbps network.

We leverage Jupyter Notebooks[1] to document and automate the deployment, execution, and analysis of our experiments. Jupyter Notebooks provide an interactive environment that allows us execute code snippets in a reproducible manner. Additional tools, such as Papermill[2], enable us to parameterize and execute notebooks programmatically, facilitating execution and result collection.

Cluster Deployment. We deploy our evaluation stack using Terraform[3], an infrastructure-as-code tool that allows us to define and manage our infrastructure in a declarative manner. Terraform enables us to automate the provisioning and configuration of the necessary resources on Grid'5000, ensuring consistency and reproducibility across our experiments. We use Terraform to define the cluster configuration, including the number of nodes and their specifications.

We use the `terraform-grid5000-k8s-cluster` module[4], which simplifies the deployment of Kubernetes clusters on Grid'5000. This mod-

[1] `https://jupyter.org/`
[2] `https://papermill.readthedocs.io/en/latest/`
[3] `https://www.terraform.io/`
[4] `https://github.com/pmorillon/terraform-grid5000-k8s-cluster`

ule handles the installation and configuration of Kubernetes on the allocated nodes, allowing us to focus on deploying and evaluating our stream processing solutions. Once the Kubernetes cluster is set up, we deploy Apache Flink using the Flink Kubernetes Operator (Section 2.4), and Kafka using the Strimzi Kubernetes Operator[5]. Both modified versions of Flink and the Flink Kubernetes Operator are shipped as Docker images. We also deploy a MinIO[6] instance to serve as an object storage backend for checkpointing and state management.

Monitoring. We use Prometheus[7], a popular open-source monitoring and alerting toolkit, to collect and store metrics from our Kubernetes nodes, and DSP system. We deploy Prometheus using the Prometheus Operator[8], which simplifies the deployment and management of Prometheus instances on Kubernetes. We configure Prometheus to scrape metrics from the Kubernetes nodes, Kafka brokers, and Flink components. Metrics are collected at regular intervals and stored in a time-series database. The notebooks used for our experiments include code to query Prometheus and retrieve the relevant metrics for analysis. To visualize the collected metrics, we use Grafana[9]. Grafana provides a powerful and flexible dashboarding solution that allows us to create custom visualizations of the metrics collected by Prometheus.

4.3.3 Benchmarking

While our analysis presented in the previous Chapter has provided valuable insights, the workloads we analyzed were subject to strict privacy constraints. The scripts used to collect and analyze the workloads had to be approved by the company's data privacy team and were executed in a controlled environment. As a result, we were unable to leverage these workloads for benchmarking purposes in our evaluations. To compare our proposed solutions against state-of-the-art approaches, we also require a well-established benchmarking framework that is widely recognized in the research community.

To this effect, we use Nexmark [Tuc+08], a widely-used benchmark for DSP systems. Nexmark simulates an online auction system, generating a continuous stream of events representing auctions, bids, and persons. Nexmark includes a set of queries that cover a range of common stream processing operations, including filtering, joining, windowing, and aggregations. These queries are designed to be representative of real-world stream processing workloads, allowing us to assess the performance and efficiency of our so-

[5] https://strimzi.io/
[6] https://min.io/
[7] https://prometheus.io/
[8] https://github.com/prometheus-operator/prometheus-operator
[9] https://grafana.com/

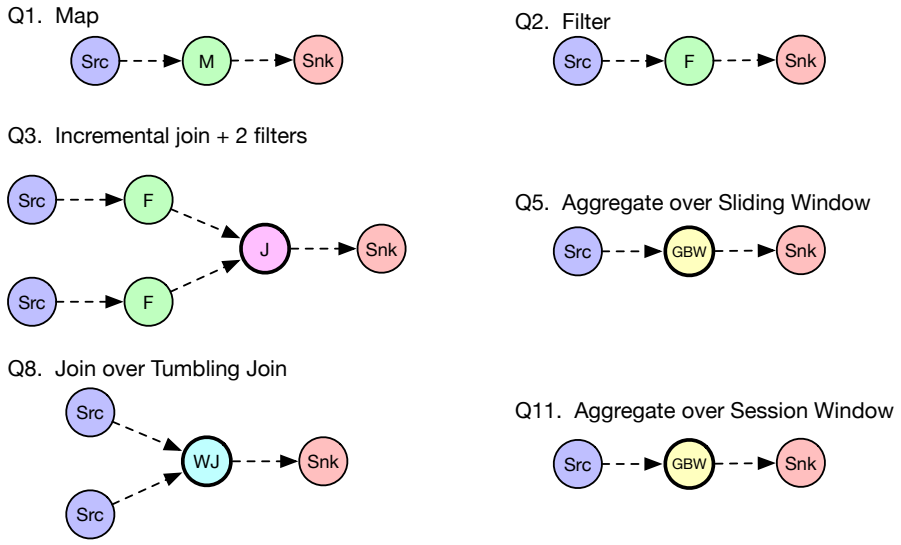


Figure 4.1: Nexmark Queries used for benchmarking. Bold edges represent stateful operators.

lutions in a controlled environment. Nexmark has been widely adopted in the research community for its relevance and comprehensiveness in evaluating DSP systems [Kal+18; Hof+19; Gu+22; Son+23; Wan+25; Mei+25].

Nexmark declares its queries in SQL, which are then translated to the native API of the target stream processing system. Appendices A.1 and A.2 present the table schemas and queries used in Nexmark, respectively. Figure 4.1 illustrates the DAG representation of the different queries. We use the implementation of Nexmark Queries and data generator provided by DS2. It includes a set of six queries written using the DataStream API of Flink, along with rate-limited data generators to simulate different workload intensities.

For our evaluation purposes, we use the following queries from Nexmark:

- **Q1** produces currency conversions using one Map operator. This query is stateless and the only computing logic is composed of a simple multiplication.
- **Q2** uses a Filter operator to select bids with a specific identifier. This query is also stateless and the computing logic is a simple comparison.
- **Q3** classifies sales based on location and categories and maps them to specific auctions, using an incremental join operator over the complete stream (i.e., without windowing) and two filter operators. This complex query is stateful due to the join operation and results in an ever-growing state due to the lack of windowing.

Query	Description	Operators	Stateful
Q1	**Currency conversion**: converts bid values from dollars to euros	1	-
Q2	**Selection**: filter bids with specific auction identifier	1	-
Q3	**Incremental Join**: classifies sales based on location and categories and maps them to specific auctions	3	J
Q5	**Hot items**: determine auctions with most bids in last period	1	WJ
Q8	**Monitor new users**: identify active users in last period	1	GBW
Q11	**User sessions**: compute number of bids each user makes while active	1	GBW

Table 4.1: Nexmark queries. The number of operators excludes sources and sinks. "Stateful" lists stateful operators used by the query: WJ for Windowed Join, and GBW for GroupBy (window). Other operators are stateless operators supporting Map or Filter operations.

- **Q5** determines the auctions with the most bids in a period and uses one stateful operator and a group-by-aggregate over a sliding window. This stateful query requires maintaining state for each window and auction. The use of a sliding window increases the computational complexity, as overlapping windows are processed.
- **Q8** monitors new active users over a period and uses a join over a tumbling window as a stateful operator. This query requires maintaining state for each window and user, making it more complex than simple filtering or mapping operations.
- Finally, **Q11** monitors user sessions by computing the number of bids each user makes while active. It uses one operator, a stateful group-by-aggregate, over a session window.

Table 4.1 summarizes the selected queries, their descriptions, the number of operators they use, and which of their operators are stateful.

4.4 Summary

Following our analysis of real-world stream processing workloads in Chapter 3, we reviewed challenges and open problems in DSP systems. We identified several key challenges related to resource management, multi-query optimization, and predictive resource provisioning in DSP.

To address these challenges, we proposed three main directions for research and development. First, we highlighted the need for more efficient resource management strategies that can dynamically allocate resources based on the actual requirements of each operator. Second, we emphasized the importance of multi-query optimization techniques that can effectively share resources and optimize execution across multiple concurrent queries. Finally, we discussed the potential of predictive resource provisioning approaches that can forecast resource requirements of queries and proactively allocate resource allocation before deploying them.

We then presented the evaluation methodology we will use to assess the effectiveness of our proposed solutions, along with the evaluation stack and benchmark we will use for our experiments.

The next chapters present our proposed solutions to the identified challenges. In Chapter 5, we propose Justin, a hybrid CPU/memory fine-grained auto scaler that addresses the challenges of inefficient resource management in DSP. In Chapter 6, we present Charon, a multi-query fine-grained auto scaler that addresses the challenges of multi-query optimization in DSP. Finally, in Chapter 7, we introduce StreamBed, a framework for capacity planning in DSP systems.

Justin: Automatic, Fine-Grained Resource Scaling

5

In this chapter, we focus on the problem of resource allocation as static, fixed-size units in DSP systems. Existing auto-scalers couple CPU and memory allocation. Each task is assigned the same amount of memory, regardless of the operator. Scaling out an operator (i.e., adding more tasks) adds a proportional amount of memory. Scaling in (i.e., removing tasks) similarly reduces the total memory available to the operator by the same factor. This coupled, one-size-fits-all allocation clashes with the heterogeneous memory requirements of operators. Some operators, such as a *filter* or a *map*, are stateless and do not require more than a minimal amount of memory to process events efficiently. In contrast, other operators, such as *joins* or *group by* over long windows, maintain state across the processing of many events [Ver+23].

Stateful operators with under-allocated memory perform poorly: The necessary state may not fit in the main memory, and accesses may resort to on-disk storage. In contrast, ample memory allocation (e.g., as the operator has been scaled out to multiple tasks) should not yield better performance than a smaller but sufficient allocation. In reality, the relation between event processing performance, state access latency, and memory requirements is more subtle than this simple intuition. As depicted in Section 2.3.4, the state of an operator is stored in a state backend, providing a key-value store interface to the tasks of this operator. The state-of-the-art state backend in Flink, RocksDB, is based on a Log-Structured Merge-tree (LSM). An LSM combines tables and caches in memory with bulk storage on disk. It uses optimizations to favor write performance and minimize the wear of modern SSD drives. The performance in response to the available memory of RocksDB depends heavily on the nature of the workload [Asy+]. Workloads formed mainly of *write* operations do not benefit from a large memory allocation. In contrast, workloads dependent on *read* operations do, in proportion to the size of their working set. The nature of an operator's access to its state is not predictable, as it often depends on the characteristics of its input events.

In this chapter, we first study the impact of memory on the performance of stateful operators in Flink (Section 5.1). The key insight from our study is that

the performance of stateful operators is not only dependent on the amount of memory allocated to them but also on the nature of their workload. Based on our findings, we propose Justin, a novel auto-scaling framework for Flink that combines both horizontal and vertical (i.e., memory) scaling operations (Section 5.2). Justin continuously collects metrics about a running query's resource usage and storage performance in RocksDB. Based on these metrics, Justin's auto-scaling policy adjusts the DS2 decisions and derives CPU and memory allocations. We leverage the existing fine-grain resource allocation capabilities of Flink to adjust the memory allocated to each operator independently. This mechanism allows practitioners to specify the amount of CPU, memory, and other resources for each operator in a query. However, the allocation of heterogeneous resources through this mechanism is only available in the Java DataStream API, and not in higher-level APIs such as SQL or Table API. Furthermore, the allocation is static, i.e. the query needs to be re-written through the Java DataStream API, re-compiled, and re-deployed to adjust these allocations. Justin extends the Flink scheduler and offer an API to adjust these allocations at runtime.

We evaluate Justin in Flink on a 7-node cluster under high loads and compare it to DS2 using the Nexmark benchmark (Section 5.3). Our results show Justin produces configurations with 48% less CPU and 27-28% less memory for **Q8** and **Q11**, two complex stateful queries of Nexmark while requiring the same or fewer reconfiguration steps to converge to an optimal configuration than DS2.

5.1 Impact of Memory on Operators' Performance

We study in this section the relationship between memory allocation and task performance in Flink. The takeaways of this section guide the design of our hybrid CPU/memory auto-scaler, Justin, detailed in the next section.

RocksDB. The state of stateful operators' tasks must be persisted across the processing of different events. As described in Section 2.3, Flink uses RocksDB [25h] as the default state backend. Writes are buffered in memory (in a MemTable) and later flushed to disk in sorted order, reducing random I/O. A hierarchy of sorted SSTables (Sorted String Tables) files is periodically merged into larger ones, reducing fragmentation and read amplification. Data is stored across multiple levels of this hierarchy, with newer data at higher levels and older data compacted into lower levels. As a result, reads may require searching SSTables across multiple levels, increasing their latency. To mask these costs, RocksDB uses an in-memory *cache* of recently accessed data, and bloom filters to avoid looking up non-existing keys.

The MemTable, implemented as a skip list, is used to buffer writes before consolidating them to SSTables. Using a large MemTable has no real impact

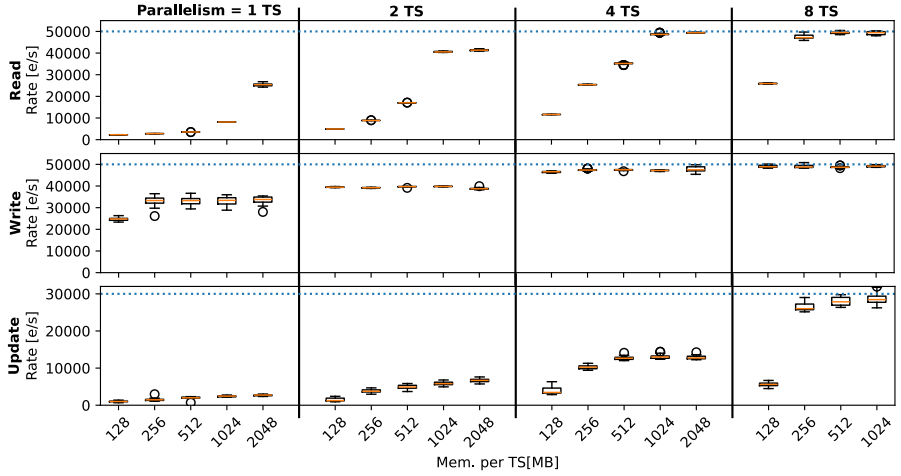


Figure 5.1: Evaluation of multiple memory-cpu configurations on three different state access patterns. Benchmarks show the maximum rate reachable against a target rate (dotted line). The impact of memory on rate significantly differs depending on the workload, read-only being the more profitable.

on write latencies, as consolidation is performed at the granularity of the first-level SSTable (of 64 MB). In contrast, read latency is directly impacted by the size of the cache and its relation to the task's working set size (i.e., the set of keys the task frequently accesses over a period). Tasks performing frequent reads, as for the *Count* operator of the word count in Figure 2.1a, may have to resort to costly exploration of SSTables on disk if the cache is not large enough. Tasks performing only or mostly writes are not impacted by the cache size.

Task Manager memory allocation. Memory sharing between task slots on the same Task Manager depends on the JVM's memory segmentation. On-heap memory, used by Flink operators for creating Java objects and subject to garbage collection, is shared among all tasks. The same applies to network memory used for communication buffers. While there is no isolation between threads for access to these segments, a Task Manager reserves a minimum amount for each task slot. In contrast, *managed memory* is specific to a thread and is not subject to garbage collection. It is used by the RocksDB instance local to each task to store its MemTable and cache.

Takeaway 1. Managed memory is reserved for all task slots in equal amounts. This memory is wasted for stateless tasks that do not use RocksDB. Stateful tasks get a one-size-fits-all allocation, regardless of their requirements.

Microbenchmarks. We evaluate the impact of memory allocation on performance using microbenchmarks. We use a single operator. The input stream is

formed of events of 1,000 B. Each event includes a key generated uniformly at random between 0 and 1,000,000 and a random payload. The RocksDB state backend is pre-populated with a value associated with each key. We consider three workloads. In the **Read** workload, the operator reads the value associated with the key in the event. In the **Write** workload, it replaces the value associated with this key without reading the previous value. In the **Update** workload, the operator reads the current value and then overrides it with the event's payload. We use a target rate of 50,000 events per second for the Read and Write workloads and 30,000 events per second for the Update workload.

Our results are presented in Figure 5.1. We consider 19 configurations for each workload, with an operator's parallelism ranging from 1 to 8 tasks and managed memory allocation of 128 to 2,048 MB. In any memory allocation, the MemTable size is at most 64 MB, and the rest is used for the cache. By default, Flink prioritizes the allocation of at least half of the memory to the cache, possibly reducing the size of the MemTable, which is allocated as a power-of-2 granularity. As a result, an allocation of 128 MB of memory results in a 32 MB MemTable and a 96 MB cache, but allocations of 256 MB or 512 MB result in a 64 MB Memtable and, respectively, 192 MB and 448 MB caches. We denote a configuration with t tasks and m MB of memory as $(t; m)$. We run each configuration for 10 minutes. We measure aggregate rate every 5 seconds and present the distribution as a box plot. The dotted line indicates the target rate. We use specific source operators that produce events at the maximal possible speed, subject to back pressure from the measured operator and capped by this target rate.

We observe that the target rate is sustained for the Read workload starting from (4; 1,024) or (8; 512). Below that, we observe that the cache hit rate (not shown on the plot) is very low, impacting processing time for every event. With 256 MB of memory, even the 8-task configuration is slightly below the target rate, possibly requiring further scale-out and the use of a lot of CPU resources.

Takeaway 2. Scaling out a read-intensive operator without appropriate memory allocation can lead to inefficient scale-out operations. The cache hit rate is a key metric for determining when the cache size is insufficient and prefer scaling up to scaling out.

The Write workload shows constant performance at all parallelisms with all memory configurations, except the smallest (1; 128), which is slightly below (1; 256). This lower performance is due to the smaller MemTable size. The target rate is reached with a parallelism of 8, with 4 tasks being very close.

Takeaway 3. The size of the cache does not impact write performance. Write-dominated operators should favor scale-out to scale-in.

Finally, the evaluation of the Update workload shows that only an 8-task

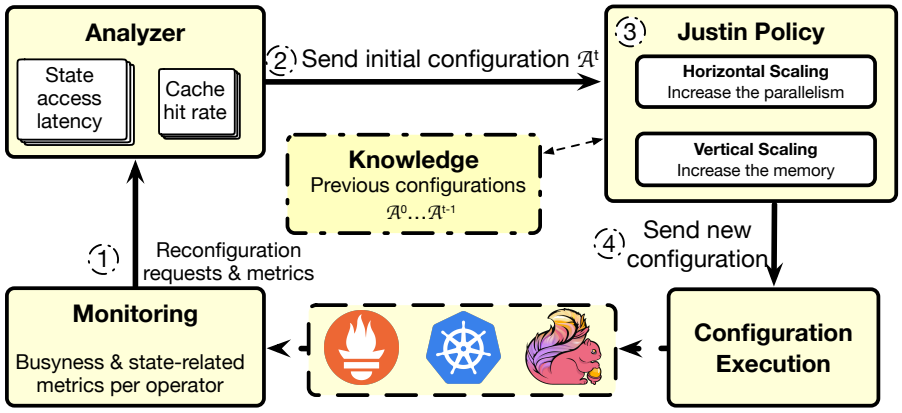


Figure 5.2: Overview of Justin components and workflow. Monitoring collects application metrics and triggers reconfigurations. Analyzer processes the collected metrics and computes relevant state-related indicators. The Justin Policy then determines the appropriate scaling decision based on these indicators and prior knowledge. Finally, Justin applies the new resource allocation decisions to the running query

configuration can sustain the workload. We also observe a *plateau* effect for 4 and 8 tasks, where adding memory does not result in significant gains (in contrast with the Read workload, where these gains are more linear). Configurations with insufficient memory (128 MB) cannot sustain the rate regardless of parallelism.

Takeaway 4. It is necessary to vertically scale (add memory) above a minimum threshold. If a scale-up does not improve performance significantly, it is preferable to scale out.

5.2 Justin: Hybrid CPU/memory Auto-Scaler

Building upon the takeaways of the previous section, we design Justin as a hybrid auto-scaler that decides on scaling *up* or scaling *out* an operator depending on its needs and avoids allocating memory to tasks that do not need or use it. Figure 5.2 presents an overview of Justin components and workflow. Justin builds upon DS2, as detailed in Section 2.3.7, and extends it with memory-aware mechanisms. We start by providing the necessary additional background on DS2 and presenting the notations used in this chapter. Then, we detail the metrics collected by Justin. This includes raw metrics from Flink and RocksDB ①, as well as derived metrics specific to Justin. Justin uses these indicators from the Analyzer components ② to apply its auto-scaling policy. To this effect, we present an auto-scaling policy that uses these indicators along with a history of previous scaling decisions ③. We then explain the

Symbol	Description
M	Default amount of Managed Memory per TS
$o \in O$	An operator in the set of operators
s_o	Use of managed memory for operator o
$\mathcal{A}^t$	Resource allocation at period t, including:
$\mathcal{A}^t.\pi_o$	Parallelism (number of tasks) for operator o
$\mathcal{A}^t.m_o$	Managed memory level for operator o
$\mathcal{A}^t.\tau_o$	Average state access latency of operator o
$\mathcal{A}^t.\theta_o$	Average cache hit rate of operator o

Table 5.1: Notations used by the Justin policy for a new configuration initiated at the end of observation period t.

mechanisms we use to enable dynamic, heterogeneous memory allocation for tasks ④. Table 5.1 summarizes the notations used in this chapter.

Elastic scaling and DS2. Elastic scaling determines the appropriate level of parallelism for all operators. From a default configuration (parallelism of tasks and memory per task) at time $t = 0$, reconfigurations are triggered, based on observations, at discrete times $t > 0$. We define a configuration $\mathcal{A}^t$ as a map between operators $o \in O$ where $\mathcal{A}^t.\pi_o$ is the parallelism π (number of tasks) of operator o at time t.

DS2 uses a primary metric, the busyness of operators, averaged across their tasks. Busyness is the fraction of CPU time spent processing events, i.e., not idling or waiting due to back pressure. A reconfiguration trigger is a high busyness for one of its operators in addition to backpressure from its upstream operator(s), indicating that the query's capacity is insufficient to cope with the current rate. When triggered at time t, DS2 determines a new level of parallelism $\mathcal{A}^t.\pi_o$ for each operator, such that the resulting busy rate is below a target. For this purpose, it uses linearity assumptions and considers that the load will be equally distributed among the scaled-out operator's tasks. This computation accounts for the cascade effects between operators: scaling out an operator increases its capacity, resulting in a higher rate for downstream operators and the need to scale them out as well. The new configuration is applied via a query reconfiguration, possibly resulting in state transfers between RocksDB instances [Car+17]. DS2 typically requires several reconfiguration steps until it reaches a stable configuration for a target rate, and setting appropriate thresholds for triggering reconfiguration and target busyness levels requires careful tuning. The auto-scaler aims to keep the average of the average businesses of an operator's tasks within a specified range.

Integrating memory-awareness. Justin decides on a heterogeneous managed memory allocation for tasks, or the absence of managed memory for

stateless tasks. It requires memory-centric indicators in addition to CPU ones, such as busyness. We note, however, that there can be a coupling between memory and CPU indicators. A task performing high-latency state accesses may have a high busyness, as state accesses are accounted for as part of the event processing time. State access latency allows distinguishing between the two cases. High average access latency $\mathcal{A}^t.\tau_o$ indicates that a significant fraction of accesses by o's tasks used the disk and that scaling out may not be the best option until the cache is appropriately dimensioned. Furthermore, RocksDB exports the number of cache hits and misses, from which we can derive an average *cache hit rate* $\mathcal{A}^t.\theta_o$ for operator o's tasks. We collect these two metrics using Prometheus. As for other metrics, these are collected and averaged over a period.

5.2.1 Justin Policy Building Blocks

The elastic scaling policy of Justin extends DS2, specifically its implementation in Flink and the associated Flink Kubernetes Operator. This choice allows building upon the complex engineering, integration, and parameter tuning already made by the open-source community and improves impact potential.

Memory allocation. Justin allocates heterogeneous memory to the tasks of different operators. By default, all tasks of an operator o receive the same amount of managed memory M, calculated as the total amount of managed memory of TMs divided by their number of task slots. $\mathcal{A}^t.m_o$ represents the managed memory, in MB, allocated to all tasks of an operator o. To facilitate the mapping of tasks to task managers, we allocate memory in *levels*. Each new level corresponds to double per-task memory, from a minimum to a maximum (*maxLevel*). For $\mathcal{A}^t.m_o = x$, we allocate 2^x times the minimum of managed memory for stateful tasks. If, for instance, the minimum managed memory M is 128 MB, $\mathcal{A}^t.m_o = 0$ means 128 MB, $\mathcal{A}^t.m_o = 1$ means 256 MB, etc. Stateless tasks are allocated no managed memory, expressed as $\mathcal{A}^t.m_o = \perp$. We denote as s_o the boolean indicating the presence of stateful metrics for operator o.

Decisions history. DS2 does not maintain a history of scaling decisions, as it only takes these in one direction (horizontally). In contrast, Justin can choose between scaling out or up. Keeping a decision history helps determine whether these previous decisions improved the query capacity. Past configurations are available in $\mathcal{A}^0 \ldots \mathcal{A}^{t-1}$. A boolean $\mathcal{A}^t.v_o$ indicates that, in $\mathcal{A}^t$, the scaling decision involved *scaling up* (vertically) the memory of operator o's tasks (i.e., $\mathcal{A}^t.v_o = \top \implies \mathcal{A}^t.m_o > \mathcal{A}^{t-1}.m_o$).

Algorithm 1: Justin's hybrid elastic scaling policy

Parameters: $\Delta_\theta \leftarrow 80\%$; $\Delta_\tau \leftarrow 1ms$; $maxLevel \leftarrow 3$
Result: A new configuration $\mathcal{A}^t$

```
1  A^t ← DS2()                                                  // Obtain initial configuration from DS2
2  for o ∈ A^t do                                               // Iterate over all operators
3      if ¬s_o                                                  // No recorded RocksDB access?
4          A^t.m_o ← ⊥                                          // Disable managed memory for o
5      else
6          if A^t.π_o ≠ A^{t-1}.π_o                             // o's capacity insufficient?
7              if A^{t-1}.v_o                                   // Vertical scaling used last time?
8                  if (A^t.θ_o > A^{t-1}.θ_o ∨ A^t.τ_o < A^{t-1}.τ_o)   // Did it improve?
9                      if (A^{t-1}.m_o + 1) < maxLevel          // Can scale-up?
10                         A^t.π_o ← A^{t-1}.π_o                // Cancel scale out
11                         A^t.m_o ← A^{t-1}.m_o + 1             // Increase memory further
12                         A^t.v_o ← True
13                 else                                         // Did not improve?
14                     A^t.m_o ← A^{t-1}.m_o − 1                 // Roll-back scale in
15             else
16                 if (A^t.θ_o < Δ_θ ∨ A^t.τ_o > Δ_τ) ∧ A^{t-1}.m_o + 1 < maxLevel   // Could
                     vertical scaling be useful?
17                     A^t.π_o ← A^{t-1}.π_o                    // Cancel scale out
18                     A^t.m_o ← A^{t-1}.m_o + 1                 // Attempt increasing memory
19                     A^t.v_o ← True
20 return A^t                                                   // Return the updated configuration
```

5.2.2 Justin Policy Algorithm

Algorithm 1 is called when the capacity of the query is insufficient and requires a reconfiguration. We use the unmodified DS2 trigger based on busyness and backpressure metrics, as detailed in Section 2.3.7 and 4.3.3.

The key principle of Justin's policy is, in appropriate situations, to prioritize a vertical scaling action to a horizontal scaling action decided by DS2. It uses memory efficiency metrics to determine if there is a gain potential and leverages insights from past scaling decisions to evaluate whether they improved capacity. The policy starts by calling the unmodified DS2 (Line 1). It then iterates over all operators. It first identifies if the operator is stateless, which is indicated by the absence of RocksDB-specific metrics in Prometheus (Line 3). If it is, we remove its portion of managed memory (Line 4) and keep the parallelism given by DS2.

For stateful operators, we consider those where a re-scaling decision is proposed by DS2 (Line 6). Other operators' capacities are deemed sufficient and do not need to scale. We then distinguish between two cases: if the operator was vertically scaled the previous time (Lines 7–14) or not (Lines 16–19)[1].

[1]Note that the initial scaling decision falls within the latter case since no prior vertical scaling exists.

If the operator was previously scaled up, we determine if this led to an improvement. As discussed in Section 5.1, this depends on the number of writes vs. reads, the size of the working set, among other factors. A scaled-up operator that does not give better capacity is unlikely to improve with further scale-up. We assess this improvement by comparing the cache hit ratio and state access latency of the current and previous periods (Line 8).[2] If there was an improvement and the operator is not already at the maximum memory level, we cancel the scale-out (Line 10) and replace it with scale-up (Line 11). If there was no improvement, we cancel the previous scale-up (Line 14) to avoid wasting memory. The parallelism recommended by DS2 will thus apply using the previous memory configuration.

If an operator was not scaled up already (or was so earlier than $t - 1$), we evaluate if the cache hit rate is *below* a threshold Δ_θ, indicating insufficient cache size for reads or if average state access latency is *over* a threshold Δ_τ, indicating a significant fraction of operations need costly disk/SSD accesses. In this case, we cancel the scale-out decision (Line 17) and increase the operator's memory level (Line 18). Otherwise, we apply the recommended parallelism.

5.2.3 Implementation

Justin is implemented in version 1.18 of Apache Flink and extends the Kubernetes Operator and its DS2 implementation. Our changes account for about 1,500 LoC. In addition to the policy described earlier in this section, Justin relies on mechanisms allowing the enactment of the decisions: (1) the support of Task Slots (TS) with heterogeneous memory allocations, and (2) the creation when necessary of newer TMs.

Flink 1.18 supports an API allowing to specify custom resource configurations for operators when submitting a new query (number of cores and quantity of heap, network, and managed memory). We extend Flink's Adaptive Scheduler module to allow such changes to be enacted at runtime via a REST API. When a new configuration $\mathcal{A}^t$ is determined by Justin, the Adaptive Scheduler computes a placement of tasks to TSs across TMs.

The artifact is available on GitHub [25d] and has been selected among the DAIS artifacts for publishing in the Science of Computer Programming journal.

5.3 Evaluation

Our evaluation aims to answer the following research questions:

[2]The algorithm uses strict inequality signs < and > for clarity, but we can use a minimum amount of improvement as a percentage as well, implementing a hysteresis.

- **RQ5.1**: For a given target rate, is Justin able to converge to efficient configurations in terms of CPU and memory usage?

- **RQ5.2**: Does the integration of vertical and horizontal scaling lead to a longer convergence time compared to horizontal (i.e., CPU-only) scaling?

We compare Justin and the Flink DS2 implementation and use the obtained rate (i.e., the capacity of a configuration), the sum of assigned CPU cores, and the sum of allocated memory as metrics. We use all six queries of the Nexmark benchmark presented in the previous section (§ 4.3.3). Among these queries, **Q1** and **Q2** are stateless, while the other four are stateful. **Q3** performs an incremental join (i.e., without winddowing), while the remaining queries perform windowed aggregations over different window types and sizes.

Experimental setup. We alternatively deploy Flink with Justin or DS2 on a testbed of 7 nodes from the Gros Cluster (see Section 4.3.3 for specifications). One node hosts the Kubernetes controller, and another hosts Prometheus and Grafana. We deploy the Job Manager on another dedicated node. The four remaining nodes are used to host TMs.

We configure each TM to use 4 CPU cores and 2 GB of memory, shared among 4 TSs. Flink reserves a portion of the available memory for framework management data and JVM-specific data. The default amount of managed memory per TS is 158 MB. With DS2, all tasks get this fixed amount. With Justin vertical scaling, a task may receive from 158 MB ($o_i.m^t = 0$) to 632 MB ($o_i.m^t = 2$, i.e., after two scale-up actions) of managed memory.

We set elastic scaling trigger parameters to keep the average busyness of operators between 20% and 80%. Through experimental analysis, we identify two thresholds that allow suitable identification of memory pressure for some operator's tasks: A cache hit rate over $\Delta_\theta = 80\%$ and an average state access latency over $\Delta_\tau = 1ms$. Similarly to DS2, we use short 2-minute decision windows and a 1-minute stabilization period, i.e., once a scaling decision is enacted, no further decision is taken for 1 minute to prevent oscillations. If no scaling is triggered at the end of the window, the autoscaler waits for the following full metrics window collection. Metrics are collected and aggregated with a granularity of 5 seconds.

5.3.1 Results

Figure 5.3 presents elastic scaling results for Justin and DS2. We show the achieved rate (capacity) and the resource allocation for the two auto-scalers as a function of time. The objective is for sources to reach the target rate

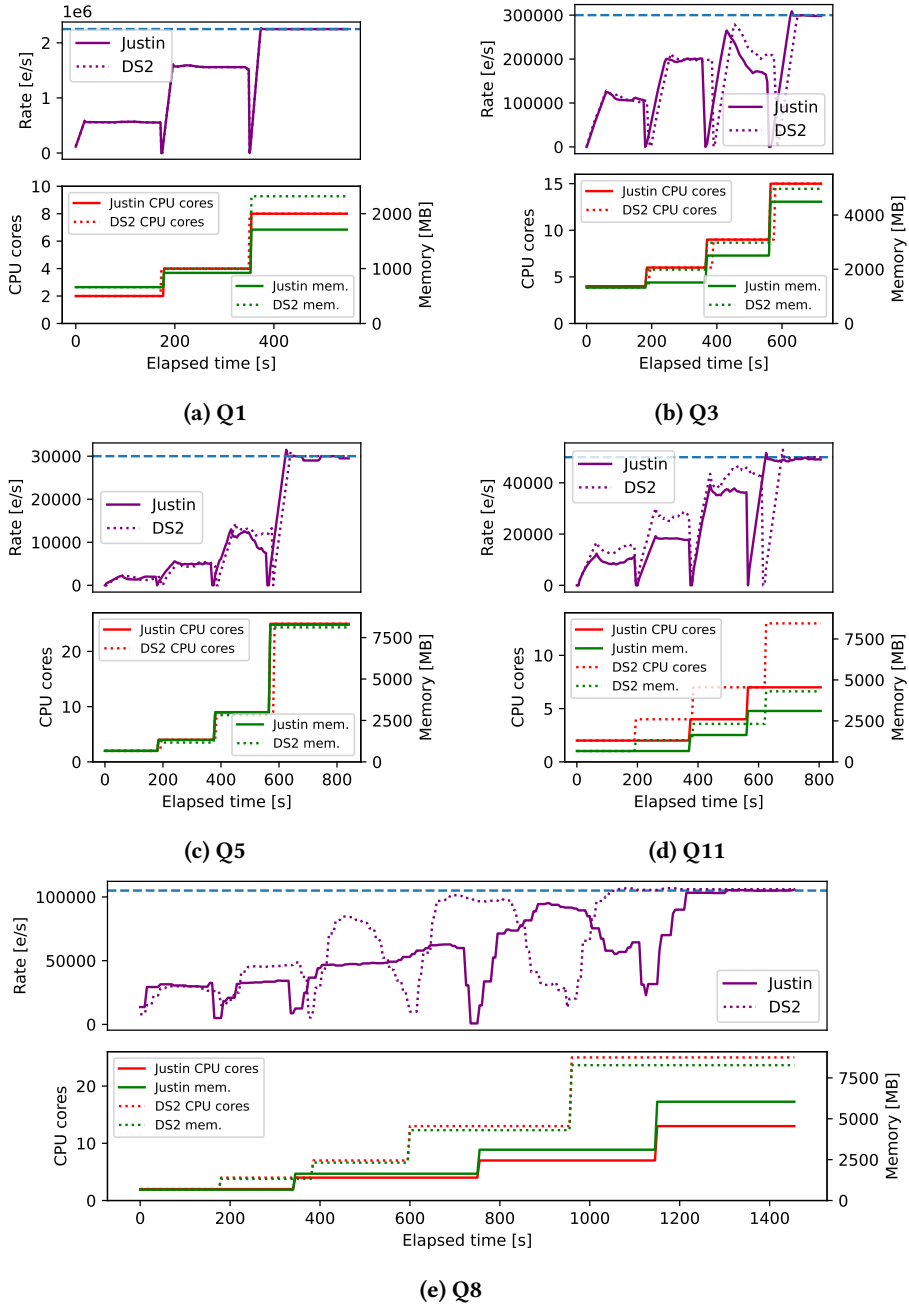


Figure 5.3: Elastic scaling performance of Justin (plain lines) compared to DS2 (dotted lines), achieved capacity (purple), and resulting overall CPU cores (red) and memory consumption (green). The horizontal dashed blue line represents the target rate.

indicated by a thin, dashed blue line after a few reconfigurations. The top plots show the achieved rate, while the bottom plots show the overall CPU and memory consumption (including heap, network, and managed memory). Note that, as in the DS2 paper [Kal+18], we exclude sources from the resource count as these are used as stateless workload injectors that would be replaced by lighter-load Flink sources connected to external sources in production. We include sink operators with a fixed parallelism of one task. Sinks have a low load across all queries and never are a bottleneck.

Q1 and Q2: stateless, single-operator queries. These two queries feature a single stateless operator. As the results for the two were highly similar, we choose only to show those for **Q1**. Figure 5.3a shows that both auto-scalers use two steps to reach a final configuration supporting the 2,250,000 event/s target rate. Both reach a parallelism of 7 tasks for the operator, but Justin disables its managed memory. As a result, the Justin configuration uses 40% less memory, from 2,317 MB for DS2 down to 1,379 MB.

Q3: two stateless and one stateful operator. **Q3** performs an unbounded incremental join, but the size of this operator state converges to a relatively small value (~8 MB), suggesting vertical scaling is likely counterproductive. The two other operators are stateless. Figure 5.3b shows that Justin can free the managed memory of the stateless operators while avoiding unnecessary scale-up for the stateful one, using 10% less memory than DS2. Both auto-scalers converge in the same number of steps. In detail, at the first reconfiguration ($t = 1$, 180s), stateless operators get configured as (1; $\perp$) in Justin and (1; 158) in DS2; after two more scaling steps ($t = 3$) the configuration of the stateful operator becomes (12; 158) in both cases.

Q5: stateful sliding-window aggregation query. **Q5** performs an aggregation over a sliding window, requiring complex access patterns to state in order to re-compute outputs over a new window frequently. Under the Nexmark workload, the state remains small (i.e., ~10 MB). As for **Q3**, vertical scaling would have no impact on the performance of this query. Justin chooses only to perform horizontal scaling. Yet, it saves a small amount of memory by removing the managed memory of the sink operator, as shown by Figure 5.3c. For both auto-scalers, the final configuration for **Q5**'s stateful operator is (24; 158).

We discuss **Q11** before **Q8** to follow the order of presentation in Figure 5.3.

Q11: stateful session window aggregation query. **Q11** results in Figure 5.3d illustrate the benefit of replacing a scale-out decision with a scale-up.

We observe a first reconfiguration a little before 200 s for both auto-scalers. While DS2 increases the parallelism of the primary operator to 3, growing the memory use accordingly, the CPU and memory of Justin remain equal but for a higher resulting capacity. This is explained by the joint deci-

sions of Justin of (1) stripping the sink operator of its unnecessary managed memory and (2) allocating the same amount of memory to the primary operator due to a scale-up decision (i.e., increasing its memory level from 0 to 1 while keeping a single task).

The resulting capacity is lower with Justin but higher when accounted for per expanded core. Follow-up reconfigurations at ~380 s and ~560 s use scale out for both Justin and DS2. However, the better per-task performance of the configuration resulting from the first reconfiguration leads Justin to require 48% lower CPU and a 28% lower memory utilization. The final configuration matching the target rate is (6; 316) for Justin and (12; 158) for DS2. Justin uses the same number of reconfigurations but converges slightly faster than DS2 with this query.

Q8: stateful tumbling window join. Finally, Figure 5.3e presents results for **Q8**. This query is more complex and requires more reconfigurations to achieve the target rate. As for **Q11**, the first reconfiguration of DS2 uses a scale-out while Justin triggers a scale-up of the primary stateful operator. Interestingly, and in contrast with **Q11**, the scale-up of Justin seems to have no real benefit, which may lead to thinking one additional round of reconfiguration will be necessary to compensate for a bad decision. The following reconfigurations show the situation is the opposite. Justin performs three scale-out reconfigurations to reach the target rate, while DS2 needs four. We note that Justin intermediate configurations take longer to stabilize than with DS2, resulting in a shorter convergence time for the latter despite the additional step. In terms of resources, however, the advantage is clearly for Justin, with 48% less CPU cores and 27% less memory than DS2 for the same capacity. The primary operator's final configuration is (12; 316) for Justin and (24; 158) for DS2.

Discussion. Our experiments with Nexmark show that Justin exploits hybrid CPU/memory scaling effectively for all queries but one. It saves memory by stripping unnecessary memory from stateless operators (in primary operators of **Q1**, **Q2**, and **Q3**, and for sink operators of all queries). It significantly reduces the overall resource consumption for a given target rate in complex stateful queries (**Q8** and **Q11**), both in terms of memory as expected and, perhaps more unexpectedly, CPU. This latter result is due to the higher efficiency of tasks not constrained by state access that, as a result, do not need to reach high parallelism for the same capacity as constrained ones. Justin results in fewer or the same number of steps as DS2 and comparable convergence time. Finally, we observe that for a query that does not really benefit from hybrid CPU/memory scaling (**Q5**), Justin does not introduce a penalty over DS2.

5.4 Summary

In this chapter, we presented Justin, a hybrid CPU and memory auto-scaler for Apache Flink. We first motivationed our work by studying the impact of memory allocation on the performance of stateful operators in Flink. We showed that the relation between memory allocation and performance is not straightforward and depends heavily on the nature of the workload. Based on these insights, we designed Justin, an auto-scaler that extends the DS2 framework to consider memory allocation in addition to CPU when making scaling decisions. Justin continuously collects metrics about the resource usage and storage performance of each operator's tasks. Based on these metrics, Justin's auto-scaling policy derives CPU and memory allocations for each operator independently. We implemented Justin in Flink 1.18 and evaluated it on a 7-node cluster using the Nexmark benchmark. Our results show that Justin can produce configurations with 48% less CPU and 27-28% less memory for complex stateful queries while requiring the same or fewer reconfiguration steps than DS2.

This work opens several interesting perspectives. First, and similarly to DS2, Justin makes implicit assumptions about the absence of skew, i.e., unbalanced key popularities leading to an imbalanced load between the different tasks of an operator. Heterogeneous memory allocation between tasks of the same operator could be an option, although it would require a significantly more complex auto-scaler. Finally, Justin and DS2 consider the auto-scaling and resource allocation of queries in isolation. An auto-scaler that considers the co-placement of query tasks, e.g., running memory-intensive tasks of some queries with stateless tasks of another query, may lead to better consolidation and cluster resource use overall. This last direction is the focus of the next chapter.

Publications. The work presented in this chapter contributed to the following publication:

Justin: Hybrid CPU/Memory Elastic Scaling for Distributed Stream Processing. Donatien Schmitz, Guillaume Rosinosky, and Etienne Rivière. In IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2025. Lille, France, 102-118.

This publication received the **IFIP Best Paper Award** at DAIS 2025 and the **IFIP DisCoTec-wide Best Paper Award** at DisCoTec 2025.

Charon: Optimized Multi-Query Task Placement and Resource Allocation

6

Chapter 5 presented the foundation for fine-grain resource allocation in DSP systems. It focused solely on the memory requirements of individual queries and proposed a method to allocate memory to operators in a way that prevents unnecessary horizontal scaling decisions, resulting in overall resource savings. While the proposed method was effective, it did not consider mostly-idled operators and focused on single-query scenarios.

In Chapter 3, we analyzed real-world workloads and highlighted the prevalence of multi-query scenarios in production environments. More importantly, we observed that queries in this context often experience highly heterogeneous workloads, leading to significant variations in CPU and memory requirements. This chapter builds upon those observations and introduces Charon, a novel approach that dynamically adapts resource allocation and leverages optimized task placement strategies to enhance the performance of multi-query processing. Leveraging the mechanism introduced in Chapter 5 for fine-grain resource allocation in a multi-query context can lead to suboptimal resource utilization due to the independent treatment of each query. Indeed, queries are reconfigured in isolation, without considering the resource requirements of co-located queries. The default task placement strategy in DSP systems, which typically relies on a round-robin approach, does not account for the specific resource allocations of tasks and can lead inefficient use of cluster resources. Charon addresses this by introducing a bin-packing-based task placement strategy that optimizes the allocation of tasks across workers, taking into account their specific CPU and memory requirements. We propose two task placement strategies: a Minimum Downtime strategy that only reconfigures tasks that do not sustain their target rate, and a Maximum Consolidation strategy that re-evaluates the placement of all tasks to optimize resource utilization.

In this chapter, we first provide a complementary background on task placement strategies in DSP systems, the motivation for our approach, and

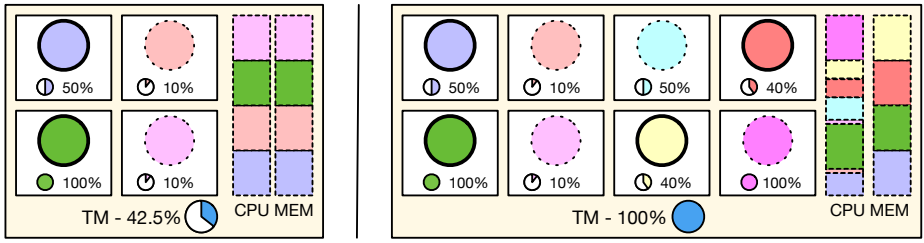


Figure 6.1: A one-size-fits-all approach to resource allocation can lead to sub-optimal resource utilization. On the left, the default Flink configuration allocates one task per available CPU core to each task, leading to CPU under-utilization and allocation of memory to stateless tasks (dotted circles). On the right, CPU is distributed as fine-grained shares, and memory is allocated only to tasks that require it (bold circles).

the overview of Charon (Section 6.1). We then present its three main components: Monitoring (Section 6.2), Resource Planning (Section 6.3), and Task Placement (Section 6.4). Finally, we evaluate Charon on the Gros Cluster (see Section 4.3.3 for specifications) using the Nexmark benchmark (Section 6.5) before concluding the chapter (Section 6.6).

6.1 Background and Motivation

As detailed in Sections 4.1 and 5.1, an essential limitation of Flink's *one-size-fits-all* resource allocation is that it provides the same amount of CPU and memory resources to each task, regardless of its actual requirements. Single-task operators with a busyness level lower than 100% are assigned a full CPU core, thereby wasting computational capacity. The allocation of managed memory to stateless operators is also a pure waste, as their tasks make no use of the RocksDB storage backend. Figure 6.1 (left) illustrates how these limitations result in an overall underused TM that is nonetheless unable to host new tasks.

As described in the previous chapter, Flink recently introduced a new feature allowing fine-grain resource allocation to tasks, i.e., selecting arbitrary amounts of managed and heap memory, and assigning *shares* of CPU resources lower than a full core [25b]. With this feature, the number of TS on a TM is no longer fixed, but is limited only by the total assigned volume of each resource. Figure 6.1 (right) shows the potential of fine-grain resource allocation, where only stateful tasks are assigned managed memory and all are assigned CPU shared corresponding to their busyness levels. While instrumental in allowing fine-grained resource management, this new mechanism suffers from two limitations. First, as explained in the previous chapter, the allocation is fixed and declared at compile time, disallowing runtime adapta-

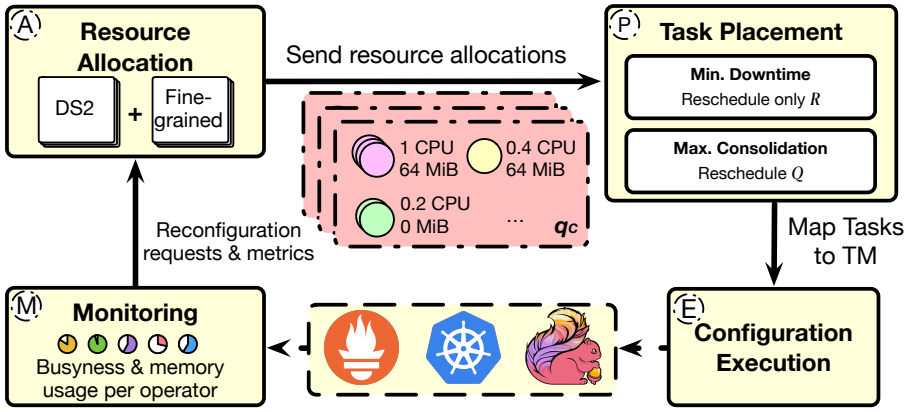


Figure 6.2: Charon components and workflow. Monitoring collects application metrics and triggers reconfigurations. Resource Allocation determines the optimal CPU and memory for query operators. The Task Placement component then determines the optimal placement of tasks on the available TM. Finally, Charon applies the new resource allocation and task placement decisions to the running queries.

tion. Second, the choice of resource amounts relies on expert prior knowledge. The appropriate resources for an operator depend, however, not only on the nature of its computation, but also on the volume and content of events it processes, which may only be known at runtime.

We also observe that existing simple and greedy task *placement* mechanisms are ill-suited for fine-grained resource allocation. First, they either do not consider the heterogeneity of tasks (e.g., for the Round Robin policy) or only consider a single dimension (e.g., the least-used TM policy only considers CPU allocation). Second, they target the placement of tasks *in isolation*, i.e., one after the other, rather than optimizing the placement of the multiple queries found in a typical DSP cluster. As a result, these placement policies achieve poor *consolidation*, and may require more TM to support a given workload than an optimized placement.

Overview. We present Charon, an integrated scheduler for multi-query DSP workloads that addresses these limitations. Charon builds upon two main principles: (1) autonomous fine-grained resource allocation integrated with state-of-the-art elastic scaling mechanisms; and (2) optimized, multi-query task placement mechanisms that aim to maximize consolidation.

Figure 6.2 presents an overview of its components, which are then detailed in the following three sections of this paper. Similarly to Justin, Charon extends Flink Monitoring with the ability to detect improper memory configurations (Ⓜ, Section 6.2). These metrics may trigger periodic re-scaling operations for some of the running queries. The Resource Allocation mod-

Symbol	Description
Default configurations	
C	Number of cores per TM
H	Available heap memory per TM
M	Default amount of Managed Memory per TS
Workload	
$q \in Q$	A query in the set of queries
$o \in O$	An operator in the set of all operators
$O_q \subseteq O$	Operators specific to query q
Inputs of Resource Allocation	
$\mathcal{R}^t \subseteq Q$	Set of queries requiring reconfiguration
b_o^t	Average observed busyness for operator o during last period
s_o^t	Use of managed memory (binary) by operator o during last period
Output of Resource allocation and inputs of Task Placement	
$\mathcal{A}^t$	Resource allocation, including:
$\mathcal{A}^t.\pi_o$	Parallelism (number of tasks) for operator o
$\mathcal{A}^t.c_o$	Share of CPU core(s) for each task of operator o (%)
$\mathcal{A}^t.m_o$	Share of managed memory for each task of operator o (%)
Output of Task Placement	
N^t	Number of necessary TM
$\mathcal{P}^t$	Map between tasks of operators O and TM $\{1, \ldots, N^t\}$

Table 6.1: Notations used by Charon for an allocation and placement workflow initiated at the end of observation period t.

ule extends elastic scaling policies with the ability to assign different CPU shares and memory allocations to different operators (Ⓐ, Section 6.3). The Task Placement module considers the resulting set of tasks and their requirements, and determines a placement that maximizes consolidation, thereby limiting the number of necessary TMs. In contrast with existing Flink greedy task placement strategies, this module considers the placement as a multi-dimensional optimization problem solved using a bin packing algorithm. We present two possible optimization policies that differ in their ability to redeploy stable queries to optimize consolidation, albeit at the cost of additional reconfiguration downtime (Ⓟ, Section 6.4). Finally, Charon applies the new resource allocation and task placement decisions to the running queries (Ⓔ). For the latter, we extended the fine-grain allocation API introduced in the previous chapter, allowing us to additionally specify the placement of each task, upon a query reconfiguration.

Table 7.1 summarizes the notations used in the rest of the chapter.

Implementation. Charon is implemented as a standalone Python service running alongside the Flink Kubernetes Operator. The artifact is open-sourced and available online [25e]. Charon leverages the monitoring capabilities of the Flink Kubernetes Operator to collect the necessary metrics for its operation. We expose a REST API to pull the output of the DS2 algorithm, as

implemented in the Operator. Charon monitors all running queries in the cluster. When it receives the scaling decisions from DS2 for all queries, it triggers its Resource Allocation component to adjust their resource allocation. The output is passed to the Task Placement component, which computes the optimal placement of tasks on TM, depending on the selected optimization policy. We leverage the constraint programming capabilities of Google's OR-Tools [PD25] to solve the corresponding ILPs. Finally, Charon applies the new resource allocation and task placement decisions to the running queries using Flink's REST API. The service accounts for around 800 lines of Python code.

To apply the new resource allocation and task placement decisions, we first extended the fine-grain resource allocation API introduced in the previous chapter. This extension allows us to specify not only the CPU and memory amounts for each task, but also the TM on which it should be placed. Finally, we modified the task allocation strategy in Flink to leverage this new API and enforce the placement decisions made by Charon. The modifications required around 1,200 lines of Java code in the Flink codebase.

6.2 Monitoring

Monitoring collects information about running queries. The two metrics of interest are: (1) the busyness of operators, i.e., how much time is spent processing events *vs.* waiting for input by their tasks and, in addition, (2) measures of the use of managed memory by these tasks, indicating access to the RocksDB state backend, or its absence thereof.

As Justin, Charon targets minimal modification to Flink and the Flink Kubernetes Operator. This applies to the periodic metrics collection and aggregation strategy. While metrics are collected by the Prometheus service at a high frequency (every five seconds by default), they are aggregated over a longer period (one minute by default) to avoid noise and short-term fluctuations. As for the previous chapter, we define $t > 1$ as the index of the last aggregation period. For all operators $o \in O$ currently deployed in the cluster, $b_o^t \in [0, 100\%]$ is the average busyness of all tasks of o, expressed as a fraction of time, for period t. Additionally, we observe the use of managed memory by examining the statistics of RocksDB access times. If these statistics are void while events were processed, we deduce that the task is stateless. This information is encoded in the binary variable s_o^t.

Monitoring is also responsible for triggering reconfiguration for queries that are considered under- or over-provisioned. We use the existing Flink mechanism, which is based on thresholds on b_o over the last observation period. A low average busyness indicates over-provisioning and the need to scale down the query containing operator o. Symmetrically, a high aver-

Algorithm 2: Charon resource allocation policy

Parameters: The previous configuration $\mathcal{A}^{t-1}$
Parameters: Query set $\mathcal{R}^t$ requiring reconfiguration
Result: A new configuration $\mathcal{A}^t$

$\mathcal{A}^t \leftarrow \mathcal{A}^{t-1}$ // Keep configuration for non reconfigured queries.
for $q \in \mathcal{R}^t$ **do**
  for $o \in O_q$ **do**
    $\mathcal{A}^t.\pi_o \leftarrow \text{DS2}(o)$ // Fetch the recommended parallelism.
    if $\neg s_o^t$ // Operator is stateless?
      $\mathcal{A}^t.m_o \leftarrow 0\%$ // Disable managed memory.
    else
      $\mathcal{A}^t.m_o \leftarrow 100\% \times M$ // Default managed memory.
    if $\mathcal{A}^t.\pi_o = 1 \bigwedge \mathcal{A}^{t-1}.\pi_o = 1$ // Stays single-task?
      $\mathcal{A}^t.c_o \leftarrow \left\lceil \frac{b_o^t}{20\%} \right\rceil \times 20\%$ // Allocate fraction of CPU.
    else
      $\mathcal{A}^t.c_o \leftarrow 100\%$ // Keep full core.
return $\mathcal{A}^t$ // Return new configuration for task placement.

age busyness indicates over-provisioning and the need to scale out the corresponding query. At the end of period t, operators crossing thresholds $b_o^t \leq 40\%$ or $b_o^t \geq 80\%$ flag the corresponding queries for reconfiguration in a set $\mathcal{R}^t \subseteq \mathcal{Q}$, i.e., $\mathcal{R}^t = \{q \in \mathcal{Q} \mid \exists o \in O_q, b_o^t \leq 40\% \vee b_o^t \geq 80\%\}$.

Reconfiguration triggers are decided at the end of every one-minute observation period. Following a reconfiguration request and the restart of queries by the resource allocation and task placement modules, which we detail next, Monitoring skips two observations as a cool-down period to allow the system to stabilize.

6.3 Resource Allocation

Resource Allocation combines elastic scaling with the determination of optimal resource allocation for operators of queries selected for reconfiguration, i.e., in set $\mathcal{R}^t$. In addition to selecting the number of instances using existing Flink algorithms, it can adapt the allocation of CPU and memory for different operators. The goal of Resource Allocation is to ensure that each operator has sufficient resources to handle its workload while minimizing resource waste and maximizing overall system efficiency.

Resource Allocation builds upon the existing elastic scaling algorithm in Flink, based on DS2 [Kal+18]. Based on Monitoring metrics, Resource Allocation adjusts the results of the elastic scaling algorithm to remove managed memory for stateless operators and adapts CPU allocation for low-traffic operators.

Algorithm 2 details Resource Allocation operation when triggered at the

end of period t. Inputs are the set of queries under reconfiguration $\mathcal{R}^t$, the average busyness of their operators b_o^t, and their use of managed memory m_o^t. Its output is a mapping $\mathcal{A}^t$ between each operator in these queries and to its resource allocation, i.e., its parallelism ($\mathcal{A}^t.\pi_o$), CPU cores ($\mathcal{A}^t.c_o$), and amount of managed memory ($\mathcal{A}^t.m_o$).

The algorithm starts from a copy of the previous configuration (Line 1), then loops over the set of queries in $\mathcal{R}^t$ and every operator of these queries. For each operator, it fetches DS2's recommended parallelism (Line 4). If the operator is stateless, it disables its managed memory by setting it to 0% of the default allocation (Line 6); stateful operators keep 100% of that default value (Line 8). We note that *source* operators' parallelism is constant, as it depends on the query initial configuration and number of connections to data sources (e.g., Kafka). Source operators are generally stateless and can get stripped of managed memory with no performance penalty.

In a second phase, the algorithm checks whether the CPU allocation is currently oversubscribed and adjusts it to a fraction of a core if possible. We only consider operators with a parallelism of one, i.e., a single task, over at least two observation periods. For this, we verify that the parallelism in the observation period $t-1$ was already one (Line 9). In this case, the algorithm allocates a fraction of a CPU core to the operator o based on its observed busyness level b_o^t (Line 10). This allocation is in discrete fractions to avoid very small CPU shares but also to make the process of task placement, detailed in the next section, less computationally expensive. Operators that do not match these criteria keep the default allocation of one CPU core (Line 12). The allocation of heap memory to tasks is the same as their allocation of CPU: a task gets $\mathcal{A}^t.c_o \times \frac{H}{C}$ heap memory, where H is the volume of heap memory for a TM and C its number of available cores.

Resource allocation outputs the new configuration $\mathcal{A}^t$ for all operators in the set of queries $\mathcal{Q}$ (Line 13), which will be the input to the task placement component we detail next.

6.4 Optimized Task Placement

After Resource Allocation determines the CPU and memory profiles for operators and the number of tasks they require, Task Placement is responsible for spawning these tasks on available TM, potentially launching new ones if necessary.

Flink's task placement strategies are designed with homogeneous tasks in mind, and deploy queries in isolation. As mentioned in Section 6.1, these strategies are ill-suited for heterogeneous requirements and result in sub-optimal consolidation.

Charon optimizes task placement, improves consolidation, and reduces

the number of necessary TM by two means: (1) it does not use greedy placement policies but models task placement as a bin-packing optimization problem, and uses a solver to derive an optimal solution; and (2) it considers the reconfiguration of multiple queries *together* rather than one by one, thereby offering greater potential for improved consolidation.

We consider two policies, differing in the set of queries considered for placement. The *Minimum Downtime (MD)* policy only considers queries scheduled for reconfiguration ($\mathcal{R}$), but does not modify the placement of other queries ($\mathcal{Q} \setminus \mathcal{R}$). The rationale for this policy is to impose no downtime beyond what is already imposed by Monitoring decisions. It is suitable for setups where queries are relatively short-lived and/or where the cost of query downtime is deemed higher than the cost of using slightly more resources. In contrast, the *Maximum Consolidation (MC)* policy favors an optimal usage of these resources but does so by potentially replacing all running queries ($\mathcal{Q}$), and not only those scheduled for reconfiguration. It is adapted for workloads formed of long-running queries, where the cost of short-term reconfigurations and associated downtimes is compensated by lower resource utilization in the long term.

Both strategies take as input $\mathcal{R}$, $\mathcal{Q}$, and the resource allocation $\mathcal{A}^t$. They output the number of necessary TM, N^t, and a mapping of tasks to these TM, $\mathcal{P}^t$. MD and MC are modeled as bin packing problems. A bin packing problem is an optimization problem in which items (tasks) are placed into bins (TM) of a given capacity (CPU and memory). The objective is to place every item in the bins in a way that minimizes the number of necessary bins. Let n be the total number of tasks to place, and m be an upper-bound on the number of required TM, i.e., when each task receives a full core among the C available on each TM:

$$n = \sum_{q \in Q} \sum_{o \in O_q} \mathcal{A}^t.\pi_o \tag{6.1}$$

$$m = \left\lceil \frac{n}{C} \right\rceil \tag{6.2}$$

Let $x_i \in \{0, 1\}$ be a decision variable equal to 1 if TM i is used, and 0 otherwise. Moreover, let $y_{ij} \in \{0, 1\}$ be a decision variable equal to 1 if task j is assigned to TM i and 0 otherwise, with $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. Finally, let P_{ij} be the set of pre-placed tasks to TM, i.e., tasks of queries not scheduled for reconfiguration in $\mathcal{R}^t$, where P_{ij} contains task j if it is currently assigned to TM i. To simplify notations, we note $q(j)$ the query to which a task j belongs and $o(j)$ its operator (thus, $\mathcal{A}^t.c_{o(j)}$ is the CPU allocation of j).

The MD policy is equivalent to solving the following Integer Linear Program (ILP):

$$\text{Min.} \quad \sum_{i=1}^{m} x_i \tag{6.3}$$

$$\text{S. t.} \quad \sum_{j=1}^{n} \mathcal{A}^t.c_{o(j)} \cdot y_{ij} \leq C \cdot x_i, \quad \forall i \in \{1, ..., m\} \tag{6.4}$$

$$\sum_{j=1}^{n} \mathcal{A}^t.m_{o(j)} \cdot y_{ij} \leq (M \cdot C) \cdot x_i, \quad \forall i \in \{1, ..., m\} \tag{6.5}$$

$$\sum_{i=1}^{m} y_{ij} = 1, \quad \forall j \in \{1, ..., n\} \tag{6.6}$$

$$y_{ij} = 1, \quad \forall i \in \{1, ..., m\}, \forall j \in P_i \text{ if } q(j) \notin \mathcal{R} \tag{6.7}$$

$$x_i, y_{ij} \in \{0, 1\} \tag{6.8}$$

The objective function (6.3) minimizes the number of necessary TM. The first two constraints, (6.4) and (6.5), ensure that the total CPU and memory requirements of tasks assigned to each TM do not exceed its capacity. The third constraint (6.6) ensures that each task is assigned to exactly one TM. Finally, the fourth constraint (6.7) pre-assigns tasks of queries not in $\mathcal{R}$ to their current TM assignation, avoiding unnecessary reconfigurations.

The MC policy gets rid of the fourth constraint (6.7) to allow all queries (Q) to be replaced if this leads to a better placement. A query that has at least one task of one operator replaced will incur a snapshot and re-deployment of the query, and an associated downtime. However, this opens the possibility of more aggressive optimization and resource savings.

6.5 Evaluation

We evaluate the Charon prototype with the goal of answering the following research questions (RQ):

- **RQ6.1**: What is the impact of Charon fine-grained resource allocation? Considering one query, does it allow reducing the necessary resources for that query while matching rate requirements, and does it require fewer reconfigurations compared to the baseline?
- **RQ6.2**: Does the multi-query, optimized placement of Charon improve consolidation compared to existing single-query policies used by Flink?
- **RQ6.3**: What is the relative impact of using the MD and MC strategies on consolidation and, for MC, on the number of reconfigurations?

Event source	Q1	Q2	Q3	Q5	Q8	Q11
Auction	—	—	600	—	600	—
Person	—	—	200	—	200	—
Bid	9,200	9,200	—	9,200	—	9,200

Table 6.2: Default rate of the three streams used by the six queries of Nexmark (in events per second).

- **RQ6.4**: Is the optimization approach used for task placement scalable?

Our metrics of interest are the number of necessary TM and the number of reconfigurations required to support target query rates. We compare the following four candidates:

- Baseline (**BL**) is the default Flink and Flink Kubernetes Operator resource allocation and placement.
- Fine Grained (**FG**) uses fine-grained resource allocation strategies as proposed in Section 6.3, but keeping the default (round-robin) placement strategy.
- Minimum Downtime (**MD**) and Maximum Consolidation (**MC**) use fine-grained resource allocation and optimized, multi-query placement with the two strategies outlined in Section 6.4.

Benchmarks. We use the same queries as for the evaluation of DS2 and Justin: **Q1**, **Q2**, **Q3**, **Q5**, **Q8**, and **Q11**. **Q1** and **Q2** are stateless queries and are primarily composed of respectively a *map* and a *filter* operator. **Q3** and **Q8** are stateful queries, with **Q3** performing an *incremental join* query and **Q8** computing a *join* over a window. **Q5** and **Q11** are both stateful queries performing *windowed aggregations*.

Nexmark uses three event sources — *Auction*, *Bid*, and *Person*. The benchmark determines the rate of event generation for these sources as a ratio: for every *Person* event generated, 3 *Auction* events and 46 *Bid* events are generated. The default rate is 10,000 events per second, with a distribution across sources and use by the six queries shown in Table 6.2. We note this default rate as "x1" and consider higher data volumes by multiplying the rates in Table 6.2 by 5 ("x5") and 10 ("x10").

Experimental Setup. We run the experiments on a testbed of eight nodes from the Gros cluster (see Section 4.3.3). One node is used solely for managing the Kubernetes cluster and for the observability stack, which includes Prometheus [24b] and Grafana [14c]. We use another dedicated node for the JM, and the remaining 6 nodes for TM. Each TM is configured with 4 CPU

Rate	Cand.	Q1				Q2				Q3			
		TM	TS	C	M	TM	TS	C	M	TM	TS	C	M
10k (x1)	BL	1	3	3	1029	1	3	3	1029	2	5	5	1715
	FG	1	3	**1.4**	**0**	1	3	**1.4**	**0**	**1**	5	**2.6**	**343**
50k (x5)	BL	1	3	3	1029	1	3	3	1029	2	5	5	1715
	FG	1	3	**1.4**	**0**	1	3	**1.4**	**0**	**1**	5	**2.6**	**343**
100k (x10)	BL	1	3	3	1029	1	3	3	1029	2	5	5	1715
	FG	1	3	**1.4**	**0**	1	3	**1.4**	**0**	**1**	5	**2.6**	**343**
		Q5				**Q8**				**Q11**			
		TM	TS	C	M	TM	TS	C	M	TM	TS	C	M
10k (x1)	BL	2	5	5	1715	1	3	3	1029	2	5	5	1715
	FG	**1**	5	**3.4**	**686**	1	3	**2.2**	**343**	**1**	5	**3.4**	**686**
50k (x5)	BL	3	9	9	3087	1	3	3	1029	4	15	15	5145
	FG	**2**	9	**7.4**	**2058**	1	3	**2.2**	**343**	4	15	**13.4**	**4116**
100k (x10)	BL	4	15	15	5145	1	3	3	1029	7	27	27	9261
	FG	4	15	**13.4**	**4116**	1	3	**2.2**	**343**	7	27	**25.4**	**8232**

Table 6.3: Comparison of of the baseline (BL) and fine-grain (FG) scaling strategies for single queries. The queries run individually at different throughput levels. The values indicate for each *sustained* configuration: the number of needed TMs (TM), the total number of tasks (TS), the total number of CPU cores (C), and the total amount of managed memory used by the query (M in MiB). Bold values indicate improvements compared to the baseline.

cores — resulting in 4 TS for the default one-size-fits-all resource allocation and placement strategy — and 4 GiB of RAM. The number of TM that are actually deployed depends on the decisions of the task placement algorithm.

6.5.1 Fine-Grained Resource Allocation: Impact on Individual Queries

We first address **RQ6.1** and the impact of fine-grained resource allocation on the performance and cost for individual queries. Table 6.3 presents the resource utilization of queries after the elastic scaling process has converged, for both the BL and FG setups (i.e., using the default round-robin placement strategy). Each query runs for a total of 20 minutes, targeting the three rate levels (x1, x5, and x10).

For each query, we report the final, stable resource allocation sustaining the target rate, with the number of necessary TM, the number of tasks (TS), allocated CPU cores (C), and managed memory used by the query (M in MB). We first observe that, in all configurations, the number of tasks (i.e., the operators' parallelism) is identical for BL and FG. For stateless queries (**Q1** and

Q2), FG uses significantly fewer resources, both in terms of CPU and memory, than BL. Observations from Monitoring allow Resource Allocation to detect and remove unnecessary managed memory from stateless operators. For stateful query **Q3**, FG manages to reduce the number of TM used from 2 to 1, while using the same number of TS. For **Q5**, FG uses one less TM than BL throughout levels x1 and x5 but requires the same number of TM for x10, although using fewer CPU cores and memory overall. **Q8** requires a single TM for both candidates at every rate level, although here again, FG uses fewer CPU cores and memory than BL. Finally, for **Q11**, FG uses one less TM than BL at level x1 but requires the same number of TM for x5 and x10, primarily due to CPU requirements. This more complex stateful query nonetheless manages to use fewer resources overall in all configurations.

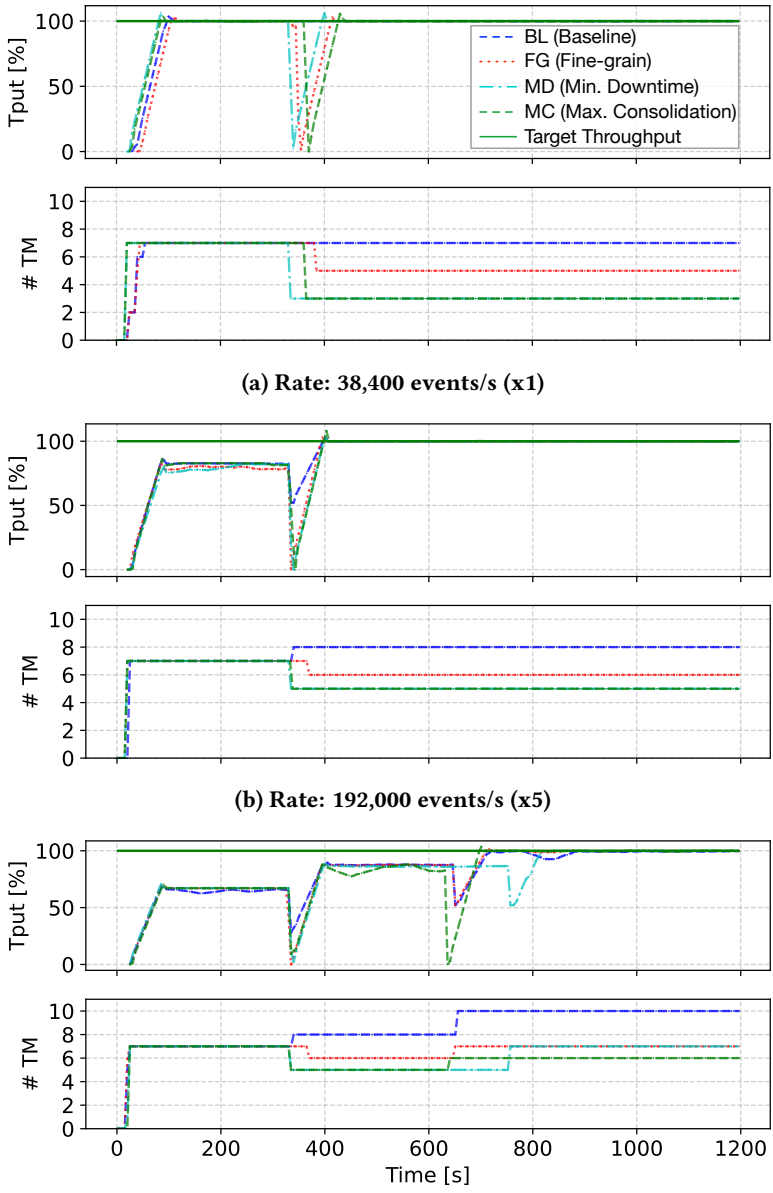
In summary, this first series of experiments allows answering positively to **RQ6.1**: fine-grain resource allocation enables more efficient resource utilization for each query. Compared to the Justin policy presented in the previous Chapter, Charon manges to reduce both CPU and memory resources for all queries, including stateful ones. We note however that the lack of smart memory allocation policy (i.e. only removing managed memory from stateless operators) limits the possible gains for stateful queries.

6.5.2 Placement with Multiple Queries

We now explore **RQ6.2** and **RQ6.3** with multi-query workloads, i.e., where the six Nexmark queries are running simultaneously and consume events from the same input flows. The total number of events consumed by the six queries combined is, respectively, 38,400 (x1), 192,000 (x5), and 384,000 (x10) events per second. We consider the four candidates: BL and FG, as in the previous subsection, plus the multi-query-aware MD and MC.

Figure 6.3 presents the achieved throughput (top) and the number of TM used (bottom) as a function of time, and for the three rate levels. For readability, the throughput is normalized as a percentage of the target rate, which is represented by a dashed green line. Reconfigurations impact the throughput of some queries (with BL, FG, and MD) or all queries (with MC), followed by a ramp-up phase to reach the new, sustainable throughput. Table 6.4 presents the number of reconfigurations for the different candidates and the different rates.

The x1 scenario shows that the BL candidate does not reconfigure any of the queries and continues to use the original 7 TM used for one-size-fits-all provisioning at the beginning. In contrast, the three other candidates perform such reconfigurations and significantly reduce the number of TM while respecting the target rate. In all cases, a single re-scaling decision is necessary. We can observe the impact of multi-query, optimized placement by compar-



(a) Rate: 38,400 events/s (x1)

(b) Rate: 192,000 events/s (x5)

(c) Rate: 384,000 events/s (x10)

Figure 6.3: Multi-query results for 4 scaling candidates for 3 rate levels. The top-most plot of each figure shows the total throughput of all queries as a percentage of the target rate (indicated by a dashed green line). The bottom plots show the number of Task Managers (TM) used by each candidate to achieve their throughput.

Rate	Cand.	Q1	Q2	Q3	Q5	Q8	Q11	Total
10k (x1)	BL	0	0	0	0	0	0	0
	FG	1	1	1	1	1	1	6
	MD	1	1	1	1	1	1	6
	MC	1	1	1	1	1	1	6
50k (x5)	BL	0	0	0	1	0	1	2
	FG	1	1	1	1	1	1	6
	MD	1	1	1	1	1	1	6
	MC	1	1	1	1	1	1	6
100k (x10)	BL	0	0	0	2	0	2	4
	FG	1	1	1	2	1	2	8
	MD	1	1	1	2	1	2	8
	MC	2	2	2	2	2	2	12

Table 6.4: Number of reconfigurations made by each candidate to reach the target rate for each query and in total.

ing the results of FG on the one hand with those of MC and MD on the other hand. The latter two candidates manage to reduce the number of necessary TM to 3, i.e., a ≃43% of what the default Flink requires.

For the x5 rate, BL scales only **Q5** and **Q11** once and ends up using 8 TM. FG, MD, and MC all reconfigure all queries once, and reduce the number of necessary TM through better consolidation. However, we observe the impact of placement optimization at time 350s, where MD and MC manage to place the resulting tasks on 5 TM while the round robin placement of FG needs 6 TM.

The x10 rate is more demanding and requires more than one reconfiguration per query to sustain the target rate. BL reconfigures all queries once, except **Q5** and **Q11** that need two reconfigurations, reaching a final resource budget of 10 TM. At time 350s, the three other candidates find the same configuration sustaining an overall throughput of ~85% of the target rate. However, FG does not consolidate well and uses one more TM than MD and MC. After a final scaling step at 650s, FG achieves the target rate using 7 TM, MD uses 7 TM, and MC uses 6 TM, 60% of the budget used by BL.

As expected, we observe that BL makes the least number of reconfigurations, as it only scales queries that cannot sustain the target rate with their initial configuration. For rates x1 and x5, FG, MD, and MC make the same number of reconfigurations for each query. Finally, in scenario x10, MC makes more reconfigurations than FG and MD, as it tries to optimize consolidation by redeploying tasks from already well-placed queries.

In summary, this experiment shows the interest of optimizing placement considering multiple queries compared to only using fine-grain resource allocation with the default, round-robin placement algorithm of Flink, positively

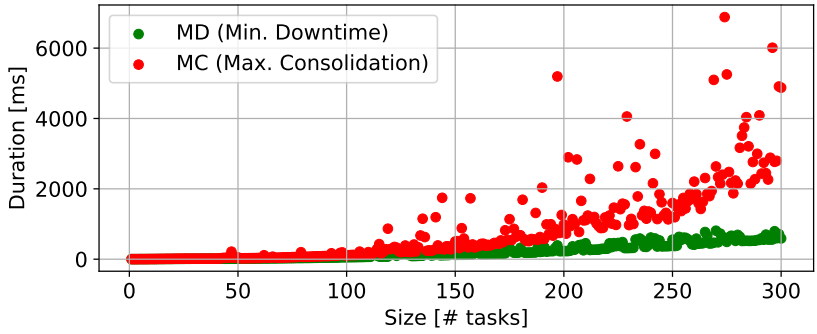


Figure 6.4: Solver efficiency for the task placement problem with varying numbers of tasks. The solver's performance degrades as the number of tasks increases. Pre-assignment of 25% tasks to TM helps reduce the search space and overall solving time.

answering **RQ6.2**. The MC strategy, by forcing the replacement of stable queries together with those identified for reconfiguration, achieves slightly better consolidation and resource utilization at the cost of additional reconfigurations than MD, also answering **RQ6.3**.

6.5.3 Task Placement Performance

We finally investigate **RQ6.4** and the scalability of the task placement optimization problem. Both MD and MC candidates utilize bin-packing optimization, which is known to be NP-hard. Our implementation uses Google's OR-Tools [PD25] to solve the corresponding ILPs. We leverage the constraint programming capabilities of OR-Tools to efficiently explore the solution space and find optimal solutions for both strategies.

To evaluate the solver's efficiency, we conduct microbenchmarks on the placement problem with large numbers of heterogeneous tasks. We compare the performance of MC and that of MD with a pre-assignment of 25% of stable tasks to TM. Figure 6.4 shows the solver's performance as the number of tasks increases for MD and MC. The solver's performance degrades as the number of tasks increases, with a significant increase in solving time for instances with more than 200 tasks for MC, which is already a significant number for a DSP cluster. Pre-assignment helps reduce the search space and allows for higher task counts, as it provides additional constraints that guide the solver towards feasible solutions more quickly. A possibility to mask the cost for very large clusters and query sets would be to pre-compute placement and resource allocation before enacting decisions, i.e., outside of the auto-scaling decision loop, although this would require more significant changes to Flink and the Flink Kubernetes Operator. It is also possible to interrupt the search

after a timeout duration for an approached solution. Overall, we conclude this evaluation by answering **RQ6.4** affirmatively for common workload sizes.

6.6 Summary

This chapter highlighted the inefficiencies of state-of-the-art resource allocation and task placement strategies in DSP systems when handling multi-query workloads with heterogeneous resource demands. Operators with varying CPU and memory requirements lead to suboptimal resource utilization due to the inability of existing systems to adapt resource allocations and placements dynamically. Furthermore, the prevalent round-robin task placement strategy fails to consider the specific resource needs of tasks, resulting in fragmented workers and increased resource wastage. Based on these observations, we designed and implemented Charon, an integrated auto scaler that combines fine-grained resource allocation with optimized multi-query task placement. By dynamically adjusting CPU and memory allocations based on operator needs and modeling placement as a bin-packing problem, Charon achieves significant improvements in resource efficiency and consolidation over the state of the art. Experiments with the Nexmark benchmark show its ability to reduce wasted resources, lower the number of required TM, and balance trade-offs between downtime and consolidation.

This work opens several interesting perspectives. First, the use of ILP solvers for task placement allows for the easy integration of additional constraints and optimization goals. For example, future work could explore the integration of network-aware placement strategies that consider data locality and inter-task communication patterns to further optimize performance. Second, Charon currently assumes homogeneous TM capabilities; extending the placement algorithm to handle heterogeneous TM configurations could enhance its applicability in diverse cloud environments. Finally, while Charon focuses on CPU and memory resources, future research could investigate the inclusion of other resource types, such as GPU or storage I/O, to provide a more comprehensive resource management solution for DSP systems.

In the last two chapters, we presented two complementary auto-scaling solutions for DSP systems. These solutions address key challenges in resource allocation and task placement, demonstrating significant improvements in resource efficiency and system performance. In the next chapter, we will explore a complementary direction by investigating how to predict, prior to deployment, the performance of DSP queries under various configurations.

Publications. The work presented in this chapter contributed to the following publication:

Charon: Fine-Grain Resource Allocation and Optimized Task Placement for Multi-Query Distributed Stream Processing. Donatien Schmitz,

Guillaume Rosinosky, and Etienne Rivière. In Proceedings of the 41st ACM/SIGAPP Symposium On Applied Computing (SAC), 2026. Thessaloniki, Greece.

The artifact associated with this publication is available on GitHub [25e].

StreamBed: Capacity Planning for Stream Processing

7

In this chapter, we address a complementary aspect of resource management in DSP systems: capacity planning. While previous chapters focused on fine-grain elasticity, we now turn our attention to the challenge of determining the appropriate resource allocation for DSP applications before deployment. To this end, we present *StreamBed*, a capacity planning framework designed to assist practitioners in estimating the required resources for their DSP jobs, prior to deployment.

We begin by introducing, in Section 7.1, the motivation behind capacity planning in DSP systems and the challenges it entails, followed by a complementary review of related work. We then provide an overview of the StreamBed framework, detailing its architecture and key components, in Section 7.2. Subsequently, we delve into the specific components of StreamBed, namely the Capacity Estimator (Section 7.3), the Configuration Optimizer (Section 7.4), and the Resource Explorer (Section 7.5). We also discuss the implementation details of StreamBed in Section 7.6. Finally, we evaluate the effectiveness of StreamBed through a series of experiments, presented in Section 7.7, and conclude with a summary of our findings and potential future directions in Section 7.8.

7.1 Motivation

It is hard to know, before the full-scale deployment of a new query, the resources budget that it will *eventually* require after elastic scaling, or how these resources will have to be configured to *sustainably* ingest a target workload (i.e., achieve a given *capacity*). As another example, the use of a total of twelve servers and the parallelism of each of the five operators in Figure 7.1's configuration at peak capacity is the result of runtime decisions that depend on the scaling behaviors of operators and their reaction to operational conditions, e.g., the amount of RAM (heap, managed, network memory) provided to each of the tasks, the performance of the storage sub-system, or characteristics of the input stream. As another example, in Chapter 5, the auto

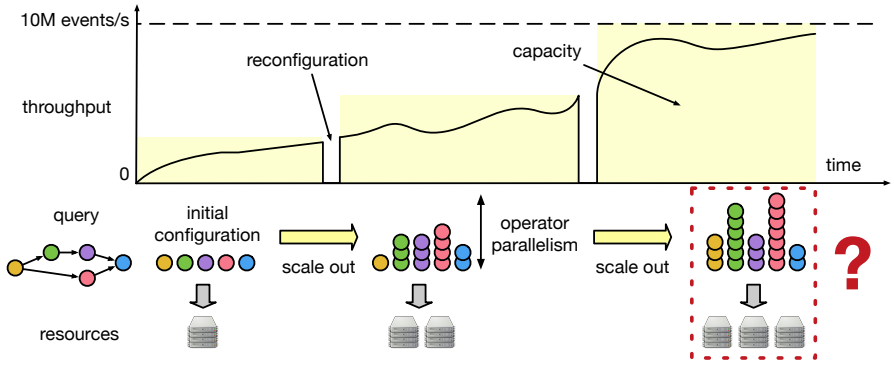


Figure 7.1: DSP engines scale out to handle increasing rates. Reconfigurations apply the necessary parallelism for a query operators (its *configuration*) to increase its capacity. Our objective is to determine *in advance* the resources budget and its configuration for a target, large rate.

scaler needed five reconfigurations to find a sustainably stable configuration for **Q8**. The scaling behavior of queries is often sub-linear, disallowing simple proportionality-based predictions. They often exhibit non-trivial resource usage and performance patterns, such as load spikes due to stragglers [FBD20] or imbalance between tasks' loads due to skew [Gul+16; Kal+18; Zou+22].

Uncertainty in the resource requirements of queries often leads practitioners to exercise caution and over-provision their infrastructure, as presented in Chapter 3. An infrastructure that cannot scale out to the needs of a query could, indeed, result in cascading failures and query termination [Pau20]. When DSP jobs are deployed in a public cloud, the problem becomes that of mastering costs, as uncontrolled scale-out may lead to paying for a large amount of on-demand resources.

Capacity planning tools for DSP queries can help practitioners to address these challenges. Figure 7.2 illustrates three use-cases of such tools. First, in a public cloud setting, capacity planning can help to estimate the cost of running a query at a target throughput for a certain duration, allowing users to budget their expenses in advance. Second, in a private cloud setting with limited resources, capacity planning can help to determine if a new query can be safely deployed alongside existing ones without exceeding the available resources. Finally, capacity planning can provide an initial configuration for a new query, reducing the number of runtime reconfigurations needed to reach the desired capacity and avoiding the associated costs and service interruptions.

Numerous prior works have studied the problem of resource provisioning for DSP queries. One line of work is based on *benchmarking*, which targets the deployement of a query at a production scale. For example, Theodo-

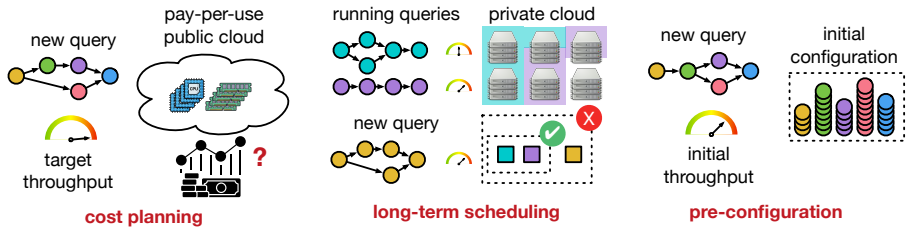


Figure 7.2: Use-cases of capacity planning for DSP queries. On the left, it helps to determine the cost of running a large query at a certain throughput and for a certain duration in the cloud. In the middle, it allows for determining if a new query can be safely deployed alongside others in a private cloud with limited resources. On the right, it returns an initial configuration for a query, avoiding the costs and service interruptions of many scale-out reconfigurations.

lite [HH21; HH22] is a systematic method for benchmarking the scalability of DSP systems, such as Flink and Kafka Streams. Theodolite implements different search strategies piloting tests with varying CPU budgets, verifying if service-level objectives (SLO) are met for each of them, and forming a scalability profile. It defines "realistic" use cases (e.g., downsampling, time-based aggregation, hierarchical aggregation) and workload dimensions (e.g., number of keys, window size) to evaluate how resource needs grow with workload. The authors also provide a cloud-native benchmark framework and implementations for Apache Flink and Kafka Streams, showing that both scale roughly linearly, but configuration choices strongly impact efficiency. However, the synthetic nature of the benchmarks limits their applicability to real-world queries. Rafiki [Pfi+22] allows determining the configuration (level of parallelism) of each operator of a query by piloting a series of runs and gradually increasing the parallelism of each operator based on observed backpressure metrics.

Another class of work proposes to *model* the DSP queries, their performance, and their scalability. Truong *et al.* [THS17; Tru+18] use queueing theory to model each DSP operator as a queue and predict system behavior under load. They measure execution time, inter-processing time, and variance at runtime, then apply approximation methods to estimate queue length, latency, and stability. By gradually increasing input rate, they determine each operator's effective utilization—the maximum safe load before queues become unstable or backpressure occurs. An algorithm then fits utilization levels to observed queue sizes, enabling prediction of throughput limits and resource needs. Some authors propose to model the performance and resource usage of queries based on general characteristics, learning from a large set of queries and identifying a general model, e.g., mixture-density

networks [Kho+15; Kho+16] or zero-shot cost models [Hei+22; Agn+23] (discussed in Section 4.1). These approaches require initial training using a large set of queries (typically, several thousands). As no dataset of *real* queries of this size exists, the authors have to resort to synthetic, randomly-generated queries that may not represent real workloads.

Prior research proposed various techniques for capacity planning in DSP systems, including benchmarking and modeling approaches. However, these methods often rely on synthetic benchmarks or require extensive training on large datasets of queries, which may not accurately reflect real-world workloads. In this work, we propose StreamBed, a capacity planning tool that models the performance and resource usage of DSP queries based on their actual execution traces.

7.2 Overview

We detail StreamBed objectives, operational model, and target workloads. Then, we provide an overview of its components.

Objectives and assumptions. StreamBed targets long-running queries that need to scale out to hundreds of cores for their execution. We consider a system formed of two clusters: a large one is dedicated to production deployments, while a much smaller one is dedicated to capacity planning. The two clusters use the same hardware and network interconnect, to make observations on one transposable onto the other.

The goal of StreamBed is to build a capacity planning model for a *specific* DSP query provided by the user. We assume that a dataset, representative of the input of the job, is also provided by that user, e.g., historical data for the corresponding source stored in a data lake or using artificial data generators. We do not require, however, that the query ran previously on this dataset or any other dataset. The query does not have to be modified by the user, but StreamBed needs to be informed of fields in events' schemas used to represent event time, if any. This information is needed to replace recorded, historical times with emulated times in controlled runs of the query.

We assume that the submitted DSP job uses an arbitrary combination of stateless and *windowed* operators including joins, as is common for continuous queries [Ver+23]. This means that operators do not accumulate state without limit, but operate on a working set formed of the events received over a window, defined based on time and/or a number of events.[1] We do

[1]We do not target *batch* processing jobs [Car+15], where the processing throughput is dictated by the job deployed capacity and not by its sources, and where some operators such as non-windowed joins may keep accumulating state for all events of an input before they can process another.

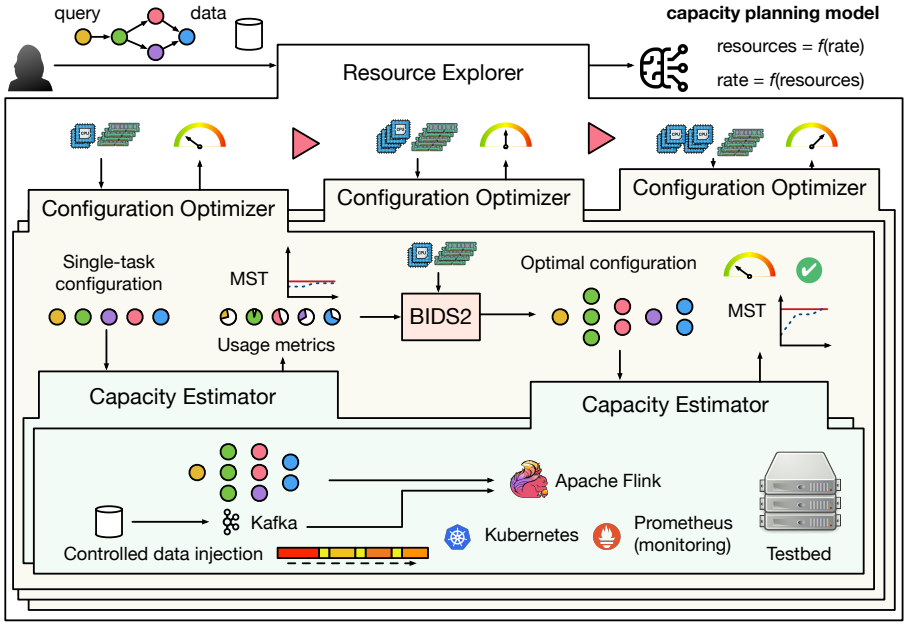


Figure 7.3: StreamBed workflow. The user submits her query and a representative dataset. The resource explorer builds a capacity planning model from capacity estimations with small resource budgets. For each budget, the configuration optimizer determines the best resource allocation and its MST via controlled benchmarking by the capacity estimator.

not make assumptions on the length of this window or the sizes of operators' working sets.

StreamBed builds a model allowing querying the necessary resources budget for different input rates (i.e., the number of task slots and their resource profile) as well as their appropriate configuration (parallelism for each operator). The returned configurations must be able to sustain the requested throughput without over- or under-provisioning.

StreamBed handles skew (and the resulting imbalance in the loads of tasks of a given operator) and the unpredictable load peaks due to stragglers by considering the measured capacity as the maximal sustainable throughput (MST [IPV17; Kar+18; CYH20]) of the query under a specific resources budget, and the evolution of this MST with varying budgets.

StreamBed components. The general workflow and the three nested components of StreamBed are illustrated in Figure 7.3. At the top-most level, the **Resource Explorer (RE)** drives the exploration of different *resource budgets*, i.e., numbers of task slots with a given resource profile. For each budget, the RE collects the achievable capacity or Maximal Sustainable Throughput (MST) and the associated configuration, which it uses to build a capacity plan-

ning model. The choice of profiles is driven by a Bayesian Optimization algorithm for a set of target candidate models. In Figure 7.3, the RE evaluates three such resource budgets.

The determination of the best configuration and of achievable MST for a given resource budget is under the responsibility of the **Configuration Optimizer (CO)**. For resource profiles that have not yet been evaluated, a first step evaluates in-situ (using the lowest-level component, the Capacity Estimator) a minimal configuration with a parallelism of 1 for each operator, to collect usage metrics. Then, the CO leverages BIDS2 (Bounded-Inverse DS2), an evolution of the DS2 algorithm able to determine, in one pass and for a bounded resource budget, the best possible configuration. In Figure 7.3, the CO determines the MST for the smallest of the budgets submitted by the RE for evaluation.

The determination of the MST of a given configuration is performed by the **Capacity Estimator (CE)**, which uses in-situ, controlled runs of the job in the small-scale testbed. Determining the MST experimentally is a complex task. Performance tends to fluctuate heavily during *ramp-up* phases, while the state of operators and buffers between tasks are not stabilized at their peak size. Performance is also generally highly unstable when operating above the MST, and queries may exhibit important variations of resource usage over time. We devise a controlled data injection strategy that can effectively determine the MST of a configuration by stress-testing it and adjusting input rates using a dichotomous strategy.

In the following sections, we detail these components in a bottom-up fashion, starting with the Capacity Estimator (Section 7.3), following with the Configuration Optimizer (Section 7.4), and finishing with the Resource Explorer (Section 7.5).

7.3 Capacity Estimator

The role of the Capacity Estimator (CE hereafter) is to determine experimentally the Maximal Sustainable Throughput, or MST, of a job in a specific configuration, upon request of the Configuration Optimizer. The MST, a common performance metric for DSP [IPV17; Kar+18; CYH20], is the highest possible average actual rate that can be processed by the configuration, that corresponds to the injected rate (i.e., no piling up of unprocessed records between the Source operator and the rest of the job). The goal of the CE is to estimate the MST in a small amount of time, while guaranteeing its accuracy, in particular with respect to the impact of stateful operators, skew, and stragglers. The CE must also take into account that injected rates higher than the MST often lead to chaotic behaviors such as unsteady actual processing rates.

Load injection. The CE uses controlled load injection, attaching rate-limited

sources to the job. These sources can connect with a distributed storage in a data lake, allowing the use of data at rest as a representative input. Alternatively, StreamBed can employ as a source an online pseudo-random event generator provided by the user. In both cases, sources attempt to inject events at up to a fixed target rate but abide by the back-pressure received from downstream operators.

Warmup phase. A newly started job will typically accept incoming load at a much higher throughput than its steady-state MST. Two factors explain this. First, each edge of the job graph is associated with buffers; back-pressure mechanisms only kick in when the fill rates of these buffers reach a threshold. Initially, empty buffers lead to a temporarily higher "absorption capacity". Second, stateful operators start with an empty state that gradually grows until it reaches a steady state (e.g., a full sliding window of events). Initially, operations requiring to access state can be much faster, possibly biasing measurements. These observations lead us to the need for a *warmup* phase prior to any measurement, to observe performance metrics on a stabilized job.

Evaluation strategy. A policy decides on variations of the input rate, and on when to measure the actual processing rate. A naive approach to determining the MST would be to start from a low target rate and gradually increase it until the source starts observing a certain back-pressure level. This approach is inappropriate for two reasons. First, it mixes the necessary warmup phase with the actual measurement phase, risking estimating the MST of a job that has not reached its steady state. Second, it does not take into account the inertia of the job under test: A change in input rate leads to cascading effects on buffer occupancies and back-pressure levels. These effects often take several seconds, and sometimes several minutes, to stabilize. Using an ever-increasing rate bears risks that this stabilization never occurs.

We propose a strategy, illustrated by Figure 7.4, that takes into account these concerns by using fixed target throughputs, including necessary stabilization phases but avoiding measurements during these phases. It starts with a sufficiently long warmup phase where we inject throughput at the maximal possible rate, building up sufficient state at stateful operators and filling buffers. Reaching this steady state can require several minutes of warmup, primarily depending on available memory and the size of the working state of stateful operators.

The warmup phase is followed by the evaluation itself, using a dichotomous approach similar to a binary search. We consider a *maximal* and a *minimal* rate, $\max_r$ and $\min_r$, initially set to ∞ and 0. The search progresses in phases. In each phase, a target rate r is tested. If the test is successful, which the algorithm determines as the actual processing rate being equal

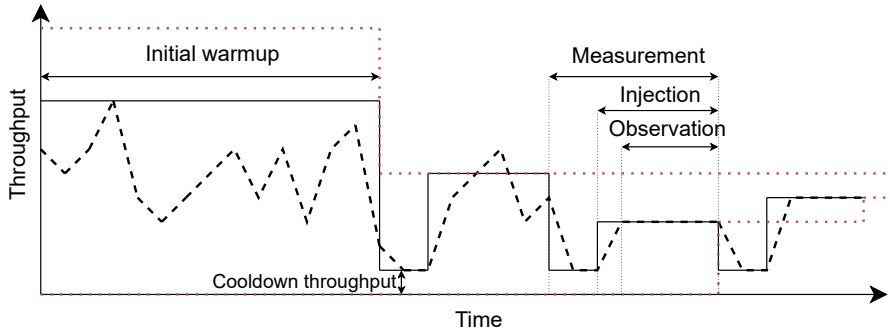


Figure 7.4: The Capacity Estimator uses an evaluation strategy based on fixed-rate load injection, with a dichotomous search following an initial warmup phase. Black plain lines show the target input rate, while black dashed lines show the actual, measured processing rate. Dashed, thin red lines present the maximal and minimal rates $\max_r$ and $\min_r$.

or very close (i.e., ≥99%[2]) to the incoming rate then $\min_r = r$. Otherwise, $\max_r = r$. The next target is the average between these two values, i.e., $r = (\max_r + \min_r)/2$. The search stops when reaching a configurable sensibility (e.g., the next value of r is within 1% of the previous) or after a maximal number of iterations.

A measurement for a target rate r starts with a *cooldown* phase, using a low (but non-zero) rate allowing operators to process events in their input buffers and recover from saturation in the previous measurement. Then, the target rate r is applied for an *injection* phase. We measure the actual processing rate after a short duration to enable a local ramp-up for the new injection. During the *observation* phase, we measure the achieved rate from the point of view of the source and its stability and collect busyness metrics and actual input rates for all operators.

The metrics for the final measurement are returned along with the MST to the Configuration Optimizer.

7.4 Configuration Optimizer

The second component of StreamBed is the Configuration Optimizer, or CO for short. It receives from the Resource Explorer a query and a resource budget, i.e., a number of task slots with a given profile (amount of RAM). Its goal is to return the optimal configuration fitting this budget, together with its MST as measured *in situ* using the CE.

[2]Due to the use of a rate limiter, the actual rate can never exceed 100%.

Symbol	Description
$\mathcal{B}$	Task slots budget
λ_o	Observed true processing rate of a task of operator o
r_o	Observed ratio of operator o's input rate over source's rate
π_o	Optimal parallelism of the operator o
λ^*_{src}	Optimal input rate of the source operator
λ^*_o	Optimal input rate of one operator instance o

Table 7.1: Notations used by the BIDS2 algorithm.

The CO requires usage metrics for the execution of a *minimal* configuration, i.e., where each operator has a parallelism of 1. These usage metrics include the *actual input rate* and the level of *busyness* of each single-task operator, i.e., the percentage of time it spends actually processing events. The CO maintains a cache of these single-task configuration metrics for different resource profiles, reusing previous measurements when possible. When no data exists, the CO requests the CE to evaluate this single-task configuration with the target profile. An exception to this reuse rule is when the Resource Explorer explicitly requests the evaluation of a single-task configuration as part of its exploration.

Based on observed metrics on the single-task configuration, we formulate the problem of determining the *optimal* capacity for the target resource budget, i.e., the highest possible source input rate. The corresponding *best possible* configuration is the one where the busyness metric of all tasks, across all operators, is the highest possible while avoiding the saturation of any task. We solve this optimization problem using an algorithm we name BIDS2, for Bounded-Inverse DS2. In contrast with the original DS2 presented in Section 2.3.7, BIDS2 does not determine the amount of necessary resources for a given input rate, but determines the optimal configuration for a *bounded* resource budget. The configuration resulting from the optimization is given as an input to a call to the CE to determine its MST and experienced busyness levels.

BIDS2 algorithm. Table 7.1 lists the variables used in the optimization. Our goal is to maximize the throughput of the source:

$$\max \lambda^*_{\text{src}} \tag{7.1}$$

To this end, we must find π_o the level of parallelism for each operator o. π_o is the decision variable for each operator o, o varying between 1 and the number of operators of the job excluding the sources. As Kalavri *et al.* [Kal+18], we assume in this optimization problem a linear relation between the pro-

cessing rate and the level of busyness: we compute the true processing rate λ_o of a task of operator o by dividing its actual processing rate by the average busyness level of its tasks. While this relation can be considered as linear at a small scale and is sufficient in this module, linearity is not guaranteed at higher scales: in StreamBed, we manage this non-linearity when building the model at the Resource Explorer level, as it will be reflected in the different capacities measured for different budgets and profiles.

We compute the optimum processing rate λ_o^* of an operator o based on its parallelism π_o, as given by Equation 7.2.

$$\forall o : \ \lambda_o^* = \pi_o \cdot \lambda_o \tag{7.2}$$

Then, we compute using Equation 7.3 the rate of each operator expressed as a function of the decision variable input rate λ_{src}^*: The ratio r_o, precomputed using the actual rates returned by the CE is used as a multiplier to estimate this proportion.

$$\forall o : \lambda_{\text{src}}^* \cdot r_o \leq \lambda_o^* \tag{7.3}$$

Finally, we set the constraint in Equation 7.4 that the sum of the parallelism for the different operators should be exactly the number of task slots $\mathcal{B}$:

$$\sum_o \pi_o = \mathcal{B} \tag{7.4}$$

The configuration resulting from the optimization is given as an input to a second call to the CE, allowing it to determine its MST and to return it together with observed metrics.

7.5 Resource Explorer

The goal of the Resource Explorer (RE) is to build the capacity planning model for a target unknown query. It drives the collection of capacity measurements and configuration information for different resource budgets (number of tasks) and resource profiles (total amount of memory per task). Based on configurations and rates received from the CO, the RE can further determine an optimal configuration for a (predicted) resource budget.

The RE models the relation between resources and capacity as a *surrogate model*, i.e., an approximation of a complex system replacing expensive simulation models [AAM20]. The RE identifies the most representative function that describes the relation between a number of TS Π (all used, i.e., $\sum \pi_o = \Pi$) using resource profiles with M MB of memory per TS and the resulting capacity λ_{src}^* as detailed in equation 7.5.

$$f(M, \Pi) = \lambda^*_{src} \tag{7.5}$$

Some jobs may have a linear scaling profile but this is not the case for all of them. In particular, jobs containing stateful operators, aggregates, and especially joins often see their improvement in performance decrease with additional resources. We must model these sub-linear scaling profiles appropriately, i.e., using monotonically increasing functions with a derivative that decreases slowly over time. Based on our experience and on results from other researchers [Gul+16], we select in StreamBed two simple functions, logarithm and square root, matching this requirement and performing well in practice. StreamBed can easily accommodate alternative functions. The three linear regressions we use are given in equations 7.6, 7.7, and 7.8, with a and b the slopes of the functions of the surrogate models, applied to the quantity of memory and task slots, and c the intercept. The goal of the RE is to determine which of the three models fits best the observations, find coefficients a, b, and c, and use the chosen model.

$$aM + b\Pi + c = \lambda^*_{src} \tag{7.6}$$

$$a \log M + b \log \Pi + c = \lambda^*_{src} \tag{7.7}$$

$$a\sqrt{M} + b\sqrt{\Pi} + c = \lambda^*_{src} \tag{7.8}$$

Overview. The RE builds concurrently the three candidate models based on a succession of measurements by the CO. The model is then used to extrapolate, for high values of λ^*_{src}, values of Π that are outside of that search space.

The construction of surrogate model candidates must balance their extrapolation capacity and the cost of collecting measurements. The collection of the MST for a given resource budget and resource profile requires running the CO and up to two instances of the CE. Due to the need for the query to stabilize to its steady state, these steps easily represent dozens of minutes of data collection in the test cluster.

The model construction happens in three phases. First, in a *candidate search* phase, we use a black-box optimization technique to determine new resource budgets and profile candidates optimizing the distance between the prediction of the current best model and the actual results of the runs. Second, in a *model selection* phase, we determine which of the three models has the best potential for extrapolation. Finally, as the best model is selected, it can be used to directly predict the MST for a given configuration.

Model performance. We measure the accuracy of the candidate models with the root mean squared error (RMSE), a quantitative measure of how well the model predicts the resource requirements by comparing predicted values

with the results obtained by the CO. Lower RMSE values indicate a better-performing model.

The predictive capabilities of the models are determined using Leave-One-Out Cross-Validation (LOOCV): for each observation in the data set, we evaluate if the model trained on all other observations provides a good (low) RMSE value. LOOCV helps to minimize overfitting and provides a reliable estimate of the model's predictive capabilities, especially in cases where there are few observations [Ng97].

Candidate search. The goal of the candidate search is to find a number of candidate couples (M, Π) and their corresponding MST that reduce the training error for the current best model.

The search space is 2-dimensional. The number of task slots Π ranges from the number of operators of the query to the number of cores in the test cluster. M is discretized using a configurable level of granularity. The maximal value for M is the largest possible in the test cluster when divided by all cores.

If we consider a query with 9 operators on a test cluster with 48 cores (39 possible values for Π) and memory ranging from 512 MB to 4 GB in increments of 512 MB (8 possible values) then the search space admits 312 different combinations.

An intensive evaluation of the search space (i.e., a grid search [DN19]) is simply too costly. We use instead Bayesian Optimization (BO) [Ant21], an optimization method targeting black-box functions, often used for expensive-to-evaluate problems such as hyper-parameter optimization [FH19]. BO is based on a *probabilistic model* (a Gaussian Process) approximating the true function coupled to an *acquisition function* guiding the search for the optimal point for a *cost function* supplied by the user. The acquisition function balances exploration (searching in uncertain locations of the search space) and exploitation (searching in regions with low predicted error) of candidate points, collected in set D. Typical acquisition functions include Probability of Improvement, Expected Improvement, and Lower Confidence Bound. The RE uses Expected Improvement as it balances the search for low mean (exploitation) and high variance (exploration) candidates.

The candidate search must identify configurations that are likely to yield better RMSE values for the currently better candidate model. The cost function in Equation 7.9 identifies the lowest LOOCV score amongst candidate models, reducing the training error.

$$\text{BestModel}(D) = \underset{\text{model} \in \{\text{linear,log,sqrt}\}}{\arg\min} \text{LOOCV}(D, \text{model}) \tag{7.9}$$

The set D is bootstrapped by collecting measurements from the 4 "corners" of the search space, i.e., 4 combinations of lowest and highest values of

Π and M, enabling to compute an initial LOOCV score for the three candidate models. The general behavior of the training process is that, following an initial chaotic phase, the RMSE decreases until new individuals start to worsen the score. Our stop criterion takes this behavior into account. We proceed to a minimum of 3 measurements in addition to the 4 corners. Then, we stop when the RMSE increases by more than 10% between two measurements, or when reaching the maximal number of 20 measurements (by default).

Following this bootstrap phase, BO proceeds to execute the candidate search phase by calling the acquisition function, a call to the CO followed by error computation using Equation 7.9 as the cost function, and then the adaptation of the internal probabilistic models based on the received candidate couple (M, Π). Note that, to account for the unavoidable measurement variations that happen when collecting MST from the CO and the CE, the RE may decide to re-evaluate a candidate couple that has been previously evaluated to reduce uncertainty. Our experience is that, indeed, variations are common between two runs with the same budget and profile in particular for complex queries involving joins and/or windowed operations.

Model selection. Following the BO, we select the most appropriate model from the three candidates based on their predictive capability. To evaluate it we use a 50:50 split on D. Since we want to compute the extrapolation capability, we use the first half of D with observations with the lowest values of Π as the training set to predict the other half as the test set, as commonly reported in the literature [SLM18; SW22].

We apply each candidate model and evaluate how well it predicts observations in the test set. The model with the lowest average RMSE for points in the test set is selected (obviously, we use the model trained on the full D and not only on the training set used for model selection).

Model usage. The model considers the resource budget and their profile as independent variables. It can be used directly to derive the capacity. Returning the necessary resource budget for a target input rate requires, however, to solve the model inversely. We adopt an iterative strategy where we consider one or several memory value(s)[3] M and we incrementally increase Π until the predicted capacity matches or exceeds the request.

StreamBed tends to estimate resource budgets for which the requested rate is very close to the maximum capacity. To guarantee that the requested rate will be sustainable, the RE applies a slight over-provisioning factor by interrogating the model with 110% (by default) of the requested rate. This tunable factor accounts for the fact that, while in our experience the RE reports a throughput close to the MST temporary slowdowns (due to the network or

[3]The user's production cluster may already be configured with a specific profile, or she may want to know which profile is best for her query.

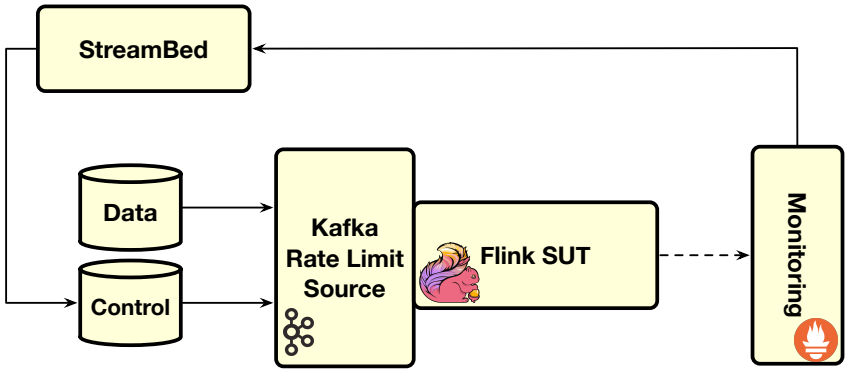


Figure 7.5: Technical components of StreamBed. StreamBed requests the execution of controlled runs of a query under test, evaluates its performance for a given configuration, and drives the exploration of resource budgets to build a capacity planning model.

JVM overheads) may lead to small decreases in capacity that are then difficult for the system to catch up.

The model returns the number of task slots and their profile but not their configuration. If requested, this configuration can be computed using a final pass of BIDS2 using the true processing rates and busyness levels collected for the largest number of cores in D with the selected resource profile(s).

7.6 Implementation

StreamBed uses a cloud-native stack with Kubernetes [25g] supporting a data lake using Apache Kafka for data ingestion, storage, and replay and Flink for processing, as described in Section 4.3.3. We use the Strimzi Kafka Kubernetes Operator [Str23] to use Kafka as a source of data for Flink and Spotify's Flink Kubernetes Operator [Spo23] to be able to deploy Flink easily with different memory and CPU settings. Because this work was done prior to the official release of Flink Kubernetes Operator, we use the alternative Spotify's Kubernetes Operator to be able to deploy Flink easily with different memory and CPU settings. Figure 7.5 illustrates the technical components of StreamBed and their interactions. The artifact associated with this work is available on GitHub [25f].

Capacity Estimator. The CE interacts with a running, unmodified instance of Flink v.1.14.1, using Apache Zeppelin [Apa23] notebooks. Runtime measurements (e.g., about processing rates and busyness) are collected using Prometheus [24b] with a 5-second aggregation period.

We deploy a Flink instance in the test cluster without auto-scaling features, using only dedicated cores for every TS without simultaneous multi-

threading. This instance is deployed by the CE on demand with a fixed, homogeneous resource profile for the TS. It is re-deployed only if the CO requests a different profile, which takes less than a minute.

The CE must be able to control precisely the rate at which the source of a query under test in the small cluster receives data. We developed a novel *rate-limited* Kafka source connector extending the regular Apache Flink Kafka source [Fli22]. This rate-limited source obtains data from a first Kafka topic as well as rate control information from a secondary topic.

Configuration Optimizer and Resource Explorer. The RE and CO are both developed in Python. Contrary to the previous Chapter that used the OR-Tools library, the CO uses the Python PuLP library [COI23] coupled to the CBC (Coin-or Branch and Cut) solver [COI17] both developed by the COIN-OR foundation. This change is only due to this work being done prior to the previous Chapter. Metrics are retrieved from CE runs in Prometheus using the prometheus-api-client library [San23]. The RE uses the scikit-optimize library [Hea+21] for Bayesian Optimization and the scikit-learn library [Ped+11] for the regression.

7.7 Evaluation

We wish to answer the following research questions:

- **RQ7.1**: Is the evaluation of the MST by the CO and CE accurate, i.e., can the RE rely on these measurements for its model training?
- **RQ7.2**: Is the CO effective at deriving the "best possible" configuration for a given resource budget?
- **RQ7.3**: Is StreamBed able to predict, based on small-scale runs, an accurate budget of resources, their profile, and their configuration, such that a production run based on this prediction is neither over- nor under-provisioned?
- **RQ7.4**: How much resources and time are necessary for StreamBed to build capacity planning models?

We first present our workloads and our experimental setup. Then, we answer questions **RQ7.1** and **RQ7.2** using micro-benchmarks at the CE and CO levels. Finally, we answer questions **RQ7.3** and **RQ7.4** using macro-benchmarks involving the RE and the full StreamBed stack as well as a comparison to production-scale deployments.

Query	Description	Min rate ($\times 10^3$ evt/s)
Q1	**Currency conversion**: converts bid values from dollars to euros	1,600
Q2	**Selection**: filter bids with specific auction identifier	3,600
Q5	**Hot items**: determine auctions with most bids in last period	50
Q8	**Monitor new users**: identify active users in last period	1,400
Q11	**User sessions**: compute number of bids each user makes while active	60

Table 7.2: Minimal rates used for the selected Nexmark queries.

7.7.1 Experimentation infrastructure

Target workload and queries. We use representative queries from the Flink SQL implementation of Nexmark. According to our target assumptions, we consider queries with no continuously increasing state, i.e., we exclude **Q3** as it consists of an incremental *join*. As described in the previous Chapters, Nexmark provides a data generator that we use with its default settings, i.e., the input event stream features 2% of events linked to persons, 6% proposing auctions, and 92% representing bids by the former to the latter. The average sizes of these events are respectively 200, 500, and 100 Bytes. We populate the data lake (Kafka) with pre-generated data streams that we use as data-at-rest input for StreamBed operations.

Experimental setup. StreamBed needs sufficiently many nodes supporting data injection and source operators (if these are under-dimensioned, it may return unsuccessfully after failing to replay a target rate). Table 7.2 presents as a reference the minimal rate that a single-task configuration supports for each query. Queries **Q1**, **Q2**, and to a lesser extent **Q8** have a high minimal rate, while **Q5** and **Q11** process fewer events per second.

When replaying data at rest, the limiting factor are our modest SSDs, with a peak capacity per Kafka server of about 1,200,000 events/s. This represents, for our target queries, a minimal number of 8 Kafka nodes for $q5$ and $q11$ and 16 Kafka nodes for $q1$, $q2$, and $q8$ to be able to inject rates decided during the RE exploration (obviously, using machines with multiple, high-performance drives would allow reducing this number). Rate-limited source operators are CPU-bound. We must also ensure they never are the limiting factor when testing a specific configuration. We identified that 64 source tasks are necessary to support the maximal rate that the Kafka nodes can serve (4 servers).

The test cluster uses 3 servers with 48 task slots to leave 2 available cores

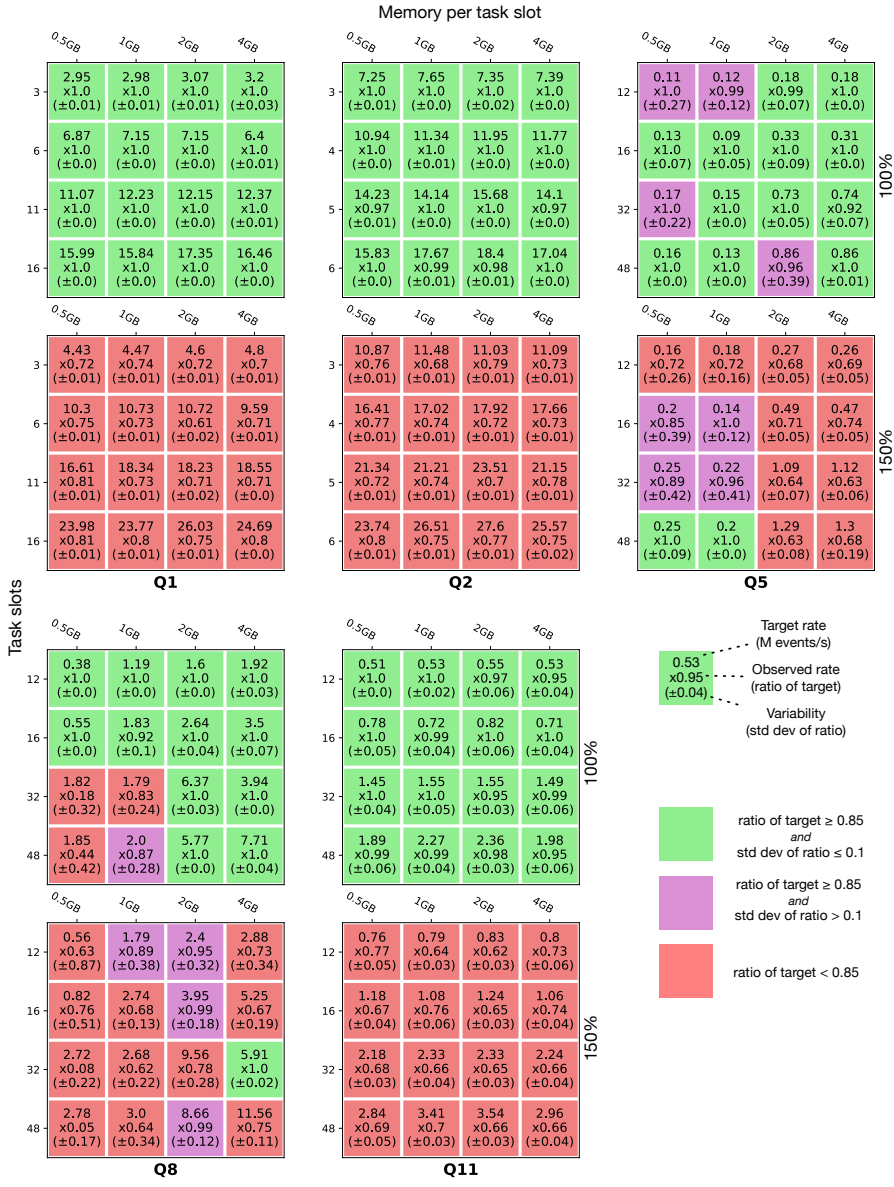


Figure 7.6: Accuracy of the MST estimation under various resource budgets and task profiles. The MST estimated by the CE for the configuration decided by the CO is replayed at 100% of the predicted rate for the upper row and at 150% of the predicted rate for the lower row. Colors represent runs that sustainably process the target rate (green), approach or meet the target with instabilities (purple), or fail (red).

per machine for system management. Similarly, we use up to 64 GB of RAM per machine, resulting in a maximum profile of 4 GB per task. As for the previous chapters, we also dedicate 3 nodes for Kubernetes, Prometheus, and the Flink Job Manager.

7.7.2 Experimentation results

RQ7.1 MST estimations. We first evaluate the accuracy of the CO and CE in identifying the MST of small-scale runs.

A run for the CE with queries **Q1**, **Q2**, and **Q11** uses a warmup of 120 seconds and measurements of 75 seconds (30 seconds of ramp-up, 30 seconds of observation and 15 seconds of cooldown). The cooldown uses a throughput of 200 events per second and per source, for a total of 6,400 events/s. One CE run performs 8 iterations for a total duration of 645 seconds. For complex queries **Q5** and **Q8**, our initial evaluations highlighted the need for longer measurements, due to the longer time necessary for our configurations to use RocksDB disk storage and not only the in-memory cache: We use a 450-second warm-up, a 900-second duration, and 7 iterations, for a total of 900 seconds (15 minutes); we also increase cooldown throughput to 12,800 events/s.

We query the CO for 16 combinations of resource budgets and profiles per query. We consider 3 to 16 tasks for **Q1**, 3 to 6 for **Q2**, and 12 to 48 for **Q5**, **Q8**, and **Q11** (the high achievable rate of **Q1** and **Q2** makes it unnecessary to test them with the full cluster). We use memory profiles of 0.5, 1, 2, and 4 GB.

We run each query with the 16 configurations returned by the CE, using their estimated MSTs as the target rate. We observe if each configuration supports the estimated rate sustainably for 10 minutes preceded by a warmup of 2 minutes. Figure 7.6 presents the results in its upper row. For each of the 16 configurations, a box presents first the target rate returned by the CE for this configuration (for instance, **Q8** with 12 TS of 2 GB has an MST of $1.92{\times}10^6$ events/s) and the observed rate, expressed as a ratio of this target. A high level of variation in the measurements, as presented by the standard deviation of this ratio across all measurements, is a sign that the job is close or past saturation.

Finally, we report in the lower row of Figure 7.6 results using a target rate of 150% of the estimated MST. A configuration passing this test means that its estimation was too conservative.

Simple queries **Q1** and **Q2** show little variations. All their tested configurations sustainably process the estimated MST at 100% but fail with 150%. Query **Q11** shows good results but slightly higher variations. This is due to the behavior of its stateful GroupBy (window) operator and resulting stragglers. For these three queries, a general linear scale behavior seems to apply

with increasing numbers of TS. The impact of memory is less clear and highlights the variability of the in-situ data that the RE receives as an input (and that the BO will address by repeating runs where the RMSE error is important).

For complex queries **Q5** and **Q8**, we observe a lower level of repeatability and lower scaling capabilities in terms of task slots. For **Q5**, our investigation is that the factor of instability lies mostly in the Join operator and its very uneven load across tasks, as we also highlight in the next experiment. For **Q8**, variability is caused by the presence of stragglers due to the windowed operators using a non-overlapping window size of 10 seconds, lower than our 5-second monitoring period, and leading to "sawtooth-like" load profiles. For both queries, results with 150% of the MST injected show that some estimations (in particular with 32 TS/4 GB and 48 TS/2 GB for **Q8**) can be too conservative. This behavior is not predictable from one run to the next, and we did not select a "good run" but the one performed in the whole series. This indicates again that the RE must accommodate variations in the data collected in particular for more complex queries.

For all memory profiles, **Q1**, **Q2**, and **Q11** achieve at worst 95% of the expected rate, and often 99-100%, this showing we manage to answer positively **RQ7.1**. For **Q5** and **Q8**, performance is much lower and very volatile with 0.5 and 1 GB. This is caused by state needs that cannot be satisfied with these feeble memory values, highlighting the impact of insufficient memory for efficient scaling. In the next sections, we focus on 2- and 4-GB profiles for these two queries.

RQ7.2 Configurations. We now evaluate the result of the CO optimization, zooming in the largest configuration for each query running at 100% of the estimated MST (i.e., bottom-right corners of matrices in Figure 7.6's first row). Figure 7.7 presents the distribution of the time series measurements over each 10-minute run. Most scaled-out operators reach their peak capacity at some point in time during the run, even if the median busyness is lower. This is the expected behavior: Operators must be provisioned to handle peak load. For Group By (window) and Joins, we observe a wide range of busyness levels, due to the skew and the stragglers resulting from upstream Group By (window) operators' uneven output rates. For the Join of **Q8**, additional tests with a forced lower parallelism (not shown) lead to saturations, back-pressure cascades, and instabilities; the measured busyness of 60% seems to be a practical maximum. Some stateless, scaled-out operators such as the first Filter of **Q5**, **Q8**, and **Q11** have relatively low busyness levels, due to an important level of back-pressure from downstream tasks. Overall, the CO can avoid underprovisioned operators with frequent busyness levels of 100%, thus answering favorably the **RQ7.2**.

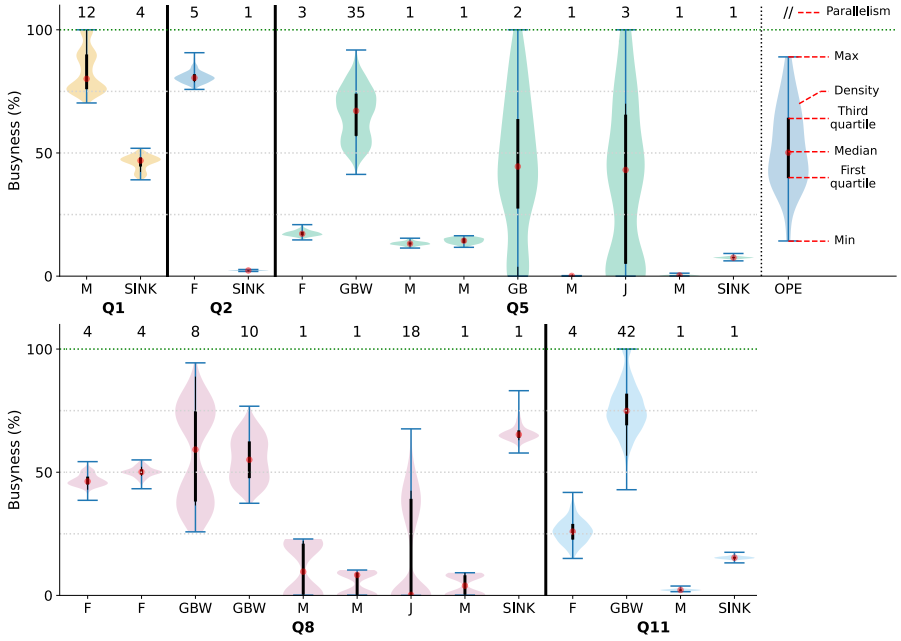


Figure 7.7: Distribution of task busyness levels measurements. The numbers above box plots are the parallelism of operators.

RQ7.3 Capacity planning: accuracy. Our final evaluation explores the capacity of StreamBed to build an effective and accurate capacity planning model **RQ7.3** and the cost of building this model **RQ7.4**.

Table 7.3 presents the uses of the models for large requested rates and Table 7.4 presents the results and building costs of the models. Queries **Q1**, **Q2**, and **Q11** are identified with a linear scaling model, while **Q5** and **Q8** respectively use the log and square root models. We can see that the models identify a minor impact of memory profiles for all queries (also clear in the values of coefficient a in the models). For **Q5**, the small variation (<1%) between the 2- and 4-GB profiles is a result of the uncertainty in the CO measurements. We can also observe that the effect of increasing rates differs between tasks, e.g., with coefficient b in the linear models of **Q1**, **Q2**, and **Q11**.

We validate the predictions using large-scale production runs. These runs use up to 85 nodes from the Gros cluster described in Section 4.3.3, using up to 69 nodes for Flink TMs (for **Q5**) and up to 36 Kafka nodes (for **Q11**). As our cluster does not allow supporting with Kafka the rates requested for **Q1** and **Q2** we inject for these queries data directly using sources paired with the Nexmark generator.

We adopt a similar strategy as for testing small-scale runs to verify that the predictions of StreamBed are neither over- or under-provisioned. A pre-

Query	Requested rate	Number of TS with profile			
		0.5 GB	1 GB	2 GB	4 GB
Q1	160×10^6	179	179	179	178
Q2	190×10^6	69	69	69	69
Q5	2.5×10^6	-	-	1069	1079
Q8	15×10^6	-	-	179	176
Q11	20×10^6	565	564	562	559

Table 7.3: Capacity planning results for the five queries.

diction is not under-provisioned if it sustainably supports the requested rate over time. A prediction is not over-provisioned if, when injecting a higher rate (we use 120% and 150% of the requested rate) the query shows signs of instability or insufficient capacity. High instability in the observed rate shows that we are above the MST and should be avoided. We also observe the *pending records* metrics, i.e., how many events generated by the fixed-rate source "pile up" at the source operator due to back pressure. While a small amount of buffering is normal in a setup close to full utilization, an ever-increasing pending records metric is a clear sign of an under-provisioned system.

Figure 7.8 presents the results of production-scale runs using the number of TS (cores) given in Table 7.3. We measure rate and pending records metrics after a ramp-up period of 5 minutes, for a duration of 30 minutes (**Q1**, **Q2**, and **Q11**), 90 minutes (**Q5**), and 120 minutes (**Q8**). For **Q1** and **Q2**, the actual rate is exactly the one requested at 100%, while 120% and 150% rates are not matched and lead to ever-increasing pending records.

For **Q5**, for readability, we present separately the 100% rate and the two other rates for the two considered profiles (2 GB and 4 GB). The 100% rate is sustained in both cases but with 2 GB per TS, we observe temporary instabilities. Resulting pending records are, however, later absorbed by the job indicating that the saturation point is not reached. With 4 GB, the query is very stable. In both cases, we observe chaotic behavior in terms of throughput for 120% and 150%, meaning the saturation point was reached.

For **Q8** we consider only the 100% and 150% cases for readability. Interestingly, we can see the impact of long-term instabilities that StreamBed can account for in the model (i.e., as these show earlier on small-scale jobs than on large-scale ones with high parallelisms for the concerned operators). Here, while the query is initially able to process 150% of the requested rate for a time, we observe that pending records gradually accumulate and provoke instabilities, making the query non-sustainable. The reason is that the working set of the query fits for a time dependent on the total memory before introducing congestion. We observe this effect both with 2- and 4-GB profiles.

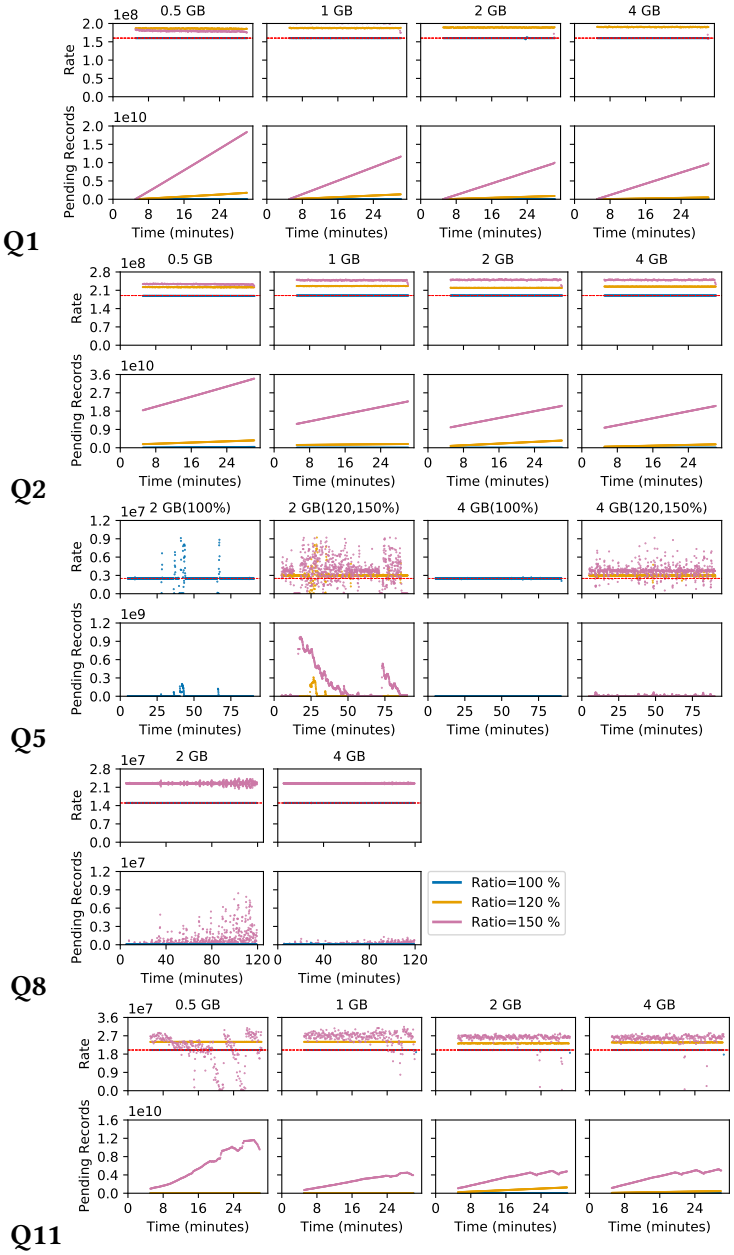


Figure 7.8: Large-scale production runs using capacity planning models (Table 7.3). For each query, we present the requested rate (thin red line) and the actual rate when injecting 100% (blue), 120% (yellow), and 150% (purple) of this rate. The second row present accumulated pending records. We present the 100% and (120%, 150%) cases separately for each of the two profiles of Q5 to improve readability.

	– Min/Max –		– Runs –				— Coefficients —		
Query	TS	RAM	#CO	#CE	Duration	Model	*a*	*b*	*c*
Q1	2/16	0.5/4	9	10	139 min.	lin	1.0	9.9E5	-7.6E5
Q2	2/6	0.5/4	14	20	248 min.	lin	7.5	3.0E6	-2.7E6
Q5	9/48	2/4	14	14	252 min.	log	-7.6E3	5.7E5	-1.2E6
Q8	9/32	2/4	13	13	234 min.	sqrt	2.6E3	1.4E6	-3.9E6
Q11	4/48	0.5/4	16	20	252 min.	lin	4.1	3.9E4	-2.1E5

Table 7.4: RE training results and costs, chosen model, and coefficients. #CO/#CE are number of calls to the two modules.

Overall, the predictions are accurate, with a slight over-provisioning in some cases as can be expected with our 110% over-provisioning factor, positively answering **RQ7.3**.

RQ7.4 Capacity planning: cost. Table 7.4 presents the results of a run of the RE for each of the five queries, with the number of calls to the CO and CE, the training time, and the resulting models. The training by the RE uses 9 (for **Q1**) to 16 (for **Q11**) calls to the CO. These calls result in 10 to 20 calls to the CE, as not all CO calls require evaluating a single-task configuration that is already in the cache. The complete duration of a CO call is about 24-minute long when the single-task configuration did not run, and about 13-minute when it did (for **Q5** and **Q8** a CO call takes 18 minutes always, as single-task configurations are tested first as part of the corners to bootstrap D). The CO optimization itself takes about 100 ms, while BO steps at the RE level have a negligible duration (<1 ms). In total, the total training duration ranges from 139 minutes for **Q1** to 252 minutes for **Q5**. These durations are reasonable considering the costs and scale of the target production deployments, answering favorably **RQ7.4**.

Summary. Our evaluation answers positively to our research questions. The collection of individual measurements by the CE is most often accurate but can yield noisy outputs when memory is insufficient for the considered query. Despite this uncertainty, our production-scale evaluation shows that StreamBed was able to correctly perform capacity planning, even for queries with sub-linear scaling profiles.

7.8 Summary

We presented StreamBed, a capacity planning system for stream processing allowing to derive scaling and configuration decisions for large-scale Flink jobs from a series of controlled, small-scale runs of a target query. StreamBed permits to estimate accurately the needed capacity to run sustainably very

large-scale queries without the need to execute them on actual production infrastructure or to build less accurate and expensive generic models. Moreover, it can identify cheap configurations able to run non-linear scaling queries that the state-of-the-art elastic scaler DS2 is not able to converge to.

StreamBed conducts a series of small-scale executions of a target query while varying resource configurations and input rates to build a model of the query's performance. It then leverages this model to predict the needed resources to run the query at large scale. Our evaluation on a set of representative Flink SQL queries from the Nexmark benchmark shows that StreamBed can estimate accurately the needed resources to run large-scale queries, while requiring only 3 nodes in the process.

Publications. The work presented in this chapter contributed to the following publication.

Streambed: Capacity Planning for Stream Processing. Guillaume Rosinosky, Donatien Schmitz, and Etienne Rivière. In Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems (DEBS), 2024. Lyon, France, 90-102.

The authors of this publication are listed in alphabetical order with Dr. Rosinosky as the first author due to his leading contribution to this work.

Conclusions

8

With the growing adoption of Distributed Stream Processing (DSP) systems in various domains, efficient resource management of such system has become a critical challenge. The high level of computation abstraction allowed by DSP systems simplifies the development and deployment of streaming applications by decoupling application logic from low-level resource management details. However, it often leads to suboptimal resource utilization resulting from the allocation of static resource amounts and naive task placement strategies. Inefficient resource management increases operational costs by requiring more worker nodes than necessary. The analysis of real-world workloads presented in Chapter 3 further highlights the prevalence of multi-query scenarios and the heterogeneous resource requirements of queries in production environments.

This thesis has explored two key aspects of resource management in DSP systems: fine-grained resource scaling and proactive resource planning. The former focuses on dynamically adjusting the resources allocated to individual operators based on their workload characteristics, while the latter aims to model and predict resource requirements of queries prior to their full-scale deployment. We discuss the main insights and contributions of this work in the following sections. Then, we outline several promising directions for future research in this area.

8.1 Summary and Insights

The goal of this thesis was to improve the resource efficiency of DSP systems. The first step towards this goal was to understand the resource requirements of streaming queries in real-world workloads. In Chapter 3, we analyzed a production DSP cluster and highlighted the prevalence of multi-query workloads and the heterogeneous resource requirements of queries. This analysis motivated the need for more sophisticated resource management strategies that can adapt to the dynamic and diverse nature of streaming workloads.

Building upon these insights, Chapter 5 introduced Justin, a hybrid CPU and memory auto-scaler for Apache Flink. Justin extends the state-of-the-art DS2 [Kal+18] auto-scaler by incorporating memory allocation into the scaling decisions. DS2 finds appropriate operator parallelism based on observed

busyness levels, but assumes a fixed memory allocation per task. Justin continuously collects metrics about the resource usage and storage performance of each operator's tasks (i.e., state access latency and cache hit rate), allowing it to adjust both CPU and memory allocations for each operator independently. We presented the Justin scaling policy that chooses between horizontal and vertical scaling actions based on their metrics. Our results showed that Justin can produce configurations using less CPU and memory for complex stateful queries while requiring the same or fewer reconfiguration steps than DS2. We note however, that for some queries with write-heavy workloads, Justin's memory adjustments has little impact on performance, as the state access pattern does not benefit from additional memory.

A key limitation of Justin is that it considers each query in isolation when making scaling decisions. This approach can lead to suboptimal resource utilization in multi-query scenarios, as it can lead to fragmented resource allocation across worker nodes without proper consideration of task placement. To address this limitation, Chapter 6 introduced Charon, an integrated scheduler for multi-query DSP workloads that combines fine-grained resource allocation with optimized task placement strategies. While Justin focuses on the memory requirement of operators at a fine granularity, Charon focuses on optimizing CPU allocation and task placement across multiple queries. Additionally, by identifying stateful and stateless operators, Charon can allocate memory only to the former, leading to significant memory savings. This approach results in the collocation of a greater number of tasks on each worker node, thereby reducing the overall number of nodes required to support a given workload. Charon employs a bin-packing-based task placement strategy that optimizes the allocation of tasks across workers, taking into account their specific CPU and memory requirements. Our evaluation demonstrates that Charon can significantly reduce the number of worker nodes required to sustain multi-query workloads compared to the default Flink scheduler, while maintaining the target throughput.

Finally, this thesis has highlighted the importance of proactive resource planning in DSP systems. It is crucial for practitioners to have tools and methodologies that can predict the resource requirements of streaming queries prior to their deployment. Accurate resource planning can help avoid over-provisioning and under-provisioning of resources, leading to cost savings and improved performance. Previous approaches to resource planning often rely on synthetic benchmarks or require large amounts of synthetic data, which may not accurately reflect the characteristics of real-world workloads. In Chapter 7, we introduced StreamBed, a framework for modeling and predicting the resource requirements of large-scale streaming queries, based on small-scale executions with real-world data. StreamBed builds a resource usage model allowing users to predict the resource requirements of a query

at a desired rate. To do so, StreamBed runs a serie of small-scale executions of the query with varying resource allocations and input rates, collecting detailed metrics about resource usage and performance. StreamBed guides the exploration of the configuration space using Bayesian Optimization, minimizing the number of necessary test runs. Using the collected metrics, StreamBed builds a resource usage model that captures the relationship between input rate, resource allocation, and performance. Our evaluation shows that StreamBed can accurately predict the resource requirements of complex streaming queries, enabling effective resource planning and management.

8.2 Opportunities for Future Work

In this section, we outline several promising directions for future research in the area of resource management for DSP systems. These directions build upon the insights and contributions of this thesis and aim to further enhance the efficiency and effectiveness of our proposed solutions.

Skew Awareness. Both Justin and Charon make implicit assumptions about the absence of skew, i.e., unbalanced key popularities leading to an imbalanced load between the different tasks of an operator. In practice, skew is a common phenomenon in streaming workloads and can significantly impact the performance and resource utilization of DSP systems. Future work could explore the integration of skew detection and mitigation techniques into the auto-scaling and resource allocation strategies proposed in this thesis. While previous skew mitigation techniques have focused primarily on load balancing through key re-partitioning [Nas+15; Nas+16], integrating resource-aware strategies could provide a more holistic approach to handling skew in DSP systems. For instance, Justin and Charon could be extended to allocate heterogeneous memory and CPU resources between tasks of the same operator based on their individual observed load. The mechanisms for fine-grain resource allocation introduced in this thesis could be leveraged to this end with small modifications to the exposed API and the Adaptive Scheduler of Flink.

Evaluating Latency. In this thesis, we focused on throughput as the primary performance metric for evaluating the effectiveness of our proposed resource management strategies. Another important performance metric in DSP systems is latency, which measures the time taken for a data tuple to be processed from ingestion to output. While the relationship between throughput and latency is linked, optimizing the throughput does not necessarily guarantee optimal latency performance. If we take the example of a system-under-test that is able to sustain a target throughput at its maximum capacity, the latency

might differ if the operators' buffers are full or nearly empty.

Numerous work have focused on improving latency in DSP systems, from intra-operator communication optimizations [Wu+18] to latency-aware elastic scaling [Hei+14; LJK15]. Future work could explore the integration of latency considerations into the auto-scaling and resource allocation strategies proposed in this thesis.

Task Placement Optimization Heuristics. Charon employs a bin-packing-based task placement strategy to optimize the allocation of tasks across workers. The algorithm used is known to be NP-hard. We have shown in the evaluation of the task placement strategy that the time taken to compute the placement of a large number of tasks can be significant, potentially impacting the responsiveness of the system to dynamic workload changes. In that case, exploring alternative optimization heuristics could lead to faster computation times while still achieving better task placement than the current approach. One possibility could be to pre-compute the different possible placements multiple tasks combinations and store them in a lookup table. During a reconfiguration, the placement algorithm could then query this table to quickly find a near-optimal placement without having to solve the bin-packing problem from scratch. A hybrid approach combining pre-computation to reduce the search space of the bin-packing algorithm could also be explored.

Integration of Additional Constraints in Charon. Charon currently focuses on optimizing CPU and memory allocation for tasks based on their busyness levels and statefulness. However, DSP systems often operate in environments with additional resource constraints, such as network bandwidth limitations and storage I/O bottlenecks. Systems such as CapSys [Wan+25] have highlighted the importance of considering network capacity in task placement decisions. Future work could extend Charon to incorporate these additional resource constraints into its resource allocation and task placement strategies.

We also note that, for instance, stateful operators doing write-heavy workloads may be bottlenecked by storage I/O rather than memory. In such cases, Charon could be extended to monitor storage I/O metrics and prevent co-locating I/O-intensive tasks on the same worker node, thereby avoiding contention and improving overall performance. This extension could be easily integrated in the bin-packing algorithm by adding a constraint that limits the total I/O load on each worker node, effectively balancing the total load across all resources.

Heterogeneous Clusters. In this thesis, we have assumed a homogeneous cluster where all worker nodes have the same resource capacities. However, in real-world deployments, clusters may consist of heterogeneous nodes with varying CPU, memory, and storage capabilities. This heterogeneousity can

arise from using different hardware configurations (i.e., different generations of CPUs, varying amounts of RAM, or different types of storage devices) or from deploying DSP systems on cloud platforms that offer a variety of instance types. Future work could explore the extension of Justin and Charon to handle heterogeneous clusters. This extension would involve adapting the resource allocation and task placement strategies to account for the varying capacities of different worker nodes. Additionally, the system could take into account the cost of different instance types when making resource allocation decisions, aiming to minimize the overall cost of running the DSP system while still meeting performance requirements.

Merging Justin and Charon. Justin and Charon address complementary aspects of resource management in DSP systems. Justin focuses on fine-grained memory and CPU auto-scaling for individual queries, while Charon emphasizes multi-query task placement and resource allocation. Merging the two approaches could lead to a more comprehensive resource management solution that combines the strengths of both.

However, we note that merging the two systems is not straightforward. The auto-scaling policy of Justin focuses only on memory allocation and does not consider CPU busyness levels, while Charon focuses solely on CPU busyness levels and does perform a simple binary memory auto-scaling. It is hard to know wether a decision made by the auto-scaler yielded better performance due to a memory allocation change, a CPU share change, or a better task placement. Additionally, fragmenting CPU and managed memory to a finer granularity could complicate the task placement problem, potentially leading to increased computation times for finding optimal placements, and eventually resulting in poorer consolidations. Future work could explore the integration of Justin's memory auto-scaling capabilities with Charon's CPU busyness-based resource allocation and task placement strategies.

Enhancing Auto Scalers with StreamBed Planning. Both Justin and Charon rely on reactive scaling strategies that adjust resource allocations based on observed metrics. While this approach allows the system to adapt to changing workloads, it still requires several reconfiguration steps to reach an optimal configuration. Integrating StreamBed's proactive resource planning capabilities into the auto-scaling strategies could enhance their effectiveness. By leveraging StreamBed's ability to predict resource requirements based on small-scale executions, Justin and Charon could make more informed scaling decisions from the outset. For instance, before deploying a new query or adjusting the resources of an existing one, the auto-scaler could use the model built by StreamBed to estimate the optimal resource allocation for the desired throughput. This integration could reduce the number of reconfiguration steps required to reach an optimal configuration, leading to improved

performance and resource efficiency.

Monitoring and Optimizing Energy Consumption. While this thesis has primarily focused on improving resource efficiency in DSP systems, an important aspect that warrants further investigation is the direct monitoring and optimization of energy consumption. In Chapter 6, we showed that better resource management and consolidation of tasks can lead to a significant decrease in the amount of TM used by a DSP cluster. We did not however measure the actual energy consumption of the cluster. Future work could explore the integration of energy consumption monitoring into DSP systems, for example using the Power API [Fie+24] or similar tools.

A promising direction would be to extend the auto-scaling and resource allocation strategies proposed in this thesis to explicitly optimize for energy efficiency. This could involve developing energy-aware scaling policies that consider the energy consumption of different resource configurations when making scaling decisions. This direction is particularly relevant in the context of heterogeneous clusters, where different instance types may have varying energy efficiency profiles. Systems such as NebulaStream [Zeu+19] allow the deployment of DSP applications on edge devices with limited energy resources.

Bibliography

[14a] *Apache Calcite.* https://calcite.apache.org/. 2014. (Visited on 09/08/2023).

[14b] *Apache Storm.* https://storm.apache.org/. 2014. (Visited on 09/08/2023).

[14c] *Grafana.* https://grafana.com/docs/. 2014.

[15a] *Apache Flink.* https://flink.apache.org/. 2015. (Visited on 09/08/2023).

[15b] *Apache Samza.* https://samza.apache.org/. 2015. (Visited on 09/08/2023).

[16] *Apache Beam.* https://beam.apache.org/. 2016. (Visited on 09/08/2023).

[24a] *Aiven.* 2024.

[24b] *Prometheus.* https://prometheus.io. The Linux Foundation, 2024.

[25a] *Apache Hadoop.* https://hadoop.apache.org/. 2025.

[25b] *Flink Fine-Grained Resource Management.* https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/finegrained_resource/. 2025.

[25c] *Flink K8S Operator.* https://github.com/apache/flink-kubernetes-operator. 2025. (Visited on 09/08/2023).

[25d] *Justin Artifact.* https://github.com/CloudLargeScale-UCLouvain/flink-justin. 2025.

[25e] *Justin Artifact.* https://github.com/CloudLargeScale-UCLouvain/flink-charon. 2025.

[25f] *Justin Artifact.* https://github.com/CloudLargeScale-UCLouvain/StreamBed. 2025.

[25g] *Kubernetes.* https://kubernetes.io/. 2025.

[25h] *RocksDB.* https://rocksdb.org/. 2025.

[AAM20] R. Alizadeh, J. K. Allen, and F. Mistree. “Managing computational complexity using surrogate models: a critical review”. In: *Research in Engineering Design* 31 (2020), pp. 275–298.

[Agn+23] P. Agnihotri, B. Koldehofe, C. Binnig, and M. Luthra. "Zero-Shot Cost Models for Parallel Stream Processing". In: *6th Intl. Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. aiDM. 2023.

[Agn+24] P. Agnihotri, B. Koldehofe, P. Stiegele, R. Heinrich, C. Binnig, and M. Luthra. "Zerotune: learned zero-shot cost models for parallelism tuning in stream processing". In: *ICDE*. 2024.

[Ahm23] S. Ahmadi. "Elastic data warehousing: Adapting to fluctuating workloads with cloud-native technologies". In: *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (Online)* 2.3 (2023), pp. 282–301.

[Al-+17] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. "Elasticity in cloud computing: state of the art and research challenges". In: *IEEE Transactions on services computing* 11.2 (2017), pp. 430–447.

[Ant21] C. Antonio. "Sequential model based optimization of partially defined functions under unknown constraints". In: *Journal of Global Optimization* 79.2 (2021), pp. 281–303.

[Apa23] Apache Foundation. *Zeppelin*. Version 0.10.1. May 1, 2023.

[Ark+21] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth. "Model-based stream processing auto-scaling in geo-distributed environments". In: *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2021, pp. 1–10.

[Arm+15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[Asy+] E. Asyabi, Y. Wang, J. Liagouris, V. Kalavri, and A. Bestavros. "A new benchmark harness for systematic and robust evaluation of streaming state stores". In: *EuroSys 2022*.

[Bal+17] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. "{TRIAD}: Creating synergies between memory, disk and log in log structured {Key-Value} stores". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 363–375.

[Bal+19] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. "{SILK}: Preventing latency spikes in {Log-Structured} merge {Key-Value} stores". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 753–766.

[Ban+23] H. Banirostam, T. Banirostam, M. M. Pedram, and A. M. Rahmani. "A model to detect the fraud of electronic payment card transactions based on stream processing in big data". In: *Journal of Signal Processing Systems* 95.12 (2023), pp. 1469–1484.

[BEI18] M. Belouch, S. El Hadaj, and M. Idhammad. "Performance evaluation of intrusion detection based on machine learning using Apache Spark". In: *Procedia Computer Science* 127 (2018), pp. 1–6.

[Bej24] B. Bejeck. *Kafka Streams in Action*. Simon and Schuster, 2024.

[Ben+09] J. Benesty, J. Chen, Y. Huang, and I. Cohen. "Pearson correlation coefficient". In: *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[Bor+08] D. Borthakur et al. "HDFS architecture guide". In: *Hadoop apache project* 53.1-13 (2008), p. 2.

[CAK12] Y. Chen, S. Alspaugh, and R. Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads". In: *Proceedings of the VLDB Endowment* 5.12 (2012).

[Cap+05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. "Grid'5000: A large scale and highly reconfigurable grid experimental testbed". In: *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* IEEE. 2005, 8–pp.

[Car+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4 (2015).

[Car+17] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. "State management in Apache Flink®: consistent stateful distributed stream processing". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.

[Car+18] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi. "Scarff: a scalable framework for streaming credit card fraud detection with spark". In: *Information fusion* 41 (2018), pp. 182–194.

[Car+20] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. "Beyond analytics: The evolution of stream processing systems". In: *ACM SIGMOD international conference on Management of data*. SIGMOD'20. 2020, pp. 2651–2658.

[Car+22] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo. "Runtime Adaptation of Data Stream Processing Systems: The State of the Art". In: *ACM Computing Surveys* 54.11s (2022), pp. 1–36.

[CL85] K. M. Chandy and L. Lamport. "Distributed snapshots: Determining global states of distributed systems". In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

[Clo21] H. Z. (Cloud). *Building an Enterprise-Level Real-Time Data Lake Based on Flink and Iceberg*. 2021.

[COI17] COIN-OR Foundation. *CBC*. Version 2.7.0. Aug. 21, 2017.

[COI23] COIN-OR Foundation. *PuLP*. Version 2.7.0. May 1, 2023.

[Cou+15] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza. "Elasticity in cloud computing: a survey". In: *annals of telecommunications-annales des télécommunications* 70.7 (2015), pp. 289–309.

[CYH20] Z. Chu, J. Yu, and A. Hamdull. "Maximum sustainable throughput evaluation using an adaptive method for stream processing platforms". In: *IEEE Access* 8 (2020), pp. 40977–40988.

[CZJ17] H. Chen, F. Zhang, and H. Jin. "Popularity-aware differentiated distributed stream processing on skewed streams". In: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE. 2017, pp. 1–10.

[DG08] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[Dim+13] M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, and T. Willhalm. "Memory system characterization of big data workloads". In: *2013 IEEE International Conference on Big Data*. BigData'13. 2013, pp. 15–22.

[Din+15] J. Ding, T. Z. Fu, R. T. Ma, M. Winslett, Y. Yang, Z. Zhang, and H. Chao. "Optimal operator state migration for elastic data stream processing". In: *arXiv preprint arXiv:1501.03619* (2015).

[DN19] J.-M. Dufour and J. Neves. "Finite-sample inference and nonstandard asymptotics with Monte Carlo tests and R". In: *Handbook of statistics*. Vol. 41. Elsevier, 2019, pp. 3–31.

[Doo22] A. W. (DoorDash). *Building Scalable Real Time Event Processing with Kafka and Flink*. 2022.

[FA16] G. Fóra and M. Andersson. *RBEA: Scalable Real-Time Analytics at King*. `https://www.ververica.com/blog/rbea-scalable-real-time-analytics-at-king`. Ververica Blog, May 2016. (Visited on 05/04/2016).

[Fan+17] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu. "Parallel stream processing against workload skewness and variance". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 2017, pp. 15–26.

[FBD20] O. Farhat, H. Bindra, and K. Daudjee. "Leaving stragglers at the window: Low-latency stream sampling with accuracy guarantees". In: *DEBS 2020*. 2020.

[FH19] M. Feurer and F. Hutter. "Hyperparameter optimization". In: *Automated machine learning: Methods, systems, challenges* (2019), pp. 3–33.

[Fie+24] G. Fieni, D. R. Acero, P. Rust, and R. Rouvoy. "PowerAPI: A Python framework for building software-defined power meters". In: *Journal of Open Source Software* 9.98 (2024), p. 6670. DOI: `10.21105/joss.06670`.

[Fli22] Flink. *Kafka Connector*. Dec. 13, 2022.

[Fli25] A. Flink. *Table API*. 2025.

[Flo+17] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. "Dhalion: self-regulating stream processing in heron". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1825–1836.

[FS21] Y. Fu and C. Soman. "Real-time data infrastructure at Uber". In: *SIGMOD 2021*. 2021, pp. 2503–2516.

[Gar98] S. R. Gardner. "Building the data warehouse". In: *Communications of the ACM* 41.9 (1998), pp. 52–60.

[GC15] G. S. Gurjar and S. Chhabria. "A review on concept evolution technique on data stream". In: *2015 International Conference on Pervasive Computing*. ICPC'15. IEEE. 2015, pp. 1–3.

[GRN20] L. Garcés-Erice, S. Rooney, and Z. A. Nagy. "Analysis of SQL Workloads on an Enterprise Datalake". In: *IEEE CLOUD*. 2020.

[Gu+22] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang. "Meces: latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 539–556.

[Gul+16] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join". In: *IEEE Trans. on Big Data* 7.2 (2016).

[Hal+16] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. "Managing Google's data lake: an overview of the Goods system." In: *IEEE Data Eng. Bull.* 39.3 (2016), pp. 5–14.

[Har+17] D. Harnie, M. Saey, A. E. Vapirev, J. K. Wegner, A. Gedich, M. Steijaert, H. Ceulemans, R. Wuyts, and W. De Meuter. "Scaling machine learning for target prediction in drug discovery using Apache Spark". In: *Future Generation Computer Systems* 67 (2017), pp. 409–417.

[HCM22] S. M. L. Hahn, I. Chereja, and O. Matei. "Evaluation of transformation tools in the context of NoSQL databases". In: *Intelligent Systems and Applications: Proceedings of the 2021 Intelligent Systems Conference (IntelliSys) Volume 2*. Springer. 2022, pp. 146–165.

[Hea+21] T. Head et al. *Scikit-optimize*. 2021.

[Hei+14] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. "Latency-aware elastic scaling for distributed data stream processing systems". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 2014, pp. 13–22.

[Hei+22] R. Heinrich, M. Luthra, H. Kornmayer, and C. Binnig. "Zero-shot cost models for distributed stream processing". In: *DEBS 2022*. 2022, pp. 85–90.

[HH21] S. Henning and W. Hasselbring. "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures". In: *Big Data Research* 25 (2021), p. 100209.

[HH22] S. Henning and W. Hasselbring. "A configurable method for benchmarking scalability of cloud-native applications". In: *Empirical Software Engineering* 27 (2022).

[Hig+20] A. S. Higginson, M. Dediu, O. Arsene, N. W. Paton, and S. M. Embury. "Database workload capacity planning using time series analysis and ML". In: *SIGMOD 2020*. 2020.

[Hof+19] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. "Megaphone: Latency-conscious state migration for distributed streaming dataflows". In: *Proceedings of the VLDB Endowment* 12.9 (2019), pp. 1002–1015.

[Hos+20] M. R. HoseinyFarahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari. "Q-Flink: A QoS-aware controller for Apache Flink". In: *IEEE/ACM CCGRID*. 2020.

[HSY01] W. W. Hsu, A. J. Smith, and H. C. Young. "Characteristics of production database workloads and the TPC benchmarks". In: *IBM Systems Journal* 40.3 (2001), pp. 781–802.

[Hua+15] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu. "Tencentrec: Real-time stream recommendation in practice". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 227–238.

[Ior+17] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. "Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling". In: *USENIX ATC 2017*. 2017, pp. 97–109.

[IPV17] S. Imai, S. Patterson, and C. A. Varela. "Maximum sustainable throughput prediction for data stream processing over public clouds". In: *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID. 2017, pp. 504–513.

[Kal+18] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows". In: *OSDI*. 2018.

[Kar+18] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th International Conference on Data Engineering*. ICDE'18. 2018, pp. 1507–1518.

[KC13] A. K. Karun and K. Chitharanjan. "A review on Hadoop—HDFS infrastructure extensions". In: *2013 IEEE conference on information & communication technologies*. 2013, pp. 132–137.

[Kho+15] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, P. P. Jayaraman, J. Hosking, and E. Abbasnejad. "Resource usage estimation of data stream processing workloads in datacenter clouds". In: *arXiv preprint arXiv:1501.07020* (2015).

[Kho+16] A. Khoshkbarforoushha, R. Ranjan, R. Gaire, E. Abbasnejad, L. Wang, and A. Y. Zomaya. "Distribution based workload modelling of continuous queries in clouds". In: *IEEE Transactions on Emerging Topics in Computing* 5.1 (2016), pp. 120–133.

[KNR+11] J. Kreps, N. Narkhede, J. Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proc. of the NetDB*. Vol. 11. 2011.

[Lan+15] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin. “A survey of open source tools for machine learning with big data in the Hadoop ecosystem”. In: *Journal of Big Data* 2.1 (2015), p. 24.

[LEB15] S. Lehrig, H. Eikerling, and S. Becker. “Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics”. In: *Proceedings of the 11th international ACM SIGSOFT conference on quality of software architectures*. 2015, pp. 83–92.

[LJK15] B. Lohrmann, P. Janacik, and O. Kao. “Elastic stream processing with latency guarantees”. In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE. 2015, pp. 399–410.

[LWW13] X. Lin, P. Wang, and B. Wu. “Log analysis in cloud computing environment with Hadoop and Spark”. In: *2013 5th IEEE International conference on broadband network & multimedia technology*. IEEE. 2013, pp. 273–276.

[Mei+25] Y. Mei, R. Xia, Z. Lan, K. Hu, L. Huang, P. Carbone, Y. Lei, V. Kalavri, H. Yin, and F. Wang. “Disaggregated State Management in Apache Flink® 2.0”. In: *Proc. VLDB Endow.* 18.12 (Sept. 2025), pp. 4846–4859. ISSN: 2150-8097. DOI: 10.14778/3750601.3750609.

[Men+16] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. “Mllib: Machine learning in apache spark”. In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7.

[Mil+16] Z. Milosevic, W. Chen, A. Berry, F. A. Rabhi, R. Buyya, R. Calheiros, and A. Dastjerdi. “Real-time analytics”. In: *Big Data: Principles and Paradigms* 2016 (2016), pp. 39–61.

[Mil97] B. Milanovic. “A simple way to calculate the Gini coefficient, and some implications”. In: *Economics Letters* 56.1 (1997), pp. 45–49.

[Mit02] M. Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2002), pp. 1094–1104.

[MK17] I. Mavridis and H. Karatza. “Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark”. In: *Journal of Systems and Software* 125 (2017), pp. 133–151.

[MK20] M. Masdari and A. Khoshnevis. “A survey and classification of the workload forecasting methods in cloud computing”. In: *Cluster Computing* 23.4 (2020), pp. 2399–2424.

[Nar+19] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. “Data lake management: challenges and opportunities”. In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1986–1989.

[Nas+15] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. “The power of both choices: Practical load balancing for distributed stream processing engines”. In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 137–148.

[Nas+16] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. “When two choices are not enough: Balancing at scale in distributed stream processing”. In: *IEEE ICDE*. 2016.

[Ng97] A. Y. Ng. “Preventing “overfitting” of cross-validation data”. In: *ICML 1997*. 1997.

[Nov24] E. Nova. *Digazu*. 2024.

[NY14] A. Nandakumar and N. Yambem. “A survey on data mining algorithms on apache hadoop platform”. In: *International Journal of Emerging Technology and Advanced Engineering* 4.1 (2014), pp. 563–565.

[Pau20] F. Paul. *Towards a Flink Autopilot in Ververica Platform*. `https://www.youtube.com/watch?v=5HvGo5vRxRA`. Oct. 2020. (Visited on 10/29/2020).

[PD25] L. Perron and F. Didier. *CP-SAT solver*. `https://developers.google.com/optimization/cp/cp_solver/`. Version v9.12. Google, Feb. 17, 2025.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011).

[Pfi+22] B. J. Pfister, W. S. Lickefett, J. Nitschke, S. Paul, M. K. Geldenhuys, D. Scheinert, K. Gontarska, and L. Thamsen. “Rafiki: Task-Level Capacity Planning in Distributed Stream Processing Systems”. In: *Euro-Par 2021 workshops*. 2022.

[Phi+92] S. Y. Philip, H.-U. Heiss, S. Lee, and M.-S. Chen. “On Workload Characterization of Relational Database Environments.” In: *IEEE Trans. Software Eng.* 18.4 (1992), pp. 347–355.

[PT07] T. Pitoura and P. Triantafillou. “Load distribution fairness in P2P data management systems”. In: *2007 IEEE 23rd International Conference on Data Engineering*. ICDE’07. 2007, pp. 396–405.

[Raj+17] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. "Pebblesdb: Building key-value stores using fragmented log-structured merge trees". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 497–514.

[Ram+17] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, et al. "Azure data lake store: a hyperscale distributed file service for big data analytics". In: *ACM SIGMOD international conference on Management of data*. SIGMOD'17. 2017, pp. 51–63.

[Rat+23] A. R. Rathinam, B. S. Vathani, A. Komathi, J. Lenin, B. Bharathi, and S. Murugan. "Advances and Predictions in Predictive Auto-Scaling and Maintenance Algorithms for Cloud Computing". In: *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*. IEEE. 2023, pp. 395–400.

[RCL23] G. Russo Russo, V. Cardellini, and F. Lo Presti. "Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources". In: *ACM Transactions on Autonomous and Adaptive Systems* (2023).

[Ren+13] Z. Ren, J. Wan, W. Shi, X. Xu, and M. Zhou. "Workload analysis, implications, and optimization on a production Hadoop cluster: A case study on TaoBao". In: *IEEE Transactions on Services Computing* 7.2 (2013), pp. 307–321.

[RM19] H. Röger and R. Mayer. "A comprehensive survey on parallelization and elasticity in stream processing". In: *ACM Computing Surveys* 52.2 (2019), pp. 1–37.

[Ros+21] G. Rosinosky, F. Schmidt, O. Bodunov, C. Fetzer, A. Martin, and E. Riviere. "Active replication for latency-sensitive stream processing in Apache Flink". In: *2021 40th International Symposium on Reliable Distributed Systems*. SRDS'21. IEEE. 2021, pp. 56–66.

[RP23] T. P. Raptis and A. Passarella. "A survey on networked data streaming with Apache Kafka". In: *IEEE Access* 11 (2023).

[RSR24a] G. Rosinosky, D. Schmitz, and E. Rivière. "StreamBed: capacity planning for stream processing". In: *18th ACM International Conference on Distributed and Event-Based Systems*. DEBS'24. 2024.

[RSR24b] G. Rosinosky, D. Schmitz, and E. Rivière. "Streambed: capacity planning for stream processing". In: *ACM DEBS*. 2024.

[Rus+21] G. R. Russo, V. Cardellini, G. Casale, and F. L. Presti. "MEAD: Model-based vertical auto-scaling for data stream processing". In: *CCGrid*. IEEE. 2021, pp. 314–323.

[RZ19] F. Ravat and Y. Zhao. "Data lakes: Trends and perspectives". In: *Database and Expert Systems Applications: 30th International Conference, DEXA 2019, Linz, Austria, August 26–29, 2019, Proceedings, Part I 30*. Springer. 2019, pp. 304–313.

[Sal+16] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang. "Big data analytics on Apache Spark". In: *International Journal of Data Science and Analytics* 1.3 (2016), pp. 145–164.

[San+20] B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri. "Plasma: programmable elasticity for stateful cloud computing applications". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[San23] A. Sanmukhani. *Python Prometheus API client*. Version 2.7.0. May 1, 2023.

[Sch+25] D. Schmitz, L. Berrewaerts, G. Rosinosky, S. Skhiri, and E. Rivière. "A Tale of Many Streams: Characterizing a Hybrid Batch-Stream Production Workload in Digazu, a Data Lake supported by Apache Kafka and Flink". In: *ACM DEBS*. 2025.

[SD15] J. G. Shanahan and L. Dai. "Large scale distributed data science using apache spark". In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 2323–2324.

[Shu23] S. Shukla. "Real-time monitoring and predictive analytics in healthcare: harnessing the power of data streaming". In: *International Journal of Computer Applications* 185.8 (2023), pp. 32–37.

[SLM18] S. Sahoo, C. Lampert, and G. Martius. "Learning equations for extrapolation and control". In: *ICML 2018*. 2018.

[Son+23] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun. "Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 301–314.

[Spo23] Spotify. *Kubernetes Operator for Apache Flink*. `https://github.com/spotify/flink-on-k8s-operator`. July 2023. (Visited on 07/19/2023).

[SRR25] D. Schmitz, G. Rosinosky, and E. Rivière. “Justin: Hybrid CPU/Memory Elastic Scaling for Distributed Stream Processing”. In: *IFIP DAIS*. 2025.

[SRR26] D. Schmitz, G. Rosinosky, and E. Rivière. “Charon: Fine-Grain Resource Allocation and Optimized Task Placement for Multi-Query Distributed Stream Processing”. In: *41st ACM/SIGAPP Symposium On Applied Computing*. SAC. 2026.

[SSL13] M. Smit, B. Simmons, and M. Litoiu. “Distributed, application-level monitoring for heterogeneous clouds using stream processing”. In: *Future Generation Computer Systems* 29.8 (2013), pp. 2103–2114.

[Str23] Strimzi. *Strimzi provides a way to run an Apache Kafka cluster on Kubernetes in various deployment configurations*. `https://strimzi.io`. July 2023. (Visited on 07/19/2023).

[SV15] K. Sharmila and S. Vethamanickam. “Survey on data mining algorithm and its application in healthcare sector using Hadoop platform”. In: *International Journal of Emerging Technology and Advanced Engineering* 5.1 (2015), pp. 567–571.

[SW22] C. Song and Y. Wang. “High-frequency wavefield extrapolation using the Fourier neural operator”. In: *Journal of Geophysics and Engineering* 19.2 (2022), pp. 269–282.

[TAW19] H. Tariq, H. Al-Sahaf, and I. Welch. “Modelling and prediction of resource utilization of hadoop clusters: A machine learning approach”. In: *Proceedings of the 12th IEEE/ACM international conference on utility and cloud computing*. 2019, pp. 93–100.

[Tem22] D. Templeton. *Twitter Sparrow tackles data storage challenges of scale*. Twitter engineering, June 2022. (Visited on 02/14/2023).

[THS17] M. T. Truong, A. Harwood, and R. O. Sinnott. “Predicting the Stability of Large-scale Distributed Stream Processing Systems on the Cloud.” In: *7th International Conference on Cloud Computing and Services Science*. CLOSER. 2017, pp. 575–582.

[Tön+14] R. Tönjes, P. Barnaghi, M. Ali, A. Mileo, M. Hauswirth, F. Ganz, S. Ganea, B. Kjærgaard, D. Kuemper, S. Nechifor, et al. “Real time iot stream processing and large-scale data analytics for smart city applications”. In: *poster session, European Conference on Networks and Communications*. sn. 2014, p. 10.

[Tru+18] T. M. Truong, A. Harwood, R. O. Sinnott, and S. Chen. “Performance analysis of large-scale distributed stream processing systems on the cloud”. In: *IEEE Cloud 2018*. 2018.

[Tuc+08] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. *Nexmark – A Benchmark for Queries over Data Streams*. Tech. rep. Oregon Health and Science University, 2008.

[Ver+23] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl. "Survey of window types for aggregation in stream processing systems". In: *The VLDB Journal* 32.5 (2023), pp. 985–1011.

[VPG23] E. Volnes, T. Plagemann, and V. Goebel. "To migrate or not to migrate: An analysis of operator migration in distributed stream processing". In: *IEEE Communications Surveys & Tutorials* 26.1 (2023), pp. 670–705.

[Wan+25] Y. Wang, L. Huang, Z. Wang, V. Kalavri, and I. Matta. "CAPSys: Contention-aware task placement for data stream processing". In: *EuroSys*. 2025.

[Whi12] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[Wol+08] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. "SODA: An optimizing scheduler for large-scale stream-based distributed computer systems". In: *ACM/IFIP/USENIX Middleware*. 2008.

[Wu+18] S. Wu, M. Liu, S. Ibrahim, H. Jin, L. Gu, F. Chen, and Z. Liu. "Turbostream: Towards low-latency data stream processing". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 983–993.

[Xin+06] Y. Xing, J.-H. Hwang, U. Cetintemel, and S. Zdonik. "Providing resiliency to load variations in distributed stream processing". In: *VLDB*. 2006.

[Zah+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster computing with working sets". In: *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*. 2010.

[Zah+13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized streams: Fault-tolerant streaming computation at scale". In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 2013, pp. 423–438.

[Zah+16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. "Apache Spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[Zeu+19] S. Zeuch, A. Chaudhary, B. Del Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl. "The NebulaStream platform: data and application management for the internet of things". In: *arXiv preprint arXiv:1910.07867* (2019).

[Zha+20] W. Zhang, N. Zheng, Q. Chen, Y. Yang, Z. Song, T. Ma, J. Leng, and M. Guo. "URSA: Precise capacity planning and fair scheduling based on low-level statistics for public clouds". In: *ICPP 2020*. 2020.

[Zon+21] Z. Zong, L. Wen, X. Hu, R. Han, C. Qian, and L. Lin. "Mespaconfig: Memory-sparing configuration auto-tuning for co-located in-memory cluster computing jobs". In: *IEEE Transactions on Services Computing* 15.5 (2021), pp. 2883–2896.

[Zou+22] B. Zou, T. Zhang, C. Zhu, L. Xiao, M. Zeng, and Z. Chen. "Alps: An Adaptive Load Partitioning Scaling Solution for Stream Processing System on Skewed Stream". In: *DEXA 2022*. 2022.

Appendices

Nexmark Queries

A

Listing A.1: Nexmark table creation.

```
CREATE TABLE person (
    id BIGINT,
    name VARCHAR,
    emailAddress VARCHAR,
    creditCard VARCHAR,
    city VARCHAR,
    state VARCHAR,
    dateTime TIMESTAMP(3),
    extra VARCHAR,
    PRIMARY KEY (id) NOT ENFORCED
);

CREATE TABLE auction (
    id BIGINT,
    itemName VARCHAR,
    description VARCHAR,
    initialBid DECIMAL(10, 2),
    reserve DECIMAL(10, 2),
    dateTime TIMESTAMP(3),
    expires TIMESTAMP(3),
    seller BIGINT,
    category VARCHAR,
    extra VARCHAR
);

CREATE TABLE bid (
    auction BIGINT,
    bidder BIGINT,
    price DECIMAL(10, 2),
    channel VARCHAR,
    url VARCHAR,
    dateTime TIMESTAMP(3),
    extra VARCHAR
);
```

Listing A.2: Nexmark queries.

```
-- Query 1: Convert bid prices from dollars to euros
SELECT
    auction,
    bidder,
    0.908 * price as price, -- convert dollar to euro
    `dateTime`,
    extra
FROM bid;

-- Query 2: Select bids for a specific auction
SELECT auction, price FROM bid WHERE MOD(auction, 123) = 0;

-- Query 3: Illustrates an incremental join (using per-key state and
    timer) and filter.
SELECT
    P.name, P.city, P.state, A.id
FROM
    auction AS A INNER JOIN person AS P on A.seller = P.id
WHERE
    A.category = 10 and (P.state = 'OR' OR P.state = 'ID' OR P.state =
        'CA');

-- Query 5: Illustrates sliding windows and combiners.
SELECT AuctionBids.auction, AuctionBids.num
 FROM (
   SELECT
     auction,
     count(*) AS num,
     window_start AS starttime,
     window_end AS endtime
     FROM TABLE(
             HOP(TABLE bid, DESCRIPTOR(`dateTime`), INTERVAL '2' SECOND
                 , INTERVAL '10' SECOND))
     GROUP BY auction, window_start, window_end
 ) AS AuctionBids
 JOIN (
   SELECT
     max(CountBids.num) AS maxn,
     CountBids.starttime,
     CountBids.endtime
   FROM (
     SELECT
       count(*) AS num,
       window_start AS starttime,
       window_end AS endtime
     FROM TABLE(
                HOP(TABLE bid, DESCRIPTOR(`dateTime`), INTERVAL '2'
                    SECOND, INTERVAL '10' SECOND))
     GROUP BY auction, window_start, window_end
     ) AS CountBids
   GROUP BY CountBids.starttime, CountBids.endtime
 ) AS MaxBids
 ON AuctionBids.starttime = MaxBids.starttime AND
    AuctionBids.endtime = MaxBids.endtime AND
    AuctionBids.num >= MaxBids.maxn;
```

```
-- Query 8 Illustrates a simple join.
SELECT P.id, P.name, P.starttime
FROM (
  SELECT id, name,
         window_start AS starttime,
         window_end AS endtime
  FROM TABLE(
            TUMBLE(TABLE person, DESCRIPTOR(`dateTime`), INTERVAL '10'
                SECOND))
  GROUP BY id, name, window_start, window_end
) P
JOIN (
  SELECT seller,
         window_start AS starttime,
         window_end AS endtime
  FROM TABLE(
         TUMBLE(TABLE auction, DESCRIPTOR(`dateTime`), INTERVAL '10'
             SECOND))
  GROUP BY seller, window_start, window_end
) A
ON P.id = A.seller AND P.starttime = A.starttime AND P.endtime = A.
    endtime;

--Query 11: Illustrates session windows.
SELECT
    B.bidder,
    count(*) as bid_count,
    SESSION_START(B.`dateTime`, INTERVAL '10' SECOND) as starttime,
    SESSION_END(B.`dateTime`, INTERVAL '10' SECOND) as endtime
FROM bid B
GROUP BY B.bidder, SESSION(B.`dateTime`, INTERVAL '10' SECOND);
```