On Optimising Rescaling Operations of Streaming Applications

Paul Osagie Omoregbee

School of Computing
Newcastle University

In Partial Fulfilment of the Requirements for the Degree of
Doctor of Philosophy

July 2023

Declaration

I hereby declare that this thesis is entirely my work, conducted under the guidance of my supervisors. I affirm that all sources used have been duly acknowledged and referenced in accordance with the established academic standards. No part of this thesis has been submitted for any other academic qualification. I take full responsibility for the content presented and affirm that it is an original contribution to auto-scaling operations of streaming applications.

Paul Osagie Omoregbee
July 2023

Acknowledgements

I am deeply grateful to my exceptional wife Mrs. Precious Omoregbee, whose unwavering support, love, and sacrifices have been the cornerstone of my success. Her unyielding belief in my abilities and her unwavering encouragement has provided me with the strength and motivation to overcome the challenges faced during this research journey.

I express profound gratitude to my supervisory team, Dr. Matthew Forshaw and Dr. Nigel Thomas, for their invaluable guidance, expertise, and constructive feedback. Their dedication and commitment to my research have been instrumental in shaping the direction of this thesis. Dr. Nigel Thomas was very supportive during some challenging phases of my research journey. His guidance and assistance were instrumental in helping me overcome the obstacles I encountered and played a crucial role in helping me navigate through what proved to be a difficult period.

I extend my heartfelt thanks to my examiners, Professor Stephen Jarvis and Dr. Paul Ezhilchelvan, for their insightful input and valuable feedback. Their expertise and willingness to evaluate my work have contributed to the rigor and quality of this thesis.

I am also grateful to Newcastle University for awarding me the Newcastle University Overseas Research Scholarship (NUORS). This prestigious scholarship has provided me with the financial support necessary to pursue my research aspirations, and I am truly honoured to have been selected.

Lastly, I would like to express my heartfelt appreciation to my parents and siblings for their unwavering support, prayers and encouragement throughout my academic journey. Their constant belief in my abilities has been a driving force behind my accomplishments.

In conclusion, I am sincerely thankful to all those who have contributed directly or indirectly to the successful completion of this thesis. Without your support, guidance, and love, this endeavour would not have been possible.

Abstract

The primary objective of an auto-scaler is to allocate resources to meet demand, while adapting to varying workloads in a distributed streaming systems. This is done in order to achieve low latency and high throughput. However, achieving optimal performance in the auto-scaling of stream processing applications can be challenging due to various factors such as workload patterns, and application state sizes. While most auto-scaling systems assume that application state sizes and offered load remain unchanged during scaling intervals, in rapidly changing workload environments, long scaling durations may exacerbate suboptimal parallelism decisions as additional state may have been accrued during a rescaling interval, thereby causing multiple rescalings. The execution of this recurring task may negatively impact the system's performance.

Similarly, accurately measuring processing capacity is crucial for optimal performance in streaming applications. This helps ensure that the system can handle the application's data volume and processing requirements without introducing bottlenecks or increasing latency. Furthermore, relying on conventional techniques, such as using offered load as a proxy for application state size, can be misleading, especially in window-based applications where both measures may not align perfectly. The stateful nature of individual window configuration creates a trade-off between memory usage and processing throughput. This can result in a false positive, causing a premature scaling decision, and leading to reduced throughput.

We address these challenges by empirically evaluating the interplay between application state size, end-to-end checkpoint duration, and the duration of scaling procedures. Large checkpointing intervals could lead to longer recovery duration due to the accumulation of more state, while short intervals can lead to high processing overhead due to the frequency and potential checkpointing overlap, a delay in a preceding checkpoint influenced by state size.

Based on our findings, we develop predictive models to provide future auto-scalers with intelligence to inform scaling decisions. Next, we conduct empirical evaluations to assess the relationship between operator throughput and state size, showcasing the relationship between the state size and the operator's throughput of a streaming application.

We explore the impact of window selectivity, an approach where the length of the window and the sliding period can impact the effectiveness and efficiency of streaming applications. In stateful operations, offered load is accumulated in a buffer until it reaches the end of the window, at which point the buffer is subsequently processed. Given that these buffers are retained in memory, a surge in offered load or larger windows may result in a rapid expansion of buffer size, thereby causing a spike in memory consumption. We therefore demonstrate first, how growing application state sizes can spuriously decrease operator throughput and trigger premature scale-

up or scaling-down decisions and secondly, the impact of windowing on instantaneous state size.

The major contributions of the thesis are as follows:

1. We investigate the task allocation mechanism employed by Flink. The present study demonstrates the non-uniform distribution of data and the disparity in throughput among individual operators.

2. We show that rescaling duration is a critical factor in auto-scaling a streaming application, demonstrating that with the accumulation of more state, the time to rescale also increases. This can lead to multiple rescaling of an application and falling short of resources.

3. We develop and evaluate a predictive model to forecast the rescaling duration of a streaming application based on the state size and end-to-end duration. This model will provide a scaling controller with knowledge of the estimated rescaling duration, enabling it to make informed scaling decisions.

4. We show that offered load is an inadequate proxy for state size, demonstrating how the state size metric varies and falls behind the arrival rate. We show the impact of growing state size on operator processing capacity. We also show the impact of windowing on the instantaneous state size.

Table of contents

List of figures **xiii**

List of tables **xv**

1 Introduction **3**

1.1 Overview 3
1.2 Research Problem 4
1.3 Thesis Contributions 5
1.4 Thesis Structure 6
1.5 Related Publications 8

2 Background **9**

2.1 Fundamentals of Data Stream Processing 9
2.1.1 Streaming Processing General System Model 10
2.1.2 Apache Flink Overview 11
2.1.3 Flink Architecture and Process Model 13
2.1.4 Flink Execution Mode 17
2.2 State Management 18
2.2.1 State Operations 19
2.2.2 Load Balancing and Scalability 21
2.2.3 Stateful and Stateless Computation 23
2.2.4 Windowing 24
2.2.5 Flink State Backends 25
2.3 Checkpointing and Restore 26
2.3.1 Consistent Checkpointing and Recovery 27
2.3.2 Impact of Checkpointing Interval on Streaming Applications 27
2.4 Streaming Application Scalability 28
2.5 Adaptation Technique for Stream Processing Systems 31

3 Stream Workload Parallelisation and Elasticity **35**

3.1 Introduction 35
3.2 Parallel Stream Processing 36
3.3 Properties of Parallel Stream Processing Systems 37
3.3.1 Sub-Stream Processing 37

3.3.2 Infrastructure Model . . . 38
3.4 Operator Parallelisation Methods and Limitations . . . 38
3.4.1 Data Parallelisation . . . 38
3.4.2 Task Parallelisation . . . 42
3.4.3 Limitations of Operator Parallelisation Methods. . . . 42
3.5 Flink FlatMap Operator Distribution Mechanism . . . 43
3.6 System Design . . . 45
3.6.1 Experimental Setup . . . 47
3.7 Summary of Experimental Results . . . 50
3.8 Conclusion . . . 54
3.8.1 Future Work . . . 55

4 Modelling of Time and Resource Requirements to Perform Rescaling 57
4.1 Introduction . . . 58
4.2 Influencing Metrics for Scaling a Streaming Application . . . 60
4.3 System Design . . . 61
4.3.1 Nexmark Workload . . . 66
4.4 Impact of Long Checkpointing Duration in Streaming Applications . . . 67
4.5 Predictive Modelling . . . 68
4.5.1 Linear Regression Analysis for Predictive Modelling . . . 69
4.5.2 Resampling . . . 71
4.6 Summary of Experimental Results . . . 75
4.7 Conclusion . . . 77
4.7.1 Experimental Challenges . . . 77
4.7.2 Future Work . . . 78

5 Impact of State Size on Streaming Operator Throughput 79
5.1 Introduction . . . 79
5.2 Evaluating The Impact of System Resource On Operator Throughput . . . 81
5.3 Application State Size Dynamics . . . 83
5.3.1 Managed State . . . 84
5.4 Evaluating The Impact of Window Selectivity Across Various Deployments . . . 85
5.5 System Design . . . 85
5.5.1 Nexmark Workload . . . 87
5.5.2 Experimental Setup . . . 89
5.6 Summary of Experimental Results . . . 100
5.7 Conclusion . . . 103
5.7.1 Experimental Challenges . . . 104
5.7.2 Future Work . . . 104

6 Conclusion 107
6.1 Thesis Summary . . . 107
6.2 Limitations . . . 108

6.3 Future Research Direction 109
6.3.1 Optimising Streaming Engine's Processing and Data Distribution Mechanism 109
6.3.2 Declining Processing Rates and the Interplay with Application State . . 110
6.3.3 Windowing Selectivity Expansion to Other Window Types 110

References **113**

List of figures

2.1 A General Stream Processing Model 11
2.2 Apache Flink Software Stack [15] 12
2.3 Flink Job Execution Process [33] 13
2.4 Flink Architecture [12] 16
2.5 State Management's Many Facets [116] 19
2.6 Balancing Load with State Awareness 22
2.7 An Example of State Computation 23

3.1 A stream topology consisting of one Sources operator, three FlatMap operators and one Sink operator. This illustrates the potential problem of workload imbalance within operators in real distributed stream processing engine [35] . . 36
3.2 Data Parallelisation Fundamental Architecture [104] 39
3.3 Key-based Splitting [104] 40
3.4 Window-based splitting is the process of dividing the input stream into windows consisting of successive data items. Every instance of the operator is responsible for processing a subset of all the windows. [104] 41
3.5 Pane-based Splitting [104] 42
3.6 Task Parallelisation Fundamental Architecture [104] 42
3.7 Experimental System Architecture (Chapter 3) 47
3.8 Measuring True and Observed processing Rate (Experiment 1) 50
3.9 Percentage difference between the True and Observed processing rate (Experiment 1) 51
3.10 Summed Value of Observed Processing Interpolation Records Over Time (Experiment 2) 52
3.11 Observed Processing Interpolated Records Over Time (Experiment 2) 52
3.12 Three Instance FlatMap Operator Processing Capacity (Experiment 3) 53

4.1 Rescaling Duration Increasing Over Larger State Sizes 59
4.2 Experimental System Architecture (Chapter 4) 62
4.3 NexMark Workload Stream Processing Directed Acyclic Graph (DAG) 67
4.4 Flink Checkpoint Duration Increasing Over Larger State Sizes. 68
4.5 Exploring the Relationship Between Variables. 69
4.7 Comparison of Model Indices. 74
4.8 Cross-Validation Resampling Method Architecture [66] 75

5.1 State Growth Despite a Declining Operator Throughput 80
5.2 Visualisation of Observed Processing Rate for Different Heap Memory Sizes of 3072 and 4144 MB. 83
5.3 Experimental System Architecture (Chapter 5) 86
5.4 Three sliding windows, with overlapping elements across windows. 88
5.5 Tumbling windows, with non-overlapping elements across windows 89
5.6 Evolution of True and Observed Processing Rates for Nexmark Query 5. 91
5.7 For Nexmark Query 5, the Evolution of Observed Processing Rates and the Interplay with Application State. 92
5.8 For Nexmark Query 3, the Evolution of Observed Processing Rates for Filter and Incremental Join operators, and the Interplay with Application State. 93
5.9 Visualisation of state size (number of bids) for a window operator of length 30 second, for different slide lengths of 5, 10 and 20 seconds. 95
5.10 Visualisation of state size (number of bids) for a window operator of length 30 second, for different slide lengths of 5, 10 and 20 seconds. 96
5.11 Empirical Cumulative Distribution Function (ECDF) plot of instantaneous state size for a sliding window operator of 30 second, with slide lengths of 5, 10 and 20 seconds. 96
5.12 Visualisation of state size (number of bids) for a window operator of slide lengths 5 second, for different window sizes of 20, 40 and 60 seconds. 98
5.13 For Nexmark Query 5, the evolution of true processing rates for different window sizes of 20, 40 and 60 seconds. 99
5.14 For Nexmark Query 5, the evolution of observed processing rates for different window sizes of 20, 40 and 60 seconds. 99
5.15 ECDF plot of the evolution of observed processing rates for a sliding window operator of 30 second, with slide lengths of 5, 10 and 20 seconds. 100
5.16 ECDF plot of the evolution of observed processing rates for a slide length of 5 second, for different window sizes of 20, 40 and 60 seconds 101
5.17 Top 5 CPU consuming processes during experiment 102
5.18 Top 5 Memory consuming processes during experiment 102

List of tables

2.1 A characterisation of methods for stateful computation. 24

3.1 Standalone hardware configuration (Chapter 3) 47
3.2 Software configuration (Chapter 3) 48
3.3 Apache Flink Configuration 48
3.4 Scaling Operator Parallelism Experiment 50

4.1 Summary of auto-scaling policies in distributed dataflow systems [58] 61
4.2 Standalone hardware configuration (Chapter 4) 62
4.3 Software Configuration (Chapter 4) 62
4.4 Apache Flink Configuration 63
4.5 First experiment configuration parameters 64
4.6 Second experiment configuration parameters 65
4.7 NEXMark Query 5 Configuration 67
4.8 Models Statistical Properties 72
4.9 Comparison of Model Performance Indices. 73
4.10 Performance Statistics Metrics for Each Model 75
4.11 Linear Models Applied to Test Data 76

5.1 Flink Job Manager and Task Manager Memory Configuration Experiment . . . 83
5.2 Standalone Hardware Configuration (Chapter 5) 86
5.3 Software Configuration (Chapter 5) 86
5.4 First experiment configuration parameters 90
5.5 Second experiment configuration parameters 92
5.6 Third experiment configuration parameters 94
5.7 Forth experiment configuration parameters 97

List of Acronyms

IoT Internet of Things

DSP Data Stream Processing

SPEs Stream Processing Engines

QoS quality of service

SLA Service Level Agreement

SLOs Service Level Objectives

ETL Extract-Transform-Load

DAG Directed Acyclic Graph

LRQ Long-Running Queries

LSM Log-Structured Merge

JNI ava Native Interface

API Application Programming Interface

ACID Atomicity, Consistency, Isolation, Durability

SASO Stability, Accuracy, Short Settling Time, and Overshooting

MST Maximum Sustainable Throughput

DS2 Data Stream 2

CEP Complex Event Processing

DSMS Data Stream Management Systems

VM Virtual Machines

NA Not Available

R^2 R-squared

R^2 **(adj)** Adjusted R-squared

RMSE Root Mean Square Error

AIC Akaike Information Criterion

BIC Bayesian Information Criterion

VIF Variance Inflation Factor

CV Cross-Validation

RBEA Rule-Based Event Aggregator

JVM Java Virtual Machine

ECDF Empirical Cumulative Distribution Function

JVM Java Virtual Machine

JVM Java Virtual Machine

DAG Directed Acyclic Graph

Chapter 1

Introduction

1.1 Overview

The emergence of Internet of Things (IoT) the has led to a growing interest in data stream processing, due to escalating demand for processing of diverse and extensive data streams. The proliferation of IoT devices and end users has resulted in a significant surge in the volume of data generated. According to a projection, it is anticipated that by 2025, a significant proportion of enterprise data, approximately 75%, will be generated remotely from the data centres [122]. These data are generated frequently and necessitate immediate processing following their production [137]. Data Stream Processing (DSP) frameworks are frequently employed as an intermediary software layer for the purpose of handling these data streams [104].

Data stream processing systems were developed to handle unbounded incoming data streams in a continuous manner while maintaining low end-to-end latency [6]. Several Stream Processing Engines (SPEs) have been suggested, such as Apache Storm [118], Apache Spark [139], and Apache Flink [15]. DSP applications are typically structured as directed acyclic graphs comprising of data transformations, also known as operators. These operators are interconnected by data streams, and collectively constitute a data processing workflow. Typically, the inputs consist of tuples within a data stream that are processed by each operator through a predetermined computation, resulting in the creation of new tuples that are subsequently transmitted to the succeeding operators.

The fundamental assumption of streaming data is that the potential value lies in the freshness of the data. These data are generated at high velocity from multiple sources and processed with low latency in real-time. Stream processing ensures that data are analyzed real-time immediately after they arrive at the stream, contrary to batch processing where the data is first persisted before they are analyzed [108].

In order to ensure a desirable level of service quality, DSP employ diverse methodologies to parallelise the implementation of their operators [110]. According to research, the majority of instances indicate that duplicating a stream processing operator results in an augmentation of its processing capability and an enhancement of its quality of service (QoS) [48]. As such, a multitude of techniques aimed at enhancing the performance of DSP systems centres around the implementation of parallelisation and elasticity strategies [104].

Stream processing parallelisation and elasticity enable streaming engines to support high quality of service in processing a large amount of data while ensuring high throughput and low latency [58]. To maximize stream processing engine throughput and improve the utilization rate of its computation resources, workloads are commonly partitioned and processed concurrently by multiple logical operator instances. The source operator being aware of the local graph sends data in tuples for processing based on a global partitioning strategy [50]. Tuples with the same key are received and processed by the same stateful operator. Stateful operator refers to an operator that has a memory space to store intermediate results called states. For example, a state can be used to record the tuples or counts of words in a sliding window. When a key is reassigned to a different operator instance, its state is usually migrated as well to ensure the correctness of the computation outcome. This is because of the binding between a key and state [35].

At a time when data is growing at an unprecedented rate, consumers are demanding not only connectivity and access, but also improved service quality and overall experience. Thus, operators view real-time analytics as a crucial enabler for accelerating the creation, delivery, and monetisation of service bundles and providing consumers with a unique network experience [115]. In this context, it is essential to develop new scalable, dynamic, and responsive data analytics solutions.

Auto-scaling refers to the automatic adjustment of computational resources based on the demand or workload of the streaming system. It allows the system to dynamically allocate or deallocate resources, such as servers, processing units, operators, data injection rate, or storage, in order to efficiently handle fluctuations in data volume or processing requirements. The goal of auto-scaling is to optimize performance, ensure smooth data streaming, and minimize costs by scaling up or down the resources as needed, without requiring manual intervention.

Auto-scaling can be accomplished via either vertical or horizontal scaling techniques. Vertical scaling is a method of enhancing the capacity of individual resources, whereas horizontal scaling is a technique of increasing capacity by incorporating additional instances of resources. Furthermore, while resource scaling both at the infrastructure level and operator scaling could be a solution for long-term workload, this solution might be too expensive for the short-term fluctuation scenario because of its temporal and random nature [101]. Hence, in order to optimise the advantages of auto-scaling, a comprehensive and meticulous assessment of all the relevant metrics and components of the streaming pipeline is imperative.

1.2 Research Problem

Maximising the benefits of auto-scaling in stream processing applications is difficult due to challenges associated with precisely estimating resource usage or accurately gauging the processing capacity of the system in the face of significant variability in client workload patterns. In this thesis, we investigate the following research problems:

Flink's Task distribution mechanism: How does Flink distribute data across multiple operators' instances in a stream processing pipeline? What is the impact of system utilization

on the level of imbalance between different operators' instances in a distributed stream processing pipeline? We address this in Chapter 3.

Impact of long scaling time: What is the effect of time and resource requirements to perform a 'rescaling' of an application during runtime? What impact does long scaling time have on streaming applications? What metrics are critical to the scaling procedure of a streaming system and how can we measure the correlation between these metrics and the scaling duration. Most auto-scaling systems do not consider the impact of scaling time and often assume that application state sizes and offered load remain unchanged during the scaling interval [51, 84, 107, 105, 58]. Chapter 4 tackles these challenges.

Impact of state size on operators throughput: How can we measure the state size of a streaming application? Is there a relationship between the application state size and offered load? Does application state size have an impact on the processing capacity of streaming operators? Most auto-scaling systems do not consider the impact of the application state and rely on offered load as a proxy for application state size. This is addressed in Chapter 5.

Windowing selectivity: What is the impact of window selectivity on instantaneous state size? Does the window size or sliding period length have any impact on streaming applications? How do we measure how a streaming operator responds to a decreasing offered load in a windowed setting with different window sizes and length configurations? This is addressed in Chapter 5.

1.3 Thesis Contributions

The research presented in this thesis offers several significant contributions.

i. We investigate the task allocation mechanism employed by Flink. The present study demonstrates the non-uniform distribution of data and the disparity in throughput among individual operators.

   I chose to exclusively utilize Apache Flink as the streaming engine for my research due to its unique blend of openness, versatility, and compatibility. Being an open-source software, Flink allows for collaborative exploration and development within a diverse community, enabling cost-effective and resourceful research. However, the level of abstraction of this research, makes adaptation to other streaming platforms relatively seamless and other streaming engines may offer similar functionalities, Flink's robustness, ease of integration, and supportive community made it the optimal choice for conducting my research experiment.

ii. We show that rescaling duration is a critical factor in auto-scaling a streaming application, demonstrating that with the accumulation of more state, the time to rescale also increases. This can lead to multiple rescaling of an application and falling short of resources.

iii. We develop and evaluate a predictive model to forecast the rescaling duration of a streaming application based on the state size and end-to-end duration. This model will provide a scaling controller with knowledge of the estimated rescaling duration, enabling it to make informed scaling decisions.

iv. We show that offered load is an inadequate proxy for state size, demonstrating how the state size metric varies and falls behind the arrival rate. We show the impact of growing state size on operator processing capacity. We also show the impact of windowing on the instantaneous state size.

1.4 Thesis Structure

Chapter 1 describes the fundamental motivations that propelled the research and highlights the principal discoveries and progressions from the investigation. Furthermore, a comprehensive record of the peer-reviewed publications produced in the period of this PhD is presented.

Chapter 2 provides a concise overview of the contextual background and technological tools employed in developing the solutions presented in this thesis.

Chapter 3 describes the task distribution mechanism of Flink over stateful operators in a distributed stream processing pipeline. We show that increasing the parallelism in a FlatMap operator does not necessarily result in a corresponding increase in processing capacity. This outcome is contingent upon various factors, including but not limited to the availability of system resources, data distribution, size of application state, bottlenecks in upstream and downstream processes, and the degree of parallelism in the upstream and downstream operators. Furthermore, in this chapter we note Flink's non-uniform distribution of data in a topology with multiple FlatMap instances as well as the disparity in throughput among individual operators.

Chapter 4 empirically evaluates application state size and end-to-end checkpoint duration and identify their correlation with the duration of scaling procedures. We evaluate the impact of long rescaling duration during an auto-scaling interval which could lead to multiple rescaling of an application when the state size growth of the application is larger and unpredictable. This repetitive task could harm the system performance. We believe that a proactive approach will be to use machine learning algorithms to develop a workload prediction module that can forecast workload characteristics and rescaling duration. Finally, we develop predictive models to provide future autoscalers with further intelligence to inform scaling decisions.

Chapter 5 undertakes empirical assessments to examine the correlation between the state size and the operator's processing capacity in a distributed streaming application. Our findings demonstrate the disparity in the rates of convergence between small and larger window configurations. The results of our study emphasise the significance of taking into

account the size of the application state when making decisions regarding auto-scaling. We also show that in a window-based applications, relying on offered load as a proxy for the application's state size can be misleading.

Chapter 6 provides a summary of the findings presented in this thesis and outlines potential directions for future research in this field.

1.5 Related Publications

Throughout the duration of my doctoral studies, I have made contributions to the following peer-reviewed publications.

[90] Omoregbee, P. and Forshaw, M. (2022). Performability requirements in making a rescaling decision for streaming applications. In Computer Performance Engineering: 18th European Workshop, EPEW 2022, Santa Pola, Spain, September 21–23, 2022, Proceedings, pages 133–147. Springer.

This paper considers the impact of long rescaling duration during an auto-scaling interval which could lead to multiple rescaling of an application when the state size growth of the application is larger and unpredictable. This paper serves as the foundation of Chapter 4 of this thesis.

[92] Omoregbee, P., Nigel, T. and Forshaw, M. (2023). A State-Size inclusive approach to autoscaling stream processing applications. In Computer Performance Engineering: 19th European Workshop, EPEW 2023, Florence, Italy, 20-23 June, 2023.

This paper makes the following contributions: first, we demonstrate that offered load is an inadequate proxy for state size; second, we model the impact of growing state size on operator processing capacity. Finally, we show the importance of carefully selecting an appropriate window size and criteria to balance the selectivity and accuracy of the window operator, taking into consideration the performance requirements of the streaming application. This paper serves as the foundation of Chapter 5 of this thesis.

[91] Omoregbee, P., Forshaw, M. and Nigel, T. (2023). Analyzing Performance Effects of Window Size on Streaming Operator Throughput. In Computer Performance Engineering: 39th Annual UK Performance Engineering Workshop, UKPEW 2023, Birmingham, United Kingdom, 7-8 August, 2023.

This paper presents two key contributions. Firstly, we analyze and model the influence of an increased collection resulting from a larger state size. Secondly, we showcase that streaming operators showcase higher processing rates and quicker responsiveness in smaller window sizes compared to larger ones. This paper comprises the latter portion of Chapter 5.

Chapter 2

Background

2.1 Fundamentals of Data Stream Processing

Stream processing is a programming paradigm defining applications that process unbounded stream of events as a collection of elements as they arrive at the stream, and this is applicable to different use cases [117]. Stream processing has become commonly used in real-time applications, due to lower processing latency compared with batch processing. In Facebook, for example, many stream processing applications require a 90 seconds end-to-end latency guarantee [82]. Stream processing is commonly categorised into stateless and stateful processing. Stateless streaming operators only consider their current input without knowledge of their past operations. Stateful streaming processing, on the other hand, keeps a copy of its state. Therefore, allowing the system to reference an event in a previous state or allowing past events to influence the way current events are processed [102].

There are open challenges in stream processing systems like Flink that still requires the attention of the research community to tackle. These includes, the manual reconfiguration of a running streaming application topology. Second, most streaming frameworks can't sustain their Service Level Agreement (SLA) when the system is overloaded because they sometimes choose the wrong execution plan on the first try. Thirdly, the default topologies are mapped to the nodes regardless of the knowledge of the workload, this causes a burden and usually slows down not only the job at hand but also the whole system, many researchers [51, 84, 107, 105] have advocated for an automatic scaling approach which has the ability to refine the topology of the system at runtime.

Hummer et al. [51], argue that while several state-of-the-art distributed streaming frameworks have the ability to manually change the operator distribution topology, none of these frameworks has yet established a mechanism to automatically adapt to the incoming workload. Auto-scaling is designed to increase the overall performance by making the refining of topology robust according to incoming workload streams in real-time, while maintaining SLA constraints [84].

The Performance metrics used by [51] are throughput (number of records out/time in seconds) and latency. These metrics do not cater for the active and ideal time of an operator which might not give an exact performance measure of the throughput of an operator. In this thesis we leverage the notion of true and observed processing rate as described in [58]. The

term `TrueProcessingRate` refers to the highest possible quantity of records that an operator instance can handle within a given unit of useful time. This operation intuitively determines the capacity of an instance of an operator. It should be noted that the useful time can be calculated by subtracting the waiting time from the duration, as stated by Kalavri et al. [58]. The `ObservedProcessingRate` refers to the number of records that an operator instance is capable of processing within a given unit of observed time. In contrast to the `TrueProcessingRates`, the observed rates are determined by simply counting the number of records processed and output by an operator instance over a unit of elapsed time, which might include any waiting time that existed over the processing period.

The occurrence of waiting time is observed in practical scenarios, with the specific reasons varying depending on the architecture of the reference system. Within the Flink framework, it is possible for an operator instance to experience blocking conditions. This can occur when the input buffers are devoid of data or when the output buffers, which have a finite capacity, are unable to accommodate additional information. Serialization and deserialization overheads can introduce delays, leading to increased waiting time. This is particularly critical in applications that require low-latency processing, such as real-time analytics and monitoring, where even small delays can impact the application's effectiveness.

The term "processing capacity" or "capacity" refers to the maximum amount of information or data that a system or operator can handle or process within a period.

Furthermore, Hummer et al. [51] do not make reference to state management which suggests that the system architecture will be unable to handle a service disruption or failure. Considering Flink's distributed data processing nature, provisions must be made to handle failures such as machine failure, killed process, network interruption, etc. My research study considers Flink checkpointing mechanism that guarantee exactly once state consistency to simulate a state recovery.

Similarly, we note Dhalion [38]. A system that essentially allows stream processing frameworks to become self-regulating. Dhalion seeks to solve the complexity of manually configuring, managing and deploying streaming processing systems. Floratou et al. [38] argue that the manual tasks involved in tuning streaming application to meet Service Level Objectives (SLOs) such as adjusting to load spikes, etc. is time-consuming, tedious and error-prone. This research examines the impact of time to redeploy a streaming application from save point. While Dhalion focused on resource utilization and performance (Throughput, latency and backpressure) this research seeks to identify the critical factors that should influence an optimal rescalling decision.

2.1.1 Streaming Processing General System Model

A stream processing system processes incoming data streams in real-time [19]. The process aggregates, filters, and analyzes data items to swiftly derive insights, respond to observed situations, and generate higher-level information. This can be seen in applications like continuous Twitter trend analysis, automated stock trading, fraud detection, and traffic monitoring. The basis of a stream processing system is the directed, acyclic operator graph, also known as its topology, which processes and forwards the input data streams [104]. In addition, the topology consists

of data sources that emit data items in streams into the graph and sinks that ingest the output. In Figure 2.1, we present a diagram of a typical stream processing system. The rectangles on the left represent data sources, while the circles represent operators. The flow of data streams is depicted by the boundaries between them. The sink is represented by the rightmost rectangle. In a distributed stream processing system, operators function across multiple interconnected processing nodes.

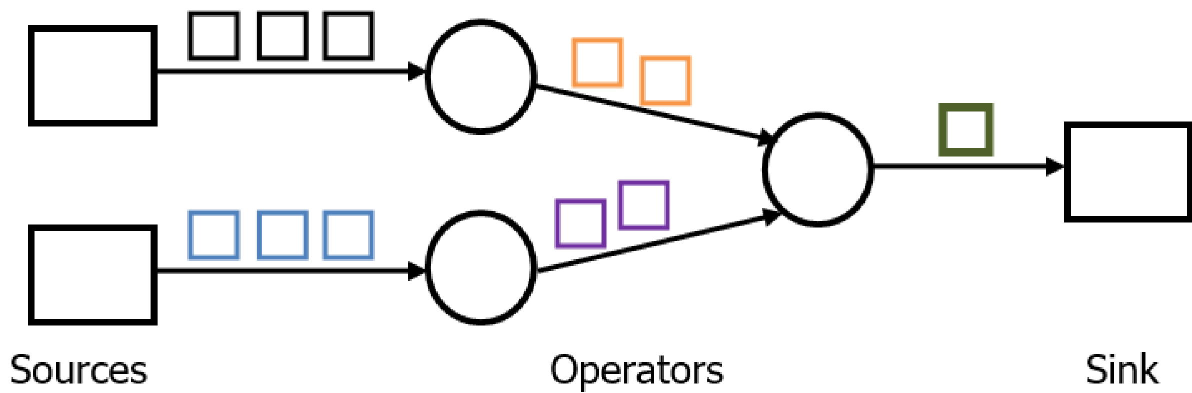


Fig. 2.1 A General Stream Processing Model

Considering that stream processing systems are intended to analyse data streams in real time, achieving low latency and high throughput are the main quality of service QoS goals. These objectives heavily influence the strategies employed for parallelization and elasticity. Failure to meet these QoS objectives can result in suboptimal system behavior, often accompanied by associated costs. Secondary QoS objectives, including load balancing, node utilization, and fault tolerance, are commonly configured to support the attainment of low latency and high throughput [104].

Operators and streams are the primary elements that make up the abstract model. Stream processing applications are typically defined at this level as directed graphs `G = (V, E)`, where V is a collection of vertices made up of data sources, operators, and sinks and E is a set of edges (i.e., streams flowing between vertices). Input streams come from data sources, which are represented by vertices without any outgoing edges. Sinks are vertices that have no outgoing edges and stand for consumers of the results generated, such as dashboards. You should be aware that at this level of abstraction, a source vertex may represent a variety of physical sources (such as sensors) that collectively generate a single logical data stream [14].

2.1.2 Apache Flink Overview

Apache Flink is an open-source framework that offers stateful stream processing capabilities that are scalable, distributed, and fault-tolerant [15].

Apache Flink enables the ingestion of vast amounts of streaming data, ranging up to several terabytes, from diverse origins and facilitates its distributed processing across multiple nodes. The resultant streams can then be directed to other applications or services, such as Apache Kafka, Databases, and Elastic search [15]. The fundamental components of a Flink pipeline comprise

of three stages: input, processing, and output. The system's operational duration facilitates expeditious data processing with exceptionally high rates of throughput while maintaining fault tolerance. The capabilities of Flink facilitate the extraction of real-time insights from streaming data and enable event-based functionalities. Flink facilitates instantaneous data analysis on streaming data, making it a suitable option for uninterrupted Extract-Transform-Load (ETL) pipelines on streaming data and for applications that are driven by events [98]. Figure 2.2 illustrates the Apache Flink software stack

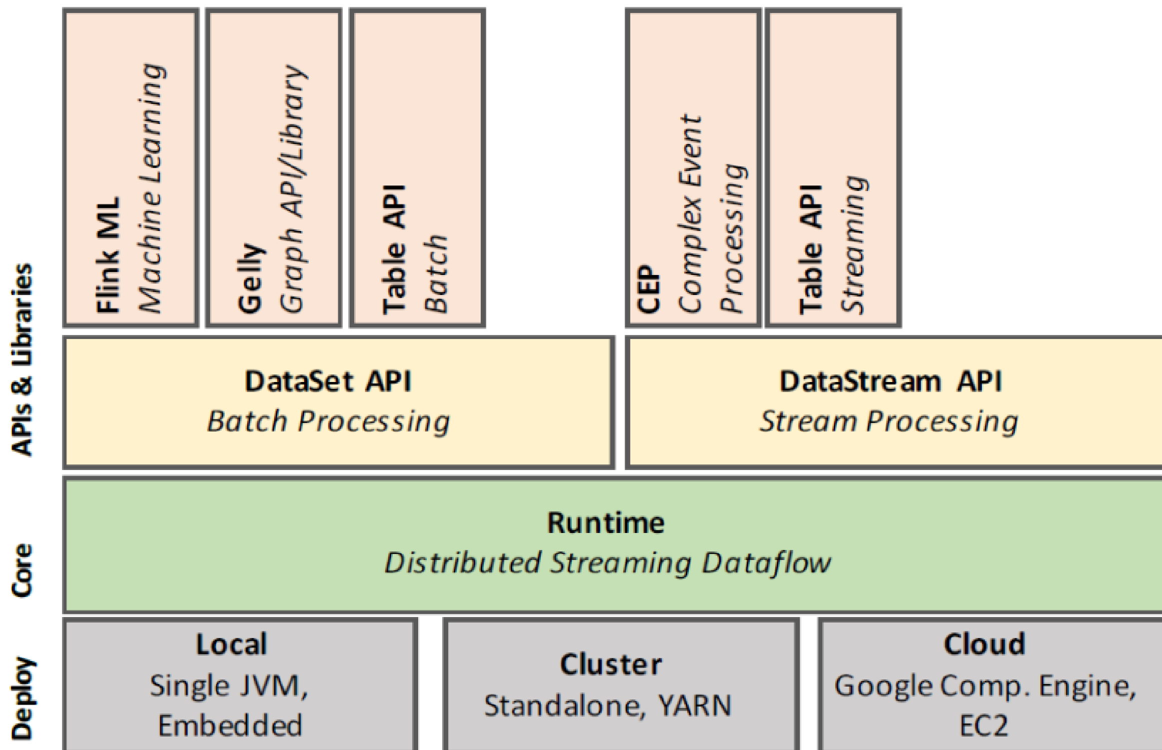


Fig. 2.2 Apache Flink Software Stack [15]

The software stack of Flink comprises of two distinct APIs, namely DataStream and DataSet, which are utilised for the processing of infinite and finite data, correspondingly. Flink provides a variety of operations for manipulating data streams or sets, including but not limited to mapping, filtering, grouping, state updating, joining, window definition, and aggregation.

Flink's primary data abstractions are DataStream and DataSet, which serve as immutable aggregations of data entities. In the context of DataSet and DataStream, it can be observed that the list of elements in the former is limited or finite, whereas, in the latter, it is unrestricted or infinite [112].

Flink applications are shown by a data-flow graph, more specifically a Directed Acyclic Graph (DAG), which is run by Flink's core, which is a distributed streaming dataflow engine. Data flow graphs are made up of operators that keep track of the state of the system and parts of secondary data streams. A varying number of parallel instances, set by the parallelism level, makes it possible for operators to run at the same time. Every instance of the parallel operator is executed within a distinct task slot on a computing machine that belongs to a cluster of computers.

Flink has a valuable set of features for building and benchmarking windows on data streams. It uses a blocking queue to manage back pressure, supports custom windows operators with a large pre-defined windowing operator and supports out of order stream. At a minimal cost Flink automatically transfers information to upstream operators upon detection of congestion [99]

2.1.3 Flink Architecture and Process Model

The Flink cluster is managed by three runtime components, namely the JobClient, JobManager, and TaskManager. The role of job manager is responsible for overseeing the allocation of resources and the distribution of tasks among the available task managers. The task manager is responsible for executing computations and providing updates to the job manager regarding the status of the tasks.

The distributed communication in Flink is facilitated through the utilisation of the Akka framework [8]. Akka is a software framework that employs an actor model to facilitate the development of applications that are capable of concurrent processing, fault tolerance, and scalability. Each actor within the Akka framework is regarded as autonomous and interacts with other actors in an asynchronous manner. The process of executing a job is illustrated in Figure 2.3 presented below. The number labels signify the sequence of events within the Flink cluster.

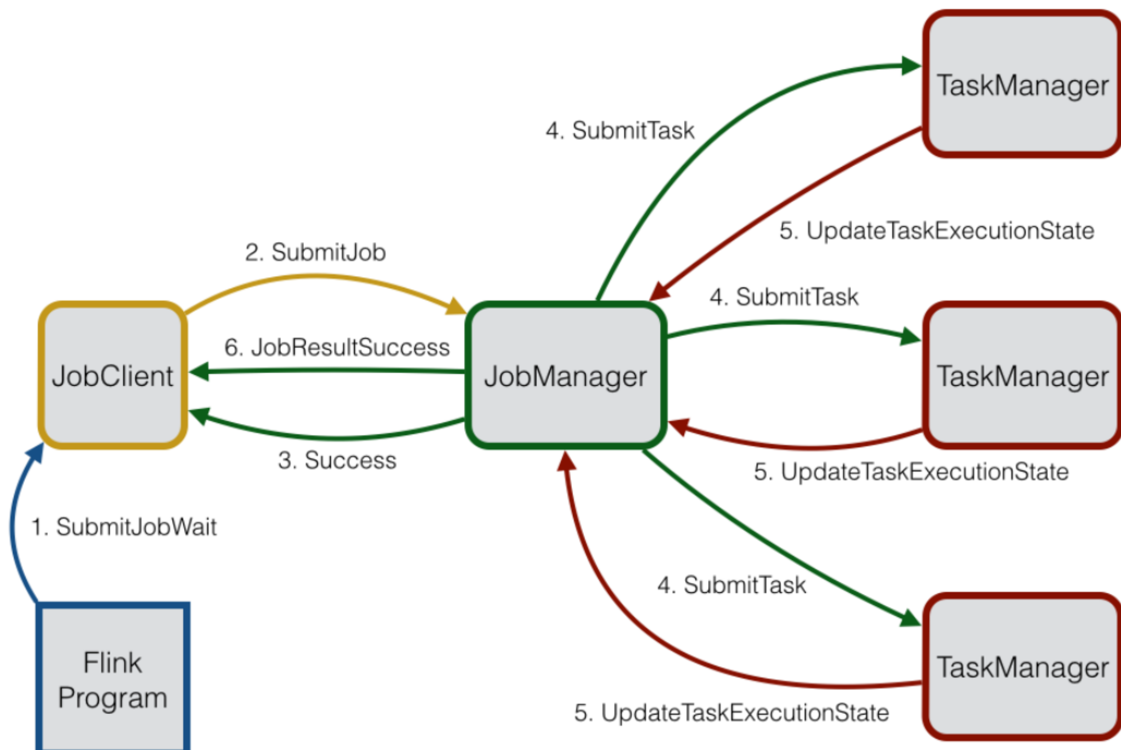


Fig. 2.3 Flink Job Execution Process [33]

Job Client

End users engage with the system through the "JobClient" module. The JobClient performs various functions, such as submitting jobs, communicating with the JobManager, retrieving the

the state of the job that is currently running, and querying the job status. After the JobClient gets the user programme, it is compiled and optimised to make a logical graph representation, which is then sent to the JobManager for processing.

The JobClient is an actor that facilitates communication through message exchange. There are two distinct messages pertaining to job submission, namely `SubmitJobDetached` and `SubmitJobWait`. The initial communication entails the submission of a task while simultaneously opting out of receiving any subsequent updates or notifications regarding the job's progress and ultimate outcome. The detached mode can be a valuable tool for submitting a job to a Flink cluster in a manner that allows for submission without the need for continuous monitoring [33].

The `SubmitJobWait` message is utilised to submit a job to the JobManager and simultaneously enrols to obtain status messages pertaining to the job in question.The internal process involves the creation of a helper actor that serves as the recipient of status messages. Upon completion of the task, the JobManager dispatches a `JobResultSuccess` message to the auxiliary actor, containing the duration and the accumulator results.Upon receipt of the aforementioned message, the assisting entity proceeds to forward the message to the respective client who had initially issued the `SubmitJobWait` message, following which it ceases to function [33].

Job Manager and Task Manager

The JobManager serves as the primary process tasked with overseeing the execution of Flink jobs. The JobManager is responsible for managing physical translation, resource allocation, and task scheduling. The JobManager is responsible for distributing tasks to various task managers [15].

Prior to the execution of any Flink job, it is necessary to initiate at least one JobManager and one TaskManager. Subsequently, the TaskManager initiates communication with the JobManager by transmitting a message denoted as "`RegisterTaskManager`". Upon a successful registration, the JobManager issues an `AcknowledgeRegistration` message. If the TaskManager has been registered with the JobManager due to the transmission of multiple `RegisterTaskManager` messages, the JobManager will issue an `AlreadyRegistered` message. In the event of registration being declined, the JobManager will issue a message denoting the refusal, namely the `RefuseRegistration` message [55].

To submit a job, a `SubmitJob` message along with the corresponding JobGraph must be sent to the JobManager. After the JobGraph is received, the JobManager proceeds to generate an `ExecutionGraph` from it. This `ExecutionGraph` functions as the abstract model of the distributed execution. The `ExecutionGraph` comprises the pertinent details regarding the tasks that necessitate deployment to the TaskManager for their execution.

The allocation of execution slots on the available TaskManagers is the responsibility of the scheduler of the JobManager. Upon allocation of an execution slot on a TaskManager, a `SubmitTask` message containing all requisite information for task execution is transmitted to the corresponding TaskManager. The `TaskOperationResult` is recognised as an indicator of a prosperous task deployment. Upon successful deployment and execution of the sources associated with the submitted job, the job submission is deemed to be successful. The JobManager apprises

the JobClient of the aforementioned condition through the transmission of a Success notification, inclusive of the relevant job identifier.

The JobManager receives updates on the status of individual tasks being executed on TaskManagers through the transmission of `UpdateTaskExecutionState` messages. By means of these update notifications, it is possible to modify the ExecutionGraph so as to accurately represent the present status of the execution.

The JobManager serves the additional function of serving as the assigner of input splits for various data sources. The responsible function involves the equitable allocation of tasks among all TaskManagers, with due consideration given to preserving data locality whenever feasible. To achieve dynamic load balancing, the Tasks solicit a fresh input split subsequent to completing the processing of the previous one. The act of fulfilling this request is achieved through the transmission of a `RequestNextInputSplit` message to the JobManager. Upon receiving a request for the next input split, the JobManager provides a response in the form of a `NextInputSplit` message. In the absence of additional input splits, the input split referenced in the message assumes a null value [15].

The Tasks are deployed in a deferred manner to the TaskManagers. Consequently, operations that require the utilisation of information are exclusively implemented subsequent to the completion of data production by one of its corresponding producers. Upon completion of the task, the producer transmits a `ScheduleOrUpdateConsumers` message to the JobManager. The communication indicates that the end-user is now able to access and review the recently generated information. In the event that the task being consumed is not currently in operation, it shall be deployed to a TaskManager. The architecture of a typical Flink cluster is depicted in Figure 2.4.

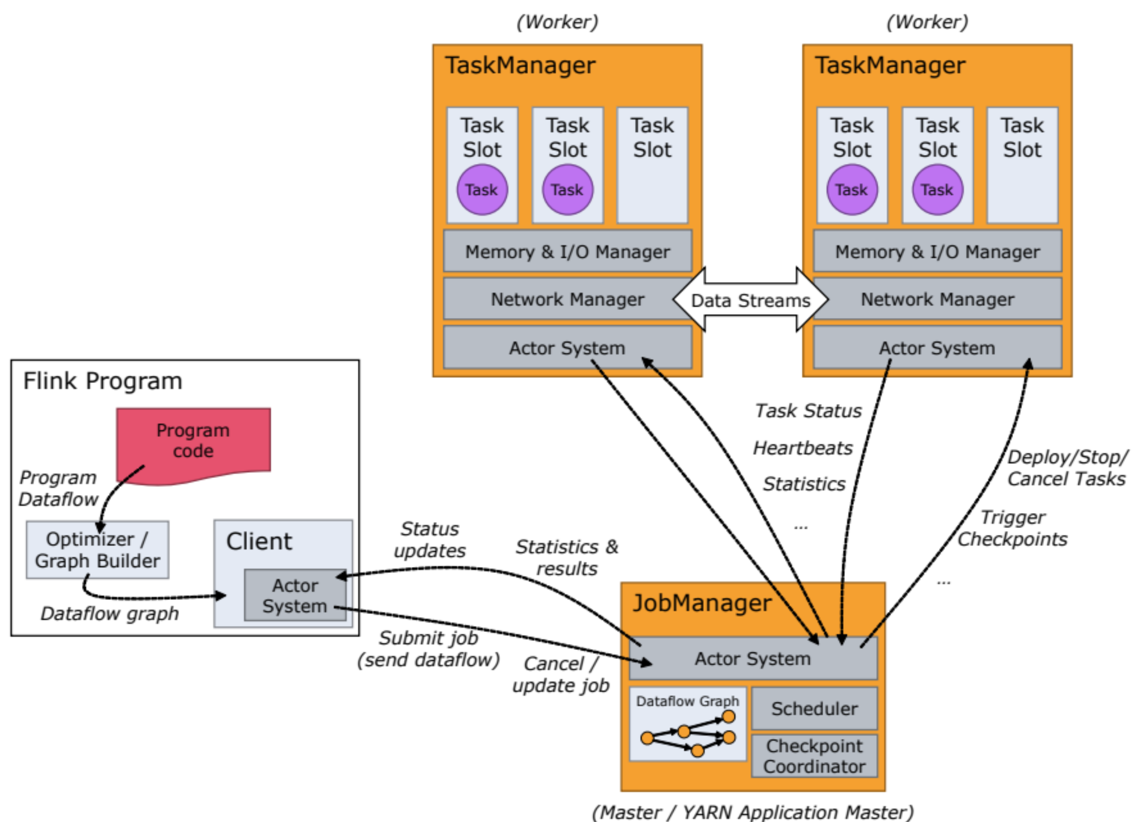


Fig. 2.4 Flink Architecture [12]

Flink Time Attributes

Flink has the capability to perform data processing operations utilising diverse time concepts namely Processing time and Event time. The system time of the machine is what the word "Processing time" refers to, also referred to as epoch time, such as the `System.currentTimeMillis()` in Java, which is engaged in performing the corresponding operation. The concept of "Event time" pertains to the computation of streaming data by utilising the timestamps that are associated with individual rows. The temporal information can be encoded by timestamps to indicate the occurrence of an event.

The time placement of a record upon its arrival at the window operator is a determining factor for its allocation to a specific window. If data is stored in a message broker and handled asynchronously from its origination, the processing time may not yield the intended outcomes, particularly when there is a significant time gap between origination and processing. The utilisation of the concept of event time can serve this objective [12]. It is imperative that every record includes a timestamp denoting the precise moment of its creation, also known as the event time. Upon reaching a window operator, the timestamp of a record is utilised to ascertain the specific window to which the record pertains. The window operator necessitates a means of ascertaining the comprehensive advancement of event time to effectuate the termination of a window subsequent to the stipulated duration. Flink employs watermarks to signify the advancement of event time, which are derived from records at the data source or immediately thereafter. Watermarks are a crucial component of the data stream, as they are strategically

placed between records to indicate that any subsequent records with timestamps lower than the watermark are not expected to appear in the remainder of the stream.

2.1.4 Flink Execution Mode

The DataStream API provides various runtime execution modes that can be selected based on the specific needs of the use case and the job's attributes. The two modes of execution modes are streaming and batching. The streaming execution mode is recommended for unbounded tasks that necessitate ongoing incremental processing and are anticipated to remain operational indefinitely. During batch execution mode, jobs are executed in a manner that bears resemblance to batch processing frameworks like MapReduce. This approach is recommended for tasks that have defined parameters and a finite input, and are not executed in a continuous manner.

The unified methodology of Apache Flink for stream and batch processing ensures that the execution of a DataStream application with bounded input will yield identical final outcomes, irrespective of the designated execution mode. It is noteworthy to consider the definition of "final" in this context: a task performed in streaming mode has the potential to generate incremental updates, whereas a batch task would solely produce a singular final outcome upon completion. Although the ultimate outcome remains unchanged with proper interpretation, the approach to achieving it may vary.

The activation of batch execution in Flink facilitates the implementation of supplementary optimisations that are exclusively feasible when the bounded nature of the input is known. Various join and aggregation techniques can be employed, along with an alternative shuffle mechanism that facilitates improved task scheduling and failure recovery performance. The subsequent section will delve into the intricacies of the execution behaviour.

Distinctive Behaviour of Streaming and Batch Execution Mode

Under the streaming execution mode, it is imperative that all tasks remain continuously online and operational. The capability of Flink to promptly handle fresh data entries throughout the entire pipeline is essential for the seamless and real-time processing of data streams with minimal delay. Consequently, it is imperative that the task managers assigned to a given task possess adequate resources to execute all the assigned tasks concurrently [50].

The process of network shuffles is executed in a pipelined manner, whereby data records are expeditiously transmitted to subsequent tasks downstream, with a certain degree of buffering implemented at the network layer. This requirement arises due to the absence of inherent temporal breakpoints for data materialisation between consecutive tasks or task pipelines while processing a continuous data stream. This stands in contrast to the batch execution mode, which allows for the materialisation of intermediate results, as elaborated upon subsequently.

The batch execution mode facilitates the segregation of a job's tasks into distinct stages that can be sequentially executed. This is feasible due to the limited scope of the input, which enables Flink to execute an entire stage of the pipeline before proceeding to the subsequent one. In contrast to the streaming mode described earlier, the process of executing tasks in stages necessitates Flink to persist intermediate results to a durable storage medium. This enables

downstream tasks to access these results even after the upstream tasks have terminated. The introduction of this approach is likely to result in a higher processing latency, albeit accompanied by other noteworthy characteristics. Firstly, this feature enables Flink to perform a rollback to the most recent obtainable outcomes in the event of a failure, rather than initiating a complete job restart. An additional consequence is that batch jobs have the ability to operate on a reduced number of resources, as measured by the quantity of available slots at task managers, due to the system's capacity to execute tasks consecutively and in succession.

Task managers are designed to retain intermediate outcomes for a duration that is at least equivalent to the time period in which downstream tasks have not yet utilised them. In technical terms, the retention of these items will persist until the pipelined regions responsible for their consumption have generated their respective outputs. Subsequently, the data will be retained for an indeterminate duration, contingent upon the availability of storage space, to facilitate the aforementioned ability to retrace previous outcomes in the event of a malfunction.

2.2 State Management

The concept of state effectively encompasses all internal side-effects that arise from a continuous stream computation. The aforementioned encompasses various elements such as operational windows and sets of data in an application, in addition to conceivably certain variables established and revised by the user throughout the operation of a stream pipeline [39]. An in-depth analysis of the disclosure and administration of state in stream processing systems reveals intriguing patterns in computer systems and cloud computing, as well as insights into the potential advancements in event-based computing.

Big data processing systems are made up of many different ideas, such as data flow operators, global scale out, and fault-tolerance. All of these are used to control, manage, or change the state. In data analytics, programmes can be shown as either directed data flow graphs or trees, as long as there are no iterations or results that are shared. From this point of view, the results of the study can be thought of as the root components, with the operators serving as the intermediary nodes, and the data acting as the leaves. Each operator node performs an action that changes the incoming data arriving into outputs. The transmission of data from the leave nodes, to the intermediary operator nodes, and then to the root nodes.

There are two distinct types of operators. Stateless operators are characterised by their purely functional nature, which enables them to generate output exclusively based on the input they receive. Stateless operators are exemplified by relational selection, merging two inputs, or relational projection without duplicate elimination. In contrast, operators that are stateful perform their output computation based on a sequence of inputs and may utilise supplementary side information that is stored in an internal data structure referred to as state [116]

Figure 2.5 organises state management into five principles and three state applications, each based on the primary topic it addresses. In the context of this research endeavour, the scope will be limited to a few pertinent aspects of state management as depicted in Figure 2.5, namely Operations, Load Balancing and Stateful and Stateless Computation.

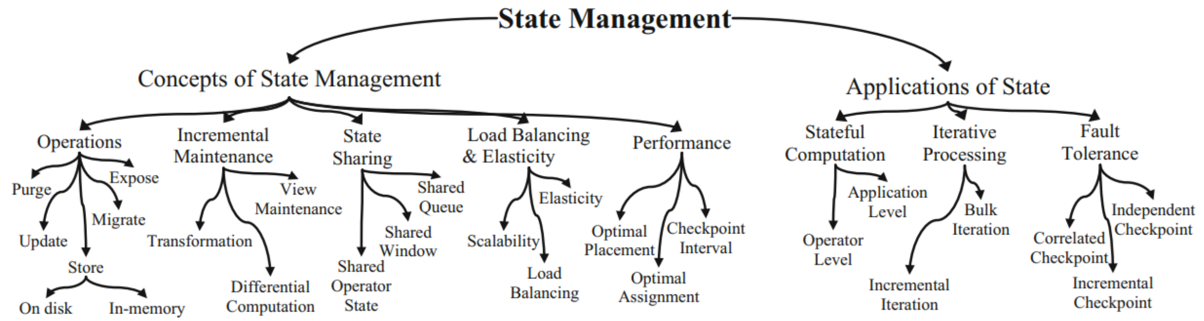


Fig. 2.5 State Management's Many Facets [116]

2.2.1 State Operations

The efficient management of state poses a multitude of technical obstacles. The migration of state among operators or nodes in a cluster has been explored in previous research [29, 37]. Additionally, exposing this state to programmers has been suggested as a means of facilitating its use [18]. Incremental maintenance of state has been proposed as a way to enhance performance, while sharing state among different processes has been suggested as a means of conserving storage [116]. The storage of state can be accomplished either remotely or locally, and various techniques such as in-memory or disc spilling can be employed [100]. Furthermore, load balancing can be achieved through these techniques, and there is even the possibility of geographically distributing state. Various operations can be performed on state, such as storing, updating, purging, migrating, and exposing [116]. In the following, we will explain some of these operations.

Storing State

The storage options available for states exhibit significant variation, with the size of a state typically dictating the location of its storage. In the case of small sizes, Ren et al. [100] suggest the utilisation of in-memory storage for state retention, which has the potential to expedite processing. However, it is important to note that this approach may also impede recovery operations in the event of machine malfunctions. In this instance, it will be necessary to replicate the state across multiple machines to enable recovery from potential machine failures, including those of a transient nature. In contrast, scholars Nicolae and Cappello [89] have devised remedies for large sizes, wherein the state is retained in persistent storage. Nevertheless, this results in increased overhead. However, determining the most suitable location for optimal storage of state is not always a straightforward task. In the following section, we will examine three potential approaches for managing large state sizes: load shedding, state spilling, and state cleanup delay.

The execution of lengthy queries on data streams, commonly referred to as Long-Running Queries (LRQ), may require a significant amount of memory due to the complexity of the queries and the large operator states involved, particularly in the case of multi-joins. In situations where there is a scarcity of system resources, and processing demands cannot be fulfilled, alternative handling methods may be utilised. This may occur, for instance, when there is a high throughput and inadequate storage or computational capacity. The technique of load shedding

is known to retain only a portion of the system's state, such as a sample, synopsis, or through lossy compression. This approach results in a reduction of workloads and an improvement in performance. However, it comes at the cost of decreased accuracy. According to Liu et al. [73] , workloads have the potential to be permanently offloaded or deferred for subsequent processing when computing resources become accessible.

In situations where accuracy is of utmost importance, the implementation of load shedding may not be a feasible remedy. Therefore, it is possible to utilise an alternative technique known as state spilling. This holds particularly true for relational operators that maintain state as they resort to flushing in-memory states to discs when the memory reaches its capacity. An additional alternative is to postpone the process of state cleanup, which involves the handling of states that are stored on discs, until the availability of resources is sufficient. All of these solutions for state handling are capable of achieving both low-latency processing and accurate results [116].

Updating State

Within this particular section, we identify four distinct concepts pertaining to the modification of state: specifically, incremental state updates, fine-grained updates, consistent updates, and update semantics. In this context, our attention is directed towards the fourth concept, which is commonly referred to as the update semantics.

Various big data processing frameworks [15, 118, 139] investigate and contrast diverse update semantics for state. The assessment of state correctness can be achieved through three distinct types of semantic guarantees, which are at-least-once, at-most-once, and exactly-once. Systems that employ at-least-once semantics ensure complete processing of each tuple. However, they are unable to provide assurance against duplication during processing, which could mean that a tuple is added to the state. In the context of at-most-once semantics, it can be observed that systems exhibit behaviour where tuples are either not processed or executed only once and subsequently added to the state. In contrast to the at-least-once semantics, the at-most-once semantics do not require that duplicated tuples be found. Systems that implement exactly-once semantics make sure that each tuple is processed once, without exception, thus offering the most robust guarantee.

Purging State

State management can eliminate data that is no longer required for subsequent operations within a system. For instance, a buffer state may remove tuples that have expired. Some scholars have suggested various methods for eliminating state in a distributed system [127].

Migrating State

Dynamic state migration is a very important operation, especially for stream processing systems. It is used to move state from one node to another in an efficient way while keeping the operator semantics. The fluctuation of workloads, data characteristics, and resource availability necessitates special attention to operations such as joins and aggregations when nodes are added or removed. This is of particular importance.

Cardellini et al. [16] note that moving from one state to another comes with two main problems: what to move and how to move? The first consideration pertains to the selection of a mechanism that minimises the synchronisation overhead and defers the production of results while migration is underway. The second consideration pertains to the determination of what data or processes should be migrated. That is, figuring out the best way to divide up tasks so that transfer costs are kept to a minimum.

2.2.2 Load Balancing and Scalability

The dynamic nature of system workloads necessitates the implementation of load balancing or elasticity mechanisms to effectively manage increased demands. Load balancing is the ability of a system to distribute its workload between its computational resources, especially in situations where certain nodes are burdened with more significant workloads than others [117]. In the event of an increase in workload within a node, it is possible to achieve a balance of workload by redistributing it to another node, as illustrated in Figure 2.6 (a). The concept of elasticity pertains to the capacity of a computing system to furnish supplementary computing resources when faced with escalating workloads.

As workloads increase, it is possible to allocate supplementary resources, such as nodes, to distribute the workload. This is illustrated in Figure 2.6 (b). Managing load balancing and elasticity in stateless operators is a relatively uncomplicated task. However, the same cannot be said for stateful operators, as their intricate nature poses a significant challenge.

Contemporary data-parallel computation frameworks address the issue of elasticity by means of preserving and transferring state during the course of active job execution. In order to facilitate state migration, it is necessary to ensure that the quantity of parallel channels is capable of dynamically adjusting to changes in the number of nodes, as well as fluctuations in computing resources and workload availability that may arise during runtime.

As depicted in Figure 2.6, the distribution of load can be either uniform or non-uniform across various nodes. Figure 2.6 (a) & (b) in the right half of the figure, we show an example of a system with uniform load distribution. In this case, the work or load is evenly distributed across all the nodes. Each node is handling a roughly equal amount of work or processing tasks. However, on the left side of Figure 2.6, we present an example of a system with non-uniform load distribution. Here, the load is not evenly distributed among the nodes. One node is handling a much heavier workload compared to the other node. This non-uniform distribution can occur due to various factors, such as varying task complexity, imbalanced data partitions, or resource limitations.

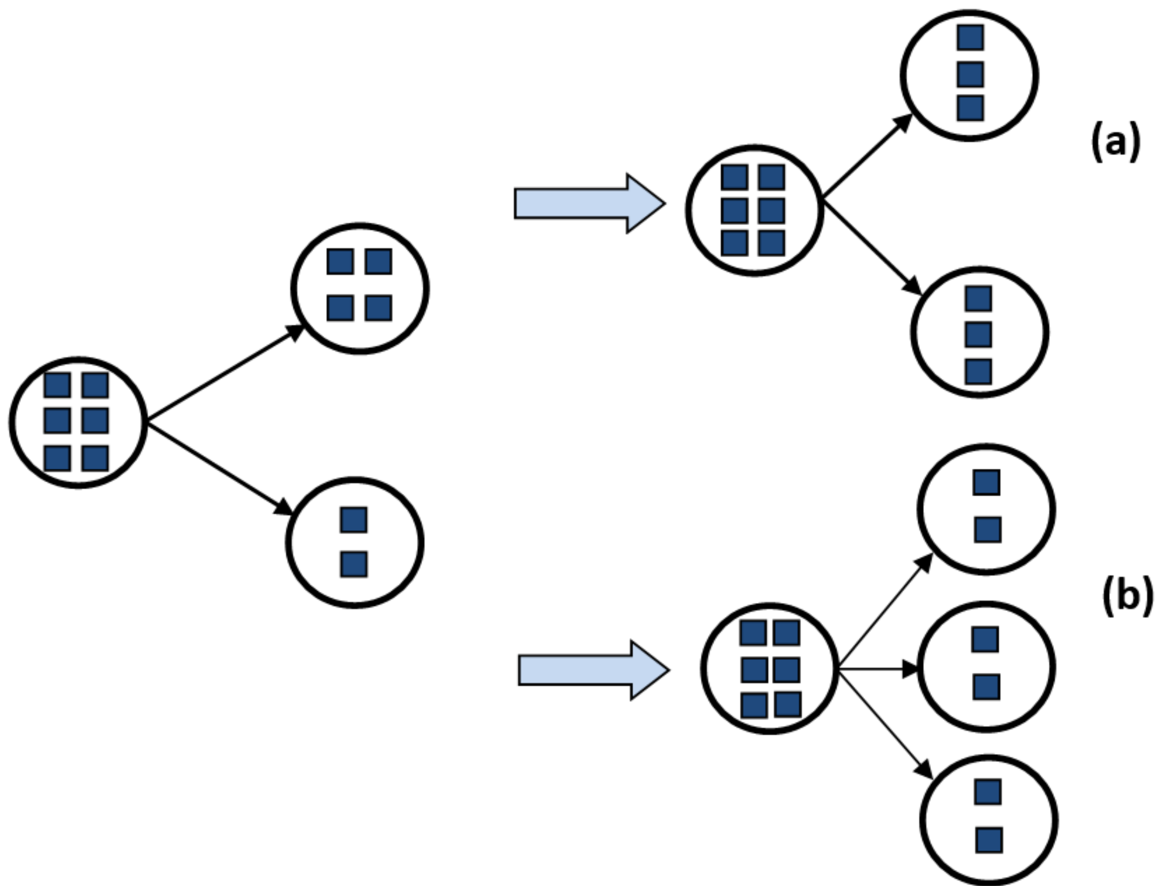


Fig. 2.6 Balancing Load with State Awareness

In a varying workload environment, redistribution and reassignment of heavy burdened node states (as illustrated in Figure 2.6 occurs to nodes that are not experiencing a heavy workload. In situations where resources are limited, it becomes necessary to reassign the states of affected tasks, such as job partitions. Therefore, it is necessary to implement partitioning techniques that facilitate the scalability and workload balance of systems.

In the context of stream applications, the concept of scalable state can be classified into two distinct categories: partitioned state and non-partitioned state, which is also known as global state [127]. The utilisation of one or both of these state types is contingent upon the characteristics of a given operation.

Partitioned State

The utilisation of partitioned state has become the established method for facilitating data-parallel computation on extensive data streams [16]. The concept of partitioned state enables the logical partitioning of state based on keys for the purpose of performing computational tasks. Each logical task is responsible for processing a specific key. The API level facilitates this process by incorporating an extra operation that is executed before stateful processing, thereby elevating the scope from task-oriented to key-oriented processing. This is analogous to the "keyBy" function in Apache Flink or the "groupBy" function in Beam and Kafka-Streams. At the physical level, you can give a physical job or computing node more than one key (or a range of keys).

Non-partitioned State

The state that is not partitioned is represented as a single entity and is assigned to physical computing tasks. The utilisation of a non-partitioned state is commonly observed in two distinct manners. Initially, to calculate overall aggregates across the entire input stream. Furthermore, it is possible to utilise it for the computation of aggregates on the physical operator level, such as determining the number of keys that have been processed by each operator. Keeping offsets while receiving logs from a physical stream source task can be made easier by using task-level state. Non-partitioned state is either about operator-specific calculations or about the sum of everything [16]. The scalability of its usage is limited and it is advisable for practitioners to exercise caution when utilising it.

2.2.3 Stateful and Stateless Computation

The stateful computations during data stream processing are facilitated by the state in a natural manner. The processing of data stream records can be classified into two categories: stateless computation and stateful computation. Stateless operators, such as filtering, do not maintain any record of prior computations. In contrast, every computation is executed in a purely functional manner, solely relying on the present input. As per the definition, operators that maintain state, such as those performing aggregations over time windows or other stream discretisation methods, exhibit interactions with preceding computations or data that have been recently observed. Therefore, as state embodies antecedent computational outcomes or formerly observed information, it is imperative to persist. Figure 2.7 depicts a streaming application's load distribution across nodes. Stateful computation, on the right, involves processing data while maintaining state, while stateless computation, on the left, processes data independently without needing past context.

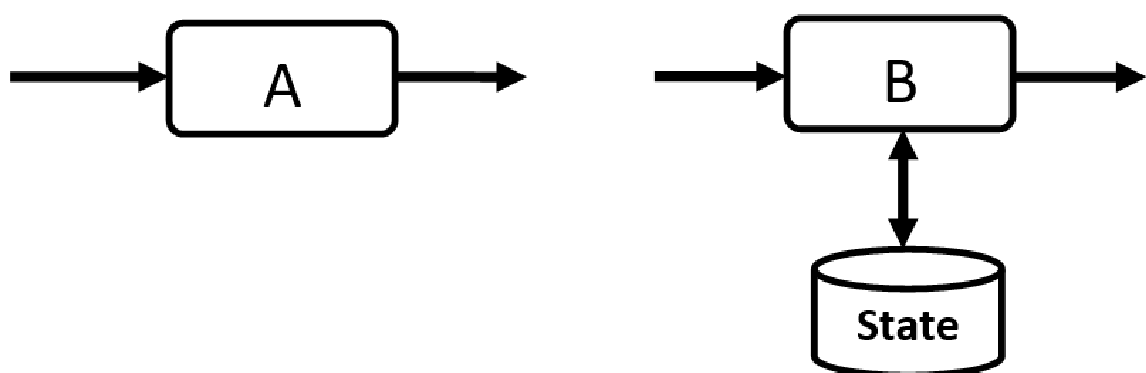


Fig. 2.7 An Example of State Computation

Contemporary data stream processing frameworks, including Flink, Spark, Storm, Storm + Trident, and Heron, have stateful operators, which is a clear indication of this trend. Although there are similarities between various frameworks, there exist divergent perspectives regarding the optimal approach to implementing state. In its initial iterations, Storm prioritised stateless processing and necessitated the management of state at the application level. The utilisation

of Storm and its extension, Trident, facilitates state management through the utilisation of an Application Programming Interface (API). Samza utilises a local database to facilitate persistence and effectively manage substantial states. The utilisation of DStream, which stands for discretised streams, allows for state computation through the implementation of Spark Streaming. Flink prioritises state as a primary element, thereby simplifying the development of applications that require stateful processing. Table 5 presents an overview of the features of stateful computation techniques in four different systems.

Table 2.1 A characterisation of methods for stateful computation.

Characteristics	Systems	Main Mechanism
Batch Processing	[75]	Data indexing
	[74]	State as explicit input
Stream Processing	[41]	Partitioned stateful operators
	[27]	Parallel patterns

2.2.4 Windowing

Windowed operators, split incoming data into parts also referred to as windows, there are special kinds of stateful operators that process these windows as a whole [43]. For example, given a stream of patient inflow into an hospital A&E department, we may want to compute the number of most common medical complains by patients in the last hour. In this scenerio, the operator will have count the number of patient and group them in windows based on their specific medical situation.

A window can be described in terms of time, number of tuples, or number of sessions. Timestamps mark the beginning and end of time-based windows, which group data from the same time period. As tuples are received, the stream processing system may set explicit timestamps or implicit timestamps. Explicit timestamps are set by data sources as tuple properties, and implicit timestamps are set by the stream processing system. Most of the time, specific timestamps are better for data about real-world events, but they can also be hard to work with, especially when sources are spread out. First, the order in which data actually comes into the system may not match the order of the stream based on the timestamps. Some events may also be lost or delayed, and it's not always clear if a window is closed or if it's still open.

A straightforward method for dealing with stragglers relies on timeouts, after which any missing tuple is treated as lost. Setting a reasonable timeout is unfortunately difficult because big values have a significant influence on processing latency, while tiny values might require the discarding of a large number of delayed tuples. Given the importance of the problem, academics

have looked into more adaptable methods for dealing with stragglers and out-of-order data. Many DSP engines, like Flink and Google Cloud Dataflow, rely on watermarks [4, 39] (also known as landmarks), which keep track of the earliest timestamp that might still exist in a stream. Any operator can use watermarks to quickly determine when a window is finished and can continue with the computation once the watermark has passed the window's conclusion. The use of punctuations [120], which are unique tuples injected into the data stream that provide progress information about one or more attributes (such as a timestamp), as a watermarking technique has been investigated [4].

The definition of alternative windows is based on either the number of sessions or counts. Count based windows refer to a method of grouping a predetermined quantity of consecutive data items, such as "the previous 1000 events." Session based windows are initiated and terminated in a dynamic manner based on specific "activity" metrics. For instance, a window is deemed complete when no further events are received within a designated time frame [14].

The magnitude of the window is determined by the number of occurrences or temporal segments. The sliding interval, also known as stride, determines the potential overlap of windows. Two types of windows are commonly distinguished: tumbling (or fixed) windows and sliding windows. The partitioning of the input stream is defined by tumbling windows, which are characterised by their non-overlapping nature. Specifically, the size and sliding interval of tumbling windows are equivalent. On the other hand, it is possible for sliding windows to exhibit overlap, resulting in the inclusion of individual tuples in multiple consecutive windows.

2.2.5 Flink State Backends

The state backend of a streaming application is responsible for determining the location and method of storing and checkpointing the application's state. Diverse state backends exhibit variations in their storage formats and employ distinct data structures to maintain a functional application. The Apache Flink framework offers three distinct backends for storing application state information, which are Memory state backend, File System state backend and RocksDB state backend. In the absence of any other configurations, by default Flink uses the MemoryStateBackend [111]

Memory State Backend

The default state backend employed by Flink is the memory state backend. The state information is stored in the heap memory of the task manager. The utilisation of objects for state persistence results in expedited read and write operations. During the checkpointing process, the state backend captures a snapshot of the state, which is subsequently stored in the heap memory of the Job Manager. The utilisation of the memory state backend is primarily intended for purposes of local development and debugging. One constraint associated with the memory state backend is that the combined state must be accommodated within the JobManager's memory.

File System State Backend

Similar to the memory state backend, the file system state backend also retains the state information within the heap memory of the task managers. After the checkpointing process, the data from the snapshot is stored in a designated file system, which may include HDFS, S3 bucket, or the local file system of the task manager. The utilisation of the file system state backend is primarily observed in high availability systems, and its application is recommended for tasks that large state size, large window size/sliding period, and large key/value states. The thesis adopts the file system state backend, based on the aforementioned recommendations.

RocksDB State Backend

The RocksDB state backend is utilised to persist state information by means of a RocksDB database. RocksDB is a data storage system that maintains a sorted collection of key-value pairs, utilising the Log-Structured Merge (LSM) approach. The *RocksDBStateBackend* consistently executes asynchronous snapshots. LSM is a specialised data structure that has been devised to enhance the efficiency of write operations. RocksDB is characterised by a notably elevated rate of both reading and writing operations, and has been optimised to provide swift and low-latency storage capabilities [111].

The RocksDB system is primarily composed of three fundamental components, namely the `memtable`, `sstfile`, and `logfile`. The `memtable` is a data structure that resides in the computer's memory, and all write operations are appended to the memtable. Optionally, writes can be recorded in the write-ahead log. Upon reaching its maximum capacity, the `memtableundergoes` a transition to a READ-ONLY state and subsequently gets substituted by a fresh active `memtable`. Periodic flushing of READ-ONLY `memtable` to disc occurs in the form of `sstfile` (Sorted String Tables). The initial retrieval of all reads is sourced from the `memtable`. In the event of an unsuccessful retrieval, the read operation proceeds to access the READ-ONLY `memtable` in a reverse chronological sequence. In the event that it remains undiscovered, the system will proceed to retrieve data from the `sstfile`, beginning with the most recently generated. Consequently, the duration of reads may be slightly longer than that of writes to achieve completion. In order to expedite the process, `sstfile` undergo compaction and are protected by bloom filters to prevent superfluous scans.

The RocksDB ava Native Interface (JNI) bridge API utilises byte, thereby limiting the maximum size that can be supported for both key and value to 231 bytes. It is crucial to note that in RocksDB, states utilising merge operations such as `ListState` have the potential to accumulate value sizes exceeding 231 bytes without any indication, ultimately resulting in failure during the next retrieval process. Presently, this represents a constraint of RocksDB JNI [95]

2.3 Checkpointing and Restore

The fault tolerance mechanism in Apache Flink is achieved through the utilisation of both checkpointing and stream replay techniques. The utilisation of a checkpointing mechanism [63] is implemented in Apache Flink to facilitate the recovery of system state and ensure fault

tolerance. Within streaming applications, operators are designated with a specific point in each input stream. This methodology involves the continuous generation of low-impact snapshots of data streams to maintain consistency, adhering to either exactly-once or at-least-once semantics, through the replay of the stream from a designated point. Snapshots are typically designed to have a low weight for streaming applications that involve small states. Frequent snapshots are taken for both running states and distributed data streams with minimal impact on overall performance. The captured images are stored in a customizable location, such as a distributed file system.

To store and process application state during normal operation and quickly restore it after a failure, checkpointing is a crucial and practical crash-tolerant strategy. However, choosing the checkpoint frequency to reduce a suitable cost function would be the primary interest question for different research use case. knowing the conditions where checkpoints are useful and where they are not will help optimize the performance of a streaming application [143, 34]. In each of our experiments, we chose a checkpointing interval that our system resource can handle. Our interest is in the amount of state produced and collecting the state size for each deployment through an API call.

2.3.1 Consistent Checkpointing and Recovery

The challenge of distributed unbounded processing has made consistent stream processing a longstanding research issue. Additionally, the absence of a formal problem specification has contributed to this challenge. The concept of consistency pertains to the assurances that a system can provide in the event of a malfunction, as well as any requirements for modification during its functioning. The act of modifying or revising a currently operational data streaming application is commonly referred to as reconfiguration within the field. An instance of this scenario pertains to situations where it becomes necessary to implement a software update to a streaming application or expand the computational nodes while ensuring that neither the precision nor the computation is compromised.

Previous studies, including SEEP [18], have emphasised the correlation between fault tolerance and reconfiguration. The SEEP system proposes an integrated strategy for scaling and restoring tasks in the event of failures. At present, the majority of stream processors operate as transactional processing systems that adhere to consistency rules and processing assurances. For example, Flink adheres to Atomicity, Consistency, Isolation, Durability (ACID) properties, providing strong consistency guarantees for stateful processing in streaming applications. Flink also supports event time processing and can achieve exactly-once processing through mechanisms like checkpointing and stateful fault tolerance.

2.3.2 Impact of Checkpointing Interval on Streaming Applications

heuristics are commonly employed to determine the optimal checkpointing interval, such as frequent or infrequent checkpointing. The implementation of frequent checkpointing facilitates expedited system recovery in the event of a failure. Nevertheless, the implementation may consume valuable resources and time that could be allocated more effectively in other areas.

On the contrary, the implementation of infrequent checkpointing results in extended durations for the recovery of system failures. In recent years, researchers in the field of streaming systems [18, 87, 31] have directed their attention towards identifying the most optimal frequency for checkpointing.

Naksinehaboon et al. [87] conducted a study on determining the most efficient placement of checkpoints in order to minimise the overall overhead, which includes both the overhead associated with rollback recovery and checkpointing. Through the utilisation of a checkpointing frequency function, an optimal interval for checkpointing can be determined by utilising a failure probability distribution provided by the user.

Fernandez et al. [18] conducted a study to assess processing latency and found that the implementation of infrequent checkpointing resulted in inconsistent latencies. The methodology employed by the researchers demonstrates that broader intervals exert a diminished influence on the processing of data, albeit at the cost of prolonging the duration of failure recovery. The proposal suggests establishing the interval for checkpointing based on the anticipated rate of system failures and the performance demands of the queries.

The impact of checkpointing intervals across methods is evaluated by Sayed and Schroeder in their study [31]. The authors provide a critique of ad hoc periodic checkpointing strategies, specifically those that involve checkpointing at fixed intervals, such as every 30 minutes. The authors note that Young's model [138] demonstrates a high level of performance close to optimality and is also practical in its application. The researchers delve deeper into more sophisticated techniques that dynamically modify the frequency of checkpointing. The results of their study indicate that these techniques exhibit a noteworthy enhancement compared to Young's model, albeit only for a limited number of systems.

2.4 Streaming Application Scalability

Previous work investigated horizontal scalability, i.e. increasing the number of workers. Karimov et al. [61] compared the performance of clusters with investigated performance for 1, 2, 4, and 6 workers. Analysis of how scalability is influenced by the throughput bottleneck, as well as the analysis of vertical and horizontal scalability for different frameworks. We take a similar approach for our horizontal scaling experiments where we carryout experiment to scale operator instance using the true and observed processing rate. However rather than toeing the part of benchmarking various frameworks, our work provides a universal approach that can be adopted by different use-case.

A variety of strategies have been put forth to manage distributed stream processing systems' flexibility in virtualised environments, They differ in the type of data that is monitored, the quality-of-service objective that is addressed, the deployment environment (cluster, cloud, fog), and the optimisation technique that is used [104].

Prior research has introduced automatic rescaling controllers that consider various metrics in arriving at a scaling decision [38, 125]. Kalavri et al. offers an autonomous scaling system called Data Stream 2 (DS2) that aims to balance resource over-provisioning and on-demand scaling [58]. DS2 measures each operator's true and observed processing ability. The number of

records that may be processed by an operator instance in one unit of useful time is known as the true processing rate (duration minus waiting time). This logically determines the operator hardware capacity. The number of records a particular operator instance processes in a certain amount of time is known as the observed processing rate (duration plus waiting time)

However, Like DS2, most auto-scaling systems do not consider the impact of application state and rely on offered load as a proxy for application state size, resulting in a false positive that will trigger a scale down. This assumption can mislead the auto-scaler to making a wrong scaling decision.

Using machine learning approaches, a different class of solutions uses measured or profiled data to identify patterns [17, 70]. Typically, patterns are improved at runtime to increase precision. To make precise scaling decisions, these systems must first undergo extensive training, which could take a very long time.

Long-running applications inevitably face various external and internal shocks capable of creating instability, for example, users' reactions to a national disaster on Twitter or a champion's league football goals. Unpredictable load variation can lead to over-provisioning or under-provisioning. Prior research has introduced automatic rescaling controllers [38, 125]. A key service distinction is the approach taken by various systems to react to changes in real-time and balance competing objectives like performance (throughput or latency) and resource utilisation during a load spike or resource failure [84, 85].

Some research has argued that memory and CPU utilisation metrics and other coarse-grained metrics are inadequate for arriving at a good scaling decision, especially in a multitenancy cloud environment where shared resources and performance interference are prevalent [58, 99]. Research is increasingly focusing on operator performance, dataflow topology, and the sustainability of throughput levels at different cluster sizes for different streaming frameworks [38, 60, 124]. We take a similar approach for our horizontal scaling experiments, investigating the operator throughput capacity. However, this thesis evaluates the performability requirement in rescaling when an application state size becomes bigger.

A suitable scaling controller should provide Stability, Accuracy, Short Settling Time, and Overshooting (SASO) properties [58]. Whereas speculative scaling techniques that violate these could lead to unnecessary costs due to sub-optimal utilisation of resources due to over-provisioning or under-provisioning, performance degradation due to frequent scaling action due to oscillation and low convergence resulting in an SLOs violation or load shedding [58]. Furthermore, an effective scaling controller must be mindful of the resource availability and the scaling duration. These factors are an important element that must be contained in the scaling policy.

Mindful of the challenges that come with decision-making on how to scale efficiently and when to scale, we study the work carried out in the DS2 project, DS2 evaluates each operator's true and observed processing capabilities regardless of the backpressure and other effects. [58]. Based on real-time performance traces, DS2 automatically determines, on demand, the optimal level of parallelism for each operator in the dataflow. The number of resources allocated to each operator is maintained as a dynamic provisioning plan. In our experiments, we use Apache Flink with the DS2 policy configuration values: 10-second decision interval (frequency of metrics

collection and policy application), 30-second warm-up period (the number of consecutive policy intervals that were ignored following a scaling action), one interval activation period (when DS2 will make a scaling decision), and a 1.0 target ratio (maximum permitted difference between the target rate and observed source rate attained by the policy).

However, like DS2, most auto-scaling systems do not consider the impact of scaling duration, especially in a volatile workload environment. Instead, more attention is focused on mitigating the impact of over-provisioning of resources for temporary load spikes and under-provisioning during peak loads.

Arkian et al. [7] introduce Gesscale (Geo-distributed Stream autoscaler), an auto-scaling mechanism designed for stream processing applications that operate in geo-distributed environments, such as fog computing. The Gesscale system employs a dynamic approach to managing the workload and performance of the system in real-time. This involves the addition or removal of replicas to or from individual stream processing operators to ensure that the Maximum Sustainable Throughput (MST) is maintained at an optimal level, while minimising resource utilisation. MST is a widely accepted metric used to evaluate the ability of a stream processing system to handle incoming data without experiencing excessive queuing delays, as documented in various sources [52, 24]. The Gesscale system utilises a performance model based on backpressure and operator utilisation to make a scale-up or scale-down decision. The implementation enables Gesscale to minimise the frequency of reconfigurations as it adopts a staggered scaling approach.

This research also reviews related research that examines workload prediction and resource allocation. On the basis of an underlying queuing model, Urgaonkar et al. [121] implemented dynamic provisioning of multi-tiered applications using virtual machines (VM). But there can only be one VM running on each physical host. Similar infrastructure is utilised by Wood et al. [130] and [121]. They mainly focus on Virtual Machines (VM) dynamic migration to facilitate VM dynamic provisioning. In order to decide whether to migrate, they construct a special metric based on consumption data for the three resources: CPU, network, and memory. The research in [121, 130] fails to link the placement method to a broader utility value for the Cloud provider. They merely make an effort to boost the application's throughput. Multiple classes are not present in the applications under consideration, which is unreasonable.

A thorough queuing model is created by Cunha et al. [25] to model virtual servers. Each class of jobs in an application is given its own virtual machine. They offer a pricing structure that promotes throughput that stays within SLA parameters and penalises throughput that exceeds those limitations. However, they put each class on a single virtual machine, which might not be economical if an application has multiple classes.

A control-theoretic method is offered by Padala et al. [96] in which each tier of the programme is run on a separate virtual computer. Black box application profiling is done by the authors, who then create an approximation model that links performance metrics like reaction time to the percentage of the processor allotted to the virtual machine that is running the application. For a virtualized context, Wang et al. [128] offer a two-level control architecture. A load balancing controller makes sure that all of the virtual machines are load balanced and that all of the applications respond uniformly.

An architecture for elastic management of cluster-based services is suggested by Moreno et al. [86]. A virtualized infrastructure layer that collaborates with a VM management and a cloud service provider makes up this system. This method aids in resource autoscaling with the least amount of user disruption. While Yang et al. [136] suggest a profile-based approach to the issue of just-in-time scalability in a cloud context, Waheed et al. [54] propose a reactive algorithm to allocate more resources to a cluster farm when workload grows. We also note the work carried out by Roy et al. [105] which Utilises a workload forecasting model-predictive algorithm for resource autoscaling. Our research evaluates the impact of long rescaling duration and provides a predictive model to forecast the rescaling duration of a streaming application based on the state size and end-to-end duration.

2.5 Adaptation Technique for Stream Processing Systems

Adaptation mechanisms refer to the available actions that can be taken to alter the configuration and behaviour of a streaming applications and their components during runtime. Cardellini et al. [16] categorises the various adaptation mechanisms into distinct groups, which include topology adaptation, deployment adaptation, processing adaptation, overload management, fault tolerance adaptation, and infrastructure adaptation. There exists a disparity in the level of attention received by various groups within the research community. The exploration of deployment adaptation mechanisms has surpassed that of other tools. The categories of processing and infrastructure adaptation mechanisms have garnered significant attention over the past decade, emerging as the most widely studied groups among the remaining categories. Thus far, the other groups have been subject to a restricted degree of scrutiny. This section examines the optimal timing for implementing adaptation measures and the appropriate time horizon for planning purposes.

Trigger

Adaptation measures may be initiated either through scheduled timers or in reaction to specific occurrences. The implementation of timer-based adaptation is relatively straightforward, as it involves the configuration of only an adaptation activation interval. This approach ensures that the adaptation policy continues to proactively plan and execute any necessary actions over a specified duration. The temporal duration separating successive adaptation rounds typically varies from a few seconds to several minutes. The activation interval for the adaptation policy must strike a balance between responsiveness and efficiency. While a short interval allows for quick responses to changes in conditions, frequent metrics collection and adaptation planning can result in overheads. Therefore, the appropriate activation interval must be determined based on this trade-off. The majority of current methodologies have implemented timer-based adaptation, as evidenced by various sources (e.g., [38, 11, 58, 71, 123]). Floratou et al. [38] and Kalavri et al. [58] proposed operator scaling techniques that involve periodic collection of relevant operator metrics, such as throughput, followed by the implementation of an adaptation policy by the system.

Several other studies have explored the concept of event-triggered adaptation actions, as evidenced by the references cited (e.g., [3, 22, 23, 59, 144]). In order to execute an adaptation policy, it is necessary to associate one or more types of events with the scheme's implementation. The arrival of a new tuple to a buffer is a commonly utilised event in literature for the purpose of load shedding, load distribution, and stream scheduling strategies. This has been demonstrated in various studies, including those referenced in sources such as [144, 62, 59]. Katsipoulakis et al. [62] propose a load distribution approach that is activated upon detection of each incoming tuple. Chaturvedi et al. [22] propose an approach for operator reuse, whereby adaptation is initiated upon the submission or termination of an application. In contrast to the aforementioned studies, Ottenwälder et al. [94] take into account triggering events that are related to the user. The placement solution for geo-distributed Complex Event Processing (CEP) involves the strategic planning and potential execution of operator migrations in order to accommodate changes in user location.

Proactivity

Another crucial aspect pertains to the time window taken into account when devising strategies for adapting to changing circumstances. Reactive approaches involve the analysis of historical data to inform decision-making regarding adaptation, potentially resulting in responses to alterations in conditions. On the contrary, proactive tactics involve making decisions based on a restricted future time frame, with the aim of preemptively adjusting applications. The attainment of proactive solutions is evidently challenging due to the necessity of forecasting future working conditions, and their effectiveness is contingent upon the precision of such prognostications. Hence, it is not unexpected that the majority of current methodologies depend on responsive adjustment strategies, such as those outlined in [3, 42, 77, 83, 109, 113, 140]. The article cites threshold-based heuristics as instances of reactive policies. These policies are characterised by the activation of actions in response to threshold violations. Typically, such violations are assessed against the most recent monitoring data, such as the average resource utilisation over the preceding minute.

Several studies have suggested proactive adaptation strategies, utilising various techniques (e.g., [13, 47, 49, 53, 64, 106]). One of the primary obstacles in formulating proactive adaptation solutions pertains to the prediction of application load in the immediate future, particularly with regard to scaling strategies for operators and infrastructure. Imai et al. [53] depend on ARMA and ARIMA forecasting models. Hidalgo et al. [47] construct a model of the incoming load via a Markov chain. Runsewe and Samaan [106] employ a more intricate state-based approach, wherein multi-layer hidden Markov models are taken into account. Buddhika et al. [13] devised a bespoke data structure, denoted as the prediction ring, to monitor the arrival of data streams and forecast the utilisation of resources. Prediction rings bear resemblance to circular buffers in their utilisation of exponential smoothing for the purpose of updating arrival rate approximations over a given period. The utilisation of the data structure is also employed for the computation of a certain task.

The utilisation of the data structure is also employed for the computation of an interference score, which serves to measure the magnitude of the effect that the placement of an extra operator instance would have on other instances located on the same machine. In addition to the matter of prediction, there exists a concern pertaining to the proactive management and optimisation of adaptive measures. Farahabady et al. [36] utilise model predictive control, a control-theoretic methodology that leverages a model to anticipate the forthcoming behaviour of a system within a restricted prediction horizon. De Matteis and Mencagli [26] employed model predictive control in their study to regulate operator scaling and enhance a multi-objective cost function. This cost function takes into consideration QoS violations, resource utilisation, and adaptation overhead. Kumbhare et al. [67] suggest a lookahead optimisation strategy that employs a predictive model to address an optimisation problem across a moving time window in order to regulate auto-scaling. The authors analyse a utility maximisation problem subject to a constraint that ensures a minimum application throughput.

Chapter 3

Stream Workload Parallelisation and Elasticity

Chapter Summary

The underlining objective of this research chapter is to identify more efficient ways of moving data within distributed architecture and modelling relevant parameters. This research leverages stateful operations, and wordcount workload, which requires the collocation of tuples with similar characteristics and produces an aggregated result.

3.1 Introduction

Stream processing parallelisation and elasticity enables streaming engines to support high quality of service in processing large amount of data while ensuring high throughput and low latency [58]. To maximise stream processing engine throughput and improve the utilisation of computational resources, workloads are commonly partitioned and processed concurrently by multiple instances of logical operators. The source operator sends data in tuples for processing based on a global partitioning strategy [50]. Tuples with the same key are received and processed by the same stateful operator. Stateful operators refers to operators that has a memory space to store results called states. For example, a state can be used to record the tuples or counts of words in a sliding window. When a key is reassigned to a different operator instance, its state is usually migrated to ensure the correctness of the computation outcome. [35].

Workload variance and skewness are common events in distributed stream processing. Workload skewness refers to an uneven or imbalanced distribution of workload across different resources or components within a system or application. It indicates a situation where certain resources or components are overloaded or underutilized compared to others, leading to an imbalance in the processing or resource utilization patterns. Workload skewness can impact system performance, efficiency, and resource allocation, and it is often a concern in distributed systems or parallel computing environments. [35]. Workload skewness also affects key-based partitioning in streaming processing that enables effective tuple distribution over threads of workers in a local operator instance [62]. According to Wang et al., operator scaling and load balancing are

resource-centric solutions for adapting to workload fluctuations. This is because executors are bound to specific resources, and elasticity can be achieved by dynamically repartitioning keys across executors [126].

Figure 3.1, presents an example to illustrate the potential problem that workload skewness can cause. In this example, there are three logic operators in the pipeline, `operator1` is the source operator instance, `operator2` are the FlatMap operators, and `operator3` is the sink. The FlatMap `operator2` has three parallel instances workers (task 1, 2 and 3) running while the source and sink has one each. The number of incoming tuples that arrives at task 1 in operator 2 is two times more than those that arrived at task 2 and 3. Due to this distribution skewness, research carried out by Fang et al., [35] assert that the efficiency of the FlatMap operator may not be optimal if the system succeed in allocating the tasks to the FlatMap nodes in a balanced manner. The potential for backpressure to occur as a result of increased latency in task 1 of the FlatMap instance within `operator2` may necessitate a reduction in processing speed by the source operator. The sink operator may encounter a scenario where it needs to await the intermediate outcomes generated by the FlatMap `operator2`.

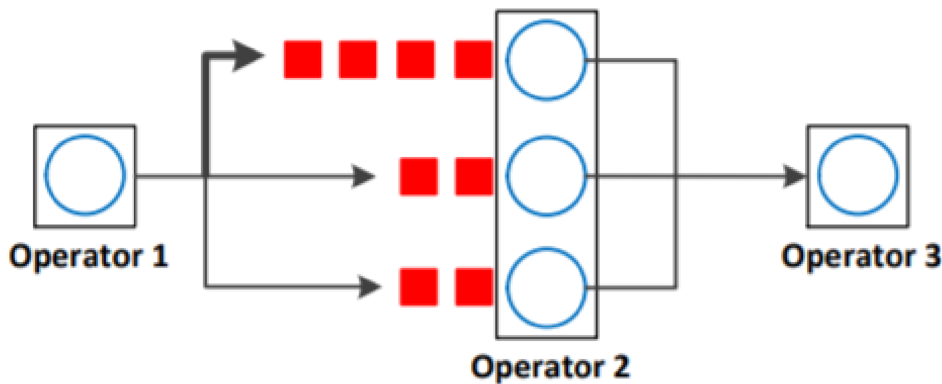


Fig. 3.1 A stream topology consisting of one Sources operator, three FlatMap operators and one Sink operator. This illustrates the potential problem of workload imbalance within operators in real distributed stream processing engine [35]

3.2 Parallel Stream Processing

If more data comes in than an operator can handle, the operator's input queue grows, resulting in data item queuing delays. In addition, back-pressure may impede operators who have to wait until the bottleneck operator further down the line finishes processing its input list. The stream processing system may not be able to meet its QoS goals. As it handles many pieces of info at once instead of one at a time, parallel stream processing reduces queuing latencies and improves throughput. Stream processing systems can parallelize their processing in a variety of methods, We shall be discussing the stream processing system properties that influence the parallelisation potential of the system and also the parallelisation techniques.

3.3 Properties of Parallel Stream Processing Systems

This section discusses two major types of streaming processing and their distinctive characteristics. General stream processing system and CEP. The properties of a streaming processor influence the parallelisation potential of the streaming processor system and the operations supported by the system.

General Stream Processing Systems

Stream processing systems usually run processes on groups of data objects in a continuous way. Every operator has the ability to generate a stream of output data items, which can be used as input for another operator, or to share the results with external applications by storing them in a data storage or transmitting them to compatible sinks. Data Stream Management Systems (DSMS), which execute continuous queries on data streams, are integral components of broader stream processing systems. A continuous query is a query that is consistently executed on a dynamically evolving dataset, such as a data stream. This differs from traditional database applications that execute various queries on a static dataset. Aurora [2] and TelegraphCQ [20] are common examples of DSMS systems that don't work in parallel. Apache Flink [15] and Apache Storm [15] are two examples of contemporary GP systems.

Complex Event Processing System (CEP)

CEP systems are stream processing systems that are made to find trends in events and get higher-level information from them. "moisture and low temperature $< 0^{o}$ " can represent "Snow". Input streams for CEP systems are made up of events that are set off by views of the outside world. The users who run CEP systems look through the streams of data for sequences of events that match the patterns. When an operator detects a pattern, it generates an output event, often referred to as a complex event, such as "Snow." In parallel execution of a Complex Event Processing CEP system, the input stream must be split into multiple parallel streams so that patterns can still be found. Automated stock trading [10, 79], financial fraud detection [5, 131], and traffic monitoring [78] are typical applications of CEP pattern detection. There are general-purpose stream processing systems, such as Apache Flink [50], that provide CEP functionality as a library.

3.3.1 Sub-Stream Processing

Operators often utilize sub-streams to parse their input streams, employing keys or windows to divide the streams into smaller segments. When using key-based extraction, data items are sorted based on a key value provided by each item, resulting in a sub-stream for each unique key value [134].

In a window-based environment, sub-streams are created using a strategy known as a "window." A window policy determines the size and sliding period of the window. The window size can be determined by the number of data items or the duration of a period, among other factors.

The window slide indicates how frequently a new input window can be initiated and can also be based on time, count, or a predicate [133, 78]

3.3.2 Infrastructure Model

The choice of the underlying framework for running a stream processing system significantly impacts the system's performance and determines the necessity and feasibility of parallel processing. This primarily relates to the specific characteristics of the processing nodes and memory design [104].

Different types of infrastructure serve as options for stream processing systems, including single-node setups, clusters, cloud-based solutions, and fog-based solutions. The scalability of single-node solutions is limited by the capacity of the underlying machine. Stream processing systems can only scale up, or add more threads, if the node size allows. Clusters offer a constant number of processing modules. Scalability of the stream processing system is constrained by cluster size. For both single-node and cluster systems, a common optimisation goal is to use as many resources as possible. Stream processing systems operating in the cloud have greater scalability. They must choose between cost and efficiency

The communication between cloud and stream sources can result in a high latency, which can harm stream processing applications with low latency requirements. In addition to its limited scalability, a fog infrastructure typically offers low communication latency because processing can be performed close to the sources. Across all types of infrastructure, the presence of diverse processing nodes can impact the processing speed of a stream processing application [104].

Memory Achitecture. Administrators typically engage in asynchronous communication through message passing. Certain systems facilitate shared memory operations for communication and state management, particularly when multiple operators are hosted on the same system. This approach decreases communication and state-migration cost but introduces additional costs associated with access synchronization [104].

3.4 Operator Parallelisation Methods and Limitations

In this section, we introduce two stream processing parallelisation concept: The data parallelisation and task parallelisation. Furthermore, we present their opportunities and drawbacks of these methods.

3.4.1 Data Parallelisation

Data parallelisation runs multiple copies of an operator, called "instances," on different parts of the original data at the same time. The degree of parallelisation of the operator is shown by the number of examples. Input streams must be partitionable to enable data parallelisation. The raw stream is split into smaller streams by a splitter component. Depending on the method for splitting, the splitter could be a separate process or part of the operator instances. A merger merges the outputs of the different instances into a single stream and, if necessary, makes sure

that the outputs are sent to the next operators in the right order. Figure 3.2 depicts a data-parallel stream processing operator's fundamental architecture.

In data parallel stream processing systems, stateful operators require special attention. The input stream should be divided among operator instances so that each instance can maintain its own state. This prevents operator instances from interfering with each other.

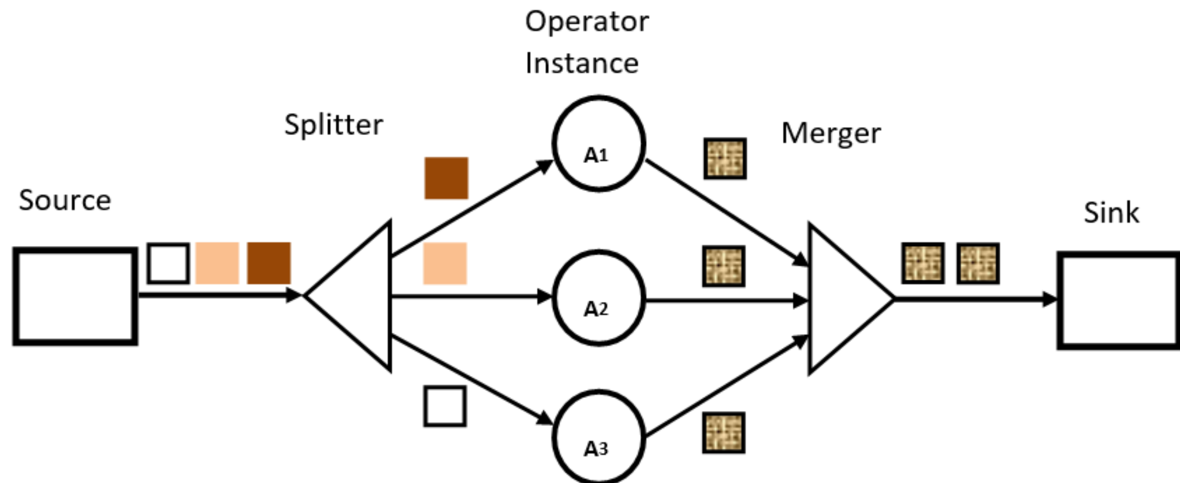


Fig. 3.2 Data Parallelisation Fundamental Architecture [104]

Following, is a description of the three common dividing strategies used in data parallelisation.

Key-based Splitting

In key-based splitting, the splitter uses keys to divide the data stream. These variables are the properties of data items. Each instance of an operator is in charge of a portion of the whole set of keys. If operators have no state or keep track of state for each key, then key-based splitting can be used. In Figure 3.3, we can tell the value of each key by its colour. Each operator instance keeps the state of a key range that has been given to it and does not combine. Similar key-range data items are forwarded to the same operator instance by the splitter.

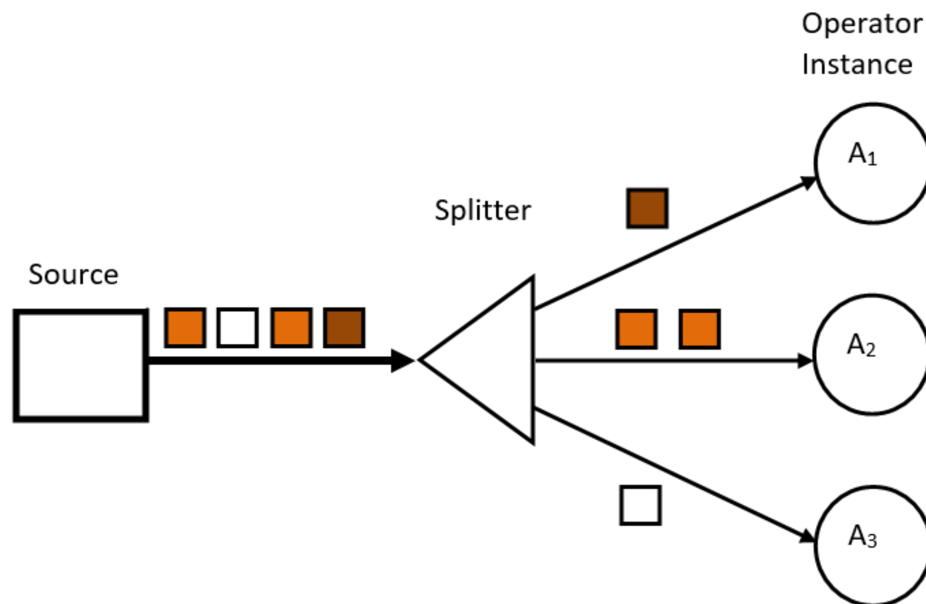


Fig. 3.3 Key-based Splitting [104]

Window-based Splitting

In the context of window-based splitting, the splitter is responsible for dividing the input stream into subsequences, specifically referred to as windows, which consist of data items. Subsequently, the windows are allocated to the instances of the operator, as depicted in Figure 3.4 below. Li et al., [68] distinguish between two types of context, namely backward context and forward context, in order to ascertain the appropriate windows for a particular data item, depending on the window policy. The backward context of a data item, denoted as e, encompasses all information regarding preceding data that has been received by the operator. On the other hand, the forward context pertains to information derived from subsequent data in the input stream following e. In the context of backward methods, when a data item "e" is being processed in the splitter, it is possible to promptly start a new window and assign it to an operator instance, if appropriate. Nevertheless, the implementation of this approach may not be practical if the window policy necessitates the inclusion of forward context.

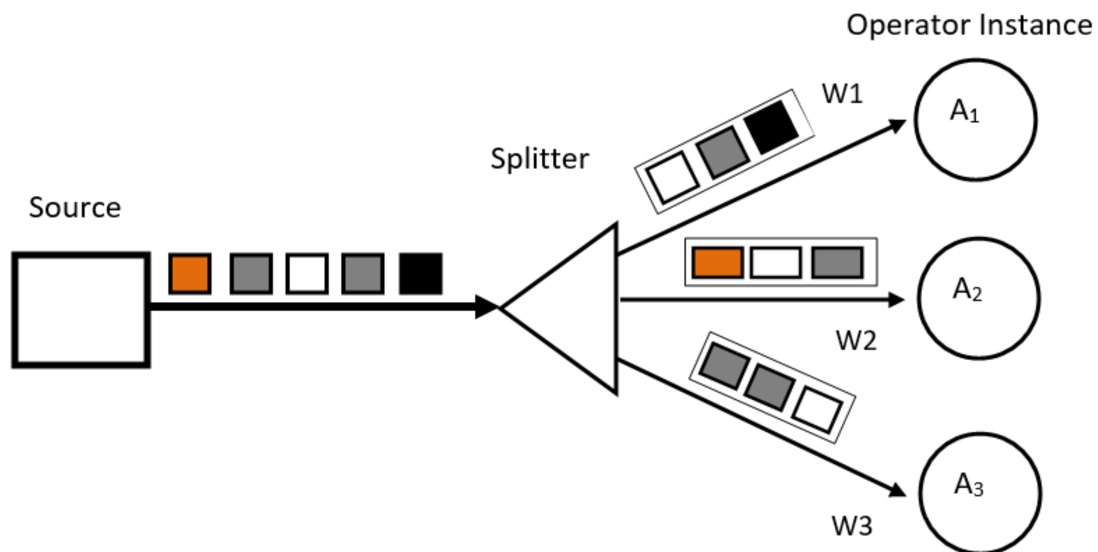


Fig. 3.4 Window-based splitting is the process of dividing the input stream into windows consisting of successive data items. Every instance of the operator is responsible for processing a subset of all the windows. [104]

Pane-based Splitting

Window-based partitioning can lead to higher communication cost when dealing with overlapping windows. Moreover, it is often unnecessary to process each window separately, and computations that produce overlapping windows can be reused by multiple windows.

In pane-based splitting, the splitter splits the input stream into groups of data times that don't overlap, known as panes. As illustrated in Figure 3.5, each window pane belongs to multiple windows. Following parallel processing of the panes, the merge operation combines the results based on the corresponding windows to which each pane was assigned. For example, when figuring out the highest temperature in a 1-minute window with a 10-second shift, the data stream can be split into 10-second chunks. At the same time, the operator instances figure out the highest number in each pane and send that information to the merge. The merger figures out the maximum value of a certain window by figuring out the maximum value of each of the window's six parts.

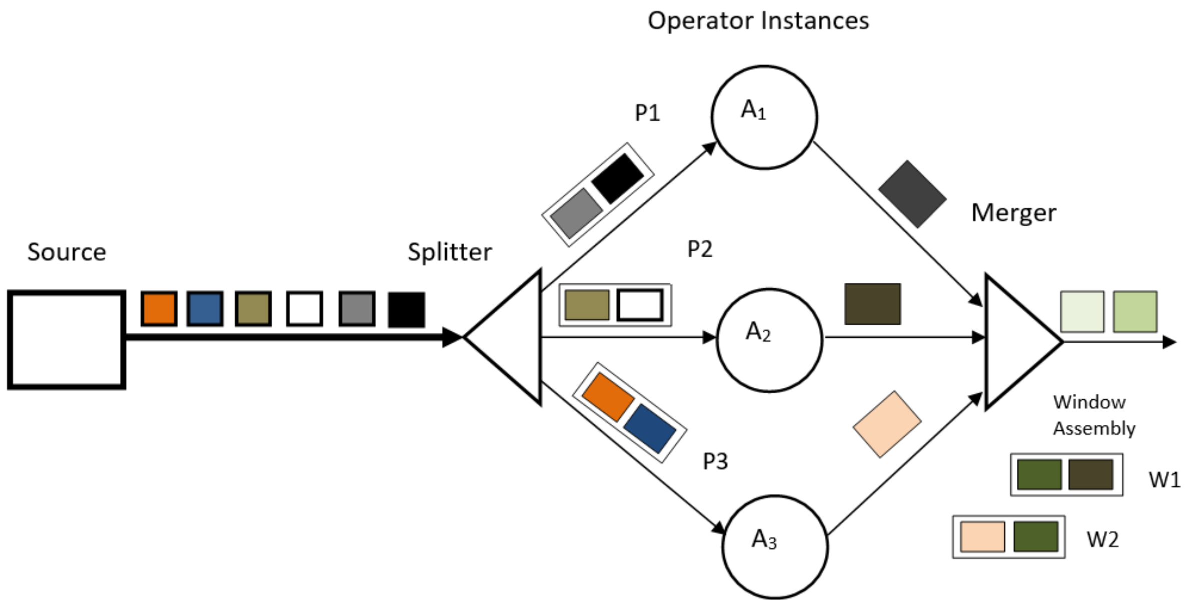


Fig. 3.5 Pane-based Splitting [104]

3.4.2 Task Parallelisation

In task parallelisation, the stream processing system simultaneously executes several action on the same data stream. Figure 3.6 shows how this works. A repeater makes copies of the data it receives and sends them to operators A, B, and C. The merger combines the sources of data into a single stream of output. Task parallelisation makes it possible to use pipelines, in which the result of one operator is the input for the next operator. In stream processing systems, pipelining breaks up big operators into smaller ones that can work at the same time. Multiple operations (i.e., tasks) must be able to execute concurrently on the same input as a prerequisite for task parallelisation. Thus, applicability is dependent on the specific application.

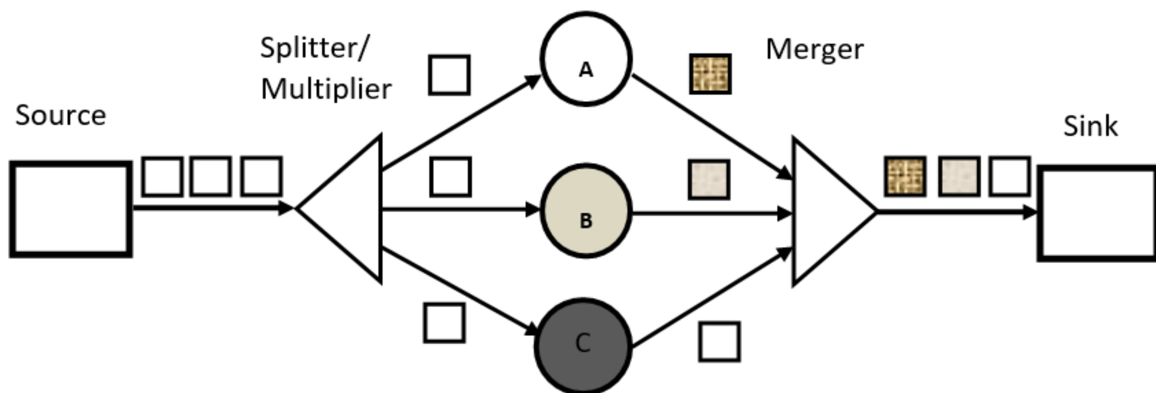


Fig. 3.6 Task Parallelisation Fundamental Architecture [104]

3.4.3 Limitations of Operator Parallelisation Methods.

Having discussed the advantages of each parallelisation method presented above, we now look at each method's drawback.

Limitations of Data Parallelisation.

The most difficult aspect of data parallelisation is achieving a well-balanced burden and delivering data items in order to downstream operators. There are three limitations to key-based data parallelisation: expressiveness, scalability, and load balancing [104]. The amount of different partitions, or key-value pairs, limits what you can do with the data and how big it can get. For example, when looking for patterns in the stock market, the key-based parallelism is limited by the amount of different company symbols. Additionally, key-based splitting is only applicable when the data includes a corresponding key. If the keys aren't spread out evenly in the input data, you have to do specific load balancing between the operator instances. It is possible for one instance to experience a high arrival rate while others remain dormant [103].

Window-based division may raise communication costs. If individual operator instances are assigned distinct overlapping windows, the data items within the overlap of these windows need to be duplicated and distributed to each respective operator instance. To minimize this overhead, window aggregation consolidates multiple overlapping windows into a single operator instance [80, 27]. This approach offers more flexibility in the input data structure since it doesn't necessitate the use of keys for division, as seen in window-based splitting.

For pane-based division, it is essential that the operator function can be split into two stages. The first stage involves processing individual panes, while the second stage involves combining the results from multiple panes to obtain the window's overall results. Nonetheless, It doesn't need associative operations instead of key-based splitting. Instead, the system keeps processing the data in the panes in sorted groups, keeping the order of the items.

Limitations of Task Parallelisation.

Task parallelisation, including pipelining, is a well-established parallelisation technique, but it has three main drawbacks [104] When each operator must receive the entire input stream, it can increase network traffic. Second, if the processing speeds of the operators vary, there is the possibility of a burden imbalance. Third, task parallelisation has limited scalability: An operator can only be divided into a restricted number of sub-operators before reaching an atomic operation or, more commonly, before the advantages of additional parallelisation are outweighed by the costs associated with distributing the processing.

3.5 Flink FlatMap Operator Distribution Mechanism

Flink uses a parallel data processing model to distribute data among different FlatMap instances. When you apply a FlatMap transformation on a DataStream, Flink internally partitions the data stream into multiple substreams based on the parallelism of the job. Each substream is then processed by a separate instance of the FlatMap operator. The partitioning is based on a key grouping or round-robin distribution. If a key grouping is used, Flink ensures that all records with the same key are processed by the same FlatMap instance to preserve the ordering of the records. On the other hand, round-robin distribution sends records in a round-robin fashion to different instances of the FlatMap operator.

The parallelism of the FlatMap operator determines the number of instances that process the data stream in parallel. You can set the parallelism by calling the *setParallelism()* method on the FlatMap operator. Flink also supports data shuffling, which means that records with the same key can be sent to different parallel instances of the FlatMap operator. This is useful when the processing of a record depends on data from other records with the same key, which are processed by different instances of the FlatMap operator. In this case, Flink ensures that all records with the same key are sent to the same downstream operator for processing.
Flink provides different levels of operator parallelism to optimise the execution of data processing pipelines. These levels include the execution environment, client level, and system level.

i. Execution Environment Parallelism: This level of parallelism is set at the beginning of the execution of a Flink job and is determined by the resources available in the environment. The execution environment parallelism is set by the Flink cluster manager and is based on the number of task manager slots and the available resources in the cluster. This level of parallelism is fixed throughout the execution of the job and cannot be changed dynamically.

ii. Client Level Parallelism: This level of parallelism is set by the Flink client when submitting a job to the Flink cluster. The client level parallelism specifies the degree of parallelism for each operator in the job and can be set using the setParallelism() method when defining the job. This level of parallelism is defined at the job level and can be adjusted dynamically based on the input data size or the available resources in the cluster.

iii. System Level Parallelism: This level of parallelism is determined by Flink's internal system parameters, such as the degree of network and disk I/O parallelism, the number of available threads, and the buffer size. The system-level parallelism is managed by Flink's runtime system and is used to optimize the performance of the job by minimizing the overhead of inter-operator communication and maximizing the utilization of available resources.

By combining these levels of parallelism, Flink can optimize the execution of data processing pipelines for different input data sizes, available resources, and performance requirements. Flink's flexible parallelism model allows users to fine-tune the degree of parallelism for each operator in the pipeline, enabling efficient processing of large-scale data in real-time.
There are differences between these levels of operator parallelism in the way the topology is deployed.

i. Execution Environment Parallelism: This level of parallelism is fixed throughout the execution of the job and is determined by the resources available in the environment. When the job is submitted, Flink's cluster manager deploys the job's topology across available task manager slots in the cluster based on the execution environment parallelism. The task manager slots are distributed across the available nodes in the cluster, and each slot runs a single instance of the operator with a fixed degree of parallelism.

ii. Client Level Parallelism: This level of parallelism is defined at the job level and can be adjusted dynamically based on the input data size or the available resources in the cluster.

When the job is submitted, the Flink client specifies the degree of parallelism for each operator in the job using the *setParallelism()* method. Flink's runtime system deploys the job's topology across available task manager slots based on the client level parallelism, and the system automatically adjusts the degree of parallelism based on the available resources in the cluster.

iii. System Level Parallelism: This level of parallelism is determined by Flink's internal system parameters and is used to optimize the performance of the job. Flink's runtime system manages the system-level parallelism by optimizing the buffer size, the degree of network and disk I/O parallelism, and the number of available threads based on the input data size and the available resources in the cluster. The system-level parallelism is not explicitly defined by the user and is managed by Flink's runtime system.

In summary, the differences between these levels of operator parallelism lie in how the topology is deployed across the available task manager slots in the Flink cluster. The execution environment parallelism is fixed and determined by the cluster manager, the client level parallelism is adjustable by the user, and the system level parallelism is managed by Flink's runtime system to optimize performance.

3.6 System Design

We present our experimental pipeline setup and the relationship among the various unit of our setup in Figure 3.7. These units are self-contained and hosted on Microsoft Azure, which makes it easy to scale the experiment. The term "easy to scale" refers to the ability to effortlessly grow computing resources without necessitating intricate integration. This setup architecture has four major areas: the data source (Wordcount workload generation), Streaming framework (Flink and DS2 scaling controller), Visualisation, and Datastore. The experimental objective is to evaluate the operators elasticity and operator's processing capacity. In this experiment, we adopt the horizontal scaling methodology by increasing the operators parallelism. This research adopts best practice benchmarks like wordcount. The benchmark must consider the diversity of workload to cover different type of application domain. We provide further details on the (4) fundamental processes in our pipeline.

i. Data Sources: input data is required for any stream processing engine and these data can be derived form either a message broker (eg. Kafka etc.), Realtime event data, event logs, reading from file, sockets etc. for this research I have a random string generator that continuously generates a set of sentences for the purpose of simulating a data source that feeds data to the streaming pipeline. The random sentence generator randomly generates a predefined set of sentences from a group of letters per seconds and persist the output to a local file.

By leveraging a combination of defined rules like the length of the sentence, etc., a random sentence generator can produce a wide array of sentences that, while not necessarily

meaningful or coherent, appear grammatically sound and structurally plausible to a certain extent.

ii. Computation: My computation is powered by Apache Flink as illustrated in Figure 3.7. The computation evaluates the stream data ingested continuously to detect the occurrence of each strings over a short period of time. Here we simulate a simple word count that records the number of times a string occurs. To achieve this, We do three things: First, we specify that my stream is a keyed window using the command `.keyBy()`. Specifying a keyed window ensures that all attributes of my incoming stream will be used as a key, thereby allowing the computation to be done in parallel by multiple task as against a non-keyed stream where the stream will not be split, rather all the window logic will be processed by a single task. Secondly, I define a window assignor which assigns all incoming stream element to one or more window. Windowing in stream processing basically determines the time frequency in which data is received for computation. Lastly, we apply a rolling aggregate to the keyed stream `(.sum ())` to aggregate the value of each key and produces a key value pair `(word, 1)`.

iii. Data Storage: All processed stream are written to a local file called the rate file. The metadata contained in this file include the worker name, instance id, number of instances, time, true processing rate, observed processing rate, true output rate and observed output rate.

iv. Visualisation: This research uses `ggplot2` graphical representation from `R` to visualise the results. We also leverage `R` to analyse the performance of the streaming engine, study the pattern and trends when the stream computation is subjected to different conditions and also to help derive experimentation results. The objective is to extract the pertinent performance data and visually represent them through graphs.

As depicted in Figure 1, each constituent of our experimental pipeline exhibits an interrelation that functions as an input to another.

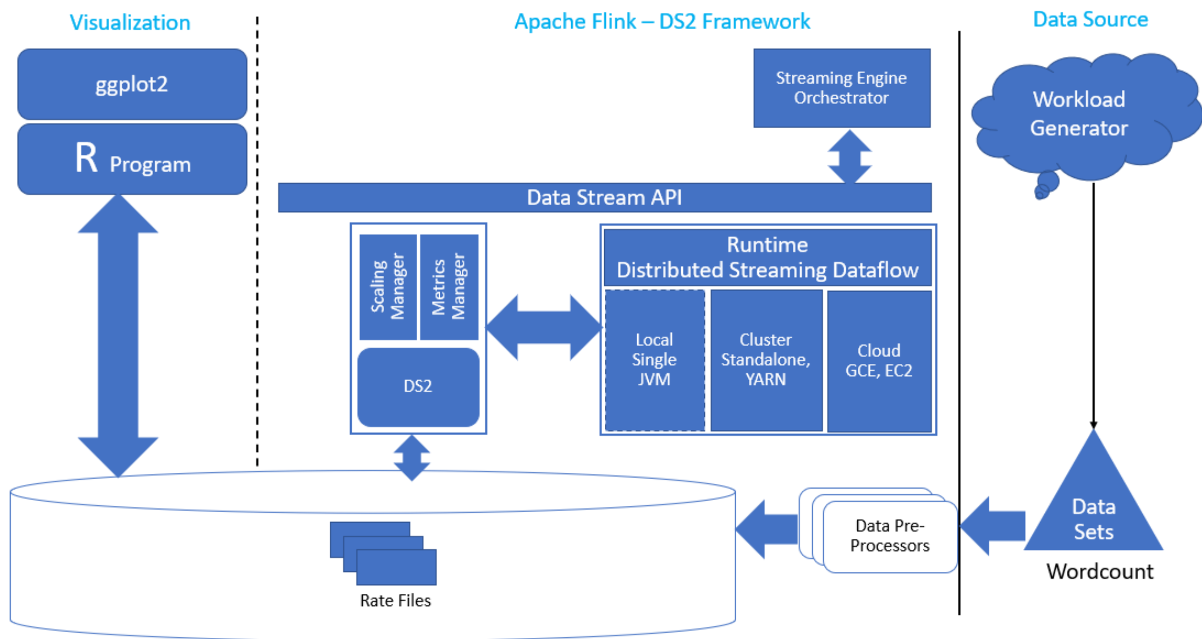


Fig. 3.7 Experimental System Architecture (Chapter 3)

3.6.1 Experimental Setup

In carrying out these experiments, we seek to know how much work is done by each flatmap operator instance, the imbalance in terms of data distribution across the different operators, does it increase or decrease the utilisation of the system, are the operator instance heavily loaded.

VM and Software Configuration

Tables 3.1, 3.2 and 3.3show the hardware and software configuration parameters as well as the Flink configuration used for this experiment.

Table 3.1 Standalone hardware configuration (Chapter 3)

Hardware	Configuration
CPU	Intel(R) Xeon(R) E5-2673 v4 @ 2.30GHz
CPU Cores	8 vCPU(s)
Memory	16GB
Disk	1TB

Table 3.2 Software configuration (Chapter 3)

Software	Version
OS	Ubuntu 18.04.3 LTS (Bionic Beaver)
Flink	1.4.1
R	Rstudio 2021.09.0 Build 351
Intellij IDEA	2019.3.3 (Ultimate Edition)

Table 3.3 Apache Flink Configuration

Configuration Parameters	Values
taskmanager.numberOfTaskSlots	3
state.backend	filesystem
state.backend.fs.checkpointdir	file:///path/to/savepoints
jobmanager.heap.mb	2048
taskmanager.heap.mb	2048

- `taskmanager.numberOfTaskSlots`. This refers to a configuration parameter that specifies the number of parallel task slots available on a Flink TaskManager
- `state.backend` . This refers to a configuration parameter that determines the type of state backend used for storing and managing the state of streaming applications.
- `state.backend.fs.checkpointdir`. This refers to a configuration parameter that specifies the directory where the checkpoints of Flink's state backend are stored on the file system.
- `jobmanager.heap.mb`. This refers to a configuration parameter that determines the maximum memory allocated to the JobManager in the Flink cluster
- `taskmanager.heap.mb`. This refers to a configuration parameter that determines the maximum memory allocated to each TaskManager in the Flink cluster

Based on the above, define two terminology that are frequently used within this chapter.

trueProcessingRate

This signifies the maximum potential number of records an operator instance can process per unit of a useful time. Intuitively, this calculates the capacity of an operator instance. Please note that `useful time = duration - waiting time` [58].

ObservedProcessingRate

This signifies the number of records an operator instance processes per unit of observed time. Contrary to the trueProcessingRates, the observed rates are measured by simply counting the numbers of records processed by an operator instance over a unit of observed time [58].

i. **Experiment 1.** This experiment seeks to evaluate the true and observed processing rate of the FlatMap operator. The present study involves the implementation of a basic topology consisting of one source operator, a FlatMap operator, and one Sink operator. We inject a source rate of 100,000 records per seconds and we measure the operators true and observed processing rate.This experiment was ran for 5 hours. Additionally, the percentage disparity between the true and observed processing rate is quantified in order to comprehend the variation in the streaming processing capability.

ii. **Experiment 2.** Building upon the preceding experiment, the experiment delves into the notion of operator rescaling. The process of rescaling generally entails the dynamic redistribution of workload among multiple operator instances or the modification of the number of instances themselves. In this study, three distinct executions were conducted. The initial implementation involves a single FlatMap, while the subsequent ones incorporate two and three FlatMaps, respectively. The process of introducing the new FlatMap was carried out on an hourly basis through the rescaling of the application and modification of the parallelism. A single source operator and a single sink operator are maintained. The source data injection rate is set at 150,000 records per second. The duration of the experiment is three hours.

iii. **Experiment 3.** The experiment was conducted for a duration of one hour, during which there were five instances of redeployment and rescaling of the source injection rate as well as the Source and FlatMap operators. As presented in Table 3.4, the initial source injection rate is 80,000 records per second, and subsequent redeployments result in a doubling of this figure. The source and FlatMap operator are subject to scaling within the range of 1 to 3 while ensuring the use of a singular Sink operator.

    Drawing from the results of the preceding experiments, the present study aims to investigate the correlation between operator instance parallelism and the distribution of tasks across multiple operator instance. It is anticipated that with the introduction of additional source operators and an increase in the rate of source data injection, the FlatMap operator will encounter difficulties in managing the influx of data, resulting in backpressure. Therefore, provisions have been made to accommodate the rescaling of FlatMap operators.

Table 3.4 Scaling Operator Parallelism Experiment

Source Injection Rate	Source Operator Parallelism	Flatmap Operator Parallelism
80,000	1	1
160,000	2	2
800.000	3	3
1,600,000	3	3
8,000,000	3	3

The insights obtained from the aforementioned experiments, namely Experiment 1, Experiment 2 and Experiment 3 will facilitate an enhanced comprehension of the techniques employed to gauge the processing rate and utilisation capacity of operators. We also learn that increasing the source operator greater than one has a multiplicative effect on the source injection rate.

3.7 Summary of Experimental Results

Figure 3.8, displays the outcomes of the first experiment, presenting both the true and observed processing rate of FlatMap operators over time in a single plot. The source injection rate is set at 100,000 records per seconds. in Figure 3.9, we measure the percentage variance between the true and observed processing rate. Our findings indicate a variance of approximately 30 percent, implying that the operators possess the potential to enhance their present processing capacity by 30%.

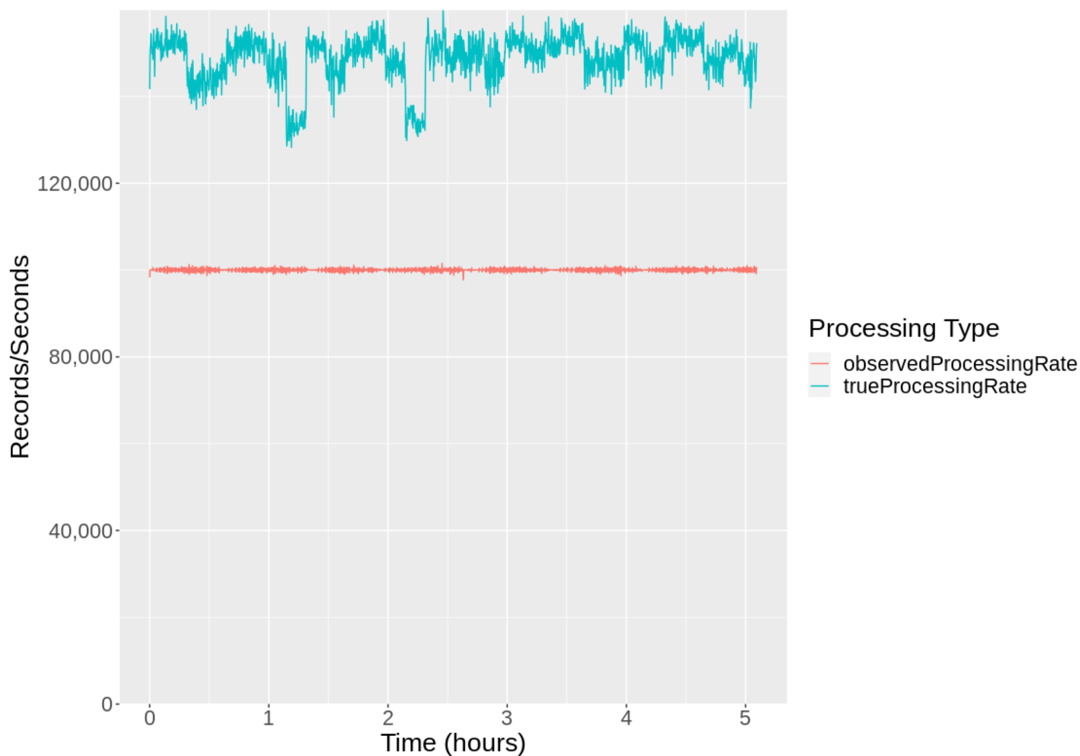


Fig. 3.8 Measuring True and Observed processing Rate (Experiment 1)

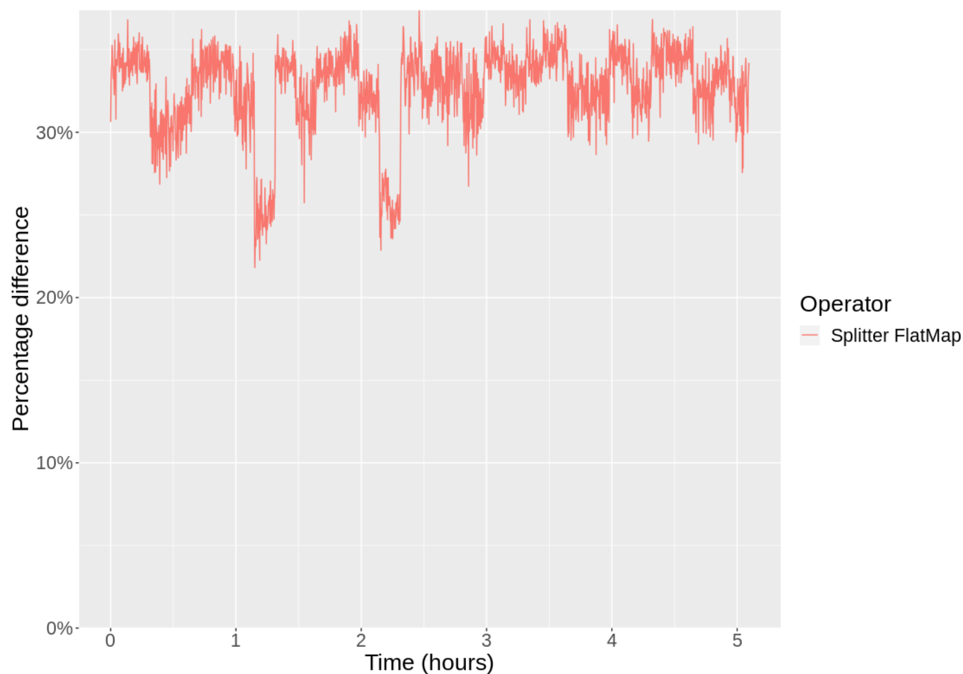


Fig. 3.9 Percentage difference between the True and Observed processing rate (Experiment 1)

The percentage difference provides a relative measure of how much the true processing rate metric differs from the observed processing rate in percentage terms. To calculate the percentage difference between the true and observed processing rate, we used the following formula:

$$\texttt{Percentage Difference} = \texttt{True} - \texttt{Observed}/\texttt{True} * 100 \tag{3.1}$$

The observed processing rate is based on the actual data processed and output within a given time frame, including any periods of waiting or delays. In streaming applications, waiting times can occur due to various factors, such as network latencies, uneven data arrival, and variations in data processing complexity. While the true processing rate, on the other hand, represents an ideal scenario where the operator processes data continuously without any waiting. It is a measure of the operator's capacity which dependent on the underlining hardware infrastructure, assuming perfect conditions and continuous data availability. This is the reason for the variation we see between the two measurement.

Figure 3.10 illustrates an example of the summation of the ObservedProcessingRate interpolated values for three operator instances over a three-hour period. The experiment result shown in Figure 3.10, is the aggregated number of records processed by each FlatMap operator instance per unit of observed time. This is grouped by timestamp and operator. This experiment displays the outcome of our second experiment (Experiment 2). The experiment contains three deployment with one FlatMap operator in the first deployment, and the introduction of a second and third FlatMap operators by changing the parallelism in the second and third deployment, respectively. The source injection rate is 150,000 records per seconds and was constant. As illustrated, The data stream exhibits a notable decrease followed by subsequent recovery, which has been observed to occur on multiple occasions. The observed data imbalance pattern can be attributed to the source data injection or node-level factor that are external to Flink.

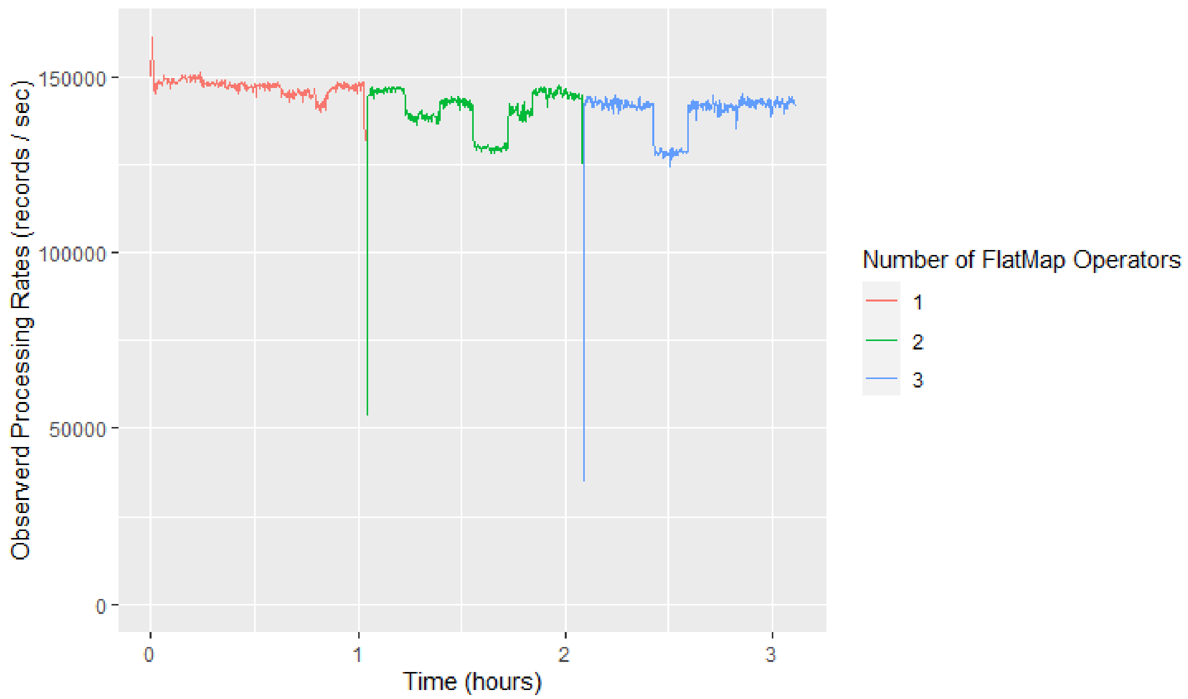


Fig. 3.10 Summed Value of Observed Processing Interpolation Records Over Time (Experiment 2)

Furthermore, we show in 3.11 the number of records processed by each FlatMap operator instance by measuring the observed processing rates. Consistent with our third experiment, this experiment ran for three hours, changing parallelism to introduce a new FlatMap operator every hour and maintaining a constant source injection rate of 150,000 records per seconds.

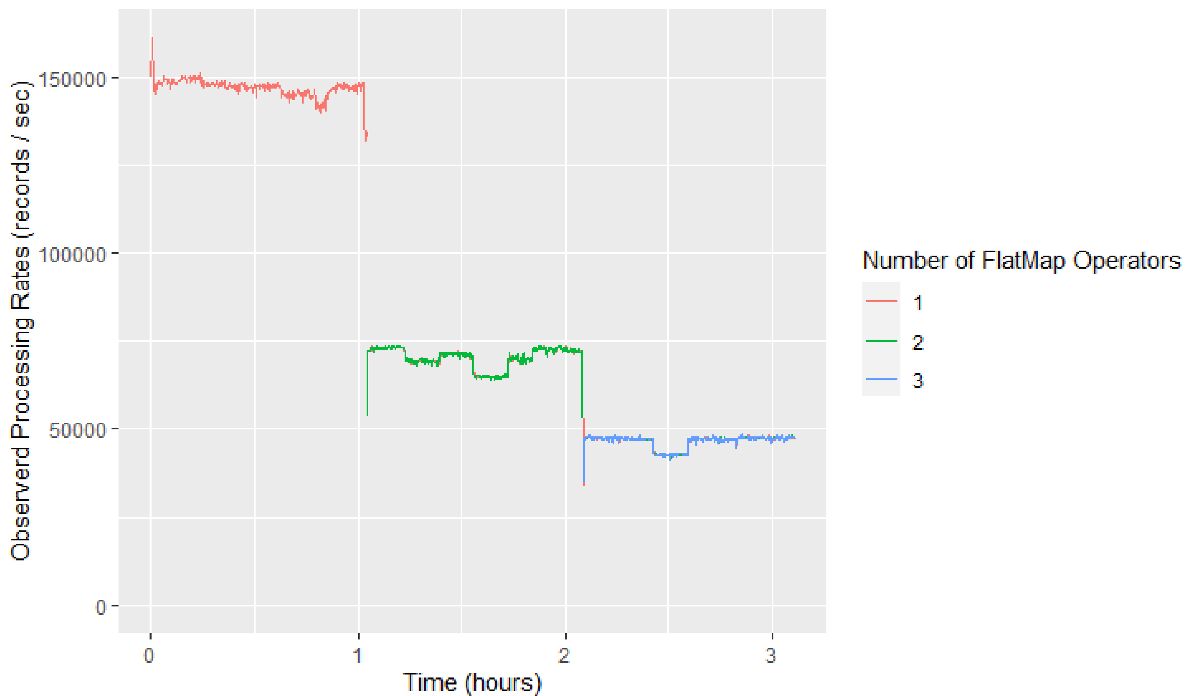


Fig. 3.11 Observed Processing Interpolated Records Over Time (Experiment 2)

The result in Figure 3.11 shows the observed processing rate for three different deployment. each having one, two and three FlatMap operators, respectively. We notice a declining processing rate in the deployments with more FlatMap operators despite increasing the FlatMap operator parallelism compared with the deployment with a single FlatMap. This behaviour is caused because we are using a single node deployment, by increasing the parallelism of the operator, we are not introducing additional compute power. Rather, Flink divides the offered load across multiple threads. in this case the different parallel operators.

During the analysis of the observed processing rate numbers, we detected slight temporal gaps between the data processing performed by the FlatMap operators. In order to obtain a comprehensive comprehension of the operators' behaviour during these temporal disparities, we have chosen to employ a linear interpolation mechanism to estimate all time gaps. As a result, the time allocated for processing the data by each instance of the operator was extended, leading to the production of Not Available (NA) values. The method of Linear Interpolation was employed to approximate the values that were expressly marked as missing NA.

Linear interpolation is a useful technique for understanding streaming data because it allows us to estimate values between known and unknown data points in a continuous manner. In streaming data, it is common for data points to arrive at irregular intervals and in different orders, making it challenging to obtain a continuous understanding of the underlying data point. Linear interpolation can help address this challenge by allowing us to estimate the value of the unknown data at any given point in time, based on the known values at adjacent points in time. This can help us to identify trends, patterns, and anomalies in the data, and make predictions about future values.

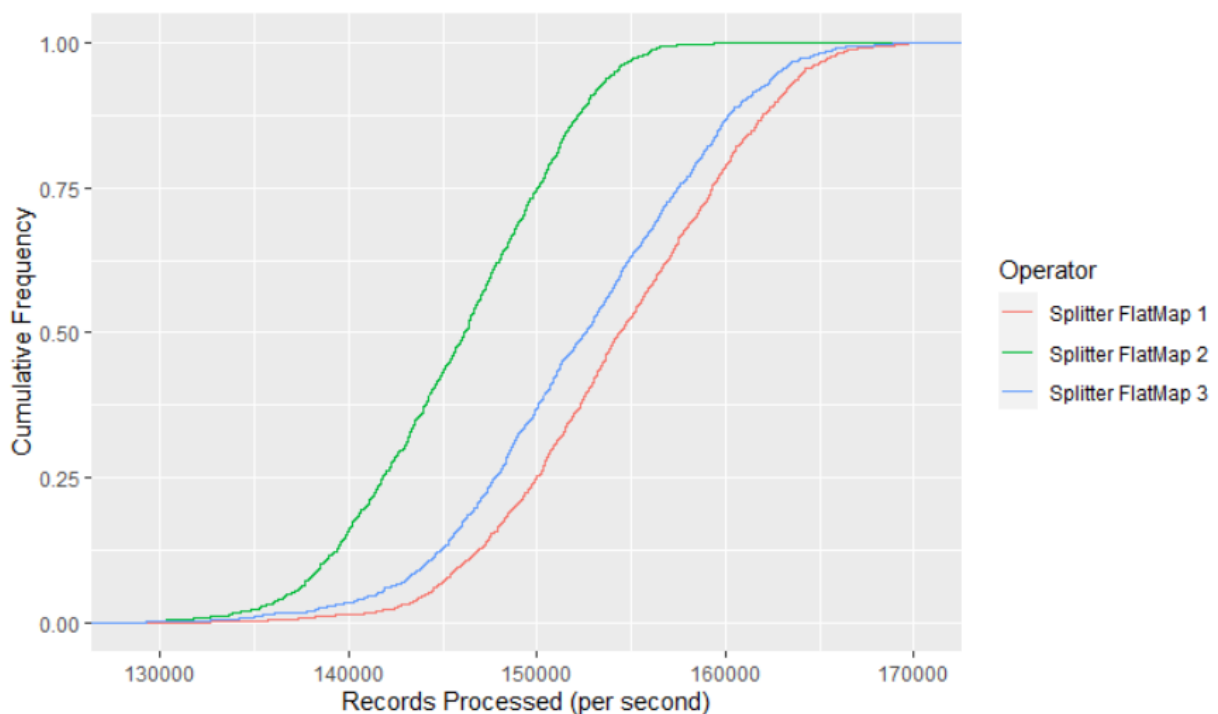


Fig. 3.12 Three Instance FlatMap Operator Processing Capacity (Experiment 3)

Figure 3.12 illustrates records processed by three FlatMap instance Flink deployment, displaying the outcome of our third experiment (Experiment 3). Scaling a stateful operator is common

requirement for streaming applications by adjusting the parallelism of by either increasing or decreasing the operator instances [50]. We plot the cumulative frequency over records processed per second as shown in Figure 3.12. This experiment is run on a standalone Flink infrastructure with the state back end set to `filesystem`. Further work would merit exploring the differing impacts of alternative state backends, such as `MemoryStateBackend` or `RocksDBStateBackend`. This experiment started with one Flatmap and subsequently scaled up at run time through the increase in parallelism introducing two more Flatmap operators, in a bid to increase throughput by enabling parallel data processing. According to [81], for keyed transformation, each operator is allocated a data with similar property type, while for non-keyed transformation (operator state), events are distributed in a round-robin fashion among operators. Based on Figure 3.12 above, there is a variation in the number of records processed by each FlatMap instance. Splitter FlatMap 1 processed more data followed by Flatmap 3 and 2. This result shows that the distribution of data is not done evenly or sequentially.

3.8 Conclusion

Our experimental findings highlight a crucial insight: the processing capacity of a flatmap operator is intricately linked to the hardware it operates on. Simply increasing the operator's parallelism does not always equate to a boost in its processing capacity. This is due to a multitude of influencing factors, including the availability of resources, data distribution patterns and application state size. There are a couple of reasons behind this phenomenon.

- Limited resources: The processing capacity of a flatmap operator is limited by the resources available on the task manager slots that host the operator. If the parallelism of the operator is increased without adding more task manager slots or increasing the resources available on existing slots, then the operator may not be able to fully utilize the available resources. This can lead to idle resources and a decrease in processing capacity.
- Uneven data distribution: If the input data is not evenly distributed across the parallel instances of the flatmap operator, some instances may have more work to do than others. For example, if some instances receive more data than others, they may become bottlenecks, leading to idle resources on other instances. In this case, increasing the parallelism of the operator may not lead to a corresponding increase in processing capacity.

Furthermore, our results suggest that Flink's task distribution is not consistent across various operators. Therefore, when assessing the combined record count handled by a uniform group of FlatMap operators in a stream processing pipeline, it's essential to individually evaluate each operator's processing capacity. One shouldn't assume that one FlatMap's processing speed matches that of another. This understanding is critical in our main goal of pinpointing more effective approaches for data transfer in a distributed architecture and modeling relevant parameters.

3.8.1 Future Work

In this sub section, we provide bullet points of future direction and next steps that could be useful and complementary to this body of research. This is listed in no specific order.

- Measure the impact of system utilisation on the level of imbalance between different operators' instance in a distributed stream processing pipeline. The first step towards achieving this will be to deploy this experiment on a system with more dedicated computational resource.
- An interesting future direction will be to investigate the task allocation mechanism being used by Flink in this kind of distribution setup with the aim of ascertaining the effect of the imbalance between each operator instance and throughput.

Chapter 4

Modelling of Time and Resource Requirements to Perform Rescaling

Chapter Summary

Autoscaling mechanisms promise to ensure QoS properties for applications while also maximising resource utilisation and operational costs for service providers. Despite the perceived benefits of autoscaling, maximising its potential is difficult due to various challenges associated with the requirement to precisely estimate resource usage in the face of significant variability in client workload patterns and trends [105]. Mindful of the challenges that come with decision-making on how to scale efficiently and when to scale, this research notes the work carried out in the DS2 project, which provides an automatic scaling system that strives to strike a balance between resource over-provisioning and on-demand scaling, by considering each operator true and observed processing capabilities regardless of the backpressure and other effects [58].

However, most auto-scaling systems do not consider the impact of scaling time. This research chapter seeks to contribute to this space. We argue that long scaling time especially in a rapidly varying workload environment, poses a potential challenge to a streaming application upon resuming from an auto-scaling procedure. The unanticipated workload characteristic could lead to poor scaling decision-making and suboptimal system performance.

Therefore, in this chapter, we analyse some metrics that can influence the scaling duration of a streaming application like state size and end-to-end checkpoint duration. We start by exploring the correlation between the state size and scaling duration, and then conduct some simulations to validate our argument that, the rescaling policy approach adopted by DS2 could lead to multiple rescaling once the state size of the application gets bigger.

Predictive models were developed and trained with all relevant associated variables collected from the experiment. These models were tested to ascertain their performance effectiveness, and the most efficient model was used to predict scaling duration based on forecasted state sizes. We believe this predictive model will provide auto-scaling controllers with more insight leading to a more robust and effective scaling decision-making process in a general use case [90].

4.1 Introduction

Stream processing technology powers numerous applications, such as continuous analytics, monitoring, fraud detection, stock trading, and mobile and network information management [9]. Dealing with applications characterised by high demand fluctuation in stream processing is difficult, as the distribution of data streams is constantly changing and unpredictable [141].

Workload variance and skewness are common features in distributed stream processing deployments. When a large amount of data is injected into a distributed system for processing, a change in the source data stream can affect the system performance resulting in Service Level Objective (SLO) violation [35]. Cloud hosting enables operators to scale horizontally or vertically based on the benefit of on-demand elasticity and economy of scale. However, without sufficient knowledge of the workload characteristics, making a scaling decision can become a complex task [106]. This could lead to over-provisioning or under-provisioning. Overprovisioning system resources can lead to increased costs and the underutilisation of computational resources. Meanwhile, under-provisioning can lead to suboptimal performance of the system, thereby affecting the QoS. Auto-scaling is automatically adjusting a system capacity or compute resource to maintain a steady or predictable performance level at the lowest possible cost.

A suitable scaling controller should provide SASO properties [58]. Whereas speculative scaling techniques that violate these could lead to incurring unnecessary costs due to sub-optimal utilisation of resources as a result of over-provisioning or under-provisioning, performance degradation due to frequent scaling action as a result of oscillation and low convergence resulting in an SLO violation or load shedding [58] Furthermore, a good scaling controller must be mindful of the resource availability and the scaling time. These factors, amongst other things, are an important element that must be contained in the scaling policy.

Mindful of the challenges that come with decision-making on how to scale efficiently and when to scale, this research notes the work carried out in the DS2 project, which provides an automatic scaling system that strives to strike a balance between resource over-provisioning and on-demand scaling, by considering each operator's true and observed processing capabilities regardless of the backpressure and other effects [58].

However, like DS2, most auto-scaling systems do not consider the impact of scaling time, especially in a volatile workload environment. Rather more attention is focused on mitigating the impact of over-provisioning of resources for temporary load spikes and under-provisioning during peak loads. We leverage Apache Flink's checkpointing fault-tolerant guarantees through the exactly-once guarantee semantics to stop, update the configuration, and restart a running application. This enables us to achieve two things: first, a failure-free execution and application state consistency and second, rescaling of the computational resources (vertical scaling) [143].

This chapter considers the impact of long rescaling duration during an auto-scaling interval which could lead to multiple rescaling of an application when the state size growth of the application is larger and unpredictable. State size is the measure of the entire content of the memory where the application state resides at a given point in time.

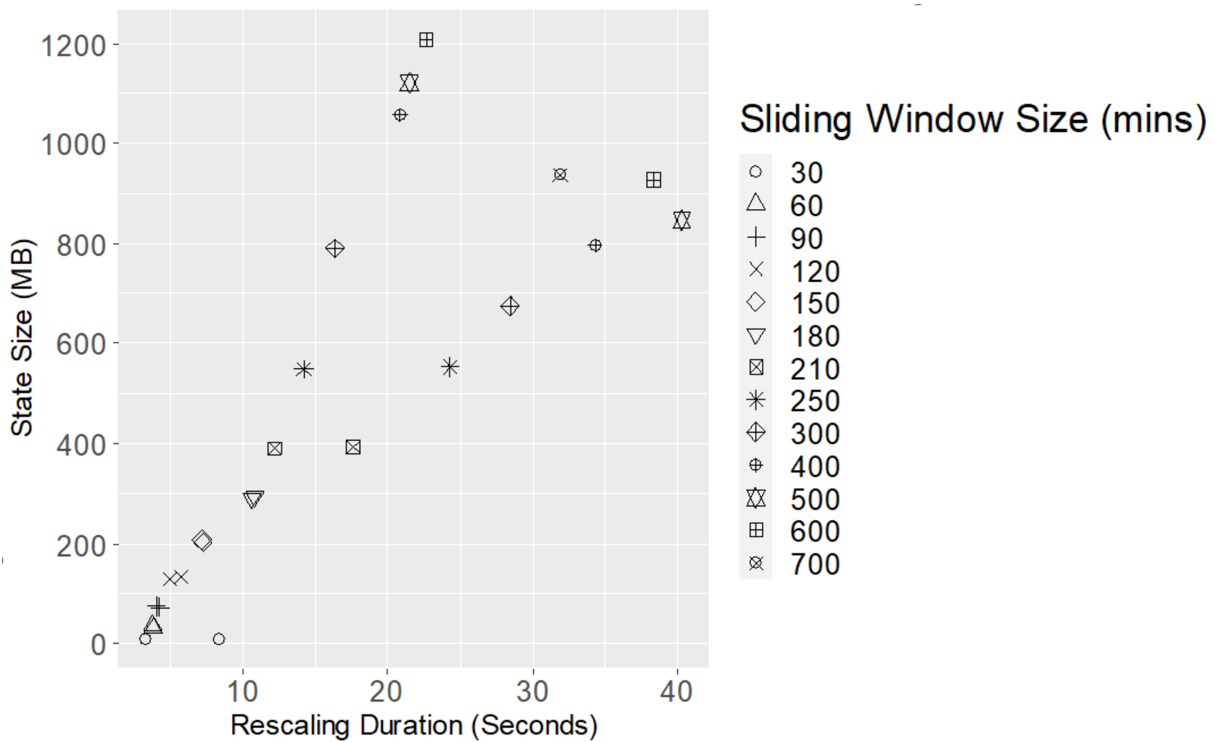


Fig. 4.1 Rescaling Duration Increasing Over Larger State Sizes

In a window-based environment, sub-streams are created using a strategy known as a "window". A window policy determines the size and sliding period of the window. A sliding window size defines a specific period of time during which data is considered for analysis. Larger window sizes result in an increased state size because more data is accumulated within the extended window [92]. Figure 4.1 shows the estimated time to auto-scale an application over different state sizes. The plot shows that it takes approximately 40 seconds to scale a running application when the application state size is 1GB. In this experiment(first experiment under section 4.3), we collect the application state information twice, every 2.5 minutes. Each shape represents the duration of the sliding window; consider each shape as the scaling duration value collected every 2:30 minutes for the different state sizes. Therefore, we have two identical shapes for every deployment (each experimental deployment is set at 5 minutes). You will observe that the points begin to disperse as the state size grows. This shows the rescaling time variance between each deployment.

Figure 4.1 shows that, as the application's state grows, so does the rescaling duration. We argue that more data could have accumulated during a rescaling process, which could mean constantly rescaling and falling short of resources, leading to multiple rescaling of the application. This repetitive task could harm system performance. We believe that a proactive approach will be to use machine learning algorithms to develop a workload prediction module that can forecast workload characteristics [142] and rescaling duration. This predictive approach should be embedded in a scaling policy to broaden the decision-making scope and ensure adequate resource allocation even when workload skewness exists.

We make the following contributions in this chapter:

- First, we show that rescaling duration is critical in auto-scaling a streaming application.

- Second, we show that state size is an important metric when rescaling.
- Third, we show that the end-to-end checkpoint duration strongly correlates with the state size.
- Fourth, we develop a machine learning predictive model to forecast a streaming application's rescaling duration based on the state size and end-to-end duration. This model will provide a scaling controller with knowledge of the estimated rescaling duration.
- Finally, we apply our model to predict the rescaling duration of an application based on forecasted state size values.

4.2 Influencing Metrics for Scaling a Streaming Application

Scaling controllers usually rely on metrics to determine when and how to scale. Several studies have focused on different metrics in building an optimal scaling policy. Most systems rely on simplistic performance models, like setting predefined thresholds and conditions. These conditions include CPU and memory utilisation, backpressure, observed processing rate, etc. However, CPU and memory utilisation can be insufficient metrics for streaming applications, especially in cloud environments where multi-tenancy and performance interference is prevalent [84].

Kalavri et al. propose a better approach (DS2) to automatically determine the optimal parallelism level for each operator in the dataflow as the computation progresses, using real-time performance traces. It maintains a dynamic provisioning plan in which the allocation of resources to each operator changes.

This paper explores other metrics like state size and end-to-end checkpoint duration. These metrics have been shown to significantly influence the scaling duration of streaming applications compared to offered load (data arrival rate).

While many stream processors support elastic runtimes and job reconfiguration via migration or externalisation of state, symptom detection and scaling actions are entirely dependent on manual intervention by the vast majority. 4.1 summarises the systems that incorporate some form of automatic scaling. We classify them according to (i) the metrics used to detect symptoms, (ii) the policy logic used to determine when to scale, (iii) the type of scaling action used to specify which operators to scale and to what level, and (iv) the optimisation objective (e.g. latency or throughput SLO).

Table 4.1 Summary of auto-scaling policies in distributed dataflow systems [58]

System	Metrics	Policy	Scaling Action	Objective
Borealis [1]	CPU, network slack, queue sizes.	Rule-based	Load shedding	Latency, throughput
Stream Cloud [45]	Average CPU, observed rates.	Threshold-based	Speculative, multi-operator	Throughput
Seep [18]	User/system CPU time.	Threshold-based	Speculative, single operator	Latency, throughput
IBM Streams [42]	Congestion, observed rates.	Threshold-based, blacklisting	Speculative, single operator	Throughput
FUGU+ [46]	CPU, processing time.	Threshold-based	Speculative, single operator	Latency
Nephele [76]	Mean task latency, service time, interarrival time, channel latency.	Queuing theory model	Predictive, multi operator	Latency
DRS [40]	Service time, interarrival time.	Queuing theory model	Predictive, multi operator	Latency
Stela [135]	Observed rates.	Threshold-based	Speculative, single operator	Throughput
Spark Streaming [93]	Pending tasks.	Threshold-based	Speculative, multi operator	Throughput
Google Dataflow [65]	CPU, backlog, observed rates.	Heuristics	Speculative, multi operator	Latency, Throughput
Dhalion [38]	Backpressure, queue sizes, observed rate .	Rule-based, blacklisting	Speculative, single operator	Throughput
Pravega [28]	Observed rates.	Rule-based	Speculative, single operator	Throughput
DS2 [58]	True processing and output rates.	Dataflow model	Predictive, multi operator	Throughput

4.3 System Design

Figure 4.2 shows our experimental pipeline setup and the relationship among the various unit of our setup. These units are self-contained, which makes it easy to scale the experiment. This setup architecture has four major areas: the data source (NEXMark workload generation), Streaming framework (Flink and DS2 scaling controller), Visualisation, and Datastore. Our experimental aim is to simulate a fluctuating workload streaming environment and measure the time required to complete a rescaling procedure using the DS2 rescaling policy. This experiment leverages Flink's consistent checkpointing of the application state on a single Flink instance [124].

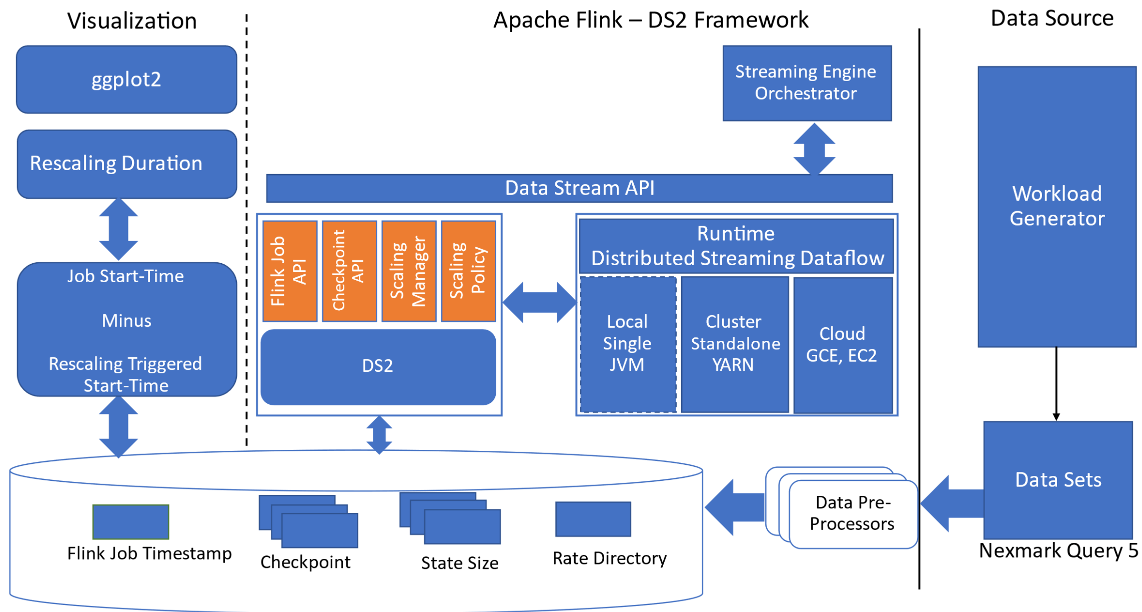


Fig. 4.2 Experimental System Architecture (Chapter 4)

Hardware and Software Configuration. Tables 4.2 and 4.3 show this experiment's hardware and software configuration parameters.

Table 4.2 Standalone hardware configuration (Chapter 4)

Hardware	Configuration
CPU	Intel® Core™ i5-8500 CPU @3.00GHz
CPU Cores	6
Memory	16GB
Disk	1TB
NIC	1000 Mbps
Kernel Version	4.15.0-74-generic

Table 4.3 Software Configuration (Chapter 4)

Software	Version	Number of instances
OS	Linux 19scompd047 84-Ubuntu	1
Flink	1.4.1	1
R	Rstudio 2021.09.0 Build 351	1
Intellij IDEA	2019.3.3 (Ultimate Edition)	1

Table 4.4, shows the single instance Flink installation configuration parameters used for this experiment. When checkpointing is triggered, the state is preserved between checkpoints to

protect against data loss and to ensure consistent recovery. The internal representation of the state, as well as how and where it is persisted between checkpoints, is determined by the State Backend selected. Flink supports multiple state backends, each specifying how and where the state is stored. This includes memory, file system and RocksDB state backends. We chose the file system state backend (`FsStateBackend`). Further work would merit exploring alternative state backends' differing impacts.

Table 4.4 Apache Flink Configuration

Configuration Parameters	Values
taskmanager.numberOfTaskSlots	4
state.backend	filesystem
state.backend.fs.checkpointdir	file:///path/to/savepoints
jobmanager.heap.mb	6144
taskmanager.heap.mb	6144

We set the data source rate at 20,000, the checkpoint interval to 2:30 minutes and the sliding window to 60 minutes. We update the time window configuration parameter using different time intervals to create different state sizes each time we run the workload. Subsequently, we consume the /jobs/:jobid/ REST API and collect values like state size and end-to-end checkpoint duration (this is the duration of a complete checkpoint measured by the time interval between the triggered timestamp and the most recent acknowledgement.) and job start time. Below is a summary of the two major experimental runs used for this experiment.

The purpose of running two experiments was to get the one that simulated our experimental objective better. We aim to show a use case where state size growth will lead to an increased rescaling duration interval. Rescaling duration is the time between when a scaling operation is triggered, and the new configuration is deployed. We derive the rescaling duration as follows:

$$\texttt{Rescaling duration} = \texttt{new job start time} - \texttt{rescaling triggered time} \quad (4.1)$$

New Job Start Time. The start time recorded in Flink for a newly deployed running application.

Rescaling Triggered Time. The recorded time when a rescaling procedure is triggered

Experiment One. This experiment was repeated 13 times. Checkpoint interval equal to five minutes, data source rate equal to 20,000 records/sec, operator parallelism (Bid-Source and Sliding window-sink) equal to 1 and 2 respectively, windows width time in minutes were alternated over the 13 experiments in the following interval (30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 400, 500, 600). At the end of each run, the state size, end-to-end checkpoint duration, new job start-time, Job ID and checkpoint triggered-start-time values are collected through the Flink APIs. We control the state size at the end of each deployment for each deployment by adjusting the window width time value of the workload.

Table 4.5 First experiment configuration parameters

Deployment	Operators Parallelism		Data Source rate	Checkpoint Interval (mins)	Window width (mins)	Back Pressure Status
	Bid-Source	Sliding window-sink				
1	1	2	20,000	5	30	Low
2	1	2	20,000	5	60	Low
3	1	2	20,000	5	90	Low
4	1	2	20,000	5	120	High
5	1	2	20,000	5	150	High
6	1	2	20,000	5	180	High
7	1	2	20,000	5	210	High
8	1	2	20,000	5	240	High
9	1	2	20,000	5	270	High
10	1	2	20,000	5	300	High
11	1	2	20,000	5	400	High
12	1	2	20,000	5	500	High
13	1	2	20,000	5	600	High

Experiment Two. We try to explore shorter checkpoint intervals to enable us to measure the scaling duration between different checkpoints. Therefore, the checkpoint interval for this experiment is set to 2:30 minutes to allow us to collect the state size twice after each checkpoint and redeployment. Like the previous experiment, we adjusted the state size after every deployment through the window width time parameter of the workload. After each deployment, we update the sliding window-sink operator parallelism to simulate a change in the data flow topology during scaling.

This research chapter provides decision support and predictions that can cater to the early deployment of newly emerging workloads with no historical operational data. Therefore, the volume of data used for this experiment supports this use case. More experiments could be run with wider checkpointing intervals. However, this would require more system resources.

We also observed the backpressure status, which becomes high when the state size grows. This experiment measures the rescaling duration when the state size grows huge. Therefore, we do not investigate the backpressure effect because it affects the offered load, which is not a priority for this experiment.

We also kept the source data rate static (20,000 records/sec) as increasing it did not significantly affect the state size as we wanted. The choice of checkpoint interval (2:30 and 5 minutes) enabled us to collect a state size value distinctive from another. Shorter intervals would have very close results, while longer intervals would require a bigger hardware resource.

Table 4.6 Second experiment configuration parameters

Deployment	Operators Parallelism		Data Source rate	Checkpoint Interval (mins)	Window width (mins)	Back Pressure Status
	Bid-Source	Sliding window-sink				
1	1	2	20,000	2:30	30	Low
	1	3	20,000	5:00		Low
2	1	2	20,000	2:30	60	Low
	1	3	20,000	5:00		Low
3	1	2	20,000	2:30	90	Low
	1	3	20,000	5:00		Low
4	1	2	20,000	2:30	120	Low
	1	3	20,000	5:00		Low
5	1	2	20,000	2:30	150	Low
	1	3	20,000	5:00		Low
6	1	2	20,000	2:30	180	High
	1	3	20,000	5:00		High
7	1	2	20,000	2:30	210	High
	1	3	20,000	5:00		High
8	1	2	20,000	2:30	250	High
	1	3	20,000	5:00		High
9	1	2	20,000	2:30	300	High
	1	3	20,000	5:00		High
10	1	2	20,000	2:30	400	High
	1	3	20,000	5:00		High
11	1	2	20,000	2:30	500	High
	1	3	20,000	5:00		High
12	1	2	20,000	2:30	600	High
	1	3	20,000	5:00		High
13	1	2	20,000	2:30	700	High
	1	3	20,000	5:00		High

A system failure caused the last deployment (13) to fail. This was caused by the hardware configuration capacity being exhauted. This deployment failure was observed when the window width became bigger translating to a larger state size. We also observed the backpressure status

which becomes high when the state size gets bigger. This experiment focuses on measuring the rescaling duration when the state size grows huge and therefore, we do not investigate the backpressure effect. We also kept the source data rate static as increasing it did not significantly affect the state size as we wanted. At the end of each experiment, the relevant system data values are collected and measured. To automatically collect these values, we develop two Java GET API classes that consume the Flink REST API. We also updated the DS2 rescaling shell script to log a timestamp whenever executed. The experiment was deployed in this order:

i. First, I execute a script to start the Flink streaming application using the NEXMark Query5 workload.

ii. Next, I trigger a redeployment script that redeploys an application from savepoint by ending the current job and redeploying a new job with the capability of updating the dataflow topology (scaling up or scaling down). This script is triggered immediately after a checkpoint operation is concluded and the status changes to complete. Furthermore, the script collects and saves the job ID of the completed job and the timestamp when the script was triggered.

iii. Next, I invoke the CheckpointApiCall.java GET API code to collect the values of the following parameters: jobid, triggered-timestamp, state_size and end_to_end_duration through the Flink API.

iv. Next, I Invoke the StartTimeApi.java GET API code to collect the jobid and start-time from the newly deployed running job.

v. Finally, we change the configuration parameters, build a new artefact, and repeat the procedure from the start.

I ran these experimental procedures multiple times with different configuration parameters as shown in Tables 4.5 and 4.6 above. This is done to analyse the behaviour of the streaming application over different computational parameters. We collect results after each deployment and measure the redeployment duration over different state sizes.

4.3.1 Nexmark Workload

NEXMark is an online auction system that allows queries over three main business objects (Person, Auction and Bid) [119]. All users register under a person's business objects to participate in an auction as either a seller or a buyer. As illustrated in Figure 4.3, a seller submits items for sale to an auction with certain information about the item to be sold. An auction has an opening and closing time, and after an auction is closed, buyers enter bids for existing auctions (Tucker et al., 2008). There are many interesting queries that NEXMark provides, which could be over single or multiple business objects and joins or unions from stored data or the business objects. Since our objective was to simulate the rescaling duration of a streaming application that accumulates a huge state, we, adopted NEXMark Query5 for our experiments.

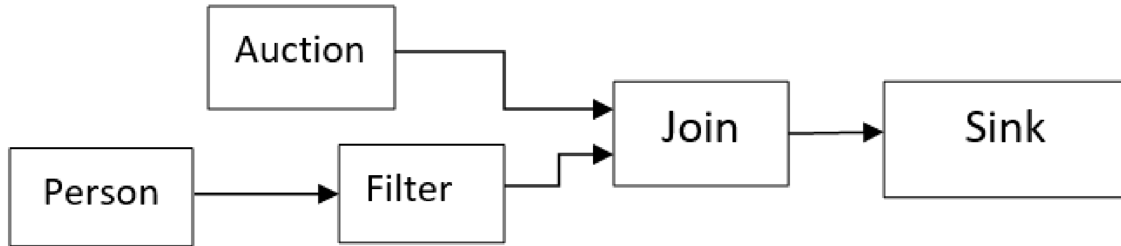


Fig. 4.3 NexMark Workload Stream Processing Directed Acyclic Graph (DAG)

The DAG comprises several key components as shown in Figure 4.3: a data source generating auction and person events, a filtering stage to eliminate irrelevant events, a join operation combining auction and person events based on person ID, an aggregation process grouping events by the person's city to compute the total number of auctions, a key-value store to maintain the state of aggregated results, and a sink to output the final results.

Query 5 selects the items that have seen the most bids over a period. These items are also referred to as HOT ITEMS. This query uses a sliding window group by operation. A sliding window groups tuples within a window that moves across the data stream in accordance with a predetermined interval [114]. Table 4.7 shows some relevant configuration parameters we update in the NEXMark Query5.java class. We run this workload multiple times with different state sizes. We update the time window configuration parameter using different time intervals to create different state sizes each time we run the workload.

Table 4.7 NEXMark Query 5 Configuration

Configuration Parameters	Values
Checkpoint Interval	2:30 minutes
Data Source Rate	20,000
Sliding Window	60+ minutes

NEXMark Query 5 was adopted as the workload for this experiment because it retains the system state based on the sliding window capabilities that meet our experimental objective to simulate the rescaling duration of a streaming application that accumulates a huge state.

4.4 Impact of Long Checkpointing Duration in Streaming Applications

The Flink checkpointing mechanism guarantees exactly-once state consistency it is the most common fault tolerance and rescaling approach used by most state-of-the-art streaming systems for stateful applications [56]. This study employed the full checkpointing mode. We

measure the end-to-end duration of each checkpoint, which is the duration of a complete process. After each checkpoint, we leverage the Flink API to collect certain parameters like the state size, triggered timestamp, and end-to-end duration values. We also configure the Flink `env.enableCheckpointing()` to set the checkpointing interval.

Checkpointing intervals are usually carefully selected as high intervals could lead to longer recovery duration, while low intervals can lead to high processing overhead due to state size [124]. Figure 4.4 shows that a larger state leads to longer checkpoint durations. The growth in state size is dependent on the window size, as illustrated in our initial experiment within Table 4.5. When the state of a system is larger, the time it takes to capture a snapshot of that state (checkpoint) increases. In scenarios where frequent checkpoints are necessary for fault tolerance, longer checkpoint durations can become a problem. This is because the system may spend a significant amount of time taking these checkpoints, potentially impacting the application's responsiveness and overall performance.

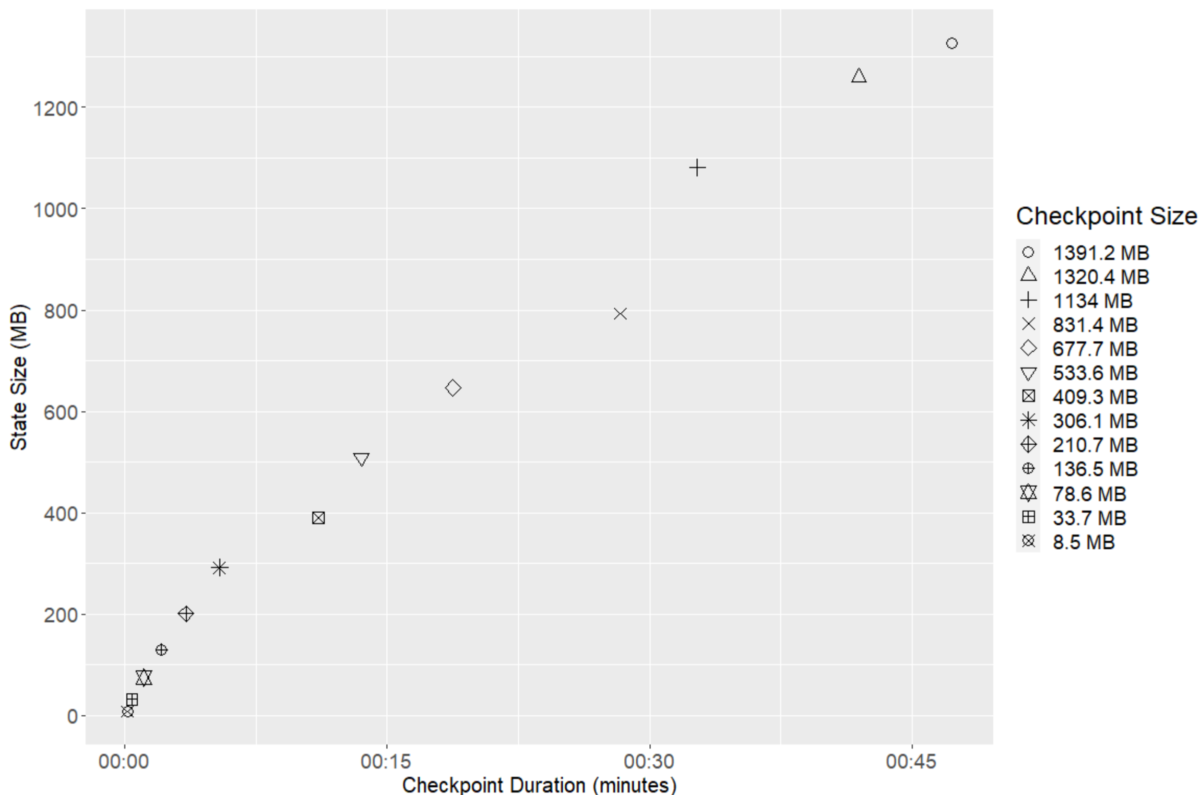


Fig. 4.4 Flink Checkpoint Duration Increasing Over Larger State Sizes.

4.5 Predictive Modelling

This section describes our approach to developing our predictive models and the methods used in testing each model's performance quality. We also show each model's prediction results when applied to known and unknown data sets.

4.5.1 Linear Regression Analysis for Predictive Modelling

Regression analysis is a statistical approach to determine the relationship between one dependent variable and one or more independent (predictor) variables. The analysis produces a predicted value for the criterion because the predictors are combined linearly. Cohen asserts (Cohen et al., 2014). Regression analysis is primarily used for prediction, classification, and explanation. To determine whether we can create a predictive model, we determine whether our predictor and response variables are related.

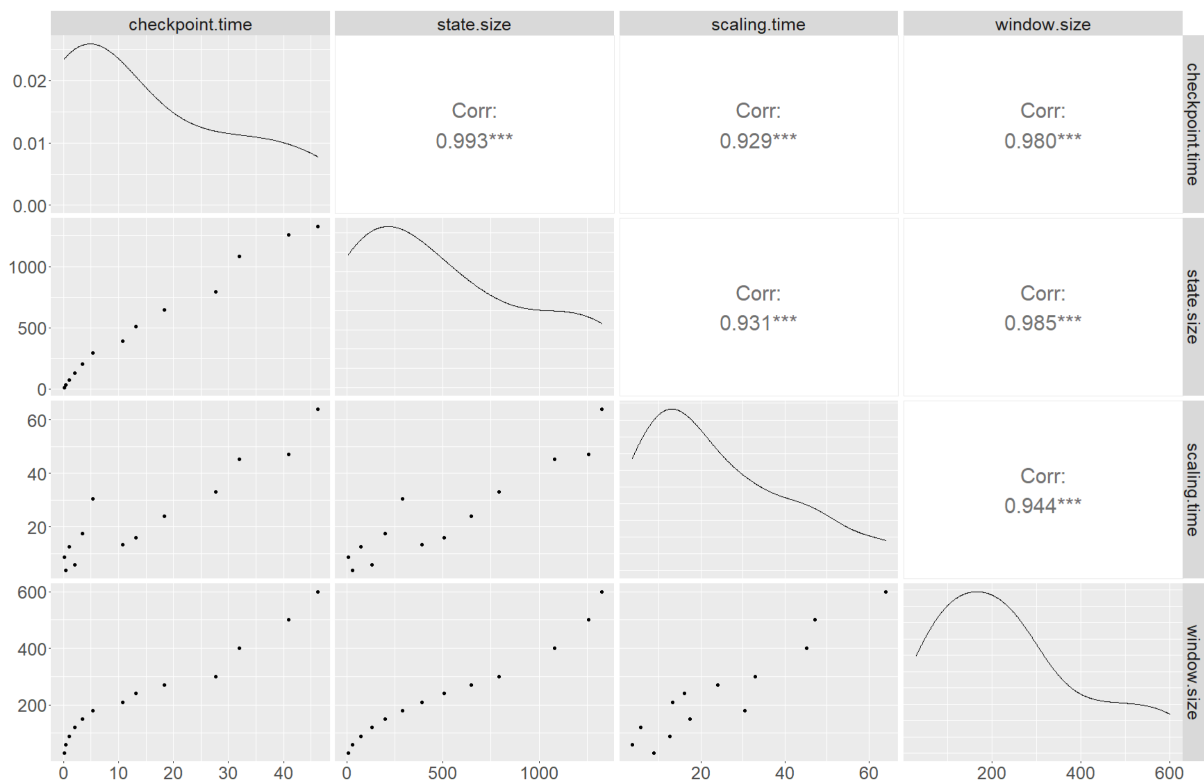


Fig. 4.5 Exploring the Relationship Between Variables.

Figure 4.5 provides some interesting trends that create curiosity to question the data further. The correlation coefficients tell how close the variables are to having a relationship. Correlation coefficients closer to 1 signify a stronger relationship. The scatter plots allow us to envision the relationship between sets of variables. Scatter plots where the points have a clear aesthetic pattern suggest a stronger connection. Figure 4.5 contains four variables from our dataset: checkpoint time (the end-to-end checkpoint duration for a complete checkpoint measured in seconds), state size (the state size of all acknowledged subtasks measured in megabytes), scaling time (this is the rescaling duration measured in seconds), and window size (the different sliding window configuration parameters used for each deployment we ran to produce different state sizes measured in minutes).

The data set used for this experiment contains two data points, because checkpoints were taken twice (2:30 and 5 minutes), and the relevant parameters, as stated above, were collected after each checkpoint was completed. Therefore, the dispersal of the data at some points shows the rescaling time variance between each checkpoint based on the state size. This suggests a

strong relationship between the dependent and independent variables, especially as the correlation coefficients are close to 1.

The relationship between state size, end-to-end checkpoint duration, window width size and rescaling duration appears to have a strong relationship with correlation coefficients close to one. We can see a consistent increase in the rescaling time as the state size, end-to-end duration, and window width size increase. We formulate the equation for our linear regression as follows.

$$Y \approx \beta 0 + \beta 1X \ + \varepsilon \tag{4.2}$$

- The Y and X represent the dependent and independent variables.
- $\beta 0$ denotes the model intercept or the point at which it crosses the y-axis.
- $\beta 1$ denotes the model slope and the direction and steepness of the line (positive or negative).
- ε is the error team that includes variability that the model cannot account for (what X is unable to reveal about Y).

We develop four linear regression models to ascertain the strength of these relationships and identify which independent variables have the closest relationship with our dependent variable. The reason for adding more predictors is to improve the model's predictive ability.

i. Model 1. Rescaling Duration $\approx \beta 0 + \beta$ (State Size) + ε

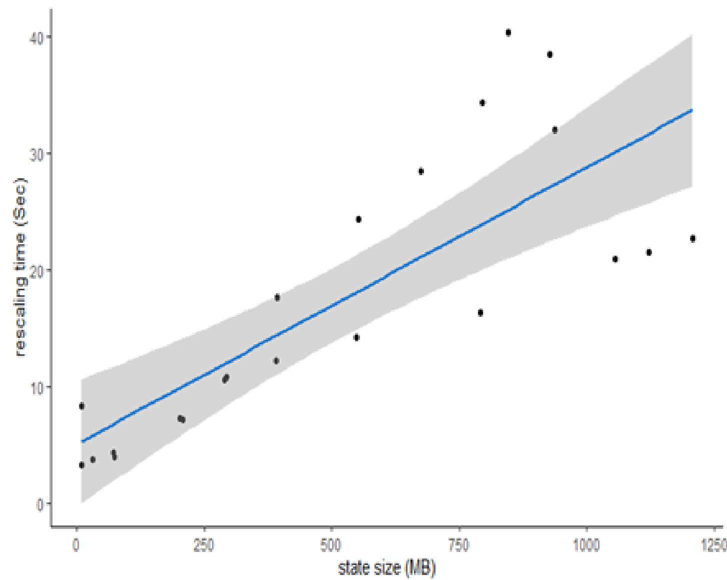
ii. Model 2. Rescaling Duration $\approx \beta 0 + \beta 1$ (State Size * End-to-End Duration) + ε

iii. Model 3. Rescaling Duration $\approx \beta 0 + \beta 1$ (State Size * Window Width Size) + ε

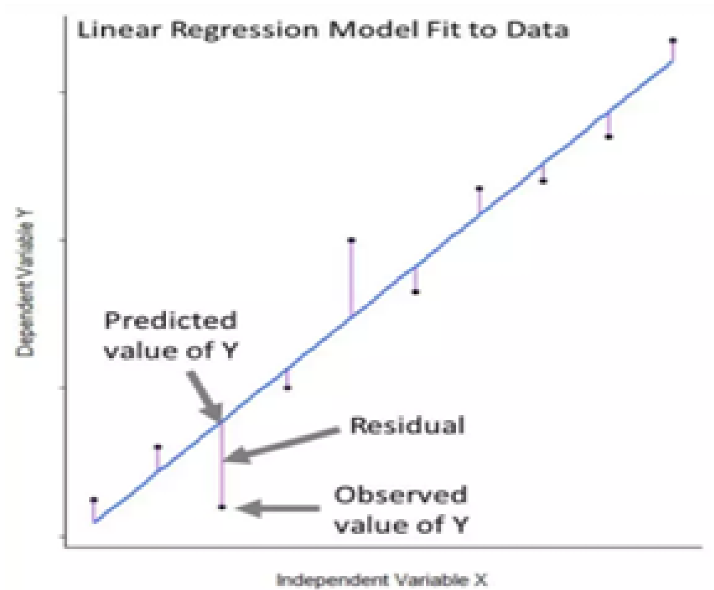
iv. Model 4. Rescaling Duration $\approx \beta 0 + \beta 1$ (State Size * End-to-End Duration * Window Width Size) + ε

Since we have numeric values and the relationship tends to have a linear pattern, we use the linear regression model for this experiment. We create a linear model engine to fit our linear model line to our data as shown in Figures 5 and 6. It fits the line, thereby minimising the sum of the squares of the residuals. This method is called "Minimising Least Squares". However, it is worth noting that even when the linear regression model seems to have a good fit, it is usually imperfect [97]. The residuals represent the differences between our observations and their model-predicted value, as shown in Figure 4.

The output of our fitted linear model is shown in Figure 5. The grey shading around the line denotes a confidence interval of 0.95, the default value for the stat_smooth() function [97].



(a) Linear Model 1 Fitted to Data



(b) Linear Regression Model Fit to Data [97]

4.5.2 Resampling

We split the dataset into a training set and a testing set. The training data set is used to fit models, tune models, estimate parameters, and compare models, among other things, In contrast, the testing data set evaluates the model's performance. It also serves as an objective source for determining the model's performance. To accomplish this, we used the rsample package to generate an object that contains information about how to split the data, followed by two further rsample methods that create data frames for the training and testing sets.

Creating a uniform guideline for splitting data into specific proportions can be exceedingly difficult. Various factors, including the size of the initial data pool and the total number of predictors can influence the data proportion. With a large volume of data, the criticality of this decision is reduced. Additionally, it is critical to consider the ratio of samples (n) to predictors (p). When n is significantly greater than p, we will have much more leeway in splitting the data.

However, when n is less than p, however, even if n appears to be a large number, modelling difficulties can arise.

We used the random sample approach to select 80% of our dataset as a training data set and the remaining 20% as test data. We note the work by Joseph et al. that proposes some evaluation criteria for choosing an optimal splitting ratio [57]. However, there is no consensus regarding the optimal data splitting ratio based on theoretical and numerical investigations [88, 30].

We used a random number seed to guarantee the reproducibility of the data. However, other strategies exist for splitting data, like the stratified random sample, non-random sampling, etc. Random sampling is a simple strategy to implement and typically prevents the process from being skewed toward any data characteristic. Stratified random sampling is mostly used to balance variables outcomes of categorical data sets. Since our population set does not have categorical variables and no imbalance in our variable distribution outcome, this data splitting strategy was not relevant to this experiment. Furthermore, the non-random sampling method is used when the data is selected based on subjective judgement. Since we have no reason to select or exclude any data element in our population pool, the random sampling method was preferable to the non-random sampling method.

Table 1 below shows the comparison of all our models' statistical properties. Working through the output of our models' statistical properties, we focus on a few evaluation metrics that tell us how well these models fit our data.

Table 4.8 Models Statistical Properties

	Model 1			Model 2			Model 3			Model 4		
	95%	CI	p-value	95%	CI	p-value	95%	CI	p-value	95%	CI	p-value
state Size	0.02	0.03	<0.001	0.00	0.02	0.077	0.00	0.03	0.10	-0.07	0.03	0.4
end-to-end duration				1.1	2.2	<0.001				-2.2	1.9	>0.9
state size * end-to-end duration				0.00	0.00	0.032				0.00	0.01	0.2
state size * end-to-end duration							0.06	0.18	<0.001	-0.03	0.20	0.2
state size * window size							0.00	0.00	0.003	0.0	0.0	>0.9
end-to-end duration * window size										0.00	0.01	0.3
state size * end-to-end duration * window size										0.00	0.00	0.024

Confidence Interval (CI) / Confidence level (95%). Sample size impacts the CI and confidence level estimate. Larger sample sizes normally lead to higher confidence levels. A confidence interval with zero value may indicate no significance in the parameter's relationship. This is frequently how confidence intervals are interpreted, but this could be incorrect, as shown by other relevant statistical properties. Rather, it indicates uncertainty. Having a confidence interval of zero indicates that an independent variable could have a positive (>0) or negative (<0) effect on the rescaling duration. However, considering the volume of our sample size, we focus on other statistical metrics like the R-squared, Adjusted R-squared and p-value.

P-value. Statistical significance is defined as a p-value less than 0.05 (<0.05). It indicates strong evidence against the null hypothesis, representing the likelihood that our data would have occurred under the null hypothesis. As a result, the null hypothesis is rejected, and the alternative hypothesis is accepted.

Table 4.9 Comparison of Model Performance Indices.

Name	R^2	R^2(adj).	RMSE	Sigma	AIC weights	BIC weights
Model 1	0.635	0.615	7.076	7.458	<0.001	<0.001
Model 2	0.937	0.925	2.937	3.283	0.158	0.579
Model 3	0.843	0.813	4.644	5.193	<0.001	<0.001
Model 4	0.964	0.944	2.212	2.855	0.842	0.421

Table 4.9 evaluates the performance quality of each model. Model 4 has the better performance indices results, followed by Model 2. Figure 5 gives a pictorial representation of the model performance indices comparison.

R-squared (R^2) is the correlation between the known outcome and model-predicted values. Values closer to 1 indicate models with a good fit; Adjusted R-squared (R^2 (adj)) is usually lower than or equal to R^2. Similarly, A model with accurate predictive capability has a value of 1, while a model with weak or no predictive value has a value of 0 or less; Calculating the Root Mean Square Error (RMSE) is one way to determine how well a regression model fits a dataset, by measuring the average distance between the predicted values of the model and the actual values in the dataset. The lower the model's root mean square error, the better it fits a dataset; Sigma (standard deviation) quantifies the degree to which a process deviates from ideal performance. Furthermore, for model selection, Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) are frequently utilised for model selection. AIC picks the best-fit model that explains the most variance with the fewest independent variables, while BIC assesses how likely a model is to be accurate. Lower AIC and Bayesian Information Criterion (BIC) scores are preferable.

Further comprehensive model checks were carried out to check each model's assumptions like collinearity, normality of residuals, linearity, homogeneity of variance and influential observation. The result indicates a multicollinearity issue in Models 2, 3 and 4, each with a Variance Inflation Factor (VIF) above 10. Concerning the normality of residual, models 2 and 4 are normally distributed, result on linearity showed that model 2 data points are fitted closer to the residual line compared to other models, which suggests that it has a better predictive performance. The result of influential observation shows that model 4 and 2 has the highest leverage and residual, which indicates a strong influence on the observation by these two models.

Model 4 and Model 2 both stand out to be the most efficient for our experiment. However, we shall investigate further to get the best amongst the two. There are a few reasons why training set statistics such as those presented in the previous section may be overly optimistic:

i. Random forests, neural networks, and other machine learning methods can effectively memorise the training data set. Re-predicting the same set of data will always produce near-perfect predictions.

ii. The training data set cannot accurately predict performance because it is not an independent data set.

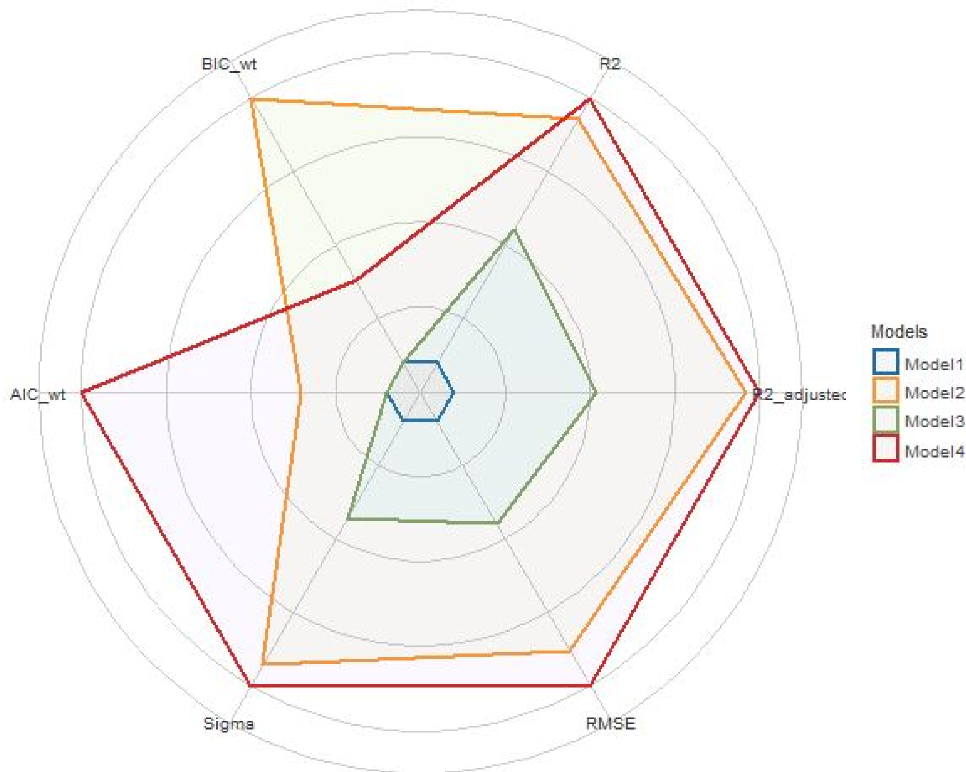


Fig. 4.7 Comparison of Model Indices.

Consequently, Cross-Validation (CV) is applied to our training data, which splits the training data further into the assessment and analysis set. An assessment set in the tidymodel framework is a set of data to measure the CV's performance, while analysis data sets are used to train and fit the models [66]. Figure 4.8 illustrates the CV resampling method architecture. The objective is to further measure our model's performance using some performance statistics. Due to the volume of our data, we created a 4-fold CV. This randomly allocates the 20 cells in the training data set into four groups ("folds") of equal sizes.

For the initial iteration of resampling, the first fold of approximately five cells is held out for performance measurement. In comparison, the remaining 75% of the data (approximately 15 cells) is used to fit the model. After training the models on the analysis set, the four models are applied to the assessment set to generate predictions. Next, we compute the performance statistics for each model based on the predictions' results. Table 4.10 shows the resampling results and performance statistics from the 4-fold cross-validation. The performance statistics used are the RMSE and the R^2.

Model 2 appears to be the most efficient model considering it has the lowest RMSE mean value and the highest R^2 mean value. However, Models 4 and 2 could compete for superior performance if subjected to a larger data set.

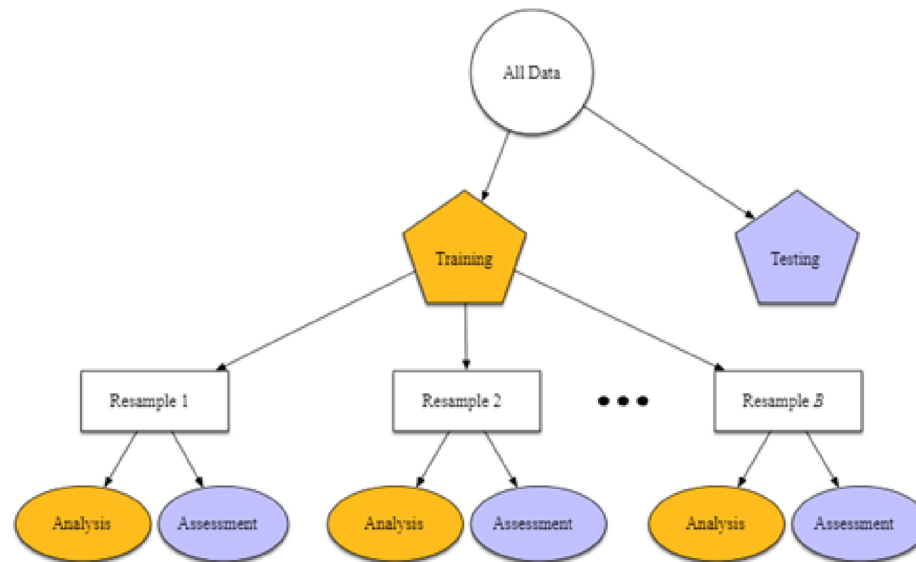


Fig. 4.8 Cross-Validation Resampling Method Architecture [66]

Table 4.10 Performance Statistics Metrics for Each Model

Models	Model 1		Model 2		Model 3		Model 4	
metrics	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2
mean	7.690	0.707	3.430	0.977	5.720	0.791	7.54	0.818
standard error	0.309	0.077	0.635	0.012	0.794	0.022	1.22	0.041

4.6 Summary of Experimental Results

In the previous section, we used different strategies to train and test the efficacy of our models. Resampling enables us to simulate how well our model performs on new data, while the test set serves as the final, unbiased validation of the model's performance. We now apply the model's predictive ability to our test data set.

Table 4.11 Linear Models Applied to Test Data

Rescaling Duration (mins)	Predicted Duration			
	Model 1 (mins)	Model 2 (mins)	Model 3 (mins)	Model 4 (mins)
3.75	5.65	2.81	0.49	**3.29**
4.91	7.94	**4.74**	8.03	6.61
5.71	7.99	**6.13**	8.04	6.88
8.30	**5.02**	2.57	-3.36	1.36
14.17	18.10	15.50	20.10	**15.00**

Table 4.11 shows the result of predicted results by each model placed side by side with the actual values of the test data set. Values in the "Rescaling Duration" column are the known test data set aside when we carried out the 80/20 split operation. The predicted rescaling duration values are the predicted values generated by each model. The predicted results (bolded) are the closest predicted values produced by each model. Following these prediction results, Models 2 and 4 have a better predictive performance capability than other models.

State Size Forecasting.

We apply our model to a real-world use case, aiming to show the general applicability of our model to common use cases. We leverage experimental data measurements from the Rule-Based Event Aggregator (RBEA) [14].

We extracted measurements of five distinct RBEA deployments, spanning from 100 to 500 GB of global state. The end-to-end duration was set to 9 milliseconds based on empirical knowledge. We applied our Model 2 to estimate the rescaling duration for each state size measurement.

The results (83 sec, 149 sec, 215 sec, 281 sec and 347 sec) show a linear growth in the rescaling duration that correlates with state size. Based on the result, it will take between 83-347 seconds to auto-scale the RBEA global state size.

We also measured Model 2's mean prediction values of rescaling duration over the forecasted state size values with the upper and lower limits prediction limits. As the state size grows, so do the upper and lower limit prediction values. An estimate will always have a level of uncertainty associated with it, which depends on the data's underlying variability and the sample size. The more variable the data, the greater our estimate's uncertainty. Similarly, as the sample size increases, we gain more information and thus reduce our uncertainty [72]. A limitation of the current modelling approach is the prediction of negative values. These negative predictions may be caused by the speculative approach of manually allocating corresponding end-to-end duration to match each forecasted state size. This can be explored in future work to prevent the model from predicting negative values.

4.7 Conclusion

Our experimental findings highlight the importance of state size during rescaling, demonstrating that with the accumulation of more state, the time to auto-scale also increases. More state could have accumulated during a rescaling time window, and this could mean constantly rescaling and falling short of resources which will lead to multiple rescaling of the application. This repetitive task could harm system performance.

Four predictive regression models were developed and trained with the existing data set, leveraging resampling cross-validation. Model 2 was chosen as the most efficient based on the performance statistics and predictive performance, followed by Model 4.

Next, we use Model 2 to predict the rescaling duration of a running application over some forecasted state size values. Results show that it will take approx. 6 minutes to auto-scale when the application state size reaches 500GB. In a constantly changing and unpredictable distributed data streaming environment, a lot could happen in 6 minutes. We, therefore, recommend that forecasting workload characteristics and state size growth rate is very relevant when developing a scaling policy to enable a streaming application to handle unanticipated resource demand.

Our predictive model can be applied by the broader user base to assess the rescaling time of streaming applications. However, it's important to note that users might need to tailor the model to their specific system environment, as different characteristics and factors in other domains like volume of data, imbalance in the variable distribution outcome can influence the model's performance and effectiveness.

4.7.1 Experimental Challenges

In this sub section, we provide bullet points of experimental challenges faced while carrying out this experiment and limitations of the overall experimental body. This is listed in no specific order.

- The volume of data used for this experiment was very small, and hence our machine learning model was limited to this low volume of data. It is my desire that a large dataset is used to train these models.
- A limitation of the current modelling approach is that it predicts negative values in the mean prediction values. Further work could explore a different model that would not permit negative values. e.g. gamma GLMs [24].
- Flink exposes the state size of a stream application after a checkpoint. This makes getting information about a running application state impossible until a successful checkpoint is taken. Checkpoint operations struggle and eventually fail when the state size becomes bigger. Even when checkpoint intervals are carefully chosen, this interval selection decision can easily become sub-optimal once a checkpoint takes a longer time to complete than expected.

- This experiment was run on my local bare metal machine. Therefore, my Flink streaming application is limited to the computational capacity of my machine. Therefore, limiting the experimental processing capacity.
- The process of redeploying and consuming Flink API, though largely done through a Java class and shell scripts but triggered manually. Updating the Nexmark workload parameter (change the state size values) and rebuilding the project artefact after each deployment is also manually done.
- Predicting a corresponding end-to-end duration to state size was difficult. Hence, we adopted a static approach of forecasting five milliseconds based on speculative assumption. This could be potential limitation of this experiment

4.7.2 Future Work

In this subsection, we provide bullet points of future direction and next steps that could be useful and complementary to this body of research. This is listed in no specific order.

- Developing an adaptive approach leveraging our models in this chapter within DS2's scaling policy to govern rescaling decisions.
- Evaluate the performance impact of a long-running checkpoint process on the streaming application with respect to overlapping checkpointing operations without delaying subsequent scheduled checkpoints [23].
- Develop a model to predict the corresponding end-to-end duration of a state size instead of the manual approach used in this experiment. The end-to-end duration is system-generated and dependent on the system state size.
- Identify and measure the effectiveness of other metrics like backpressure, etc. and check for correlations between this metric, the application state, the operators true processing rate and any other interesting correlation and trend that should be considered when rescaling.

Chapter 5

Impact of State Size on Streaming Operator Throughput

Chapter Summary

In stream processing applications, accurately measuring a system's processing capacity is critical for ensuring optimal performance and meeting SLOs. Traditionally, operator throughput has been used as a proxy for the application's state size, but this approach can be misleading when dealing with window-based applications. In this chapter, we explore the impact of window selectivity on the performance of streaming applications, demonstrating how a growing application state can artificially decrease the operators' throughput, resulting in false positives that could trigger premature scaling-down decisions. To address this problem, we conduct empirical evaluations to assess the relationship between operators' throughput and state size. Showcasing that the state size pattern typically does not correspond to the operator's processing rate in window-based applications. Our findings highlight the importance of considering the state size of the application in performance monitoring and decision-making, particularly in the context of window-based applications [92].

5.1 Introduction

Dataflow execution in streaming processors involves encapsulating distributed operator logic centred on records, intending to describe complex data pipelines. In Flink-based data processing pipelines, a consistent application state is a critical element and persisted using a modular state backbone. When required, the system orchestrates failure recovery and reconfiguration (scale-out/in) without significantly impacting execution or violating consistency [9]. Apache Flink is an open-source framework that offers stateful stream processing capabilities that are scalable, distributed and fault tolerant.

A distributed system involves a computing setup where processing tasks, data, and resources are dispersed across multiple interconnected nodes or machines. These nodes collaborate to process and analyze an uninterrupted flow of data in an effective and expandable way. Generally, Flink's distributed systems are designed to present a unified front to the user, hiding any

challenges tied to their decentralized arrangement. When utilizing such a distributed computing system, we often need to assume the state of a pipeline being employed in real-world situations at a particular time.

The technique of snap-shotting in streaming systems has been utilised for a considerable duration [143]. It has been rigorously tested in production at some of the largest stream processing deployments on thousands of nodes handling hundreds of gigabytes of state [14].

Checkpointing is a crucial and practical crash-tolerant strategy to store the process state during normal operation and quickly restore it after a failure. However, choosing the checkpoint frequency to reduce a suitable cost function would be the primary interest question for different research use cases. Knowing the conditions where checkpoints are useful and where they are not will help optimize the performance of a streaming application [34]. In this experiment, we investigate the utilisation of the application state from two distinct layers. Initially, we utilise the Flink API to gather and exploit Flink's checkpointing state. Subsequently, we analyse the constituents of the state in our workload to comprehend its internal mechanisms and track the progression of these constituent elements over time.

Flink's runtime ensures that managed state is always consistent, in the event of a partial failures or changes in the workflow topology, like upgrades to the application code or parallelism change. The size of an application's state is the amount of memory or storage space needed to store the application's current state. Data like user preferences, current settings, and any other details required for the application to operate properly might be included in this. During the design and development process, the size of the application state is frequently a crucial factor to consider because it can significantly affect the program's performance and usability.

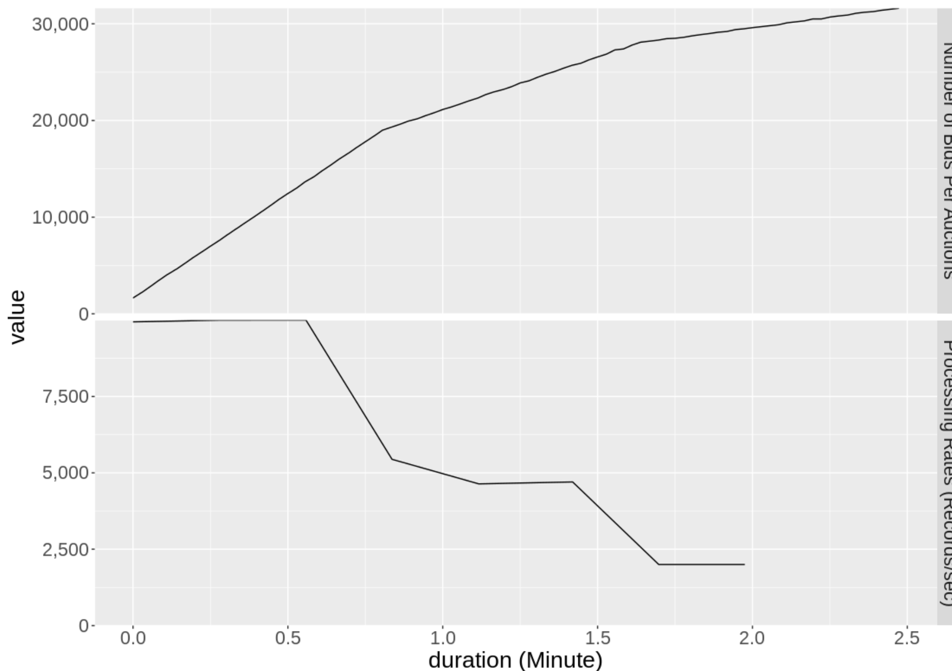


Fig. 5.1 State Growth Despite a Declining Operator Throughput

Offered load refers to the volume of data that the system needs to process at any given time. It is a measure of the amount of data that is being ingested or generated by streaming

sources. The offered load is typically measured in terms of the number of data units (e.g. events, messages, records) that are being processed per second. The offered load can be affected by various factors, such as the number of users, the complexity of the tasks being performed, and the system resources available. In this chapter, more emphasis is placed on the state size and its impact on operator throughput as well as the impact of the choice of window length and the sliding period chosen. The objective is to evaluate the processing capacity of our streaming operators over bigger state sizes and to analyse the impact of window selectivity amongst various deployments. To achieve this, we adopted two workloads: Nexmark Query 3 and Query 5.

Figure 5.1 illustrates a non-alignment between the state size and the operator's throughput of a streaming application. Without sufficient knowledge of the relationship between these two metrics poses a potential challenge for auto-scalers. In a scenario where an operator, for example, has a window of 1 hour, if the offered load (arrival rate) stops or reduces either because a sensor fails or due to a lack of customers, most auto-scalers will recommend downscaling, but the state size of that operator could still be maintained due to the window. This will mean scaling down too quickly while the state size is still large and creating additional time to recover. Our previous paper shows the impact of state size on auto-scalers' rescaling time [90].

In this chapter, we make the following contributions in this chapter:

- First, we demonstrate that offered load is an inadequate proxy for state size.
- Second, we model the impact of growing state size on operator processing capacity.
- Third, we show that flink heap memory has an effect on the growth of the state and the operator's performance.
- Finally, we show the importance of carefully selecting an appropriate window size and criteria to balance the selectivity and accuracy of the window operator, taking into consideration the performance requirements of the streaming application.

5.2 Evaluating The Impact of System Resource On Operator Throughput

Most public and private cloud systems require users to specify resource needs for running their workloads efficiently. For example, users must select the type and quantity of VM they will rent on public cloud platforms; in a Kubernetes cluster, users must specify the number of pod replicas and the resource limits for individual pods; Google requires users to identify the number of containers they require as well as the resource limits for each [107]. These restrictions allow the cloud architecture to provide appropriate resource utilization estimates, which makes cloud computing possible.

Limits, however, are (often) frustrating to the user. It is challenging to predict how many resources a job will require to function at its best, including the proper ratio of CPU power, memory, and the number of copies operating simultaneously. Load tests may be useful to

discover an initial estimate. However, because many end-user serving activities have daily or weekly load patterns, and traffic changes over longer time scales as a service gets popular, these estimates will become outdated as resource needs vary over time. Finally, as the underlying software or hardware stack gets updated, optimised, or added new capabilities, the resources required to handle a particular load change accordingly. When the requested resources are exceeded, the performance may suffer if the CPU is capped, or the process may be terminated because the memory is exhausted [107].

Apache Flink Heap Memory

Apache Flink is a distributed streaming data processing framework that uses heap memory to store and process data. Heap memory is a region of memory that is dynamically allocated by the Java Virtual Machine (JVM) at runtime and is used to store objects and data structures. In Flink, heap memory is used to store the state of a running job, as well as to buffer and cache data. The amount of heap memory that is allocated to Flink can have a significant impact on the performance and stability of a Flink job.

When Flink is running, it will dynamically adjust the amount of heap memory that it uses based on the amount of data that it needs to process. If there is not enough heap memory available, Flink may experience performance issues or even fail. To avoid these issues, it is important to configure Flink with an appropriate amount of heap memory.

In order to configure heap memory for Flink, it is possible to specify the JVM options `jobmanager.heap.mb` and `taskmanager.heap.mb:` during the initiation of the Flink job manager or task manager. The `jobmanager.heap.mb` option sets the maximum amount of heap memory allocated to the job manager that Flink can use during the `taskmanager.heap.mb` option sets the maximum amount of heap memory allocated to the task manager. It is recommended to set the `jobmanager.heap.mb` and `taskmanager.heap.mb` options to the same value to avoid heap memory fragmentation [50]. It's important to monitor the heap usage during the execution of a Flink job. If you see that the heap usage is reaching the maximum heap size, it means that the job is running out of memory and you should consider increasing the heap size or reducing the data size to prevent the job from failing.

Flink Heap Size Impact Assessment Experiment

To determine the impact of Flink's memory on the processing capacity of the operators, we run an experiment with the following parameters as contained in Table below. This experiment consists of 3 deployments, each with a constant source rate and window size. However, the heap memory value is reduced by 50% for each deployment, starting with an initial heap memory of 6144 MB.

Table 5.1 Flink Job Manager and Task Manager Memory Configuration Experiment

Deployment	Data Source rate	Checkpoint Interval (mins)	window size(mins)	State Size (MB)	Heap Size
1	10,000	02:00	240	435	6,144
2	10,000	02:00	240	435	3,072

The source rate is set to 10,000 records/second. Each deployment time is set to 2 mins and a checkpoint is also triggered every 2mins. In the end, we measure the observed processing rate for each deployment, and we show this in Figure. The result shows a that the deployment with bigger heap memory enabled the operator to process more records compared to the other deployment with smaller heap memory. These results therefore suggest that, in configuring a Flink infrastructure, the allocation and choice of heap memory must be carefully allocated to consider the workload, state size and desired operator processing rate.

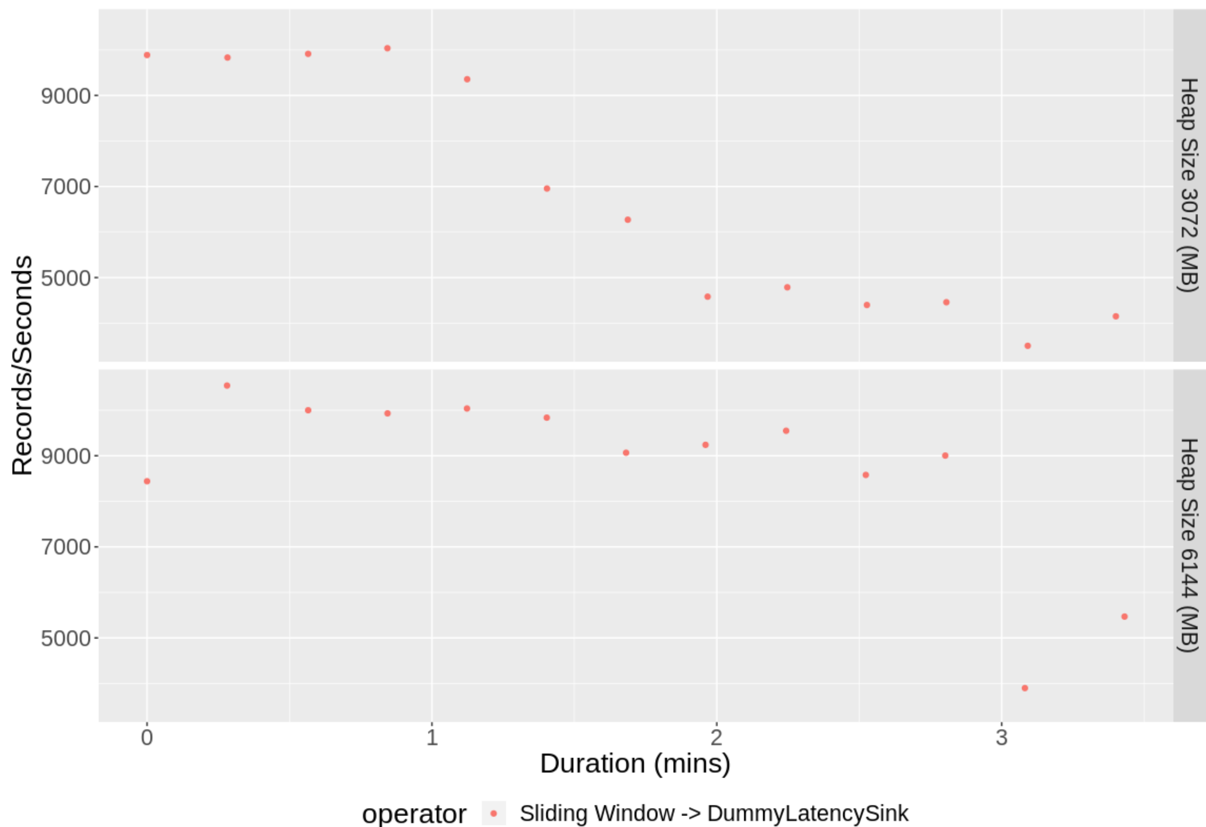


Fig. 5.2 Visualisation of Observed Processing Rate for Different Heap Memory Sizes of 3072 and 4144 MB.

5.3 Application State Size Dynamics

Distributed stream processing engines like Flink are commonly designed to provide users with the perception of a unified entity while simultaneously masking any challenges associated with

their distributed architecture. In the context of utilising a distributed computing framework such as Flink, it is common practice to establish certain presumptions regarding the current state of a pipeline that is actively employed within real-world situations. When rolling back a computation's full execution to the moment that its global state was captured, it is crucial to refer to the entire distributed state of the calculation as an atomic unit.

When reconfiguration is necessary or a partial failure prevents the pipeline from running properly, this is critical. The term "rollback recovery" is typically used to describe this strategy [32]. Distributed snapshotting protocols make it possible to roll back recovery by making an exact, full copy of the current state of a distributed execution. [21]. A directed graph of nodes and edges is an abstract representation of a distributed system, which is essentially a group of processes linked by data channels. The nodes and edges of such a network always represent the whole state during continuous system execution. In order to prevent the loss of any computational state or data, a consistent snapshot should capture the full state while taking execution dependencies into account..

5.3.1 Managed State

To retain a summary of the data viewed thus far, each stream action in Flink can declare its own state and update it continually. State is a fundamental component of a pipeline since it contains the complete status of the computation at any given point [14]. The conceptual foundations of the managed state are divided into two areas. For purely data-parallel stream operations, like getting an average for each key, the calculation, its state, and any other streams that go with it can be properly scoped and done separately for each key. Similar to how a relational GROUP BY assigns rows with the same key to the same set to calculate grouped aggregates, this also computes aggregates. This state is referred to as Keyed State. State can be stated at the level of an Operator State job, which is a parallel physical data flow task, for local per-task computations like a partial machine learning training model. The system's runtime manages and transparently partitions Keyed State and Operator State. Moreover, the system may ensure that any update actions on managed state will be mirrored exactly-once with respect to the input streams.

Keyed-State

Any computation in a data-parallel stream that is mapped to a user-defined key space will scope with the computation and any related state. The system typically receives data-stream records that contain a domain-specific key, like a user session ID, geo-location, or a device address. Flink allows a user to use the key in the broadest sense by: $S \longrightarrow K$ operation offered by the `DataStream` is an abstract type that lets you map any record from its model domain to a given keyspace. Under the key scope, state can be flexibly assigned within a user-defined function by using certain collections that the model makes available through the API and changing based on the state's properties.

Operator-State

Within the granularity of each concurrent instance of a task is another scope upon which state and computation can be stated (task-parallel). When a computation only applies to one particular physical stream partition, when a key cannot be used to scope state, the *Operator-State* is employed. This scope is used, for instance, by a Kafka ingesting source operator instance that must maintain offsets to specific Kafka partitions [132]. *Operator-State* follows a distribution scheme that, if possible, breaks state into smaller, more granular units. This enables the system to redistribute state when the operator's parallelism changes.

5.4 Evaluating The Impact of Window Selectivity Across Various Deployments

Window selectivity is an important concept in stream processing applications, where data is processed as it arrives in real time. Selectivity refers to the ability to filter and process only the relevant data within a specific time window. However, in practice, the length of the window and the sliding window period can vary widely between deployments, leading to variations in selectivity. These variations can impact the effectiveness and efficiency of stream processing applications, as well as the accuracy and reliability of any downstream analysis or, in this case, the query result [44]. Therefore, understanding the likely performance implications of the window size and slide period configuration choice is crucial to meet optimal business needs of streaming applications that rely on window-based processing.

5.5 System Design

Figure 5.3 shows our experimental pipeline setup and the relationship among the various unit of our setup. These units are self-contained, which makes it easy to scale the experiment. This setup architecture has four major areas: the data source (Nexmark Query 3 & Query 5), Streaming framework (Flink and DS2 scaling controller), Visualisation (we interrogate the offered load and state size and show their relationship and impact), and Datastore (this is stored in a local file). Our experimental aim is to simulate a growing application state size of a streaming environment and measure the impact on the operator's observed processing rate. We also collect the system resource utilisation metrics to know the highest resource-consuming system users during the experiment.

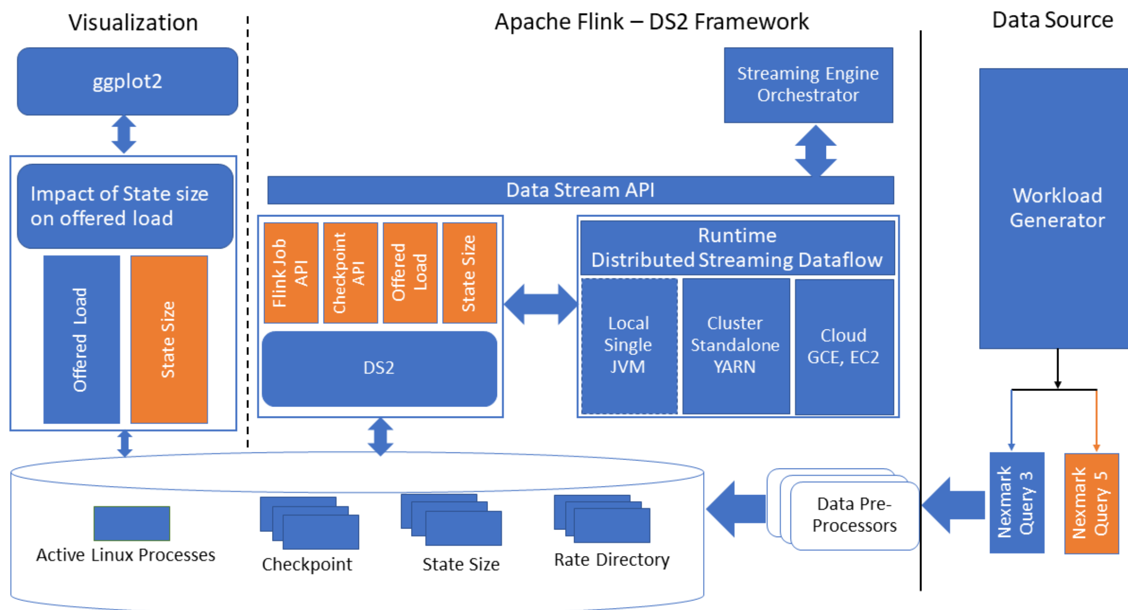


Fig. 5.3 Experimental System Architecture (Chapter 5)

Hardware and Software Configuration. Tables 5.2 and 5.3 show this experiment's hardware and software configuration parameters.

Table 5.2 Standalone Hardware Configuration (Chapter 5)

Hardware	Configuration
CPU	Intel® Core™ i5-8500 CPU @3.00GHz
CPU Cores	6
Memory	16GB
Disk	1TB
NIC	1000 Mbps
Kernel Version	4.15.0-74-generic

Table 5.3 Software Configuration (Chapter 5)

Software	Version	Number of instances
OS	Linux 84-Ubuntu	1
Flink	1.4.1	1
R	Rstudio 2021.09.0 Build 351	1
IntelliJ IDEA	2019.3.3 (Ultimate Edition)	1

5.5.1 Nexmark Workload

Nexmark Query 3

Nexmark Query 3 is a complex data processing query that involves filtering, joining, and aggregation operations. The architecture of Nexmark Query 3 typically involves the following components:

i. Data source: The source generates a stream of auction and person events. Here we have two java classes the `PersonGenerator.java` and the `AuctionGenerator.java` class. the `PersonGenerator.java` generates person events, which contain information about the person, such as their name, email, credit card, state, and city, while the `AuctionGenerator.java` generates auction events, which contain information about the auction, such as the auction ID, the seller ID, the initial bid and the description.

ii. Filtering: The filtering component filters out irrelevant events based on certain criteria, such as the event type and the auction state.

iii. Join: The join component combines the filtered auction events with the filtered person events based on the person ID. The join operation results in a stream of auction-person pairs.

iv. Aggregation: The aggregation component aggregates the joined events based on the person's city. The aggregation operation computes the total number of auctions for each city.

v. Key-value store: The key-value store is used to store the state of the aggregated results. This store is updated every time a new event is processed, and the results are aggregated.

vi. Sink: The sink component consumes the final results of the aggregation operation and can output the results to a file, a database, or a message queue.

The architecture of Nexmark Query 3 is designed to process a large volume of events in real-time, the choice of implementation and configuration of the system will depend on the specific requirements of the use case. In this experiment, our objective is to instrument the workload to generate a bigger state size that is required to test our hypothesis. The workload consists of various classes and components working together to perform the operations required by Nexmark Query 3 and process a stream of events in real-time. The summary of the workload operation is summarized in the equation below:

```
SELECT Istream(P.name, P.city, P.state, A.id)
FROM Auction A [ROWS UNBOUNDED], Person P [ROWS UNBOUNDED]
WHERE A.seller=P.id AND
              (P.state='OR' OR P.state='ID' OR P.state='CA')}
```

In the case of Nexmark Query 3, the flatMap function is used to apply a series of operations to the stream of auction and person events. The function starts by filtering the events using the `PersonFilter` and `AuctionFilter` classes, which only allow events that match certain criteria to pass through. The filtered events are then joined using the Join class, resulting in a stream of auction-person pairs. Finally, the joined events are aggregated using the Aggregator class, which computes the total number of auctions for each city. The flatMap function is an important part of the Nexmark Query 3 architecture, as it allows the system to process a large volume of events in real-time and produce the final results of the aggregation operation. To enable my workload to retain more state, I disabled the default operation of removing the `person.id` from the `auctionMap` collection.

Nexmark Query 5

In Nexmark Query 5, the state size management of a windowing operation is a crucial factor that can impact the query's performance. It allows the query to process events in a sliding time window, which can be used to aggregate events over a certain period [119]. The size of the window determines the amount of time for which events are aggregated and stored in state. When a sliding window is used, the state size can increase over time as new events are added to the window and older events are discarded [114]. State size management involves controlling the size of the state by managing the window size, the number of windows, and the number of events stored in each window. The objective is to keep the state size within acceptable limits while ensuring that the query maintains an optimal latency and throughput. The selection of a parameterisation method for a sliding window has an impact on the state size.

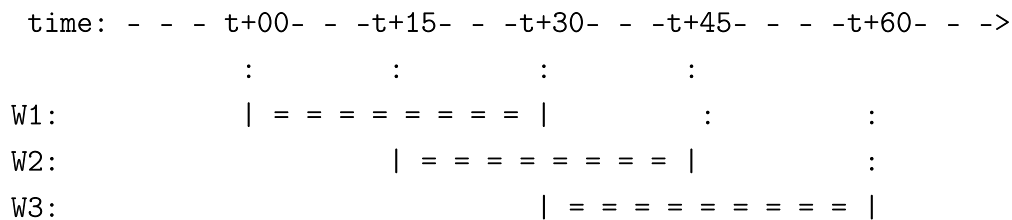


Fig. 5.4 Three sliding windows, with overlapping elements across windows.

A sliding time window can overlap, as shown in Figure 5.4 but it still represents time periods in the data stream. For instance, each window might record data for 30 seconds, but a new window might start every 15 seconds. The period refers to how frequently sliding windows open. As a result, our example's window and period would both be 30 seconds long. in my query 5 workload, I update the following parameters to generate more state.

```
.timeWindow(Time.minutes(60), Time.minutes(1))
```

The aforementioned parameter pertains to the configuration of the time window for a stream processing application in Apache Flink. Sliding window size can be determined by the number of data items or the duration of a period, among other factors. Sliding window period indicates how frequently a new input window can be initiated and can also be based on time, count, or a

predicate. Time.minutes(60) sets the sliding window size to 60 minutes. Data in the stream will be grouped and processed every 1 minute, across a past horizon of 60 minutes. Time.minutes(1) defines the sliding window period. Each processing step advances the window by 1 minute. Thus, each processing step creates a new window that spans data from the previous minute to the present minute and discards the prior window.

Other common windowing approaches are tumbling or fixed windows, as shown in Figure 5.5 below, where the size of the window is fixed, and events are processed in batches. Tumbling windows have the advantage of being simple to implement and easy to manage, but they can result in high latency and may not be suitable for processing real-time events. Increasing the size of the sliding window period can result in a larger state size and a higher memory footprint, even when the window period is not completed, due to the increased amount of data that needs to be stored in memory

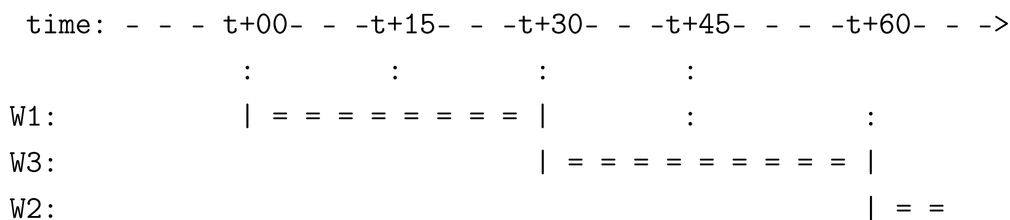


Fig. 5.5 Tumbling windows, with non-overlapping elements across windows

as intermediate results [69]. This trade-off must be carefully considered when configuring the sliding window period to ensure that the application can handle the increased state size while still delivering the desired performance and scalability. State size management is a crucial aspect of designing a scalable and performance streaming application, and it is important to choose the appropriate windowing operation and state size management strategy based on the requirements of the application [9]. Additionally, larger windows may result in longer processing times, as more data needs to be processed for each window, which can impact the overall performance and scalability of the application [69].

5.5.2 Experimental Setup

In this chapter, we ran four major experiments using two different workloads. The first and second experiments aim to show the impact of state size on operators processing capacity. In these experiments, we leverage Nexmark Query5 and Nexmark Query 3 workloads. The third and fourth experiment seeks to examine the impact of window selectivity on state size. This experiment also uses Nexmark Query5 workload.

First Experiment (State size impact on capacity using NexMark Query 5)

Modelling impact of offered load on state size. The aim of this study is to demonstrate the impact of a growing state size on the processing capabilities of streaming operators. We keep a constant arrival rate of 20,000 records/seconds with a transient window size (20, 40, 60, 80,

100, 120, 140, 160, 180, 200, 400 and 600 minutes). The reason for selecting these window sizes was to establish a consistent upward trend in state size. The study involved conducting 12 deployments, each with a consistent checkpoint interval of 10 minutes as shown in Table 5.4. The total deployment circles were carried out during a single experimental period. Upon completion of each experiment, the observed and true processing rates are collected for each sliding window configuration as well as the state size metrics for each deployment.

Table 5.4 First experiment configuration parameters

Deployment	Data Source rate	Checkpoint Interval(mins)	Window size(mins)	Back Pressure Status
1	20,000	10:00	20	OK
2	20,000	10:00	40	Low
3	20,000	10:00	60	High
4	20,000	10:00	80	High
5	20,000	10:00	100	High
6	20,000	10:00	120	High
7	20,000	10:00	140	High
8	20,000	10:00	160	High
9	20,000	10:00	180	High
10	20,000	10:00	200	High
11	20,000	10:00	400	High
12	20,000	10:00	600	High

After concluding 12 executions of the first experiment measuring the effect of state size on capacity using NexMark Query 5, the harmonised results from the 12 executions is depicted in Figure 5.6 illustrate the processing capability of the true and observed processing rates in Query 5. As indicated previously, we maintained a constant arrival rate of 20,000 records per second with a window size of 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 400, and 600 minutes.

During the initial warm-up phase of this experiment as shown in Figure 5.6 system resources are typically underutilized, leading to a surge in the true processing rate, reflecting the system's higher processing capability when resources are readily available. However, as the system becomes fully engaged and resources are more efficiently utilized, the true processing rate decreases from its peak, aligning more closely with the actual processing capacity of the system

under sustained workloads. In contrast, the observed processing rate accurately reflects the number of records processed (20,000/records), considering real-world factors such as serialisation/deserialisation and processing complexities, providing a more practical measure of the system's performance in steady-state conditions. This dynamic interplay between the true and observed processing rates showcases the system's ability to adapt and optimize its resource utilization as it transitions from an initial high-capacity state to a more stable operational mode. A consistent decrease in the processing rate is observed with an increase in state size caused by the increase in window size as demonstrated by the data spanning from 1:30 minutes to 2 hours.

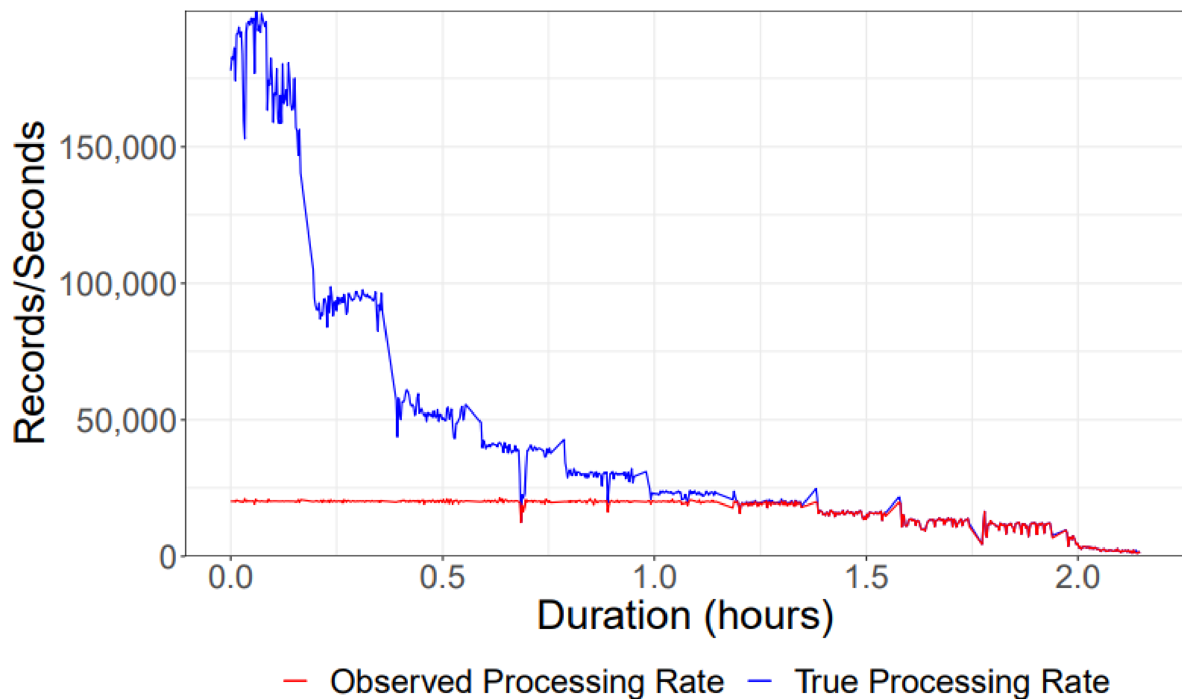


Fig. 5.6 Evolution of True and Observed Processing Rates for Nexmark Query 5.

The findings depicted in Figure 5.7 demonstrate the relationship between the observed processing rate and the growth of state size. we measured the observed processing rate because it gives the exact number of records processed by the operator. A decrease in the processing capacity of operators is noted with the increase in state size over time. The findings of this study provide evidence in favour of our hypothesis that a higher state could impact the throughput of streaming operators.

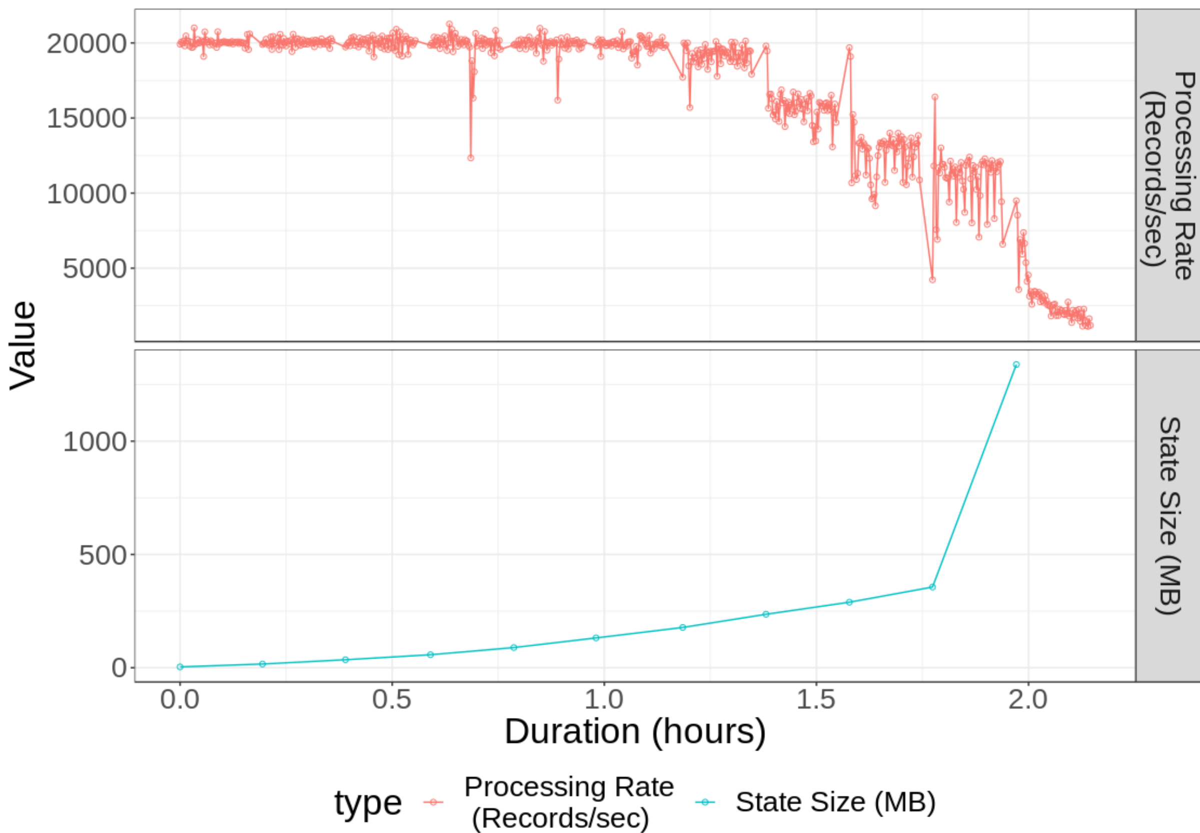


Fig. 5.7 For Nexmark Query 5, the Evolution of Observed Processing Rates and the Interplay with Application State.

Second Experiment (State size impact on capacity using NexMark Query 3)

The preceding experiment were conducted utilising Nexmark Query 5, which serves as an exemplar of sliding window and combiner. Consequently, we proceed to evaluate the state size impact on operator capacity using Nexmark Query 3, which serves as an exemplar of incremental join and filter. Query 3 comprises five operators. (Auction, Person, Filter, Incremental Join and Sink). The value of each operator's parallelism has been set to one since we are running on a single-node cluster installation.

Table 5.5 Second experiment configuration parameters

Deployment	Source Rate		Checkpoint Interval (mins)	Deployment Time	Back Pressure Status
	Auction	Person			
1	1,000,000	500,000	2	15mins	OK

The workload comprises two source operators, namely the Auction operator and the Person operator. The Auction source has a data processing rate of 1,000,000 records per second, whereas the Persons source has a rate of 500,000 records per second as shown in Table 5.5. The arrival rate specified is deemed adequate for producing the necessary state size as per the

requirements of this experiment. The duration of deployment is 0.25 hours, while the frequency of checkpoint intervals is 2 minutes. During the experimental deployment, the system underwent seven checkpoints and state metrics were collected for each of them.

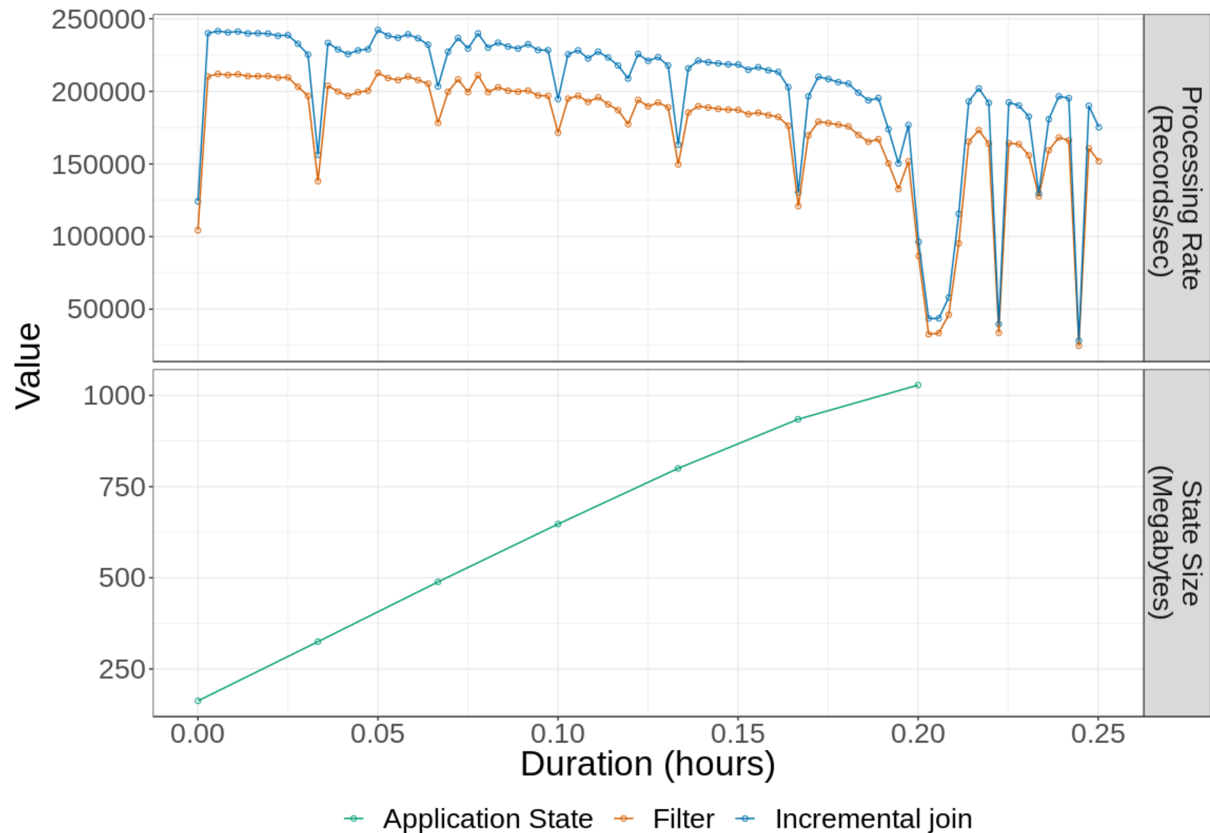


Fig. 5.8 For Nexmark Query 3, the Evolution of Observed Processing Rates for Filter and Incremental Join operators, and the Interplay with Application State.

Unlike the Query 5 workload, We experienced no back pressure effect in for this experiment, giving us a fare judgment without a backpressure effect. We collect the true and observed processing rate of the Filter and Incremental Join operator and the result also shows a decline in the operator's true and observed processing rate over time as the state size grows. Similarly, Figure 5.8 the evolution of processing rates for Filter and Incremental Join operators, and the interplay with application state.

Third Experiment (Relationship between parameters of sliding window length and state accumulation)

Building upon the findings of the second experiment, a comprehensive analysis is conducted on the individual data components that comprise the Query 5 workload state. The objective is to augment our comprehension of the internal mechanisms of the application state and to evaluate and compare the impact of various window size configuration on a stream processing system. A transient offered load is implemented, consisting of three distinct tasks. Each task exhibits an arrival rate that initially increases and subsequently decreases, following a predetermined sequence (100,000, 200,000 and 50,000). Each task has a duration of one minute.

The available window sliding periods are 5, 10, and 20 seconds. A sliding window size of 30 seconds is upheld for all deployments, while the overall duration of the experiment is estimated to be around 3 minutes. Following this, the auctions, bids, and date time values are collected, and subsequently, the quantity of bids produced in each window configuration is assessed. The absence of back pressure in this experiment allows for a focused examination of the effects of windowing in a narrow window size, contingent upon the chosen window size configuration.

Table 5.6 Third experiment configuration parameters

Deployment	Source Rate			Deployment Time (mins)	Window size (sec)	Sliding Period (sec)
	Task 1	task 2	task 3			
1	100,000	200,000	50,000	3	30	5
2	100,000	200,000	50,000	3	30	10
3	100,000	200,000	50,000	3	30	20

In the following, we present Figures 5.9 and 5.10, which depict the impact of windowing on instantaneous state size with a sliding window size of 30 seconds, with sliding periods of 5,10 and 20 seconds respectively. Figure 5.9 illustrates how state size evolves over time. We see comparable results during the rising arrival rate phase in the first 1.5 minutes. We see the greatest divergence when the arrival rate drops. We see the 5-second slide length responding most rapidly, while longer slide lengths retain elevated state sizes for a longer period. This elevated state size represents a gap not currently captured by approaches which leverage instantaneous arrival rate as a proxy for state size.

Figure 5.10 depicts the facets pertaining to the configuration of the individual sliding window, which aligns with the source injection rate. Reducing the sliding period yields increased granularity in the obtained results, as demonstrated below.

The percentile value of state size over the experimental period for various window configurations is illustrated in Figure 5.11 below. A consistent window size of 30 seconds is upheld. Each line on the ECDF plot represents a different sliding period: the blue line represents a sliding period of 5 seconds, the red line represents a sliding period of 10 seconds, and the green line represents a sliding period of 20 seconds. Each step represents a change in the cumulative proportion of sliding window periods that have observed a particular number of bids or fewer. We observe a slower reaction in the longer sliding length configuration than the smaller length, specifically around 0.75 proportion. This corresponds to the portion of our third experiment, in which the offered load is decreased from 200,000 to 50,000 records per second. Similarly, we demonstrate this trend in Figure 5.9, and Figure 5.10. We see a a greater degree of granularity for the shorter window length of 5 seconds compared to sliding lengths of 10 and 20 seconds. It is worth noting that due to the duplication of data elements that occurs in a sliding window, a

sliding period with shorter length yields increased granularity than a longer sliding period [114]. This is illustrated in Figure 5.10.

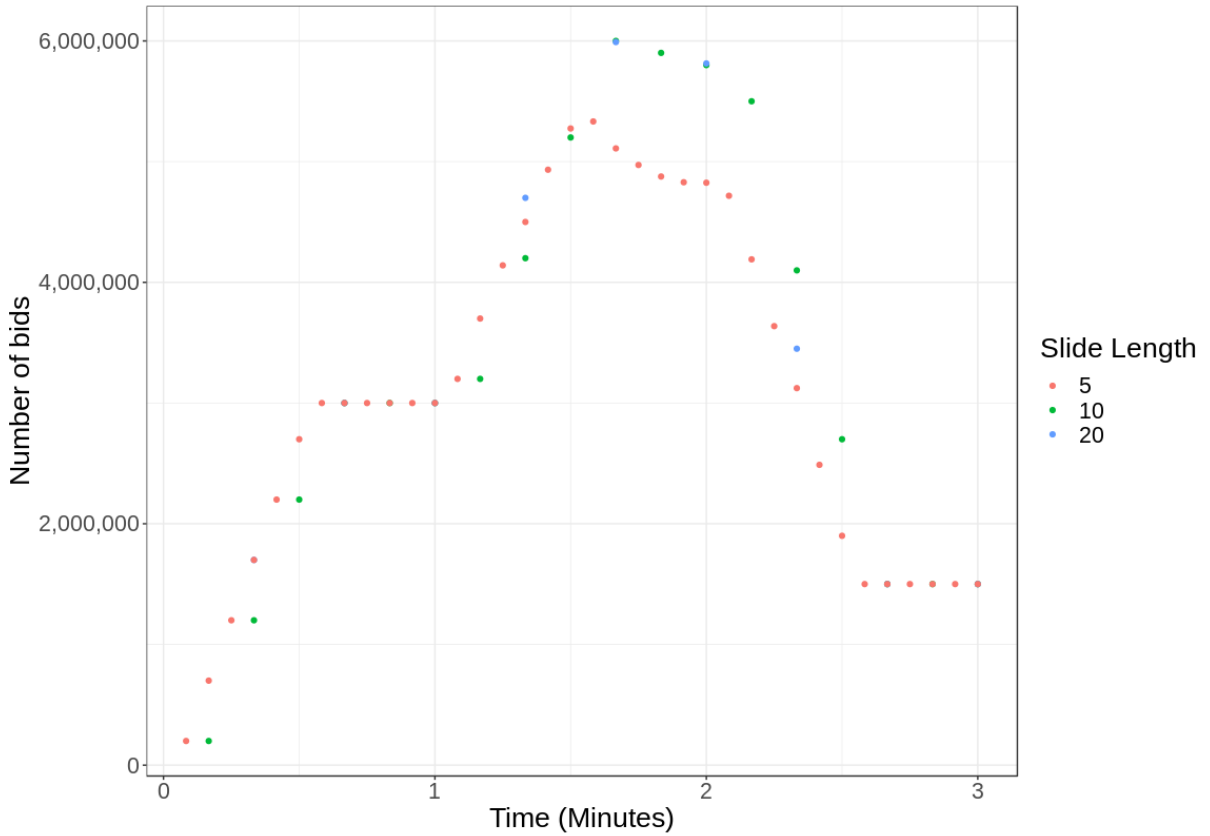


Fig. 5.9 Visualisation of state size (number of bids) for a window operator of length 30 second, for different slide lengths of 5, 10 and 20 seconds.

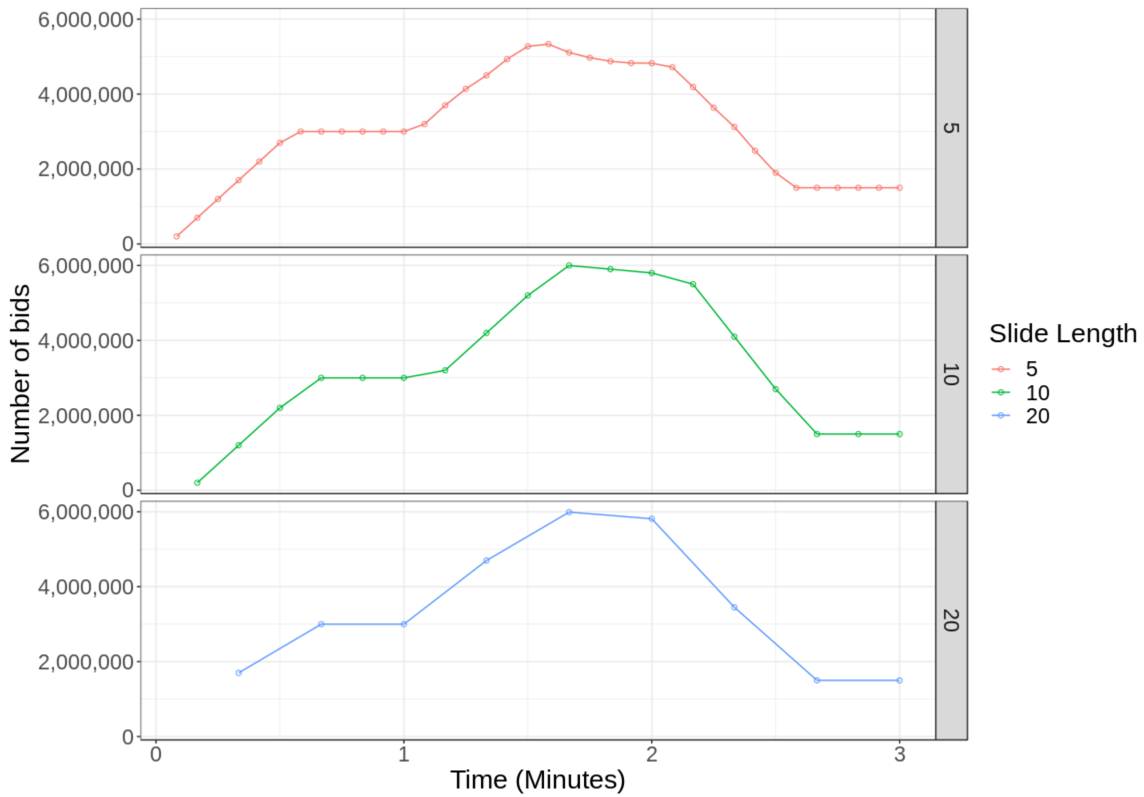


Fig. 5.10 Visualisation of state size (number of bids) for a window operator of length 30 second, for different slide lengths of 5, 10 and 20 seconds.

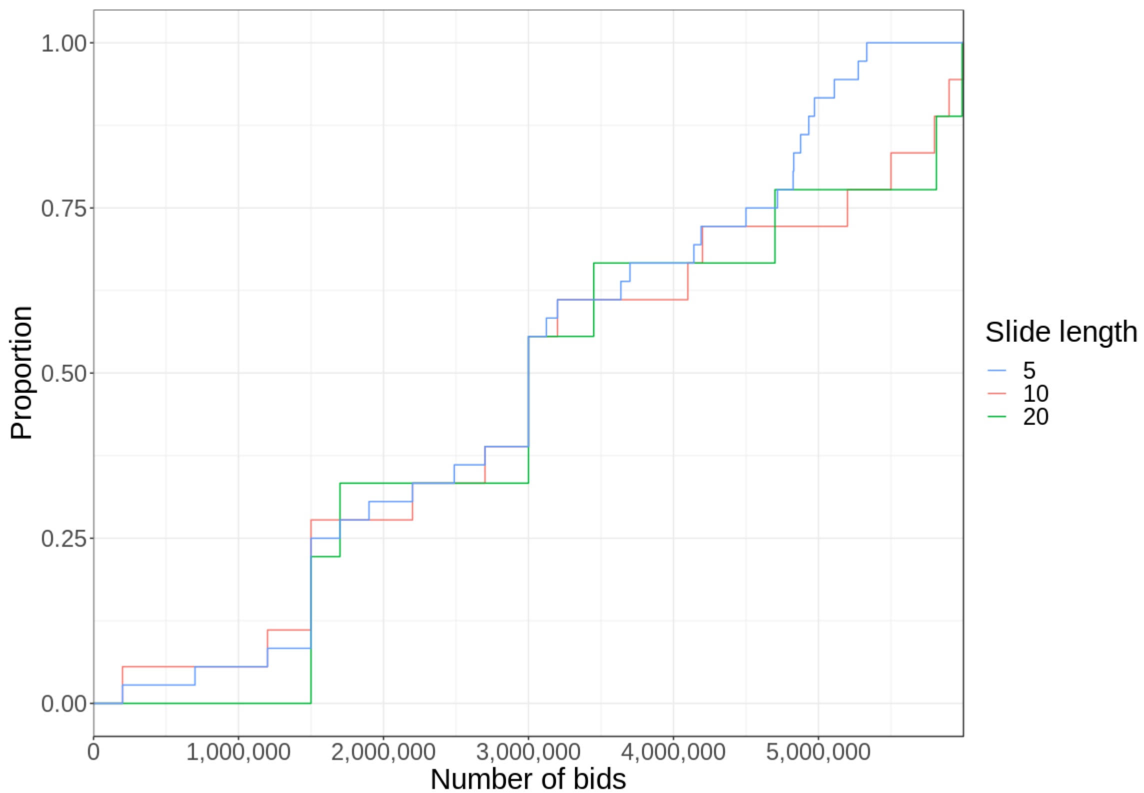


Fig. 5.11 ECDF plot of instantaneous state size for a sliding window operator of 30 second, with slide lengths of 5, 10 and 20 seconds.

Forth Experiment (Relationship between parameters of sliding window size and state accumulation)

Similar to the third experiment, the aim of this experiment is to examine the impact of different sliding window sizes on the state size accumulation. A transient offered load consisting of three distinct tasks is adopted. Each task has an arrival rate of 100,000, 200,000 and 50,000. Each task has a duration of one minute. The available window size durations are 20, 40, and 60 seconds. A sliding period of 5 seconds is upheld for all deployments, while the overall duration of the experiment is estimated to be around 3 minutes.

Our experimental findings as illustrated in Figure 5.12, indicate that the 60 seconds window, accumulated a higher number of bids in comparison to the 40 and 20 seconds window respectively. When using a sliding window of a larger size, more data is accumulated in each window, compared to a smaller window size. This is because the larger window size covers a longer period of time or more events, which means that more data points are included in each window [114].

Table 5.7 Forth experiment configuration parameters

Deployment	Source Rate			Deployment Time (mins)	Window size (sec)	Sliding Period (sec)
	Task 1	task 2	task 3			
1	100,000	200,000	50,000	3	20	5
2	100,000	200,000	50,000	3	40	5
3	100,000	200,000	50,000	3	60	5

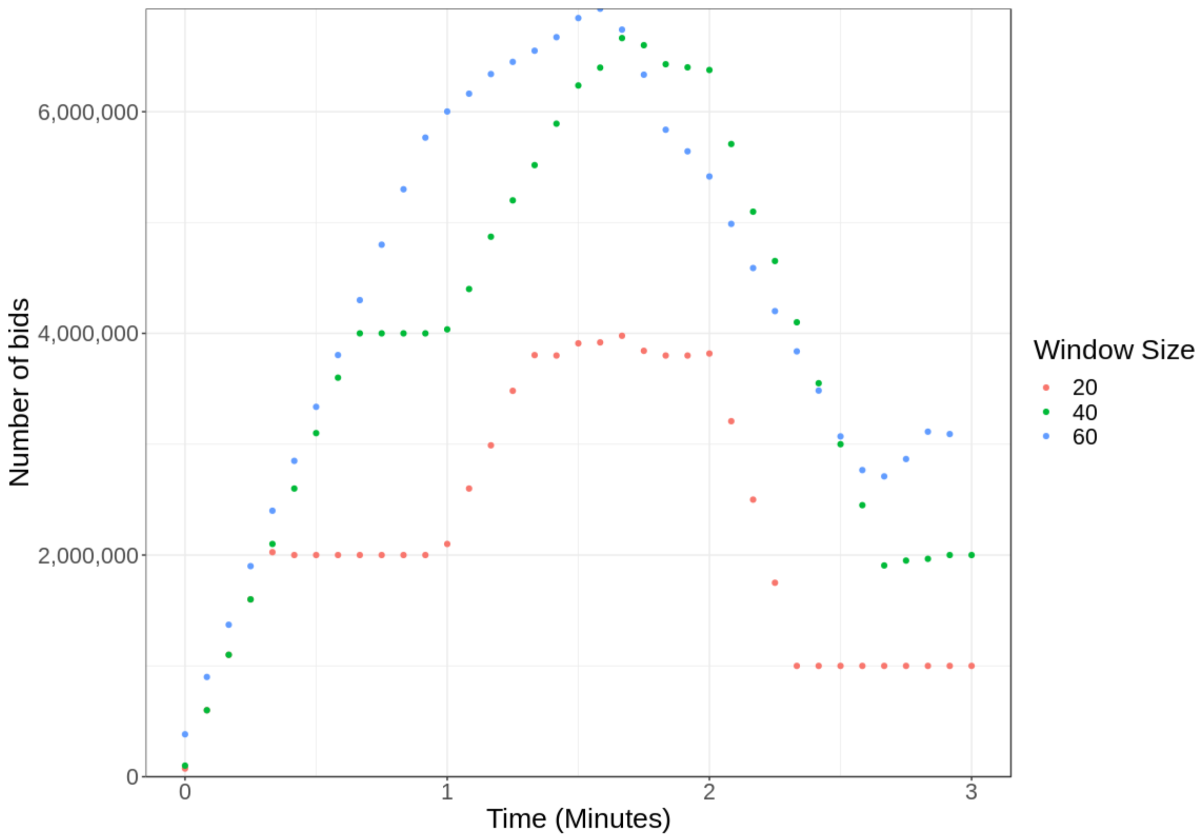


Fig. 5.12 Visualisation of state size (number of bids) for a window operator of slide lengths 5 second, for different window sizes of 20, 40 and 60 seconds.

Figure 5.13 and Figure 5.14, demonstrate the impact of different window size configuration parameters on the operators' true and observed processing rate. The findings in both figures provide evidence to substantiate the claim that streaming operators exhibit superior processing capabilities in scenarios where the window sizes are comparatively smaller. Window size of 20 seconds has a higher processing potential than the 40 & and 60 seconds window.

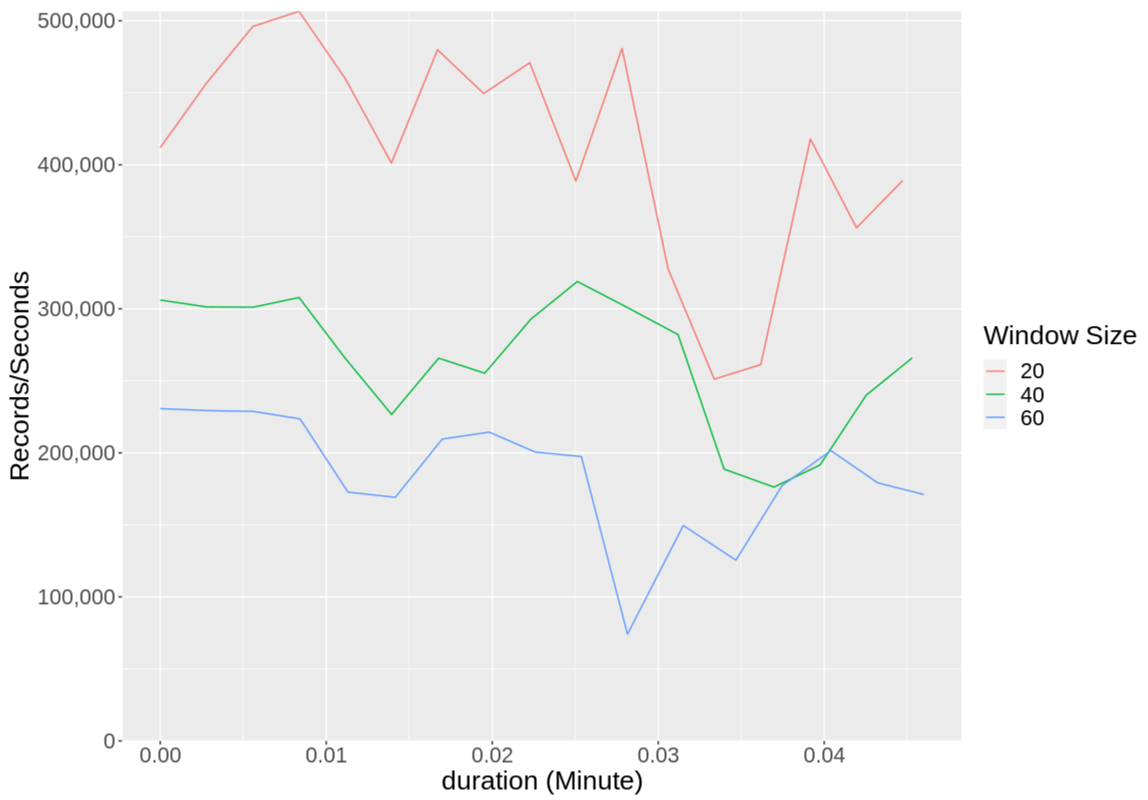


Fig. 5.13 For Nexmark Query 5, the evolution of true processing rates for different window sizes of 20, 40 and 60 seconds.

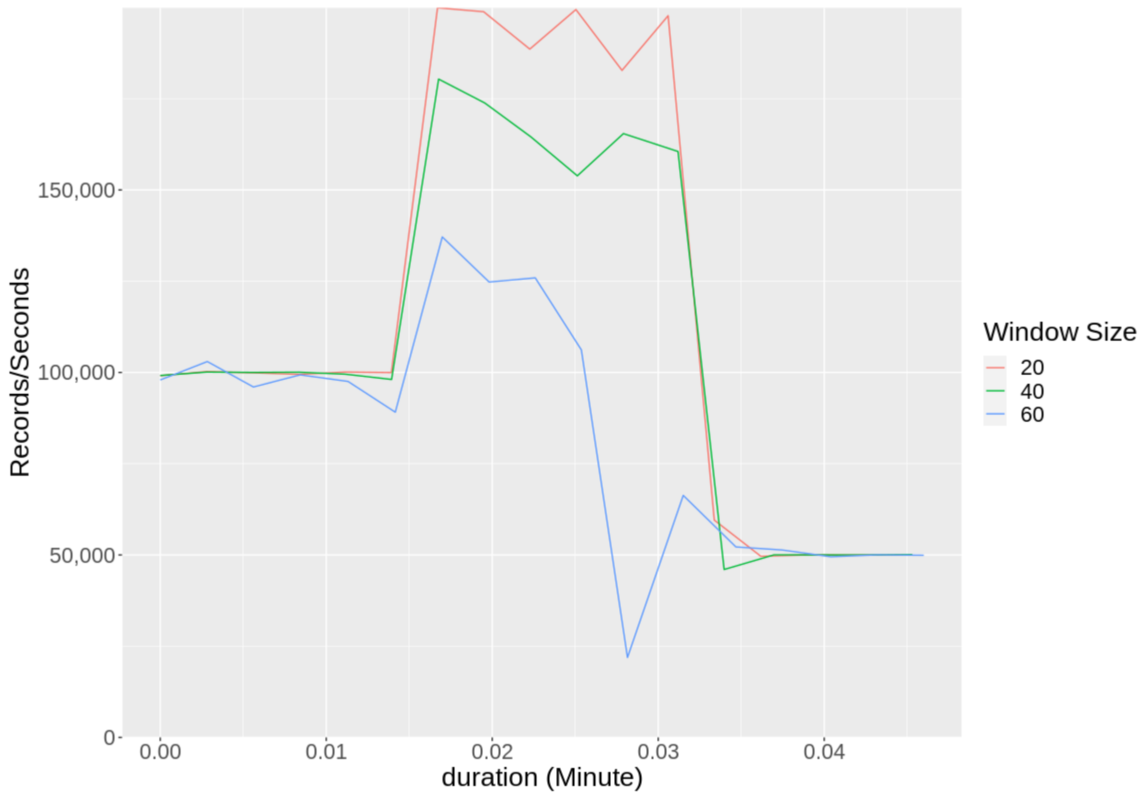


Fig. 5.14 For Nexmark Query 5, the evolution of observed processing rates for different window sizes of 20, 40 and 60 seconds.

5.6 Summary of Experimental Results

Drawing from the results of the third and forth experiment, we analyse the influence of window selectivity on operator processing to determine whether a corresponding effect exists between the operators and the state size. The ECDF depicted in Figure 5.15 and Figure 5.16 portrays the observed processing rate of the operators as observed during our windowing experiment. A consistent window size of 30 seconds is upheld. Each line on the ECDF plot represents a different sliding period. Each step corresponds to the change in the cumulative proportion of the sliding window period.

Figure 5.15, shows a comparable processing rate for each windowing length configuration during the rising arrival rate phase. However, we observe a slower reaction in the observed processing rate in the longer sliding length configuration compared to the smaller length, specifically around 0.65 proportion. This corresponds to the portion of our third experiment, in which the offered load is decreased from 200,000 to 50,000 records per second. This suggests a potential challenge to operators reaction to offered load in a window based application.

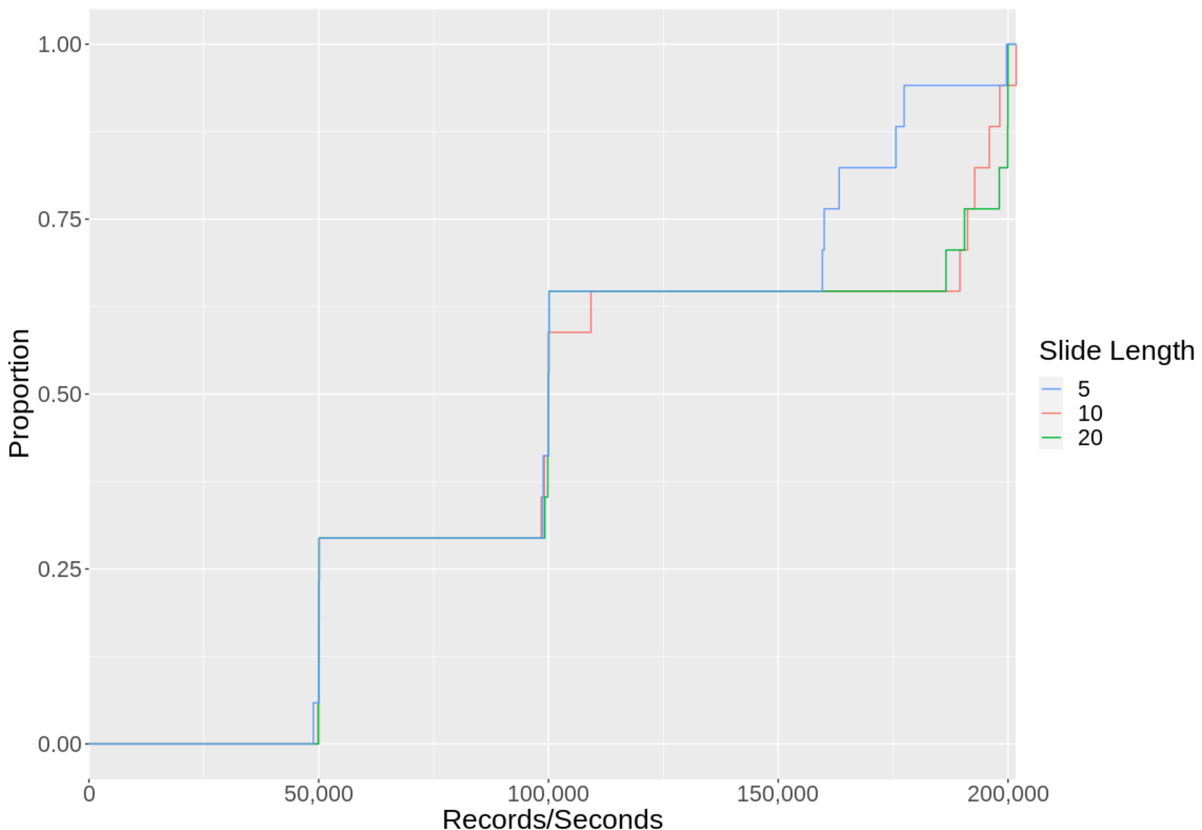


Fig. 5.15 ECDF plot of the evolution of observed processing rates for a sliding window operator of 30 second, with slide lengths of 5, 10 and 20 seconds.

In Figure 5.16, A consistent window length of 5 seconds is upheld with different sliding window sizes of 20, 40 and 60 seconds. Each line on the ECDF plot represents a different sliding window size. We see a different behaviour in the operators processing rates for different window size configurations compared to the window length. We observe a higher processing rate for the smaller window size of 20 seconds compared to the 40 and 60 seconds window size. This is seen

as early as 0.25 proportion, but it's more common around 0.65 proportion, when the system has accumulated more state.

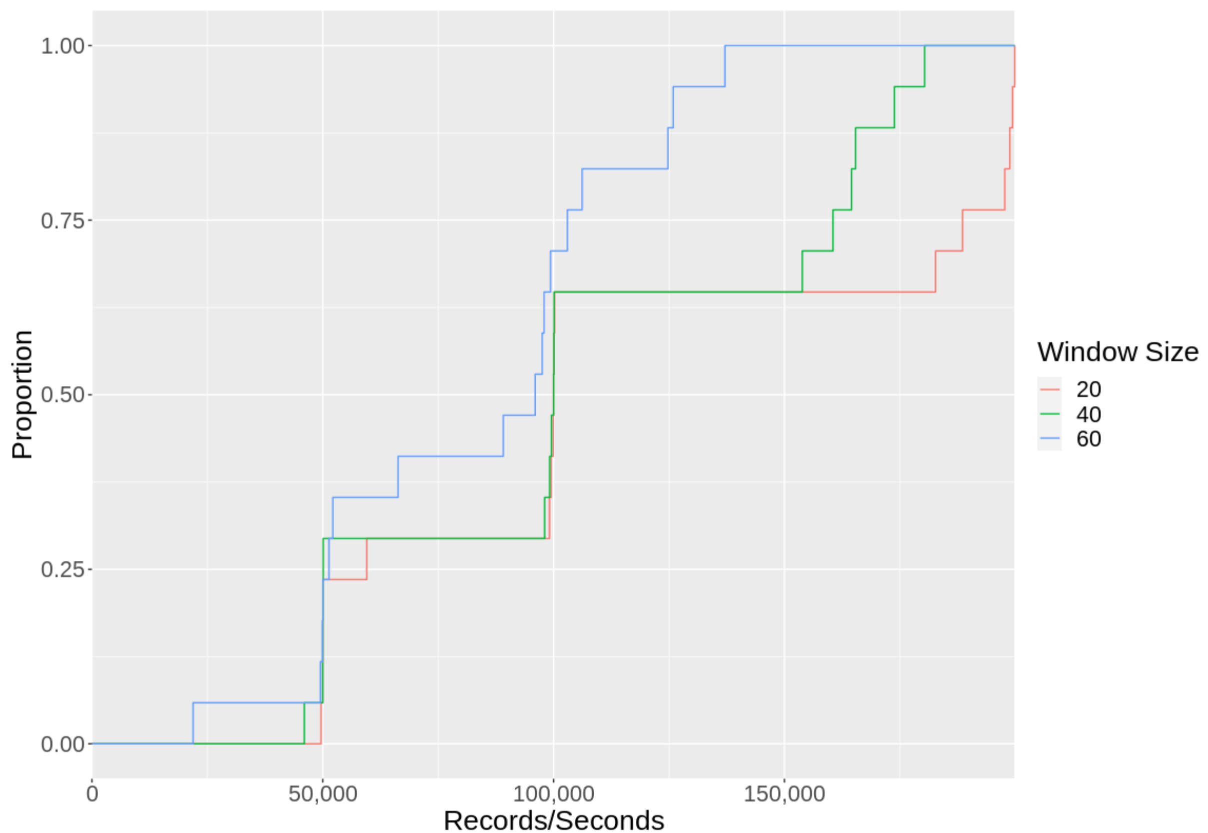


Fig. 5.16 ECDF plot of the evolution of observed processing rates for a slide length of 5 second, for different window sizes of 20, 40 and 60 seconds

In addition to the results shown above where we the see the impact of state size on the operators observed processing rate, we further interrogate the system resource utilization capacity, by checking the 10 top highest CPU and memory consuming user and our findings reveal that the java runtime being the highest system CPU and memory consuming user process. This is illustrated in Figure 5.17 and 5.18. The values for this experiment came from our first experiment, which had 12 executions, each with a consistent checkpoint interval of 10 minutes and a total experiment time of two hours. We keep a constant arrival rate of 20,000 records/seconds with a transient window size (20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 400 and 600 minutes).

This results shows the instantaneous time value of CPU time and share of physical system memory the tasks has used since the last update. Following the results from the various experiments carried out in this chapters, we observe a similar trend from the two workload (Nexmark Query 3 & 5) used.

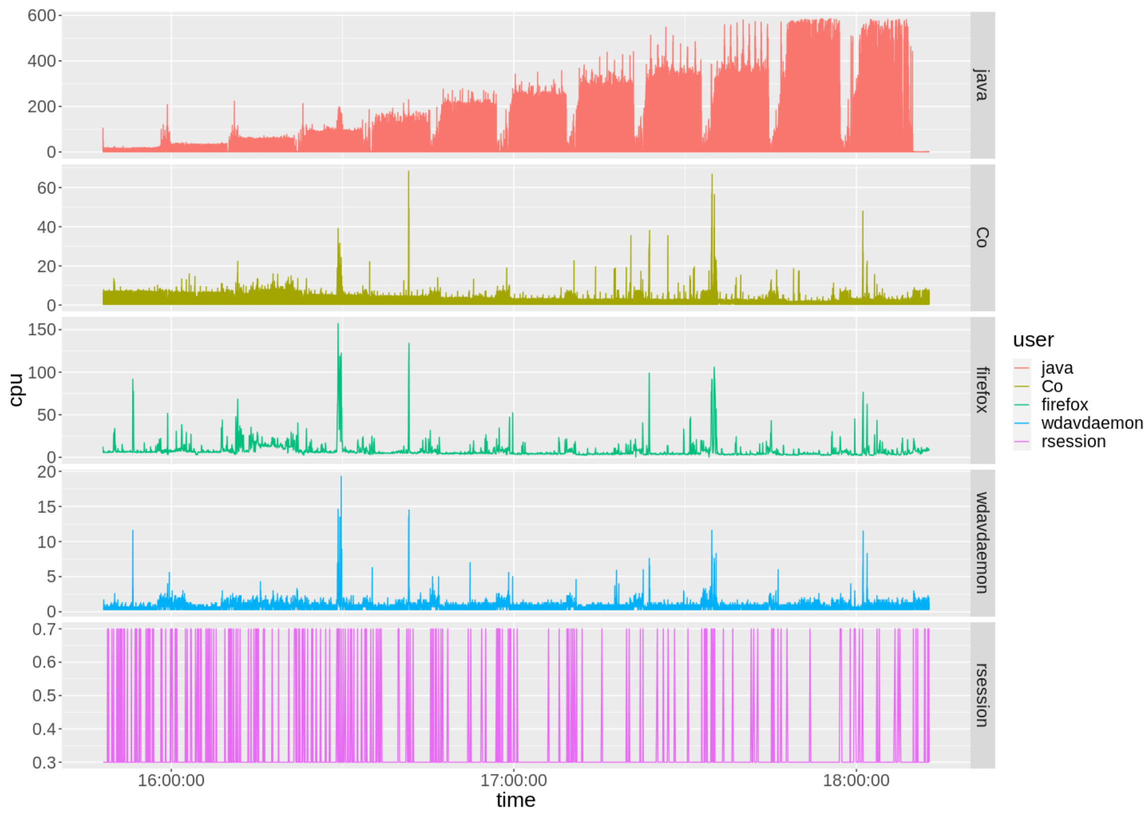


Fig. 5.17 Top 5 CPU consuming processes during experiment

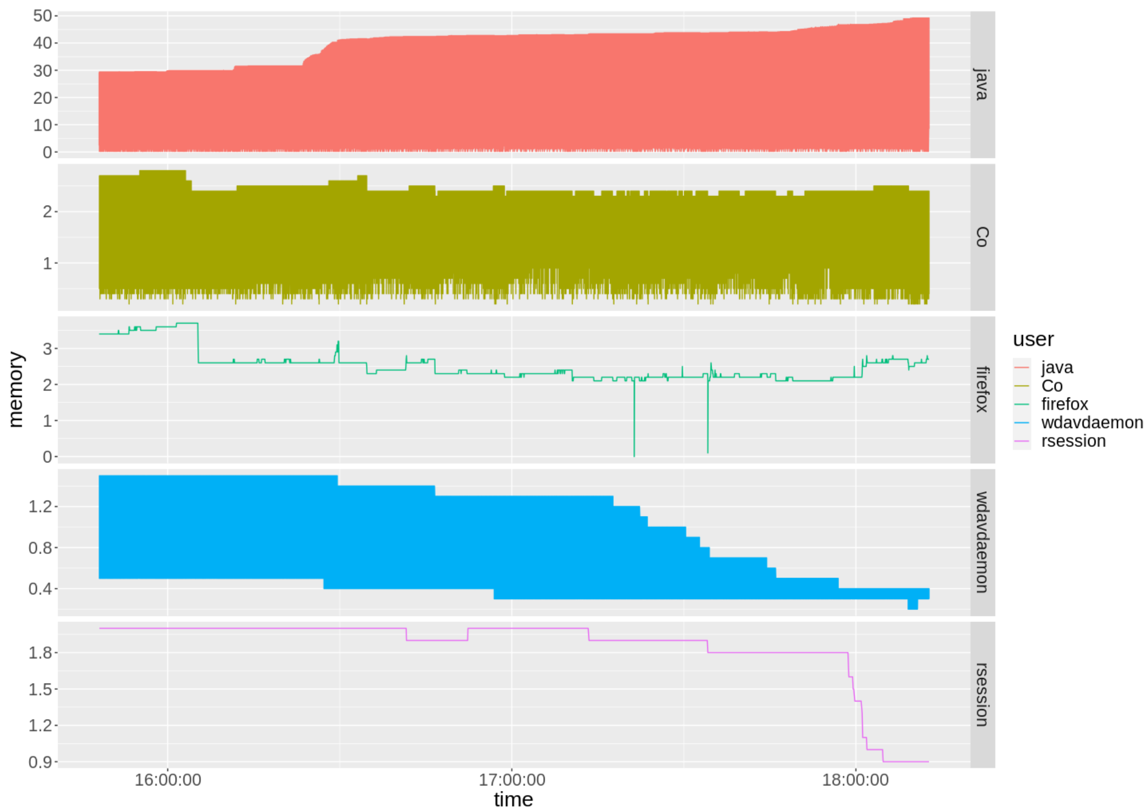


Fig. 5.18 Top 5 Memory consuming processes during experiment

Our experimental findings show that without sufficient knowledge of the relationship between these two metrics pose a potential challenge for autoscalers. Streaming applications with larger

state or long window size can lead to a slow convergence between the state size and the offered load. Like DS2, that relies on the true and observed processing rate to make a rescaling decision, most autoscalers do not consider the impact of state size when making the decision on when and how to scale up or scale down a streaming application computational resource. Our experimental results show that metrics like state size can force an autoscaler to scale down too quickly due to a reduced offered load. However, this reduction in the observed processing rate is artificial and not a natural reduction in the number of events injected per second. This leads the autoscaler to make a wrong call, and will struggle to recover from the scaling process leading to inefficiency.

5.7 Conclusion

The experimental findings strongly support the assertion that larger window sizes lead to the accumulation of a greater amount of state, thereby exerting a notable impact on the throughput of streaming operators. This outcome is attributed to the fact that processing a larger number of elements within the window significantly influences the observed processing rate of operators, in contrast to smaller windows that encompass a smaller state. The evidence gathered from the experiments underscores the critical role that window size plays in shaping the performance and efficiency of streaming application operators, providing valuable insights for optimizing and fine-tuning these systems in a window-based environment.

Stream processing applications that have a significant state or extended window duration may experience variations in how the application state aligns with the operator throughput. Many auto-scaling mechanisms do not account for the impact of state size when determining how to scale. Our practical observations reveal that as the state grows, an auto-scaling system might quickly downscale due to a decline in the operator's processing rate. However, this decrease is contrived and does not accurately depict the arrival rate. Such scaling decisions made by the auto-scaler could pose challenges for effective rescaling and ultimately reduce efficiency.

To evaluate the effects of window selectivity options in streaming applications, an experiment was done. We demonstrate the impact of different sliding window length configurations on instantaneous state size. Therefore, our results lend credence to the claim that window size selectivity is essential for streaming programmes to run as efficiently as possible in a windowed environment.

To motivate early researchers who are unfamiliar with state management within the Flink streaming framework, an experiment was conducted to assess the effect of system resources on the throughput of streaming operators. Specifically, the impact of Flink's memory on the processing capacity of the operators was investigated. The findings indicate that the deployment utilising a larger heap memory exhibited a greater capacity for processing records, whereas the deployment utilising a smaller heap memory demonstrated a reduced processing capacity as the state size increased. The aforementioned outcome implies that when setting up a Flink infrastructure, the allocation and selection of heap memory should be meticulously considered with regards to the workload, state size, and desired operator processing rate. Additionally, it has been demonstrated that with an increase in state size, the Flink execution process utilises a significant proportion of the central processing unit and physical memory of the local machine.

Subsequently, the expansion of the state size and depletion of system memory resources can lead to process crashes.

5.7.1 Experimental Challenges

In this sub section, we provide bullet points of experimental challenges faced while carrying out this experiment and limitations of the overall experimental body. This is listed in no specific order.

- Longer window length enables bigger state size. However, this forces the experiment to crash when a longer (for example, 10mins) checkpoint interval is set. We therefore went for shorter checkpoint intervals to avoid checkpoint process overlap and avoid crashing the system.
- I observed that increasing the sources rates alone on both workloads was not enough and had no significant impact on the state size as I initially assumed, especially in Query 5. Therefore instrumenting the workload is required to influence a growing state size.
- I could not simulate a scenario where the observed processing rate of a deployment with an initial high source rate increases when the state size begins to reduce. I wasted time trying to build an experiment to show this scenario. Unlike the other experiment, where I maintained a constant source injection rate and changed the window length intermittently, I could not do the reverse for the reason stated above.

5.7.2 Future Work

In this subsection, we provide bullet points of future direction and next steps that could be useful and complementary to this body of research. This is listed in no specific order.

- Develop a dynamic scaling resource multiplier. Experimental Options for developing a dynamic multiplier include:

  Version 1. P(baseline)

  Version 2. P * i%, e.g. Do what DS2 says but based on the rescaling duration, increase them by 20% every minute.

  Version 3. p * forecast of where offered load might go
  where

  P = the parallelism which DS2 suggests

  i = the predicted time to rescale

- Evaluate the impact of state size on the streaming application data initialisation injection efficiency. When an offered load is assigned to Flink, for example, 10,000 records per second, we observe a staggered growth in the processing of records during the initial

startup face that later normalises over time. The optimisation of the initialisation period and the ability for the system to reach its optimal processing capacity within a short time can be a distinctive factor amongst streaming systems, especially when real-time data for analytics is required. Future research can explore measuring the time the operator takes to reach its optimal processing rate over different state sizes.

- Future research can explore the impact of a larger collection arising from a larger state size. Iterating over a larger number of elements could have a resulting impact on the operators observed processing rate.
- A useful next step is to juxtapose the influence of sliding window selectivity with alternative windowing methodologies in terms of managing state size.

Chapter 6

Conclusion

6.1 Thesis Summary

This thesis explores auto-scaling in streaming applications, with a particular emphasis on identifying factors that may impede auto-scaling controllers from fully realising their potential. The aim of this study was to enhance the existing body of research on the optimisation of rescaling operations in streaming applications. We hope to achieve this by identifying the constraining factors and suggesting an improved approach that will equip auto-scaling controllers with additional insights. This, in turn, will lead to a more resilient and efficient scaling decision-making process in a broad range of use cases.

Chapter 3 examines the Apache Flink engine and the development of a basic distributed topology to gain comprehension of Flink streaming applications. We examine the issue of workload imbalance and disparate data injection rates among operators in a real-time distributed stream processing engine. In addition, we assess the notion of rescaling an application, a technique employed to recover from a system outage or reconfigure a running streaming topology by adjusting computational resources either upwards or downwards. We conduct different experiments on rescaling operator's parallelism with the aim of understanding how to scale-up and scale-down streaming operators and evaluate the impact of these topology adjustments on the overall system. The results of our study suggest that Flink exhibits non-uniform task distribution across various operators.

Chapter 4 builds upon the knowledge gained in Chapter 3 and examines the effects of scaling time and resources on the attainment of an optimal scaling process. We show that extended scaling time, particularly in a workload environment that fluctuates rapidly, presents a possible obstacle to a streaming application when it resumes after an automatic scaling procedure. The experimental results emphasise the significance of state size and end-to-end checkpoint duration in the process of rescaling. They indicate that as the state size grows, the duration required for auto-scaling also increases. It is possible that additional state may have been accrued during a rescaling time frame, resulting in a recurring need for rescaling and potential resource depletion, ultimately leading to multiple rescalings of the application. The recurrent nature of this task has the potential to negatively impact the performance of the system.

In addition, the existing dataset was utilised to develop and train predictive regression models, utilising resampling cross-validation techniques. The selection of the optimal model was based on performance statistics and predictive accuracy. This model was subsequently utilised to forecast the duration of rescaling for a running application across various forecasted state size values. The results of our study support our hypotheses that in a distributed data streaming environment characterised by constant change and unpredictability, a larger volume of data can accumulate rapidly within a rescaling time window. It is recommended that the forecasting of workload characteristics and the growth rate of state size be taken into consideration when formulating a scaling policy for facilitating a streaming application to effectively manage unforeseen resource demand.

Chapter 5 reveals that the utilisation of offered load as a proxy for state size, as assumed by most existing auto-scalers, is inadequate. We explore the impact of window selectivity on the performance of streaming applications, demonstrating the impact of different sliding window length configuration on instantaneous state size. Our results lend credence to the claim that window size selectivity is essential for streaming programmes to run as efficiently as possible in a windowed environment. Furthermore, we conduct empirical evaluations to assess the relationship between operators' throughput and state size. Our empirical findings indicate that the growth of the state can compel an auto-scaling system to downscale rapidly, owing to a decrease in the operator processing rate. Nonetheless, the decrease in question is contrived and does not accurately depict the arrival rate. Such decisions by the auto-scaler may result in difficulties in rescaling, leading to reduced efficiency.

6.2 Limitations

We present the limitation of our research and subsequently examines the potential challenges to the validity that emanate from these limitations. The primary limitations of this research are deemed to be the following.

L1 Single streaming platform & Single cluster instance The results of this experiments were generated using Apache Flink running on a single node cluster.

L2 Single auto-scaler The experimental outcomes presented in this study were obtained utilising a single state-of-the-art auto-scaler, DS2.

L3 Single window type Our windowing experimental results leveraged a single window type, Sliding window.

Subsequently, an examination of the limitations will be conducted, with a focus on the potential risks to the construct, internal and external validity [129].

Construct Validity This experiment was run on Ubuntu `4.15.0-74-generic` with a single instance of `Flink version 1.4.1.` **(Limitation L1)**. Although not the most current version, this particular version of Flink has proven to be a reliable and consistent option for our experimental purposes. It could be deemed advantageous to transfer this research

setup to a more recent version of Flink. Moreover, operating on a single instance entails distributing tasks across numerous threads during the rescaling operator's parallelism, as opposed to augmenting the processing potential of the streaming pipeline. We chose the file system state backend (`FsStateBackend`). Further work would merit exploring the differing impacts of alternative state backends, such as `MemoryStateBackend` or `RocksDBStateBackend`. This research also caters for the early deployment of newly emerging workloads with no historical operational data. Therefore, the volume of data used for our experiments supports this use case.

Internal Validity The presented experimental study utilises DS2 as the exemplar auto-scaler **(Limitation L2)** and incorporates its concepts of true and observed processing rates [58]. This feature offers the advantage of enabling the measurement of not only the number of records handled by operators in the topology, but also the duration of productive versus unproductive time that the operators expend. This study focuses on the streaming application's observed processing rate, which gives the exact number of records processed by each operator. This is a widely used approach for measuring operators throughput, and even though we only use one auto-scaler, our method can be used with any auto-scaling system for a wide variety of use cases. So, the use of DS2 doesn't reduce the relevance of this study.

External Validity Our window selectivity experiment considers only sliding window approach **(Limitation L3)**. The lack of testing of our hypothesis using alternative windowing approaches may pose a constraint on the generalisability of our findings. Nonetheless, this fulfils our aim by showcasing the influence of varying sliding window length configurations on the size of instantaneous states. Hence, the outcomes of our study provide support for the assertion that the selectivity of window size plays a crucial role in optimising the performance of streaming applications in a windowed setting. Our configuration possesses the potential to be implemented in a broad range of use cases.

6.3 Future Research Direction

Here we identify several potential areas for future research, based on insights gained during the course of the PhD programme.

6.3.1 Optimising Streaming Engine's Processing and Data Distribution Mechanism

An interesting future work will be to optimise stream processors' data initialization period and the ability for the system to reach its optimal processing capacity within a shorter period. This could become a distinctive factor amongst streaming systems, especially when real-time data for analytics is required.

It is recommended that further research work be conducted to measures the impact of system utilisation on the level of imbalance between different operators' instance in a distributed stream

processing pipeline. The first step towards achieving this will be to deploy this experiment on a system with more dedicated computational resources. Furthermore, An interesting future direction will be to investigate the task allocation mechanism being used by Flink in this kind of distribution setup with the aim of ascertaining the effect of the imbalance between each operator instance and throughput.

6.3.2 Declining Processing Rates and the Interplay with Application State

A useful next step is to evaluate the performance impact of the streaming application iterating over a larger collection arising from a growing state size. This could arguably be a contributing factor to the declining operators throughput.

To demonstrate this impact we can run an experiment with two arrival rates. The first with a lower arrival rate, while the other with much larger arrival rate. Then drop the arrival rate to like 10 or 20% of what the max throughput should be. then measure how much state has been accumulated from that first phase to the second phase. We can then measure the true processing rate and say based on the amount of time this operator is able to service the same amount of load, how much slack does there appear to be? My hypothesis will be with increased state you will see a lower throughput for the same hardware.

6.3.3 Windowing Selectivity Expansion to Other Window Types

A useful next step is to expand the selectivity of windowing techniques beyond the commonly used sliding window approach. While sliding windows have been widely employed in data analysis, they have limitations, such as fixed window sizes that remains constant throughout the analysis which may not be suitable for all data and analytical requirements. In the sliding window technique, a window of a specific size moves sequentially over the data, capturing a fixed number of data points within each window. While this approach is suitable for certain applications, it may not be optimal in scenarios where the data distribution or patterns change over time.

For example, if the data being analysed exhibits varying temporal dynamics or if there are fluctuations in the data's statistical properties, a fixed window size may not capture the relevant information effectively. Certain events or patterns that occur outside the fixed window boundaries may be overlooked or inaccurately represented in the analysis. This limitation can potentially lead to incomplete or biased results, particularly when dealing with dynamic or evolving data.

By exploring alternative window types that offer more flexibility in adjusting the window size or capturing different subsets of data, users can make a choice of windowing approach that best fits their business needs. Alternative window types, including expanding window, tumbling window, and session-based window. The analysis will involve evaluating factors such as window size, adaptability to changing data patterns, and computational efficiency. By considering these alternative window types, we aim to enhance the flexibility and applicability of window-based analyses in future research endeavors.

Expanding windows allow for dynamically adjusting the window size based on data characteristics, accommodating variations in data patterns. Tumbling windows offer the ability to

segment data into non-overlapping windows, which may be advantageous in certain scenarios. Additionally, session-based windows provide a means to capture sequential events within a session context. Through this exploration of alternative window types, we aim to broaden the scope of windowing selectivity and enable more nuanced and adaptable analyses in future work.

References

[1] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al. (2005). The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289.

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

[3] Abdelhamid, A. S., Mahmood, A. R., Daghistani, A., and Aref, W. G. (2020). Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2455–2469.

[4] Akidau, T., Begoli, E., Chernyak, S., Hueske, F., Knight, K., Knowles, K., Mills, D., and Sotolongo, D. (2021). Watermarks in stream processing systems: Semantics and comparative analysis ofapache flink and google cloud dataflow. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[5] Alevizos, E., Artikis, A., and Paliouras, G. (2017). Event forecasting with pattern markov chains. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 146–157.

[6] Andrade, H. C., Gedik, B., and Turaga, D. S. (2014). *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press.

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

[8] Arshi Saloot, M. and Nghia Pham, D. (2021). Real-time text stream processing: A dynamic and distributed nlp pipeline. In *2021 International Symposium on Electrical, Electronics and Information Engineering*, pages 575–584.

[9] Asyabi, E., Wang, Y., Liagouris, J., Kalavri, V., and Bestavros, A. (2022). A new benchmark harness for systematic and robust evaluation of streaming state stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*.

[10] Balkesen, C., Dindar, N., Wetter, M., and Tatbul, N. (2013a). Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 3–14.

[11] Balkesen, C., Tatbul, N., and Özsu, M. T. (2013b). Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 15–26.

[12] Biernat, N. A. (2020). *Scalability Benchmarking of Apache Flink*. PhD thesis, Kiel University.

[13] Buddhika, T., Stern, R., Lindburg, K., Ericson, K., and Pallickara, S. (2017). Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569.

[14] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. (2017). State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729.

[15] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4).

[16] Cardellini, V., Lo Presti, F., Nardelli, M., and Russo, G. R. (2022). Runtime adaptation of data stream processing systems: The state of the art. *ACM Computing Surveys*, 54(11s):1–36.

[17] Cardellini, V., Presti, F. L., Nardelli, M., and Russo, G. R. (2018). Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, 87:171–185.

[18] Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736.

[19] Chakravarthy, S. and Jiang, Q. (2009). *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*, volume 36. Springer Science & Business Media.

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

[21] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75.

[22] Chaturvedi, S., Tyagi, S., and Simmhan, Y. (2019). Cost-effective sharing of streaming dataflows for iot applications. *IEEE Transactions on Cloud Computing*, 9(4):1391–1407.

[23] Chen, X., Vigfusson, Y., Blough, D. M., Zheng, F., Wu, K.-L., and Hu, L. (2017). Governor: Smoother stream processing through smarter backpressure. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 145–154. IEEE.

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

[25] Cunha, Í., Almeida, J., Almeida, V., and Santos, M. (2007). Self-adaptive capacity management for multi-tier virtualized environments. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 129–138. IEEE.

[26] De Matteis, T. and Mencagli, G. (2017a). Elastic scaling for distributed latency-sensitive data stream operators. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 61–68. IEEE.

[27] De Matteis, T. and Mencagli, G. (2017b). Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *International Journal of Parallel Programming*, 45(2):382–401.

[28] DeSimone, S. (2017). Storage reimagined for a streaming world. [Online; posted 9-April-2017], http://blog.pravega.io/2017/04/09/storage-reimagined-for-a-streamingworld/,2017.

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

[30] Dobbin, K. K. and Simon, R. M. (2011). Optimally splitting cases for training and testing high dimensional classifiers. *BMC medical genomics*, 4(1):1–8.

[31] El-Sayed, N. and Schroeder, B. (2014). Checkpoint/restart in practice: When 'simple is better'. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84–92. IEEE.

[32] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.

[33] Ewen, S. (2023). Akka and actors. [Online; posted 05-March-2021], https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors.

[34] Ezhilchelvan, P. and Mitrani, I. (2023). Checkpointing models for tasks with widely different processing times. In *Computer Performance Engineering: 18th European Workshop, EPEW 2022, Santa Pola, Spain, September 21–23, 2022, Proceedings*, pages 100–114. Springer.

[35] Fang, J., Zhang, R., Fu, T. Z., Zhang, Z., Zhou, A., and Zhu, J. (2017). Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*.

[36] Farahabady, M. R. H., Samani, H. R. D., Wang, Y., Zomaya, A. Y., and Tari, Z. (2016). A qos-aware controller for apache storm. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 334–342. IEEE.

[37] Feng, Y.-H., Huang, N.-F., and Wu, Y.-M. (2011). Efficient and adaptive stateful replication for stream processing engines in high-availability cluster. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1788–1796.

[38] Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. (2017). Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836.

[39] Fragkoulis, M., Carbone, P., Kalavri, V., and Katsifodimos, A. (2020). A survey on the evolution of stream processing systems. *arXiv preprint arXiv:2008.00842*.

[40] Fu, T. Z., Ding, J., Ma, R. T., Winslett, M., Yang, Y., and Zhang, Z. (2017). Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on networking*, 25(6):3338–3352.

[41] Gedik, B. (2014). Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539.

[42] Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. (2013). Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463.

[43] Golab, L. and Özsu, M. T. (2003). Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14.

[44] Gou, X., He, L., Zhang, Y., Wang, K., Liu, X., Yang, T., Wang, Y., and Cui, B. (2020). Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1015–1025.

[45] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365.

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

[47] Hidalgo, N., Wladdimiro, D., and Rosas, E. (2017). Self-adaptive processing graph with operator fission for elastic stream processing. *Journal of Systems and Software*, 127:205–216.

[48] Hirzel, M., Soulé, R., Schneider, S., Gedik, B., and Grimm, R. (2014). A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):1–34.

[49] Huang, X., Shao, Z., and Yang, Y. (2020). Potus: Predictive online tuple scheduling for data stream processing systems. *IEEE Transactions on Cloud Computing*, 10(4):2863–2875.

[50] Hueske, F. and Kalavri, V. (2019). *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. O'Reilly Media.

[51] Hummer, W., Satzger, B., and Dustdar, S. (2013). Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345.

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

[53] Imai, S., Patterson, S., and Varela, C. A. (2018). Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 62–71. IEEE.

[54] Iqbal, W., Dailey, M., and Carrera, D. (2009). Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud. In *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, pages 243–253. Springer.

[55] Javed, M. H., Lu, X., and Panda, D. K. (2017). Characterization of big data stream processing pipeline: a case study using flink and kafka. In *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 1–10.

[56] Jayasekara, S., Harwood, A., and Karunasekera, S. (2020). A utilization model for optimization of checkpoint intervals in distributed stream processing systems. *Future Generation Computer Systems*, 110.

[57] Joseph, V. R. and Vakayil, A. (2022). Split: An optimal method for data splitting. *Technometrics*, 64(2).

[58] Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M., and Roscoe, T. (2018). Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[59] Kalyvianaki, E., Fiscato, M., Salonidis, T., and Pietzuch, P. (2016). Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, pages 541–553.

[60] Karakaya, Z., Yazici, A., and Alayyoub, M. (2017). A comparison of stream processing frameworks. In *2017 International Conference on Computer and Applications (ICCA)*, pages 1–12. IEEE.

[61] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., and Markl, V. (2018). Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE.

[62] Katsipoulakis, N. R., Labrinidis, A., and Chrysanthis, P. K. (2017). A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment*, 10(11):1286–1297.

[63] Kim, S. N., Medelyan, O., Kan, M.-Y., and Baldwin, T. (2013). Automatic keyphrase extraction from scientific articles. *Language resources and evaluation*, 47:723–742.

[64] Kombi, R. K., Lumineau, N., and Lamarre, P. (2017). A preventive auto-parallelization approach for elastic stream processing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1532–1542. IEEE.

[65] Krishnan, S., Gonzalez, J. L. U., Krishnan, S., and Gonzalez, J. L. U. (2015). Google cloud dataflow. *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*, pages 255–275.

[66] Kuhn, M. and Johnson, K. (2019). *Feature engineering and selection: A practical approach for predictive models*. CRC Press.

[67] Kumbhare, A. G., Azimi, R., Manousakis, I., Bonde, A., Frujeri, F. V., Mahalingam, N., Misra, P. A., Javadi, S. A., Schroeder, B., Fontoura, M., et al. (2021). Prediction-based power oversubscription in cloud platforms. In *USENIX Annual Technical Conference*, pages 473–487.

[68] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005). Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322.

[69] Li, S., Gerver, P., MacMillan, J., Debrunner, D., Marshall, W., and Wu, K.-L. (2018a). Challenges and experiences in building an efficient apache beam runner for ibm streams. *Proceedings of the VLDB Endowment*, 11(12):1742–1754.

[70] Li, T., Xu, Z., Tang, J., and Wang, Y. (2018b). Model-free control for distributed stream data processing using deep reinforcement learning. *arXiv preprint arXiv:1803.01016*.

[71] Liao, X., Huang, Y., Zheng, L., and Jin, H. (2019). Efficient time-evolving stream processing at scale. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2165–2178.

[72] Littler, S. (2022). The importance and effect of sample size. [Online; posted 24-April-2022], https://select-statistics.co.uk/blog/importance-effect-sample-size/.

[73] Liu, B., Zhu, Y., and Rundensteiner, E. (2006). Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 347–358.

[74] Logothetis, D., Olston, C., Reed, B., Webb, K. C., and Yocum, K. (2010). Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62.

[75] Logothetis, D. and Yocum, K. (2009). Data indexing for stateful, large-scale data processing. In *Proceedings of NetDB*.

[76] Lohrmann, B., Janacik, P., and Kao, O. (2015). Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410. IEEE.

[77] Madsen, K. G. S., Zhou, Y., and Su, L. (2016). Enorm: Efficient window-based computation in large-scale distributed stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 37–48.

[78] Mayer, R., Koldehofe, B., and Rothermel, K. (2015). Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, 2(4):274–286.

[79] Mayer, R., Slo, A., Tariq, M. A., Rothermel, K., Gräber, M., and Ramachandran, U. (2017a). Spectre: Supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 161–173.

[80] Mayer, R., Tariq, M. A., and Rothermel, K. (2017b). Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 54–65.

[81] mehmetozanguven (2020). Datastream api. [Online; posted 22-May-2020], https://medium.com/analytics-vidhya/apache-flink-series-4-datastream-api-21ffdd8f2bc0/.

[82] Mei, Y., Cheng, L., Talwar, V., Levin, M. Y., Jacques-Silva, G., Simha, N., Banerjee, A., Smith, B., Williamson, T., Yilmaz, S., et al. (2020). Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE.

[83] Mencagli, G., Torquati, M., Danelutto, M., and De Matteis, T. (2017). Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2608–2624.

[84] Mohamed, S., Forshaw, M., and Thomas, N. (2017a). Automatic generation of distributed run-time infrastructure for internet of things. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 100–107. IEEE.

[85] Mohamed, S., Forshaw, M., Thomas, N., and Dinn, A. (2017b). Performance and dependability evaluation of distributed event-based systems: a dynamic code-injection approach. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 349–352.

[86] Moreno-Vozmediano, R., Montero, R. S., and Llorente, I. M. (2009). Elastic management of cluster-based services in the cloud. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24.

[87] Naksinehaboon, N., Liu, Y., Leangsuksun, C., Nassar, R., Paun, M., and Scott, S. L. (2008). Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783–788. IEEE.

[88] Nguyen, Q. H., Ly, H.-B., Ho, L. S., Al-Ansari, N., Le, H. V., Tran, V. Q., Prakash, I., and Pham, B. T. (2021). Influence of data splitting on performance of machine learning models in prediction of shear strength of soil. *Mathematical Problems in Engineering*, 2021.

[89] Nicolae, B. and Cappello, F. (2013). Ai-ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 155–166.

[90] Omoregbee, P. and Forshaw, M. (2023). Performability requirements in making a rescaling decision for streaming applications. In *Computer Performance Engineering: 18th European Workshop, EPEW 2022, Santa Pola, Spain, September 21–23, 2022, Proceedings*, pages 133–147. Springer.

[91] Omoregbee, P., Forshaw, M., and Thomas, N. (2023a). Analysing performance effects of window size on streaming operator throughput. In *Computer Performance Engineering: 39th Annual UK Performance Engineering Workshop, UKPEW 2023, Birmingham, United Kingdom, 7-8 August, 2023, Proceedings*. Springer.

[92] Omoregbee, P., Thomas, N., and Forshaw, M. (2023b). A state-size inclusive approach to autoscaling stream processing applications. In *Computer Performance Engineering: 19th European Workshop, EPEW 2023, Florence, Italy, 20-23 June, 2023, Proceedings*. Springer.

[93] Or, A. (2015). Dynamic allocation in spark. [Online; posted 11-June-2015], https://www.slideshare.net/databricks/dynamic-allocation-in-spark.

[94] Ottenwälder, B., Koldehofe, B., Rothermel, K., Hong, K., Lillethun, D., and Ramachandran, U. (2014). Mcep: A mobility-aware complex event processing system. *ACM Transactions on internet technology (TOIT)*, 14(1):1–24.

[95] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996). The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385.

[96] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. (2007). Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302.

[97] Paruchuri, V. (2022). Using linear regression for predictive modeling in r. [Online; posted 23-March-2022] https://www.dataquest.io/blog/statistical-learning-for-predictive-modeling-r/.

[98] Qadah, E., Mock, M., Alevizos, E., and Fuchs, G. (2018). A distributed online learning approach for pattern prediction over movement event streams with apache flink. In *EDBT/ICDT Workshops*, pages 109–116.

[99] Rameshan, N., Liu, Y., Navarro, L., and Vlassov, V. (2016). Hubbub-scale: Towards reliable elastic scaling under multi-tenancy. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE.

[100] Ren, K., Diamond, T., Abadi, D. J., and Thomson, A. (2016). Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1539–1551.

[101] Richard, T. Z. F. J. D., Ma, T., and Zhang, M. W. Y. Y. Z. (2015). Drs: Dynamic resource scheduling for real-time analytics over fast streams. *arXiv preprint arXiv:1501.03610*.

[102] Richter, S. (2017). A deep dive into rescalable state in apache flink. [Online; posted 01-September-2020], https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html/.

[103] Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., and Sericola, B. (2015). Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 80–91.

[104] Röger, H. and Mayer, R. (2019). A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52(2):1–37.

[105] Roy, N., Dubey, A., and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE.

[106] Runsewe, O. and Samaan, N. (2017). Cloud resource scaling for big data streaming applications using a layered multi-dimensional hidden markov model. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE.

[107] Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., et al. (2020). Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16.

[108] Sakr, S., Liu, A., and Fayoumi, A. G. (2013). The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46(1):1–44.

[109] Schneider, S., Wolf, J., Hildrum, K., Khandekar, R., and Wu, K.-L. (2016). Dynamic load balancing for ordered data-parallel regions in distributed streaming systems. In *Proceedings of the 17th International Middleware Conference*, pages 1–14.

[110] Shukla, A. and Simmhan, Y. (2018). Model-driven scheduling for distributed stream processing systems. *Journal of Parallel and Distributed Computing*, 117:98–114.

[111] Sree Kumar, S. (2021). External streaming state abstractions and benchmarking.

[112] Streams, D. (2012). An efficient and fault-tolerant model for stream processing on large clusters. *Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. HotCloud.*

[113] Sun, D., Gao, S., Liu, X., You, X., and Buyya, R. (2020). Dynamic redirection of real-time data streams for elastic stream computing. *Future Generation Computer Systems*, 112:193–208.

[114] Tangwongsan, K., Hirzel, M., and Schneider, S. (2019). Sliding-window aggregation algorithms. In *Encyclopedia of Big Data Technologies*, pages 1516–1521. Springer International Publishing, Cham.

[115] Theeten, B., Bedini, I., Cogan, P., Sala, A., and Cucinotta, T. (2014). Towards the optimization of a parallel streaming engine for telco applications. *Bell Labs Technical Journal*, 18(4):181–197.

[116] To, Q.-C., Soto, J., and Markl, V. (2018). A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872.

[117] Toliopoulos, T. and Gounaris, A. (2020). Adaptive distributed partitioning in apache flink. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 127–132. IEEE.

[118] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. (2014). Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156.

[119] Tucker, P., Tufte, K., Papadimos, V., and Maier, D. (2008). Nexmark–a benchmark for queries over data streams (draft). Technical report, Technical Report. Technical report, OGI School of Science & Engineering at ....

[120] Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. (2003). Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568.

[121] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. (2008). Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1–39.

[122] van der Meulen, R. et al. (2018). What edge computing means for infrastructure and operations leaders. *Smarter with Gartner*.

[123] Van Der Veen, J. S., Van Der Waaij, B., Lazovik, E., Wijbrandi, W., and Meijer, R. J. (2015). Dynamically scaling apache storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 154–161. IEEE.

[124] Van Dongen, G. and Van Den Poel, D. (2021). Influencing factors in the scalability of distributed stream processing jobs. *IEEE Access*, 9.

[125] Vogel, A., Griebler, D., Danelutto, M., and Fernandes, L. G. (2022). Self-adaptation on parallel stream processing: A systematic review. *Concurrency and Computation: Practice and Experience*, 34(6):e6759.

[126] Wang, L., Fu, T. Z., Ma, R. T., Winslett, M., and Zhang, Z. (2019). Elasticutor: Rapid elasticity for realtime stateful stream processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 573–588.

[127] Wang, S., Rundensteiner, E., Ganguly, S., and Bhatnagar, S. (2006). State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd international conference on Very large data bases*, pages 619–630.

[128] Wang, Y., Wang, X., Chen, M., and Zhu, X. (2008). Power-efficient response time guarantees for virtualized enterprise servers. In *2008 Real-Time Systems Symposium*, pages 303–312. IEEE.

[129] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

[130] Wood, T., Shenoy, P. J., Venkataramani, A., Yousif, M. S., et al. (2007). Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17.

[131] Woods, L., Teubner, J., and Alonso, G. (2010). Complex event detection at wire speed with fpgas. *Proceedings of the VLDB Endowment*, 3(1-2):660–669.

[132] Wu, H., Shang, Z., and Wolter, K. (2019). Performance prediction for the apache kafka messaging system. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 154–161. IEEE.

[133] Xia, Y., Xu, Y., and Gou, B. (2019). A data-driven method for igbt open-circuit fault diagnosis based on hybrid ensemble learning and sliding-window classification. *IEEE Transactions on Industrial Informatics*, 16(8):5223–5233.

[134] Xiao, F. and Aritsugi, M. (2018). An adaptive parallel processing strategy for complex event processing systems over data streams in wireless sensor networks. *Sensors*, 18(11):3732.

[135] Xu, L., Peng, B., and Gupta, I. (2016). Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE.

[136] Yang, J., Qiu, J., and Li, Y. (2009). A profile-based approach to just-in-time scalability for cloud applications. In *2009 IEEE International Conference on Cloud Computing*, pages 9–16. IEEE.

[137] Yasumoto, K., Yamaguchi, H., and Shigeno, H. (2016). Survey of real-time processing technologies of iot data streams. *Journal of Information Processing*, 24(2):195–202.

[138] Young, J. W. (1974). A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531.

[139] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28.

[140] Zhang, B., Jin, X., Ratnasamy, S., Wawrzynek, J., and Lee, E. A. (2018a). Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252.

[141] Zhang, F., Chen, H., and Jin, H. (2019). Simois: A scalable distributed stream join system with skewed workloads. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[142] Zhang, Q., Yang, L. T., Yan, Z., Chen, Z., and Li, P. (2018b). An efficient deep learning model to predict cloud workload for industry informatics. *IEEE transactions on industrial informatics*, 14(7).

[143] Zhang, Z., Li, W., Qing, X., Liu, X., and Liu, H. (2021). Research on optimal checkpointing-interval for flink stream processing applications. *Mobile Networks and Applications*, 26(5).

[144] Zhou, Y., Wu, J., and Leghari, A. K. (2013). Multi-query scheduling for time-critical data stream applications. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12.