



Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:

Jayasekara, Weragoda Achchillage Sachini

Title:

Efficient Stateful Computations in Distributed Stream Processing

Date:

2020

Persistent Link:

<https://hdl.handle.net/11343/244125>

Terms and Conditions:

Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.

Efficient Stateful Computations in Distributed Stream Processing

Weragoda Achchillage Sachini Jayasekara

ORCID: 0000-0003-2618-2984

Submitted in total fulfillment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems

THE UNIVERSITY OF MELBOURNE

May 2020

Copyright © 2020 Weragoda Achchillage Sachini Jayasekara

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the author.

Abstract

Stream processing is used in a plethora of applications that deal with high volumes and varieties of data. The focus towards scalable and efficient stream processing solutions has been increasing due to the vast number of time-sensitive applications such as electronic trading, fraud detection that have low latency requirements. While stream processing systems were originally envisaged to use only stateless computations, the use of stateful computations has grown to accommodate a greater range of complex stream processing applications in various domains.

Unlike stateless computations, supporting stateful computations requires addressing new challenges including state distribution to achieve scalability and state sharing among resources in a distributed environment. For instance, most stateful computations have synchronization requirements that need to be satisfied to guarantee the correctness of the results. Therefore, efficient mechanisms that can support scalable state distribution and state sharing while ensuring correctness of the results are needed to satisfy the low latency requirement of stream processing applications. Moreover, a fault-tolerance mechanism to recover state after failures is an essential functionality required to support stateful computations and minimizing the overhead imposed by the fault-tolerance mechanism is another challenge associated with stateful computations.

This thesis first focuses on providing models to support complex stateful use cases. Windows that are used to partition the continuous input streams expected in streaming applications are a main component of stream processing systems. The existing models that define window semantics do not represent use cases that have a hierarchy of window stages and therefore, we propose a generic model for stream processing that supports a hierarchical approach to windowing. Then we propose a communication model to support iterative computations which is one of the most common stateful computation types. Due to communication restrictions that limit the ways to share the state of iterative computations, existing approaches used to represent iterative computations have limitations in terms of scalability and efficiency. We address these scalability issues and provide an

efficient way to share the state of iterative computations in a distributed environment. We demonstrate that our model can support different iterative algorithms that have complex communication patterns and show the scalability and high performance of the proposed model compared to the traditional approaches used for constructing iterative streaming applications. For example, our model outperforms existing state-of-the-art solutions 72% in terms of throughput and 65% in terms of latency in some cases.

Next, we investigate the most common fault-tolerance approach used by existing systems, checkpointing and address how we can minimize the overhead imposed by the checkpointing process. We derive an expression for the optimal checkpoint interval that gives the maximum system utilization using a theoretical model and validate the model using a set of simulations. To the best of our knowledge, this is the first theoretical optimization framework for stream processing systems that use a global checkpointing approach. Our model yields an elegant expression for the optimal checkpoint interval, interestingly showing the optimal checkpoint interval to be dependent only on the checkpoint cost and the failure rate of the system.

Next, we use the derived optimal checkpoint interval in real-world streaming applications and demonstrate that the theoretical optimal interval can improve the performance of practical applications. We demonstrate that our theoretical optimal checkpoint interval can achieve utilization improvements from 10% - 200% for a range of failure rates from 0.3 failures per hour to 0.075 failures per minute compared to the default checkpoint interval of 30 minutes used by most systems. Moreover, we show that the optimal interval results in lower latency and higher throughput, with 54% throughput increase and 58% latency decrease for some cases.

Then we investigate the multi-level checkpointing approach which is introduced to address the inefficiencies of single-level checkpointing and derive the optimal checkpointing parameters that minimize the overhead of the multi-level checkpointing process. This work is the first to present a theoretical framework for determining optimal parameter settings in a multi-level global checkpointing system that uses a single periodic checkpoint interval. We demonstrate that our solution outperforms existing single level optimizations in terms of utilization by as much as 36% in some cases.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Weragoda Achchillage Sachini Jayasekara, May 2020

Acknowledgements

First of all, I would like to thank my principal supervisor, Prof. Shanika Karunasekera and my co-supervisor, Dr. Aaron Harwood for their continuous guidance and immense support throughout my PhD candidature. Their advice and research insights were key to make my PhD journey a success. I am extremely grateful for their invaluable time, patience and encouragement which helped me grow as a researcher.

I would also like to express my gratitude to A/Prof. Wally Smith, my advisory committee chair for his continuous support and words of encouragement.

I acknowledge the financial support provided by the Melbourne Research Scholarship and Defence Science and Technology Group. I am also grateful to School of Computing and Information Systems for providing the facilities to pursue my research.

I am thankful to all my friends in the department, Kushani Perera, Namrata Srivastava, Partha De, Nicholas Akinyokun, Gayashan Amarasinghe, and Sadia Nawaz for their support and company throughout my PhD journey. I am also grateful to Dr. Xunyun Liu for his support as a senior student.

I would like to thank my parents and my sister for their love and support. It would not have been possible to pursue my PhD without their guidance and encouragement. I am also grateful to my cousin, Hashini and her family for all the love and comforting words. Last but not the least, I am thankful to my husband, Sameera, for always being there for me and for being my source of strength.

Preface

The main contributions of the thesis are discussed in Chapters 3-6 and are based on the following publications:

- **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “A Multi-stage Hierarchical Window Model with Application to Real-Time Graph Analysis,” *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
- **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “Enhancing the Scalability and Performance of Iterative Graph Algorithms on Apache Storm,” *2018 IEEE International Conference on Big Data (Big Data)*, 2018.
- **Sachini Jayasekara**, Xunyun Liu, Shanika Karunasekera, and Aaron Harwood. “Communication Model for Parallel Iterative Stream Processing,” *2018 IEEE International Conference on Big Data (Big Data)*, 2018.
- **Sachini Jayasekara**, Aaron Harwood, and Shanika Karunasekera. “A Utilization Model for Optimization of Checkpoint Intervals in Distributed Stream Processing Systems,” *Future Generation Computer Systems*, vol. 110, pp. 68-79, 2020.
- **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “Optimizing Checkpoint based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice,” *Software: Practice and Experience*. (Under review).

- **Sachini Jayasekara**, Aaron Harwood, and Shanika Karunasekera. “Optimal Multi-Level Global Checkpointing for Exascale Stream Processing Systems,” *IEEE Transactions on Parallel and Distributed Systems*. (Under review).

To my father, mother and husband

Contents

1	Introduction	1
1.1	Challenges in Stateful Computations in Distributed Stream Processing . .	5
1.2	Research Problems and Objectives	7
1.3	Thesis Contribution	9
1.4	Thesis Organisation	10
2	Literature Review	13
2.1	Introduction	13
2.2	State Management	14
2.3	Windowing	18
2.4	Stateful Computations	24
2.5	Fault-Tolerance	31
2.6	State-of-the-art Stream Processing Systems	43
2.7	Summary	54
3	Complex Stateful Computational Support for Stream processing Systems	55
3.1	Introduction	55
3.2	Window Model	59
3.3	Intra-Operator Communication Model	66
3.4	Parallel Processing Applications	74
3.5	Implementation	77
3.6	Evaluation	84

3.7	Conclusion	97
4	A Utilization Model for Optimization of Checkpoint Intervals in Stream Processing Systems	98
4.1	Introduction	98
4.2	Fundamental Utilization Model	100
4.3	Utilization of a Stream Processing System	106
4.4	Evaluation	113
4.5	Conclusion	117
5	Optimizing Checkpoint-based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice	119
5.1	Introduction	119
5.2	Evaluation Methodology	120
5.3	Experiments	124
5.4	Conclusion	136
6	An Optimization for Interval-based Multi-level Checkpointing in Stream Processing Systems	138
6.1	Introduction	138
6.2	Probabilistic Checkpointing	141
6.3	2-level Checkpointing Utilization Model	143
6.4	2-level Checkpointing for Stream Processing	152
6.5	L -level checkpointing Utilization Model	157
6.6	L -level Checkpointing for Stream Processing	159
6.7	Experimental Results with Apache Flink	162
6.8	Conclusion	166
7	Conclusions and Future Work	168
7.1	Complex Stateful Computational Support for Stream processing Systems .	169
7.2	Optimization of Checkpoint Intervals	170

7.3	Impact of Optimal Checkpoint Interval on Real-world Stream Processing Applications	171
7.4	Optimization for Multi-level Checkpointing	172
7.5	Future Work	173

List of Figures

1.1	Graph of operators representing a stream processing application.	3
2.1	Stateful operator types.	15
2.2	Sliding window of size 4 and slide 2.	19
2.3	Tumbling window of size 4.	19
3.1	Window-based computations with a window of length 4 and operator A calculating the maximum element.	56
3.2	DAG of operators. Dashed arrows indicate the communication paths not allowed in a DAG.	58
3.3	Approaches for iterative stream computation.	59
3.4	Multi-stage windowing example.	63
3.5	Windowing example of three stages.	64
3.6	High level data flow of an iterative computation.	67
3.7	Internal data flow paths of operator instances.	67
3.8	Dependency Graphs.	69
3.9	Operator instance handling inputs for iterative computations.	70
3.10	Bitonic Mergesort communication pattern.	71
3.11	API usage: top- k implementation. n = total instances.	73
3.12	API usage: global barrier implementation. n = total instances.	73
3.13	Parallel top- k computation approaches.	75

3.14	Achieving synchronization of k -means clustering.	76
3.15	A directed graph distributed among 3 operator instances.	80
3.16	Data flow of the off-loading approach.	82
3.17	Data flow of the distributed memory approach.	82
3.18	Data flow of intra-operator communication model.	83
3.19	PageRank computation time of the three models.	86
3.20	SSSP computation time of the three models.	86
3.21	PageRank and SSSP computation speedup of the three models.	87
3.22	PageRank and SSSP computation time over 2 hour period for 1 minute window slide.	88
3.23	Dynamic graph edge distribution over 2 hour period for 1 minute window slide.	89
3.24	Throughput comparison of the three models.	89
3.25	Performance comparison with and without message compression.	90
3.26	PageRank comparison with existing systems.	92
3.27	SSSP comparison with existing systems.	92
3.28	Top- k calculation time.	93
3.29	Top- k calculation time over 1 hour period for 1 minute slide.	94
3.30	K -means calculation time.	95
3.31	Bitonic mergesort calculation time.	96
3.32	FloydWarshall calculation time.	96
4.1	Checkpointing with period T , checkpoint overhead c	101
4.2	Failure resulting in lost time $F(T)$	103
4.3	Failure resulting in lost time $F(T)$ and recovery time R	104
4.4	Utilization for $\lambda = 0.005$ per minute, $c = 5$ minutes, $R = 10$ minutes.	104
4.5	Simulation results for $R = 10$ minutes $c = 5$ minutes.	105

4.6	DAG of operators representing a streaming application with 3 operators. .	107
4.7	Checkpointing process of the streaming application represented in Fig. 4.6.	107
4.8	Alternative view of the checkpointing process.	108
4.9	Failure during T'	110
4.10	Utilization of a DAG of operators.	112
4.11	Utilization of a single operator and a DAG of operators.	112
4.12	Utilization for $R = 10$ minutes $c = 5$ minutes and $\delta = 0.5$ minutes.	113
4.13	System failure rate and the utilization increase gained using T^*	114
4.14	Decrease in utilization with the increase of n using T^*	115
4.15	Optimal checkpoint interval comparison.	116
4.16	Percentage increase in utilization.	117
4.17	Checkpoint cost and optimal checkpoint interval comparison.	118
4.18	Utilization of static and dynamic T for different c ranges. c range- case 1: 49-51, case 2: 45-55, case 3: 25-75, case 4: 1-99 seconds.	118
5.1	Overview of the system.	121
5.2	Flink checkpointing process at the source operator.	124
5.3	Word count use case.	126
5.4	Fraud detection use case.	126
5.5	Tweet analysis use case.	127
5.6	Throughput comparison: $T = 30$ minutes, $T = T^*$ and without checkpoints.	131
5.7	Latency comparison: $T = 30$ minutes, $T = T^*$ and without checkpoints. .	132
5.8	Checkpoint cost analysis.	134
5.9	Restart cost analysis.	136
6.1	Different types of failure levels.	139
6.2	Multi-level checkpointing with 3 levels.	143

6.3	2-level checkpointing with period T	143
6.4	Level-1 failure.	145
6.5	Level-2 failure.	146
6.6	Level-1 failure and recovery.	148
6.7	Level-2 failure and recovery.	148
6.8	Intersection point of $\frac{\partial U}{\partial T}=0, \frac{\partial U}{\partial p_1}=0$	149
6.9	Optimal utilization.	149
6.10	Optimal parameters.	150
6.11	Utilization of 2-level checkpointing.	151
6.12	Comparison of 2-level and 1-level checkpointing.	152
6.13	2-level checkpointing of a streaming application.	153
6.14	Utilization of 2-level checkpointing for a streaming application.	157
6.15	Utilization of L -level checkpointing.	160
6.16	Utilization of 3-level checkpointing for a streaming application.	162
6.17	Performance comparison of different models using parameters in Table 6.4.	166
6.18	Comparison between theoretical and simulation utilization.	167

List of Tables

2.1	Window processing optimizations summary.	24
2.2	Methods for supporting iterative computations.	31
2.3	Factors considered in existing work proposed to determine the optimal checkpoint interval.	37
2.4	Comparison of stream processing systems.	52
2.4	Comparison of stream processing systems (continued).	53
3.1	Methods supported by abstract Storm bolt.	72
3.2	Partitioning tables of the graph shown in Figure 3.15.	79
3.3	Dataset statistics.	85
5.1	Notations used in the evaluation.	125
5.2	Experimental results of the three applications using a five node Flink cluster.	128
5.3	Experimental results of selected algorithms using a fifty node Flink cluster.	130
6.1	Impact of λ_2 to p_1^* , T^* , and U for $\lambda_1 = 50$ per day, $c_1 = r_1 = 20$, $c_2 = r_2 = 50$ seconds.	152
6.2	Comparison of 3-level and 2-level checkpointing.	163
6.3	Experimental results using Apache Flink.	164
6.4	Parameters used for comparison from Di et al. [120].	165

Chapter 1

Introduction

The era of Big Data presents tremendous challenges due to the exponential growth of data generation at high rates from a vast number of sources. The amount of data generated is estimated to grow more rapidly over time with 2.5 quintillion bytes of data created every day¹ and Gartner forecasting 14.2 billion connected things to be in use in 2019 and 25 billion by 2021². The amount of real-time data is predicted to represent nearly 30% of the global datasphere by 2025 [1] arising the need for efficient real-time data processing to analyze the high volumes of data generated to get useful information. The real-time data generated from different systems and devices can be structured or unstructured data that can have useful information pertaining to multiple domains ranging from finance, infrastructure, communication to social networks.

With the growth of real-time data arises the need for time sensitive processing such as fraud detection, which has increased the focus towards stream processing. Stream processing is a programming paradigm that processes continuous streams of data to generate outputs regularly. Numerous high volume, low latency data processing applications use stream processing to gain insights from the processed data and to make decisions in real-time. Finance, retail, marketing, transportation and defence are some domains where stream processing plays a major role. For example, stream processing is used to make decisions in high-frequency trading to place trades within milli seconds

¹<https://www.domo.com/learn/data-never-sleeps-5>

²<https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-trends>

and to identify anomalies and fraudulent patterns as soon as they occur such as credit card frauds. Stream processing also enables early warning detection of natural disasters, situational awareness and emergency response coordination in crisis management. Selecting routes based on traffic conditions, predicting travel time in transportation applications and improving customer experience, proximity marketing in retail are few other use cases among many real-world stream processing applications.

In contrast to traditional data processing, the stream processing paradigm deals with a continuous stream of data and predefined queries are executed on the arriving data to provide outputs regularly in a timely manner. Moreover, streaming applications are inclined to perform in-memory processing due to the low latency requirement and are distributed to support large volumes of data. Stream processing applications define the queries that need to be executed on the continuous stream of data. Streaming use cases often require stream processing systems to perform different types of queries/computations on input data to get the final result and one streaming application can be responsible for performing multiple tasks which can result in multiple outputs. Therefore, most streaming applications have multiple operators and are often represented as a graph of interconnected operators as indicated in Fig. 1.1a.

In the graph of operators, operators without input streams are sources, operators without output streams are sinks and all the operators are interconnected by streams of data. Moreover, each operator performs a specific task on the input stream(s) of data and may produce one or more output streams to downstream operators. Distributed stream processing applications use multiple instances of an operator to support parallelism to deal with large volumes of data as depicted in Fig. 1.1b. The separation of tasks into multiple operators makes it easier for the users to modify individual operators when needed without changing the entire application and allows the users to define the degree of parallelism of each operator based on the computational complexity and workload.

Operators of a stream processing application can be divided into two main categories; stateless and stateful. Stateless operators such as filtering simply perform computations on input data without the need for additional information such as information about pre-

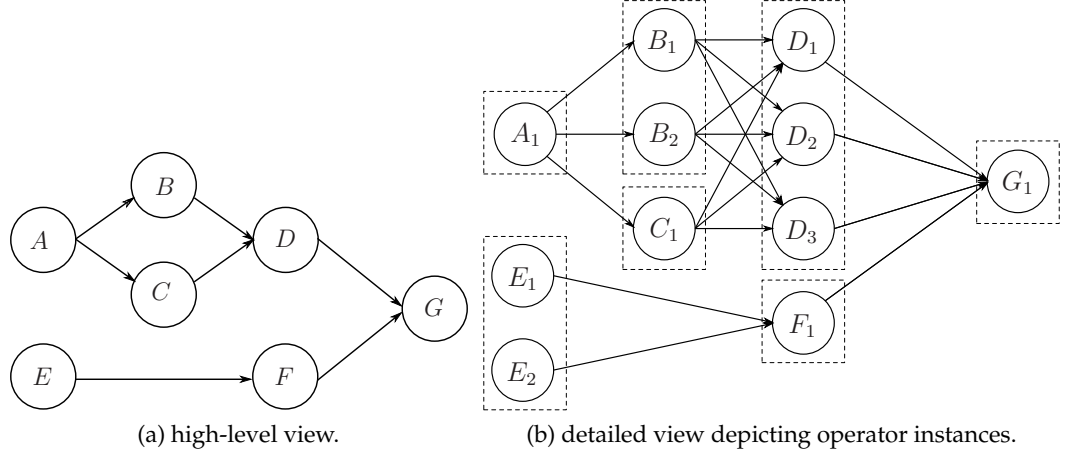


Figure 1.1: Graph of operators representing a stream processing application.

viously processed data. In contrast, stateful operators require additional information to perform computations on input data which is referred to as the state. Matteis et al. [2] define stateful operators as "operators that maintain and update a set of internal data structures while processing input data" and To et al. [3] define state as "the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow". An operator that computes sum keeps the sum of the previously processed data as its state and the intermediate results of iterative computations that have to be kept to perform new iterations are another example of state.

Stream processing systems incorporate state when handling continuous data by selecting a subset of the data such that computations can be performed on a bounded set of data. A window can be defined as a bounded, contiguous set of data from a stream of continuous data. Windows are the main abstraction used in stream processing systems to bound continuous data streams and the defined computations are performed on windowed data regularly as new windows get created over time. Data can be assigned to windows based on different policies such as time-based policies, count-based policies [4] and state is integrated into streaming systems to maintain the data in each created window.

While stream processing systems were originally envisaged to use only stateless oper-

ators, the use of stateful operators has grown to accommodate a greater range of complex stream processing. However, the introduction of stateful computations has raised new challenges in terms of state management, especially in distributed environments where the state is scattered across several resources that could be geographically distributed in different locations. Unlike stateless operators where using multiple operator instances to distribute the workload is an easy option, having multiple operator instances of a stateful operator requires the state to be distributed in a scalable manner while providing a way to communicate the state between parallel instances if needed. For instance, the early representation of stream processing applications using a strict Directed Acyclic Graphs (DAG) of operators has been relaxed to allow cycles in the operator graphs to support a wider range of stateful computations such as iterative computations.

Fault-tolerance mechanisms to recover state in case of failures are an essential part of supporting stateful computations. Especially, in large-scale distributed stream processing systems as failures are expected at any time during the application execution ranging from application-level failures to different types of hardware failures. When the scale of the stream processing system increases, the mean time to failure (MTTF) of the stream processing system as a whole decreases which arises the need for efficient fault tolerance approaches. Hence, state-of-the-art distributed stream processing systems have provided support for checkpointing to enable fault-tolerance for stateful stream processing.

In summary, the many use cases in stream computing have made it necessary for stream processing systems to provide support for stateful computations. Stateful computations ranging from iterative algorithms, graph processing algorithms to machine learning algorithms are common examples that require state to be maintained and communicated to perform the computation. Therefore, managing states such that storing, sharing and communicating the state will not affect the efficiency and scalability of the stream processing applications while making sure the computational state can be restored after failures is of paramount importance. However, there are several challenges that emerged with the usage of stateful operators in streaming applications that has to be addressed to ensure the low latency requirement of streaming applications is always met despite the

computations defined in the application.

1.1 Challenges in Stateful Computations in Distributed Stream Processing

1.1.1 Representation of complex stateful computations

Stream processing applications are often inclined to perform in-memory processing due to the low latency requirement. However, with stateful computations, keeping the entire state in-memory could be problematic due to state size. To provide scalability, state has to be distributed to handle large states similar to an operator having multiple instances to handle large workloads. When performing stateful computations, not only the state distribution but how to efficiently communicate the distributed state among parallel instances also has to be decided. If there is no mechanism to communicate/share the state, then operators cannot have multiple instances which limits the scalability. If all the instances have their local state in-memory, then sharing state over the network can be expensive depending on the state size resulting in higher application latency. If the state is kept in a shared storage that can be accessed by all the instances, then synchronization issues and consistent issues have to be addressed for cases such as multiple instances updating the state concurrently. Furthermore, reading and writing to shared storage frequently can increase latency significantly. Therefore, mechanisms to share state in a distributed stream processing system has to be carefully chosen to ensure the state sharing mechanism has a minimum overhead on application performance.

Windowing is the main abstraction that is used to bound the continuous input streams expected in stream processing systems so that computations can be performed regularly based on user requirements. However, windowing models to date do not go far enough to aid in the development of well defined complex analysis, that requires multiple stages of window computations. Due to the lack of multistage windowing models defining window semantics, results of multi-stage window based applications can differ from one

stream processing implementation to another. Therefore, a generic model for stream processing that can provide a hierarchical approach to define windows in multiple stages is useful to understand the behavior of complex event processing use cases.

Iterative computations are a common example of stateful operations where the state changes in every iteration based on the results of the previous iteration. Since graph algorithms, machine learning algorithms are common stream processing use cases that have iterative computations, support for iterative algorithms is an essential functionality in stream processing systems. When stream processing applications strictly followed a DAG, representing iterative algorithms was non trivial and not scalable because a new operator has to be devoted for each iteration or the application has to use another third-party system that can support iterative algorithms such as using a graph processing system when iterative graph algorithms need to be performed in the stream processing application. Because of this restriction, stream processing systems introduced cycles to support iterative computations. Unlike non-iterative algorithms, most iterative algorithms executed in distributed environments have synchronization requirements to ensure new iteration is started only when the previous iteration is completed by all the instances that run in parallel to guarantee the correctness of the results. However, enforcing the additional requirements such as synchronization that comes up with the introduction of cycles to the operator graph is challenging and has to be carefully handled to make sure system performance is not impacted while ensuring the final results are correct despite computations being performed in a distributed setting.

1.1.2 Fault-tolerance

Stateful operators depend on their state to process new data and therefore need fault-tolerance mechanisms to restore the state to continue processing new data after failures. Checkpointing is the common approach used in stream processing systems to support fault-tolerance. Enabling checkpoints reduces the efficiency of the system as resources have to be allocated to perform checkpoints periodically without solely working on the

actual computation. For instance, if a stream processing system performs checkpoints more frequently, then the system has to spend more time on checkpointing than doing the actual computations and if the system performs checkpoint less frequently, then the system has to spend more time on reprocessing the lost data instead of processing new data. Hence it is important to find a checkpoint interval that minimizes the overhead of checkpointing so that we can reduce the impact of checkpointing on the system performance. Current stream processing systems use a nominal value for the checkpoint interval that does not take into account the salient aspects of the checkpoint process. Therefore, finding an optimal checkpoint interval based on application specific parameters is important to ensure system efficiency.

When stream processing systems move towards exascale computing, the checkpoint-based fault-tolerance used in existing stream processing systems becomes too inefficient to scale with high failure rates expected in exascale systems. Hence, investigating more efficient options such as multi-level checkpointing used in some HPC systems is a good alternative. Multi-level checkpoint consists of different levels of checkpoint with each level tailored to specific kinds of failure unlike the single-level checkpointing where the same checkpoint type is used to recover from all expected failure types. However, it is important to find out the optimal checkpointing parameters related to multi-level checkpointing to minimize the impact of the checkpointing process on the system performance.

1.2 Research Problems and Objectives

The focus of this research is on increasing the scalability and efficiency of stateful computations in distributed stream processing systems. To address the challenges that impose overhead on stream processing systems due to stateful computations, this thesis explores the following key research problems:

- **How to support complex stateful use cases in a scalable manner?** Stream processing systems cannot efficiently represent some complex computations that incorpo-

rate stateful processing and in some cases, different implementations can give different results for the same use case. For example, applications that have windows can produce different results due to the lack of consistency among different window semantics. Moreover, supporting stateful computations like iterative algorithms where operator state often changes in every iteration, results in inefficient system performance due to communication restrictions. Therefore, models to define the semantics of windowing and to support complex computations types such as iterative graph processing, machine learning algorithms are needed to minimize the inefficiencies and inconsistencies associated with current stateful processing support.

- **How to minimize the performance overhead of checkpointing in distributed stream processing systems?** Failures are inevitable in stream processing applications as streaming applications often run for a long period of time. Therefore, stream processing systems should support fault-tolerance mechanisms to handle various types of failures ranging from hardware failures to application-level failures. Especially, stateful operators require fault-tolerance techniques to support restoring of the computational state to proceed with the computations after a failure. Checkpointing is the most common fault-tolerance technique used in stream processing systems as it only requires minimal additional resources compared to other techniques such as replication. However, to minimize the overhead imposed by checkpointing it is required to find an optimal checkpoint interval that gives the highest system utilization.
- **How well the theoretical optimal checkpoint interval can improve the efficiency of practical applications deployed in distributed stream processing systems?** Theoretical optimal checkpoint interval is defined based on several assumptions and the theoretical models cannot take into account all the different aspects in a practical system such as network delays, caching, garbage collection issues. Hence it is essential to investigate whether the theoretical optimal can improve the efficiency of practical stream processing applications.

- **How to improve the inefficiencies of checkpointing for high failure rates expected in Exascale stream processing systems?** The exponential growth of data requires stream processing systems to move towards exascale computing. However, the expected failure rates of exascale systems are very high making the existing checkpointing process used in stream processing systems inefficient. Therefore, methodologies like multi-level checkpointing and the optimal parameters for such methods have to be investigated to reduce the high overhead imposed by single level checkpointing in exascale systems.

1.3 Thesis Contribution

Main contributions of this thesis can be summarized as follows:

1. An in-depth investigation of stateful computational models, fault-tolerance approaches and optimizations used in distributed stream processing.
2. Complex stateful computational support for stream processing systems.
 - A hierarchical window-based approach to handle unbounded streaming data.
 - A new model based on intra-processor communication to enhance the efficiency of iterative algorithms.
 - Implementation of the model on Apache Storm and an API using which iterative streaming applications can be implemented.
3. A utilization model for optimization of checkpoint intervals in distributed stream processing systems.
 - A mathematical model focusing on maximizing the utilization of stream processing systems to determine the optimal checkpoint interval.
 - Simulation of stream processing applications to validate the accuracy and efficacy of the mathematical model.

4. A case study on the impact of optimal checkpoint interval on the performance of distributed stream processing systems.
 - An empirical study that explores the impact of checkpointing on the performance of streaming applications.
 - Validation of the performance benefits of theoretical optimal checkpoint interval in practical stream processing systems.
5. An optimization for multi-level checkpointing in distributed stream processing systems.
 - A probabilistic multi-level checkpointing model to reduce the overhead of single-level checkpointing.
 - A mathematical model to define the optimal parameters to maximize the utilization of stream processing systems that supports multi-level checkpointing and validation of the model through simulations.

1.4 Thesis Organisation

The chapters of this thesis are organized as follows. Chapter 2 gives an introduction to stateful computation models, state management and fault-tolerance mechanisms adopted by stream processing systems followed by optimizations used to improve fault-tolerance mechanisms. Chapter 3 discusses a window model to support applications with multiple windowing stages and a communication model to share the state between operators in the distributed stream processing applications with minimum impact on the latency. Chapters 4 and 5 focus on deriving a theoretical optimal checkpoint interval for stream processing applications and validating the accuracy and performance benefits of the derived theoretical model when applied to a practical system respectively. Chapter 6 proposes a theoretical derivation to optimize the utilization of stream processing systems when multi-level checkpointing is used. Chapter 7 concludes with a summary of the thesis and discusses the future research directions. The core chapters of the thesis are

derived from publications completed during my PhD candidature, which are listed as follows:

- Chapter 3 proposes models to support complex stateful processing use cases; a windowing model and a communication model to efficiently share the state of parallel iterative stream processing. This chapter is derived from:
 - **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “A Multi-stage Hierarchical Window Model with Application to Real-Time Graph Analysis,” *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
 - **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “Enhancing the Scalability and Performance of Iterative Graph Algorithms on Apache Storm,” *2018 IEEE International Conference on Big Data (Big Data)*, 2018.
 - **Sachini Jayasekara**, Xunyun Liu, Shanika Karunasekera, and Aaron Harwood. “Communication Model for Parallel Iterative Stream Processing,” *2018 IEEE International Conference on Big Data (Big Data)*, 2018.
- Chapter 4 provides a theoretical model that provides the optimal checkpoint interval that maximizes the utilization of stream processing applications. This chapter is derived from:
 - **Sachini Jayasekara**, Aaron Harwood, and Shanika Karunasekera. “A Utilization Model for Optimization of Checkpoint Intervals in Distributed Stream Processing Systems,” *Future Generation Computer Systems*, vol. 110, pp. 68-79, 2020.
- Chapter 5 evaluates how well the theoretical model derived in Chapter 4 represents real-world stream processing applications and confirms the performance benefits achieved using the theoretical optimal checkpoint interval. This chapter is derived from:

- **Sachini Jayasekara**, Shanika Karunasekera, and Aaron Harwood. “Optimizing Checkpoint-based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice,” *Software: Practice and Experience*. (Under review).
- Chapter 6 proposes a probabilistic multi-level checkpointing model suitable for environments with high failure rates and provides optimal parameters to maximize the utilization of stream processing applications. This chapter is derived from:
 - **Sachini Jayasekara**, Aaron Harwood, and Shanika Karunasekera. “Optimal Multi-Level Global Checkpointing for Exascale Stream Processing Systems,” *IEEE Transactions on Parallel and Distributed Systems*. (Under review).

Chapter 2

Literature Review

2.1 Introduction

With the increased use of stream processing in a wide range of domains, stream processing applications have grown to accommodate stateful operators that can support a greater range of complex computations. As opposed to stateless computations, supporting stateful computations in stream processing introduces challenges in different aspects in terms of state management, scalability and fault-tolerance.

With the thesis objective of improving the scalability and the efficiency of stateful computations in stream processing, the literature review chapter starts with an overview of the main state types and the approaches used to store state and share state in distributed environments. Then we provide an introduction to windows which are used in streaming applications to bound the continuous streams of input data and present models used to defined window semantics and methods proposed to support the parallel window processing. Next, we discuss various parallel computational models used to support stateful computations including iterative algorithms and then move on to fault-tolerance approaches used to recover the state of stateful computations after failures. This chapter concludes with a survey on well-known stream processing systems and their capabilities and limitations with respect to supporting stateful computations and discusses how the theoretical aspects proposed in the literature can be utilized to improve the state-

ful computational support.

2.2 State Management

State which can be defined as "the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow" [3], is used in various stream processing use cases and represents information pertaining to application specific computations. State can be divided into sub categories and all the different state types have to be maintained systematically to ensure state can be easily accessed and updated without any inconsistencies. Several state management tasks including state storage, state distribution and state sharing have to be implemented to provide support for stateful computations. This section provides an overview of different state types and the approaches proposed to support state management tasks such as storing state, handling large states and sharing states.

2.2.1 State types

The operators in a streaming application can be divided into two main categories; stateless and stateful. The stateful operators can be further divided into partitioned stateful operators and stateful operators [5]. The state of the partitioned stateful operators is maintained individually for multiple partitioned keys using a specific data structure, whereas stateful operators do not require a data structure as the state is treated as a single entity. Figure 2.1a depicts a stateful operator that counts the number of input elements without a partitioned key and Figure 2.1b depicts a partitioned stateful operator that counts the occurrence of each different character. As shown in the figure, the state of the stateful operator is a single value which keeps the count of processed input elements while in the partitioned stateful case, a separate count value is maintained for each unique character processed by the operator (in this case, a count value is maintained for characters a,b,c and d).

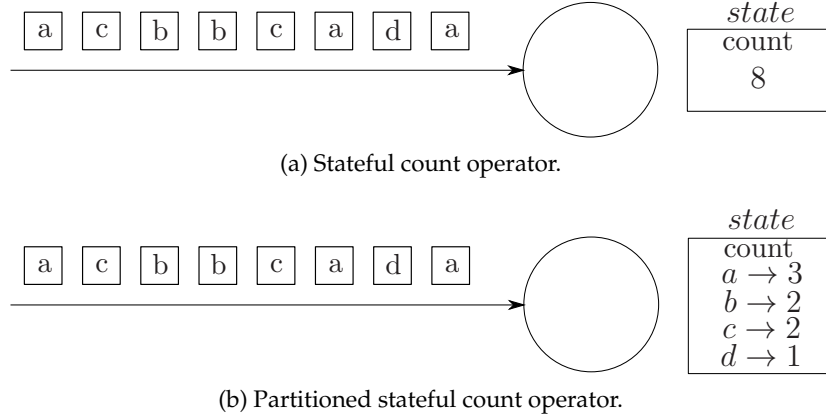


Figure 2.1: Stateful operator types.

Fernandez et al. [6] further divide an operator state into three sub states: processing state, buffer state and routing state. The processing state maintains the summary of processed tuples such as the number of encounters of processed words in a word count application. The buffer state maintains the output tuples of an operator that were sent to downstream operators and the routing state maintains mappings of keys to downstream operator instances such that the system knows the exact instance each output tuple should be directed to.

2.2.2 State Storage and sharing

When performing stateful computations, it is essential to have efficient and scalable state storage mechanisms. Keeping the state in memory is the most efficient way to read and update the state. However, most of the large-scale stateful applications have states that are too big to be kept in memory in a single commodity machine. In such cases, it is recommended to use persistent storage [3]. Moreover, in distributed environments, the state is usually partitioned among multiple operator instances and therefore state sharing mechanisms are essential to share state between operator instances. This section presents various approaches proposed for storing and sharing states.

Distributed memory: A widely used approach for storing state in distributed systems

is the use of shared or distributed memory. Systems such as Trinity [7], Kineograph [8], NScale [9], Spark structured streaming use a distributed key-value store to keep state. SGuard [10] uses a distributed replicated file system to manage state while Brito et al. [11] propose using a transaction memory to manage large states while maintaining state consistency among parallel processors.

Secondary storage: Another approach used to manage large states is the use of secondary storage to store the state. In this approach, a part of the state is read to in-memory from the secondary storage when required. This allows the processing of large volumes of data using a single machine [12, 13]. However, observations of this approach have shown that the execution time of applications is governed by the bandwidth of the secondary storage, which could result in undesirably high execution time for applications with large states. Chaos [14] uses the same approach to handle large states, but allows multiple machines to access state from the secondary storage to support parallel processing.

State spilling: State spilling [15] is another approach used to manage state where state stored in memory is flushed to disk temporarily when the memory capacity exceeds. The part of the state flushed to the disk is decided such that the throughput is maximized and when resources become available, the spilled state is read back to memory to provide the rest of the results.

Load shedding: Dropping some of the events when the state is too large is referred to as load shedding [16] and this method is also used in some systems to handle large states. The load shedding process is based on three main decisions: when to shed load, where to shed load and how much load to shed. However, load shedding is not a viable option for applications that require accurate results reflecting all the input data.

State sharing: State sharing is an essential part of distributed environments and state sharing is considered safe when the three main conditions: ensuring state visibility, avoiding race conditions and safe memory management, are satisfied [17]. State sharing is often done through message passing or shared memory in distributed environments.

For example, Trinity [7] uses a dedicated network communication module to pass state from one machine to another. Using distributed memory to facilitate intermediate result communication [18, 19] and the use of secondary storage to store the intermediate results [20] are some common examples of state sharing. When the systems cannot efficiently support state communication, performing computations is delegated to a third party system along with the entire state [21]. However, this approach has several drawbacks such as additional time taken to convert the entire state to a format recognized by the third party system and the requirement of more resources to maintain two independent systems.

Optimization techniques have been introduced to facilitate efficient state management. ChronoStream [22] partitions the operator state into slices to handle workload fluctuations and to achieve fast recovery. However, ChronoStream does not support iterations and slices created are computationally independent which is not the case with most iterative algorithms. Another example of optimizing the state size is the use of state difference referred to as delta state where only the delta state based on new data is calculated instead of calculating the entire state based on all the data [23]. This method calculates the current state using the previous state and the delta change based on the new input value, but this approach is not globally applicable as the correct state cannot be always computed simply by looking at the new data.

Several challenges have emerged due to distributing the state. For example, distributed replicated file systems used to store state are optimized for reading and writing large volumes of data [3]. Hence, this approach is not well-suited for algorithms that only change a small part of the state such as algorithms with incremental iterations where only a part of the results changes between iterations. Moreover, sharing the distributed state has a set of associated problems such as synchronization issues, inconsistent state and deadlocks [3]. Therefore, synchronization requirements, read-write conflicts have to be properly managed to provide an overall solution to sharing state in a distributed system. Kineograph [8] uses a global progress table to keep track of the updates and ensure consistency. S-store [24] is an example that provides a solution guaranteeing atomicity,

consistency, isolation, durability (ACID) and stream oriented guarantees for shared state.

For stream processing applications that have low latency requirements, reading and writing to persistent storage or shared memory to update the state can increase the latency. However, if the state is saved locally, hidden from other parallel instances in the streaming application, then this hinders the parallelism of operators [25]. Therefore, distributing state among multiple machines such that the state can be kept in memory while providing an efficient communication method to share the state when needed is a more desirable approach for streaming systems [26].

Deciding the placement of each operator of a streaming application over a set of machines is another challenge associated with stream processing and scheduling operators to gain maximal performance is considered as an NP-Hard problem [27, 28]. Xing et al. [29] have proposed an algorithm to decide the initial scheduling of streaming operators. But static scheduling can become inefficient as system load can vary over time. Therefore, dynamic scheduling algorithms are required to ensure better system performance. However, with stateful operators, this becomes more challenging as the state also has to be migrated when scheduling algorithms make dynamic decisions to migrate streaming operators from one machine to another [28].

2.3 Windowing

Stream processing applications process continuous streams of data and therefore, applications have to bound the input data to subsets before performing computations. Windowing is the main concept used in stream processing to bound the continuous input streams. Windows are created and updated over time based on user-defined policies such as time-based, count-based, delta-based and punctuation-based policies [4]. Once the window is defined, specified computations are performed on the content of the window as the window gets updated over time. For example, common operations such as aggregations, joins, pattern matching are often defined on a window.

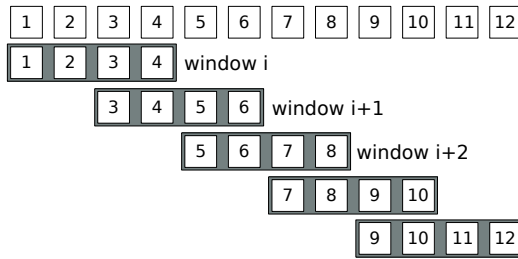


Figure 2.2: Sliding window of size 4 and slide 2.

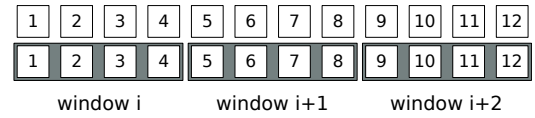


Figure 2.3: Tumbling window of size 4.

Sliding windows and tumbling windows are two of the most common windows types used in streaming applications. Sliding windows are defined based on two properties window size and slide. A new window is created at every slide and some of the old elements of the previous window are not included in the next window based on the slide size. Hence, in a sliding window, consecutive windows have overlapping elements. Figure 2.2 depicts a count-based sliding window of size four and a slide of two, where each window has four elements and a new window is created after every two elements.

Tumbling windows are a variant of sliding windows where the window size is equal to the window slide. Tumbling windows keep elements until the window size is completed and a new window gets created as soon as the previous window reaches its size and all the elements of the previous window are not included in the new window. Therefore, consecutive windows do not have overlapping elements. Figure 2.3 depicts a count-based tumbling window of size four, where each window has four elements. Partition windows are another frequently used window type where the input stream is initially divided into substreams based on a grouping function and then a window is defined for each substream individually.

Gedik [30] provides a categorization of sliding and tumbling windows and associated policies and discusses the semantics of different window types. The author defines three main operations for maintaining elements in a window: insertion, eviction and trigger. Insertion defines how a new element is added to a window, eviction defines how elements are removed from a window and trigger defines when the windowed elements are sent to processing. For tumbling windows, eviction and trigger overlap because the

window elements are discarded as soon after they are sent to processing. The author lists three types of eviction and trigger policies used for sliding windows: count-based, delta-based and time-based. In addition to these three policies, tumbling windows can be defined by punctuation-based eviction policies.

Although different window types and policies have been presented by many, due to different execution semantics, results of window queries differ from one stream processing system to another and the syntax for defining windows varies from one system to another [31]. Therefore, several models defining the window semantics have been presented to explain the behavior of different window types separating them from implementations. Next, we look into a set of models proposed to define window semantics.

2.3.1 Models defining window semantics

Botan et al. [31] present a descriptive model, SECRET to analyze the behavior of windows and results of window-based queries focusing on time-based windows. This model uses two types of timestamps on elements processed, t^{app} , application time provided by the data source, t^{sys} , system time defined by the system and usually t^{sys} is the time each element is received by the system. In this model, time windows are defined based on t^{app} value. This model defines four dimensions of a window: scope, content, report and tick. Scope defines the interval associated with the window and content maps the intervals defined by scope to window content. Report defines when the window content is sent to perform computations while tick defines when to re-evaluate a window. Although the model is used for time-based windows, it can be extended to analyze the semantics of count-based windows which are as frequently used as time-based windows. However, this model can only define windows for a single windowing stage. In our work, presented in Chapter 3, we recognize that time windowing is useful at multiple different time scales in one system and at different stages of processing and so we propose a generic model for stream processing that supports a hierarchical approach to windowing, extending the approach proposed by Botan et al. as suggested in their work as future

work.

Patroumpas et al. [32] define basic elements of a window and provide a categorization of different windows. They define four basic properties of windows: upper bound, lower bound, extend and mode of adjustment. These properties provide information regarding the timestamps of window content, window size and how the window is updated over time. The authors divide windows into two types, physical and logical. Physical windows are defined by the number of elements in the window and count-based windows and partitioned windows are a few examples. Logical windows are defined based on timestamps and landmark windows, fixed-band windows and time-based sliding windows are common examples. The authors also provide algebraic expressions for common window types to define their semantics. Kramer [33] also presents similar logical algebra defining the semantics of three common window types: time-based sliding window, count-based sliding window and partitioned window.

Li et al. [34] provide a similar framework to define window semantics. In this framework, a window is defined using three parameters: RANGE, SLIDE and WATTR. RANGE defines the window length, SLIDE defines the step by which the window moves and WATTR defines the attribute which defines RANGE and SLIDE. This model assigns a range of window-ids to each element in the data stream and the framework defines the window semantic by providing a mapping between window ids and elements.

2.3.2 Parallel window processing

Scalability is an essential part of streaming applications as the amount of data that is processed by streaming applications is unbounded and can arrive at high rates. High volumes of data arriving at high rates can impact window size as the window can grow based on how the window is defined. For instance, a window can be defined to keep data arrived in the last 10 days which can include millions of elements. Hence, methods to efficiently process window data are essential to provide low latency results. Several approaches have been proposed to support parallel execution of windows and next we

discuss different methods used to parallelize window processing.

Ivanova et al. [35] present two methods to support parallel window processing, window split and window distribute. Window split divides the window into small sub windows and sub windows can be processed in parallel by multiple workers, then a combine function is used to get the final result by merging the results of all the sub windows. However, the split function and combine function are operator-specific and therefore, users have to define these functions. In contrast, the window distribute method can be applied to any operator. But window distribute can only be used when there are sufficient resources to ensure processors are not overloaded. Traub et al. [36] and Madsen et al. [37] present similar techniques to support parallel window processing.

Mencagli et al. [38] also present two models to support parallel window processing, the agnostic model and the active worker model. In the agnostic model, when a window is triggered the whole window is sent to one of the workers and the workers only perform the defined computations to the received windowed elements and are not involved in triggering windows or evicting elements from a window. Instead of sending the whole window to a worker, the active worker model sends each tuple to workers and the workers determine when to trigger a window and when to perform computations on the window.

Matteis et al. [2] present four parallel patterns that can be used to perform parallel window operations using shared-memory or distributed-memory to reduce processing latency: window farming, key partitioning, pane farming and window partitioning. Window farming is the approach of using different workers to perform computations on different windows as results of consecutive windows are independent of one another. Astorga et al. [4] propose a similar pattern to support parallel window processing. Key partitioning is a variant of window farming but with limited parallelism. This approach allows windows of different substreams to be processed by workers in parallel with the exception that all the windows of a particular substream have to be processed by the same worker. Pane farming breaks each window into non-overlapping parts called panes and panes are processed in parallel. The final result is computed by combining the results

of each pane. However, pane farming can only be done when the computations can be divided into two parts such that the first part can be performed in parallel and the final result can be computed by combining individual results of the first part. In window partitioning, each worker keeps a partition of the window and the worker performs computations on the local partition and a reduce phase is used to compute the final result by combining the result of each window partition. Nathan et al. [39] propose a similar punctuation based window partitioning method to support parallel window processing.

Balkesen et al. [40] propose two partitioning methods to facilitate parallel sliding window processing, batch-based partitioning and pane-based partitioning. These two techniques are used to mitigate the drawbacks of window farming where consecutive windows have overlapping elements and therefore, the need to replicate elements results in increased data volume. Batch-based partitioning groups a set of consecutive windows to a batch and a batch is assigned to one worker to minimize the number of elements replicated. SABER [41] uses a similar batch-based approach to support parallel and incremental window processing. Pane-based partitioning is similar to the pane farming discussed by Matteis et al. [2] but uses a ring-based method to assign the panes among workers to avoid the need for a single worker to manage the merging process of all the pane results to get the final result.

As discussed above, performing window operations in parallel sometimes require overlapping window elements to be replicated in multiple workers. Processing consecutive windows by the same worker can reduce the number of replicated elements but can result in skewed workloads among parallel workers. Therefore, Mayer et al. [42] propose a method to address the trade-off between the workload and the number of elements replicated and their evaluations have shown that the proposed model can provide the optimal number of consecutive windows that need to be grouped to sent to the same worker.

Table 2.1: Window processing optimizations summary.

Technique	Objective	Examples	Systems
Parallel window processing	Increase efficiency and scalability	Window farming Pane farming	[2, 38, 4]
Window grouping/ window partitioning	Facilitate parallel processing and reduce replication of overlapping window elements	Batch-based Pane-based key-based	[40, 42, 41]
Incremental window processing	Avoid redundant computations	Saber Slider	[41, 45]

2.3.3 Optimizations

In addition to parallel processing, several other approaches have been explored to improve the efficiency of window operations and the use of incremental computations to process data in windows is one such approach as incremental computations avoid the need to reread the entire window content once the window is completed [43]. Patroumpas et al. [44] discuss common update patterns of common window types and how the identified patterns can be exploited to efficiently maintain window elements that change over time. Table 2.1 summarizes the different techniques proposed to improve window processing.

2.4 Stateful Computations

Stateful operations from graph processing to machine learning are widely used computations in data processing applications. The wide-spread use of stream processing in various domains has increased the need to support these stateful computations in stream processing systems. In this section, we present a set of widely used stateful computations in stream processing and the distributed computation models used in data processing to facilitate efficient and scalable execution of common stateful computations. When supporting any stateful computation, the most common decisions the systems have to make are: where to store state, how to handle large states and how to share state, which we dis-

cussed in Section 2.2. However, more challenges arise when supporting complex stateful computations, where additional requirements such as synchronization have to be satisfied. This section explores how state has to be managed when supporting such complex stateful computations. We first explore the models used to support generic iterative algorithms, because iteration is a complex operation in terms of managing state and is used in many other high level stateful algorithms, and move onto more specific use cases, graph algorithms and machine learning algorithms.

2.4.1 Iterative computations

Iterative computations are a widely used stateful operation in stream processing. Iterative algorithms require the results of the previous iteration to perform the next iteration. Therefore, the intermediate results of each iteration need to be communicated among the operator instances that run in parallel. Due to synchronization requirements and computational dependencies, iterative algorithms introduce high performance overhead resulting in inefficient system performance [46]. Stream processing systems often do not provide support for iterative computations [47] as the DAG model does not allow cycles that are essential to communicate intermediate results of iterations between operator instances that run in parallel. Often techniques like unrolling the loop [46] are used to model iterative algorithms to fit the DAG model. However, such techniques are not efficient and cannot be applied to iterative algorithms in general. For example, unrolling the loop cannot be used for algorithms that do not specify the number of iterations prior to the execution. These issues resulted in several proposals that introduce new frameworks or improve existing paradigms to support iterative computations. Next, we present a set of approaches used to support efficient iterative computations by existing work, namely, micro steps for incremental iterations, feedback loops, map-reduce based methods, dynamic task creation and algorithmic skeletons.

Incremental iterations: Ewen et al. [46] divide iterative algorithms into two categories, bulk iterations and incremental iterations. Bulk iterations are the iterations where

the result of a new iteration is completely different from the previous iteration and in incremental iterations, results of an iteration differ partially to the result of the previous iteration. The authors exploit the incremental nature to improve the efficiency of incremental iterative computations as opposed to most systems that handle incremental iterations as bulk iterations. They introduce micro steps that avoid the need for synchronization after each iteration and show that their method can achieve significant performance improvements.

Feedback loops: Murray et al. [47] present the timely data flow model to support iterations. This model allows asynchronous message communication with the help of a distributed coordination mechanism based on timestamps to guarantee consistent results. Naiad [48] is an implementation based on this model which uses loops to communicate results of iterations through feedback edges. Naiad [48] introduces a loop context with feedback to support iterations and the loop context requires additional nodes to support iterations. In the distributed implementation of Naiad, updates are broadcast to all the workers to manage the progress of the workers. However, this incurs an unnecessary cost for cases where there is no need to broadcast messages to all the workers.

Map-reduce: The map-reduce paradigm which exploits data parallelism can also be used to support stream processing[49, 50]. However, map-reduce cannot efficiently perform iterative computations and require new map and reduce tasks for each iteration [51, 19, 18] to provide a complete iterative solution. Hence systems like Twister [18] and HaLoop [19] have improved the map-reduce paradigm to support efficient iterative computing. These systems perform multiple map and reduce tasks to support iterations within the same job. Both Twister and HaLoop use distributed memory to facilitate communication of intermediate results. In addition, Ekanayake et al. [51] provide iterative algorithm support using CGL-MapReduce, a stream-based map-reduce solution and Kambatla et al. [52] present a solution introducing local map-reduce and global map-reduce which provide different levels of synchronization to support iterations.

Dynamic tasks: Ciel [53] is another stream processing engine that supports iterative computations. Ciel supports iterations by creating dynamic tasks as the Ciel job pro-

gresses. A task can spawn new tasks to support data-dependent computations. All the tasks including the newly spawned tasks represent a DAG. Optimus [54] is a similar engine that dynamically changes the data flow graph by extending the mechanisms used in Ciel. In addition to messages that communicate results of defined computations, Optimus uses a special type of message called graph rewriting messages to communicate the dynamic updates that need to be applied to the data flow graph. Optimus consists of a graph rewriter module that adds a new sub graph to the existing data flow graph for each new iteration until a predefined stopping condition is met.

Algorithmic skeletons: Algorithmic skeleton frameworks are another option that can be used to support iterative computations. These frameworks provide high-level abstractions for generic patterns of parallel program support functions such as map and reduce. However, multiple skeletons have to be combined together to represent complex parallel programs. Therefore, using skeletons to support iterative algorithms that have complex synchronization requirements is not scalable for stream processing applications and may require the implementation of new skeletons [55]. Another drawback of skeletons is that they do not address data partitioning methods, which are crucial to communicate intermediate results of iterative algorithms in a distributed environment [56].

Table 2.2 provides a summary of common techniques used by existing work to facilitate iterative computations.

2.4.2 Graph computations

Graph computations are used to analyze data in multiple domains including social networks, sensor networks and traffic flow networks and most graph computations are iterative in nature. Different graph processing frameworks have been introduced to handle large-scale graphs and next, we look into common models used to facilitate parallel graph processing.

One of the well-known systems for parallel graph processing is Pregel [57] which

was introduced by Google. Pregel introduces a vertex-centric approach for distributed graph processing based on the Bulk Synchronous Parallel (BSP) model [58]. GPS [59], GraphX [60], Giraph [61], Mizan [62] and Pregelix [63] are some of the other graph processing systems built based on Pregel concepts.

In the vertex-centric model, vertices are the main execution element and algorithms are defined as a function of a vertex [57]. Similar to most graph algorithms, algorithms executed in this model are divided into iterations called supersteps. At each superstep, all the vertices execute the defined vertex function in parallel. In each iteration (superstep), all the vertices pass the results of the vertex function to the neighboring vertices. These passed results will be available at the destination vertices to use as input in the next superstep. In the vertex-centric model, synchronization happens after each superstep. Once all the vertices completed a superstep, they signal the end of the superstep. The next superstep can start only after all the vertices notify the end of the previous superstep. This is called a global synchronization barrier as every vertex has to wait till all the other vertices reach the end of the current superstep to start the next superstep. GraphLab [64] provides a similar model, but uses shared memory to communicate information about vertices and edges with each other without the need for message passing over the network. When implementing vertex-centric algorithms in streaming applications one of the main challenges is maintaining synchronization among distributed operator instances to ensure a new superstep starts only after all the vertices in all the instances finish the current superstep, which we address in Chapter 3.

The subgraph-centric model is another distributed graph processing approach explored in Giraph++ [65] and NScale [9] where the graph is partitioned into a set of subgraphs. This model divides the vertices of the graph into partitions and a user-defined function is implemented for a single partition. This function can access all of the vertices of a partition along with the edges, the vertices link to. This model also uses a synchronization barrier to synchronize computation between each partition. Blogel [66] introduces a similar approach referred to as the block-centric approach where a block is defined as a connected subgraph of the complete graph.

PowerGraph [67] exploits Gather-Apply-Scatter paradigm to support parallel graph processing. In the gather phase, vertices collect information about neighboring vertices and edges. The apply phase is used to modify the value of a vertex based on the collected information and in the scatter phase, values of neighboring edges are updated based on the modified vertex value. Similar to GraphLab [64], this model uses shared memory to support information communication between vertices and edges.

2.4.2.1 Dynamic graph processing

Most of the above discussed approaches focus only on static graph processing. When processing graphs in stream processing systems where continuous graph updates are expected, handling updates to the graph has to be taken into consideration in addition to efficient graph computations [68, 69]. Existing data processing systems have been improved to support dynamic graphs [70, 71, 72]. Some dynamic graph processing models require integration of different special purpose systems to provide a real-time distributed graph processing system. One method used for dynamic graph processing is combining graph processing and stream processing systems. In this model, when real-time data arrives, the stream processing system identifies the updates to be done to the graph. Once changes are identified, it calls the graph processing system. Then the graph processing system executes necessary algorithms and generates results. We refer to this model as the off-loading model in our work in Chapter 3. GraphCEP [21] is a real-time graph processing system that combines a complex event processing system and a graph processing system. Wickramarachchi et al. [73] propose a similar approach for processing dynamic graphs. GraphTau [71] built on top of Apache Spark is another example of a dynamic graph processing system based on an existing data processing framework.

Kineograph [8] is a dynamic graph processing system that analyses a series of snapshots of changing graphs. The graph storage layer of Kineograph is a distributed key-value store. Trinity [7] is another graph processing engine that is based on a distributed key-value store to support parallel processing. Kineograph breaks the graph into a set

of snapshots and calculations are performed on the snapshots. In our work presented in Chapter 3, we compare our proposed model with a similar model that exploits in-memory key-value storage. Chronos [74] suggests a similar snapshot based approach for temporal graph processing. However, Chronos assumes all the graph history details are available before the computation starts which is not true when processing real-world graph-structured data where updates are generated continuously. Kim et al. [75] investigate a similar approach that converts a dynamic graph into a set of static graphs ordered by timestamps to support dynamic graph processing.

Another dynamic graph processing approach is the distributed memory based model, where the parallel operators of a stream processing system use a distributed memory system to communicate intermediate results of graph computations among each other. Tornado [20] uses an external storage to store intermediate results while NScale [9], a static graph processing system utilizes Redis¹ as the distributed memory. Moreover, graph databases have been proposed to support dynamic graphs [76, 77] and GPU-based graph processing systems [78, 79, 80] have been proposed to support static as well as dynamic graphs.

2.4.3 Machine learning

Machine learning algorithms are another class of algorithms that heavily depend on stateful computations, in particular iterative computations. Existing frameworks have been extended to support iterative machine learning algorithms such as Mahout² built on top of Hadoop which uses a new map-reduce job for each iteration. Rosen et al. [81] introduce a new looping construct as an extension to map-reduce to support iterative machine learning algorithms. The loop construct requires three inputs: initializer, body and condition. The initial input of the loop body is given as the initializer and condition is used to check whether the loop has to be continued or terminated. The body consists of a set of map-reduce operators that computes the defined tasks and a sequential operator that

¹<https://redis.io/>

²<https://mahout.apache.org/>

Table 2.2: Methods for supporting iterative computations.

Method	Description	Systems
Dynamic task creation	Adding new operators to the operator graph dynamically for every new iteration	[54, 53]
Map-reduce extensions	Improving map-reduce paradigm to support iterations	[18, 52, 19]
Feedback edges	Allowing cycles in the operator graph to support intermediate result communication	[48, 47]
Distributed memory	Using a shared/distributed memory to communicate intermediate results	[9, 20]

updates the machine learning model.

Tamano et al. [82] presents another approach using map-reduce to support machine learning. Although the suggested approach uses a new job per iteration, the authors use several optimization techniques to improve system performance such as memory-based execution, overlapping computations with I/O operations in the map phase and job integration techniques to avoid duplicate data loading by multiple jobs that perform different iterations of the same computation. In addition, Chu et al. [83] propose a map-reduce based solution for machine learning computations in multi-core architecture and GraphLab [64] provides an abstraction based on asynchronous communication to support machine learning algorithms.

There exists work that investigates techniques to efficiently support common stateful computations in stream processing. In our work, presented in Chapter 3, we propose a communication model to improve the efficiency of stateful computations such as iterative algorithms in distributed stream processing systems. The next section focuses on the fault-tolerance techniques used to recover state after failures.

2.5 Fault-Tolerance

Failures are an inevitable part of large scale systems and a system failure can be defined as "an event that occurs when the delivered service deviates from correct service" [84].

Failure handling can be divided into four main categories, namely, fault prevention, fault tolerance, fault removal and fault forecasting [85]. In this section, we focus on the fault-tolerance aspect which ensures a system can continue processing even after a failure.

Fault-tolerance is an essential functionality of any large-scale data processing systems as failures ranging from application-level to hardware-level are expected to occur at any point in time. As the system size grows, the number of individual components that can fail increases resulting in increased frequency of failures. Especially, in stream processing applications that can run for a long period of time, failures during application execution are inevitable and therefore, fault-tolerance mechanisms are required to handle different types of failures. For stateless computations, handling failures can be as straightforward as continuing the computations in another available node after a node failure. However, with stateful computations state of the computations needs to be available to continue processing after a failure. Several techniques are being used to support state recovery and fault-tolerance for stateful computations and some of the most common fault-tolerance techniques are checkpointing, replication, upstream backup and optimistic recovery. Next, we discuss each of these techniques in detail.

2.5.1 Checkpointing

The most common approach used for fault-tolerance in stream processing is the checkpoint based approach where the snapshot of the current state is persisted periodically and in case of a failure, the persisted snapshot is used to restore the state. This type of fault-tolerance is referred to as passive fault-tolerance.

The checkpointing process can be divided into coordinated checkpointing and uncoordinated checkpointing. The coordinated checkpointing process is where all the operators in a stream processing application have one checkpoint interval and therefore all of the operators perform checkpoints simultaneously. This approach is also referred to as global checkpointing. Most streaming systems perform periodic checkpoints using a token-based mechanism [86]. In this approach, a token is sent from source operators to

the succeeding operators in the application and at the arrival of the token, each operator performs the checkpoint and sends the token to its output operators. For cases where an operator has multiple inputs, it waits for the token from all the inputs to start the checkpoint and when the operator receives a token from an input, it blocks that input channel until it receives tokens from other inputs. A checkpoint is considered as completed only when all the operators in the application complete their individual checkpoints. Apache Flink³ and Storm⁴ are example systems that use this approach. In this approach, only the source operators buffer the tuples processed by the application after the last checkpoint and when a failure occurs, all the operators restore their state from the last completed checkpoint and the buffered tuples at the sources are replayed to generate the lost state. In this model, the downstream operators are idle until the upstream operators reprocess the tuples.

Another approach is to only restore the state of the failed operator. However, to make sure events are not duplicated or lost due to other operators not rolling back to their previous states, the systems have to persist all the events communicated between operators. In the uncoordinated checkpointing approach, each operator has a unique checkpoint interval and therefore each operator performs its checkpoint independently. This method also requires storing the tuples sent from one operator to another to avoid tuple lost and tuple duplication. Meteor Shower [87] follows a similar approach where each operator performs checkpoints independently. Buffering data at every operator is not desirable as it requires a significant amount of additional storage and would become infeasible with high volume data streams.

Some systems use different ways of checkpointing such as key-based checkpointing [88], checkpoint combined with other fault-tolerance approaches [89]. For example, Spark checkpoints the RDDs asynchronously and uses a lineage graph which represents the graph of operations that used to build RDDs to restore the state [89]. However, lineage based failure recovery can be expensive if the lineage graph is large and computation logic is expensive [22]. In MillWheel [88] input and output records have an assigned

³<https://flink.apache.org/>

⁴<https://storm.apache.org/>

key and checkpointing is done per key. In addition, checkpoint-based fault-tolerance mechanisms have been introduced specifically for iterative algorithms. Graph processing systems which support iterations, usually checkpoint and store the state at the boundary of an iteration [57, 61, 63]. Xu et al. [90] introduce head and tail checkpointing for iterative graph algorithms that perform checkpointing in an unblocking manner.

2.5.1.1 Optimizations

Performing checkpoints negatively impact the system performance as resources have to be allocated to perform the checkpoint which otherwise could have been used for actual processing. Moreover, most of the checkpointing mechanisms stop the computations to perform checkpoints which impacts the execution time. For example, Naiad [48] pauses all of the workers and message delivery threads before checkpointing and IBM Streams [91] pauses tuple processing to complete the checkpointing. Therefore, several techniques have been proposed to minimize the overhead of the checkpointing process.

Checkpoint interval: The frequency of performing checkpoints directly impacts the system performance. For instance, more frequent checkpoints result in systems spending more time on checkpoints than actual computations and less frequent checkpoint results in systems reprocessing lost data due to failures instead of processing new data. In practice, however, the checkpoint interval is often configured to nominal values such as 30 minutes or 1 hour [92] without considering the salient aspects of the checkpoint process resulting in inefficient system performance. Therefore, determining an optimal frequency is essential to minimize the performance overhead imposed by checkpointing.

Several approaches have been proposed to determine the optimal checkpoint interval. Young [93] introduces a model to find the checkpointing frequency that minimizes the time wasted due to failures. However, this model does not take into account failures during recovery and checkpointing. Daly [94, 95] improves this model and uses a cost function to determine the frequency that gives the minimum total wall clock time to complete an application. The wall clock time consists of time spent on actual computations,

checkpointing time, rework time and restart time. The model proposed by Jin et al. [96] for High Performance Computing (HPC) environments requires the sequential workload of the application. However, for streaming systems determining the time to complete an application is irrelevant since the workload is unbounded.

Naksinehaboon et al. [97] propose approaches focusing on reducing the wasted time of a system and Ling et al. [98] and Ozaki et al. [99] propose approaches based on calculus of variations to approximate the optimal checkpointing frequency. In addition, Tang et al. [100] investigate how power capping affects the checkpointing process and propose a power capping aware model to optimize the checkpoint interval. Rahman et al. [101] also provide a model for volunteer computing environments which minimizes the completion time. This approach assumes that a faulty process can only start after a checkpoint interval and time to detect failure and restart is negligible, which is not the case with streaming applications. Therefore, the mathematical models presented in Chapters 4 and 6 to derive optimal checkpoint interval, consider failures during checkpoints as well as recovery and take into account different factors specific to stream processing systems such as depth of the streaming application and message delay between operators.

Fialho et al. [102] propose a model for uncoordinated checkpointing where each processor performs checkpoints independently, but this is not the case for most streaming systems. Zhuang et al. [103] also present an optimal checkpointing model for stream processing applications where each operator has an independent checkpointing interval. However, these models cannot be applied to existing stream processing systems such as Apache Storm and Flink which require a unique checkpointing interval for the application, not for each operator. Furthermore, for large-scale applications running on large clusters, as the number of nodes in the cluster increases, the time between failures can reduce from hours to minutes. Therefore, assumptions made on existing models, such as the sum of the checkpoint interval and checkpoint cost being significantly less than the mean time between failures [103] may no longer be valid. Table 2.3 summarizes the factors considered by existing work when deriving the optimal checkpoint interval.

Most of the proposed models have shown the performance improvements achieved

using the optimal checkpoint interval through simulations rather than experiments on real systems. Hence, how the checkpoint interval can affect important application parameters such as throughput, latency and how the optimal interval can positively influence such parameters have not yet been examined properly with practical applications. Chapter 5 demonstrates how the checkpoint optimization model presented in Chapter 4 can improve the utilization and efficiency of practical real-world streaming applications through experiments using a state-of-the-art stream processing system.

Dynamically changing the checkpoint interval based on varying failure rates [104, 105, 106, 107] is another approach investigated to reduce checkpointing overhead. However, the system has to ensure that the overhead of maintaining past failure event information and dynamic calculations is less than the overhead of performing periodic checkpoints.

Checkpoint cost: Reducing checkpoint cost is another way of minimizing the overhead of checkpoints. The checkpoint cost can depend on several factors such as state size and write speed of the persistent storage. Next, we explore different techniques proposed to reduce the checkpoint cost.

The use of incremental checkpoints [111, 112, 113, 114] is a common method used to reduce the checkpoint cost. In this approach, instead of persisting the complete state when performing the checkpoints, the system only persists the changes to the state that occurred after the last checkpoint. This can reduce the size of the state that needs to be persisted resulting in reduced checkpoint cost, especially for computations that do not result in changes to the entire state frequently. For example, incremental checkpointing is useful for windows where the window elements overlap in consecutive windows. Sebeopou et al. [115] propose an incremental checkpoint approach for window-based computations.

However, incremental checkpointing does not provide a significant performance improvement for computations where state changes frequently such as algorithms that has bulk iterations, in which the state completely changes after a new iteration compared to

Table 2.3: Factors considered in existing work proposed to determine the optimal checkpoint interval.

	Young [93]	Daly first-order [95]	Daly [94]	Jin et al. [96]	Naksinehaboon et al. [97]	Ling et al. [98]	Zhuang et al. [103]	Liu et al. [108]	Shastry et al. [109]	Zhu et al. [110]
Failure during computation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Failure during checkpoint	×	✓	✓	✓	✓	×	✓	✓	✓	✓
Include restart	×	✓	✓	×	✓	×	✓	✓	✓	✓
Failure during restart	×	×	✓	×	×	×	×	×	×	×
Multiple failures during checkpoint interval	×	×	✓	✓	×	✓	✓	✓	×	✓
Multiple failures during restart	×	×	✓	×	×	×	×	×	×	×

the state of the previous iteration. Another drawback of incremental checkpointing is the high recovery cost. As the system only persists changes to the state, when a failure occurs all the persisted states have to be read and combined together to get the complete state which is more time consuming than reading the last persisted checkpoint. In order to reduce the recovery cost, the entire state is checkpointed from time to time instead of the changes to the state [111]. Hence when a failure occurs, the system only has to read the states from the last completely checkpointed state.

Zheng et al. [116] propose a method to reduce the communication overhead of the checkpointing process by storing all the checkpoints twice, locally in the same node and in another node called a buddy node without using reliable storage to reduce the high communication cost of writing/reading to a reliable storage. In this approach, when a

node fails it can recover state from the checkpoint stored in the buddy node. However, this approach cannot recover state when both nodes fail simultaneously. Compressing the checkpoint to reduce the checkpoint cost [117, 118, 119] is another way used to reduce checkpoint overhead.

SGuard [10] reduces the time taken to persist checkpoints by partitioning large checkpoints into smaller chunks and writing the chunks to the persistence storage in parallel. In addition, SGuard performs checkpoints asynchronously without disrupting the normal processing of the application.

Multi-level checkpointing: Even though the optimal checkpoint interval can improve application efficiency, the overhead of performing checkpoints becomes significant especially in large scale systems where more frequent failures are expected. Therefore, more elaborate checkpointing approaches such as multi-level checkpointing [120, 121, 122] have been introduced to further reduce the checkpointing overhead. Multi-level checkpointing considers multiple failure types and each failure type has a different checkpoint cost and a restart cost. In this approach, less expensive checkpoint types are performed more frequently while more expensive and more resilient checkpoint types are performed less frequently. For example, Moody et al. [123] and Mohror et al. [121] present Markov models for a multi-level checkpointing system. Di et al. [124] introduce a mathematical model for multi-level checkpointing in HPC applications which focuses on minimizing the wall clock time. However, this requires the job-length for the optimization which cannot be determined beforehand in a streaming application and assumes failures do not occur during checkpoints and recovery. They also propose a two-level checkpointing model for HPC applications assuming that failures do not occur during recovery [120]. In this model, they provide two independent checkpoint intervals for each level for jobs with an unknown length. However, having independent intervals results in overlapping checkpoints and lower utilization. Benoit et al. [125] also propose a similar model for HPC systems assuming no failures during checkpoints and recovery, which is not true for real-world systems.

In summary, there exists work that investigates techniques for minimizing the over-

head of fault-tolerance techniques, especially for the checkpoint-based fault-tolerance. However, there is a gap in optimizing the checkpointing approach in stream processing systems that deal with unbounded workloads and use global checkpointing. Therefore, our work presented in Chapters 4,5 and 6 focuses on optimizing single-level checkpointing as well as multi-level checkpointing for stream processing applications.

Approximate fault-tolerance: Approximate fault-tolerance addresses the trade-off between accuracy and performance and provide guarantees that errors due to failures are bounded [126, 127]. This approach checkpoints the state and the unprocessed data only when the number of errors due to failures is higher than the user-defined bound. Approximate fault-tolerance does not result in highly inaccurate results based on the fact that errors due to lost data are compensated after processing a sufficiently large number of data. Although this approach does not always guarantee accuracy, it gives better performance as the number of performed checkpoints are fewer compared to a system that does periodic checkpoints.

Jacques-Silva et al. [114] present another partial fault-tolerance method where users can define which parts of the application require fault-tolerance guarantees. Users can decide which operators of the stream processing application need to perform checkpoints which can reduce the checkpoint cost compared to complete application performing periodic checkpoints.

Improving recovery process: The recovery process of checkpoint-based fault-tolerance consists of time taken to read the persisted checkpoint and the time taken to reprocess the data that was processed by the systems after the last completed checkpoint. Several approaches have been presented to reduce the recovery cost as higher recovery cost results in poor system performance.

Scaling out the persisted state to multiple available nodes can speed up the recovery as lost data due to the failure can be reprocessed by multiple nodes in parallel [6, 128]. This approach combines the failure recovery process with the scaling-out process that dynamically determines when to scale out an operator based on the workload and bot-

tlenecks of each operator. To facilitate this process, the system should be able to partition the operator state while ensuring the accuracy of the results.

Su et al. [129] present a partial recovery method to facilitate failure recovery. This method recovers failed operators based on the availability of resources and some partitions of the failed operator are recovered first and the rest of the failed partitions are recovered gradually as new resources become available.

2.5.2 Replication

Replication-based fault-tolerance categorized as active fault-tolerance is another common fault-tolerance approach used in several systems. In replication-based fault-tolerance, operators have replicas with all the replicas processing the same data to ensure consistency. Hence, the state is duplicated in multiple replicas in different nodes so that in case of a failure of one machine, the state from one of the other replicas can be used to continue computations. Maintaining replicas involves two main tasks, ensuring all the replicas receive the same data in the same order and ensuring results are accurate and not duplicated [130].

Borealis stream processing system [131] uses the replication-based approach where the entire query network is executed by multiple nodes in the system. Therefore, when a node detects a failure from an input stream, it can start consuming input from another replica that outputs the same input stream. Borealis uses a unique operator to ensure all the replicas consume the input in the same order to guarantee consistency. Flux [132] which adopts a replication-based approach, uses an intermediate operator to ensure no events are lost or duplicated to ensure consistency. Stormy [133] uses a similar approach where each query is duplicated in multiple nodes while E-Storm [134] replicates all the stateful operators and makes use of an asynchronous recovery protocol to retrieve lost state from alive replicas. Semi-active and semi-passive are two variations of replication [135] that use different approaches to update the replicas to get to the correct state.

Although different approaches are used to ensure all the replicas are consistent, replication requires a substantial amount of additional resources to duplicate the computations and therefore not practical and economically viable for large-scale systems [87, 136, 137]. Besides, additional effort is required to duplicate the data and to ensure all replicas process data in the same order. Moreover, at the event of multiple node failures, this approach may not be able to recover when there are only a few replicas.

2.5.2.1 Optimizations

This section explores a set of optimization techniques proposed to improve replication-based fault-tolerance. Bellavista et al. [138] present the Load-Adaptive Active Replication method that dynamically changes the number of replicas based on the system workload. This approach monitors data sources and the incoming data rates and for high workloads, some of the replicas are dynamically deactivated to handle the workload spike. Fang et al. [139] propose a technique to handle high workloads of individual nodes in the system. This method balances workloads of nodes by deactivating computations in nodes with high workloads and activating one of the replicas in nodes with a lower workload. Dynamic Lazy Insertion [140] is another approach presented to handle high workloads that selectively decide which data to be replicated. Passive and partially active fault-tolerance [141] is a hybrid of checkpointing and replication where only selected operators will have replicas while others use checkpoints. In this approach, the number of operators that use replication is determined by the availability of the resources.

Brito et al. [135] propose an optimistic delivery of messages to improve the latency of replication. This approach allows processing of messages before knowing the correct order but later handles cases of unordered messages. A new data representation method called Multilevel Counting Bloom Filter [140] has also been proposed to reduce the network and memory requirement of replication-based fault-tolerance to minimize the high resource requirement.

2.5.3 Upstream backup

In upstream backup, all the operators buffer the tuples sent to downstream operators until they are acknowledged. This approach can become complex when handling cases where an operator has multiple downstream operators [142]. Upstream backup is often used with other fault-tolerance approaches [130]. For example, some checkpointing approaches require operators to buffer output tuples and when a downstream operator performs a checkpoint, the output tuples processed before the checkpoint is removed from the buffer of the upstream operator.

2.5.4 Optimistic recovery

Optimistic recovery [143, 144] is another approach used for failure recovery in stream processing systems focusing on iterative algorithms. This approach is designed for fix-point algorithms that converge to the correct result from different intermediate states [143]. Optimistic recovery uses a user-defined compensation function to recover instead of using a persisted state. Hence users have to define a compensation function suitable for the defined computations. After a failure, the compensation function converts the algorithm to a consistent state from which execution can proceed to find the final result. Although this method avoids the cost of performing checkpoints, compensation functions can be defined only for some iterative algorithms and therefore cannot be applied to generic streaming applications.

2.5.5 Message processing guarantees

Different fault-tolerance approaches provide different types of guarantees on the accuracy of the state and the guarantees provided by systems can be divided into three categories: at-most-once, at-least-once and exactly-once.

At-most-once: This category ensures that all of the messages are processed at most

once and no message is duplicated. Therefore, the systems state is never changed by a single message twice. The drawback of this approach is that some messages will be lost due to failures as messages will not be replayed after failures.

At-least-once: This category ensures that all of the messages are processed at least once which can result in some messages processed multiple times by the systems. Therefore, the system state can change multiple times due to duplicate message processing. Duplicate message detection functionally is not required by the systems that ensure at-least-once, but these systems will replay all the lost messages after failures.

Exactly-once: This category ensures all of the messages are processed exactly once resulting in system state reflecting all the processed messages accurately. To ensure exactly-once guarantee, systems will replay messages lost due to failures but duplicate message detection techniques are deployed to ensure no message is processed twice.

2.6 State-of-the-art Stream Processing Systems

This section focuses on well-known stream processing systems and presents how each stream processing system supports stateful computations in terms of state management, windowing support and fault-tolerance mechanisms.

2.6.1 Storm

Storm ⁵ is a widely used stream processing system in which the streaming applications are referred to as topologies and events processed by the topology are referred to as tuples. A topology consists of operators which are categorized into two groups: spouts and bolts where spouts are the data sources and the bolts are the data processors. Recent versions of Storm have introduced support for stateful bolts to support stateful processing.

⁵<https://storm.apache.org/>

State management: The latest Storm version only provides support for key-value based states. However, custom states can be implemented by the user to support complex state types. In addition to in-memory, the default state storage, Storm also supports Redis or HBase based state storage to persist states.

Windowing: Storm provides built-in support for sliding and tumbling windows based on time and count policies. Window size and slide can be based on the same policy such as time based window size and time-based slide or a window can have size and slide based on different policies such as time-based size and count-based slide. Moreover, users can configure the maximum number of tuples that can be kept in-memory to handle large windows where all the tuples in the window cannot be kept in-memory. In such cases, Storm restores tuples in the window when necessary from the backend state storage.

Fault-tolerance: Storm initially did not provide support for checkpointing but later introduced checkpointing to support fault tolerance in stateful computations. Initially, there was no state recovery mechanism for stateful computations in Storm and at-least-once guarantee was provided which ensures that each input tuple of the topology will be processed at least once using an acknowledge based mechanism, where the spout that generates tuples discard the tuple only when the output associated with that tuple leaves the topology [145]. However, new versions of Storm introduced token-based global periodic checkpoints to support stateful operators. When checkpoints are enabled in Storm, a checkpoint spout is automatically added to the application and this spout generates the token and sends it to downstream bolts periodically. There are a few limitations in Storm checkpointing approach. For example, as Storm depends on the acknowledge based mechanism to support replaying of events in case of failures, some events can be duplicated which results in an incorrect state. Moreover, Storm does not handle cases where a bolt has multiple inputs from upstream bolts. A bolt performs the checkpoint as soon as it receives one checkpoint token and does not wait for the token to arrive from all the upstream bolts, which results in inconsistent state and duplicate event processing when a failure occurs.

2.6.2 Trident

Trident is an extension of Storm which provides support for stateful computations. This was introduced because Storm did not provide support to manage state at the time. Trident processes streams of data by dividing the stream into small batches. However, Trident is well-suited only for small states and results in high latency for big states [3].

State management: Trident allows state to be kept in an external database or in-memory with HDFS backup to support state persistence. Trident supports three different states and spouts: non-transactional state/spout, transactional state/spout and opaque transactional state/spout. Moreover, all three state types support key-value based states.

Windowing: Trident provides built-in support for time-based and count-based sliding and tumbling windows. Unlike Storm, built-in Trident sliding windows only support window size and slide based on the same policy.

Fault-tolerance: Each batch of data in Trident is given a unique transaction id and state updates are ordered among batches to guarantee exactly-once processing semantics. Different combinations of Trident spouts and states provide different message processing guarantees and exactly-once processing guarantee can be achieved with the combination of transactional state and transactional spout or with the combination of opaque transactional state and transaction/opaque transactional spout.

2.6.3 Heron

Heron⁶ is developed at Twitter to overcome limitations of Storm. Similar to Storm, Heron application is referred to as a topology that consists of spouts and bolts [146].

State management: Unlike Storm, Heron supports stateful spouts in addition to stateful bolts. Stateful spouts and bolts support key-value based states and by default state is stored in the local file system. Heron also provides support for HDFS or Book-

⁶<https://apache.github.io/incubator-heron/>

Keeper based state storage.

Windowing: Windowing support of Heron is similar to Storm and has built-in implementations for sliding windows and tumbling windows based on time and count policies.

Fault-tolerance: Similar to Storm, Heron uses checkpointing and tuple acknowledgement to facilitate fault-tolerance. Heron cannot provide exactly-once guarantee for stateless topologies and only provides at-most-once or at-least-once semantics. However, for stateful topologies, Heron provides exactly-once guarantee, if the stateful operations are idempotent. However, most stateful computations used in stream processing are not idempotent resulting in inconsistent states and incorrect results if Heron is used as the stream processing system.

2.6.4 Flink

Flink⁷ stream processing engine was introduced to support stateful streaming computations. Flink applications consist of streams and transformations, where streams refer to the continuous data and transformations refer to the operations performed on the streams of data.

State management: Flink supports two types of states: keyed state and operator state. The keyed state is based on key value pairs similar to Storm and the operator state is used for non-keyed computations. In addition, Flink provides an operator state called broadcast state to support broadcasting the state to all downstream tasks in a Flink application. Keyed state and operator state can be divided further into two categories, managed state and raw state. The managed state supports several various state types including list-based states, map-based states and the raw state is used for data structures that are not controlled by Flink runtime. Moreover, Flink provides a feature called queryable state that allows users to access state from outside of Flink during runtime.

⁷<https://flink.apache.org/>

Windowing: Flink supports time-based and count-based sliding windows and tumbling windows. In addition, two other window types are supported by Flink, namely, session windows and global windows. Session windows do not have a fixed size but a window gets closed when it does not receive tuples for a predefined period of time.

Fault-tolerance: Flink uses checkpoints to support fault-tolerance where a checkpoint token referred to as the stream barrier periodically is sent to the streaming application and when an operator receives the barrier, it performs the checkpoint. Flink's checkpointing process is coordinated and managed by Flink job manager which is the master process of the Flink cluster. Unlike Storm which guarantees at-least-once event processing guarantee, Flink guarantees exactly-once semantics which makes sure the output events reflect all the input events exactly once and the application state is consistent [86].

2.6.5 Spark streaming

Spark ⁸ provides support for stream processing through Spark streaming, an extension of Spark API. Similar to Trident, Spark streaming divides the input stream into batches before processing. Discretized streams (D-Streams) are the main element of Spark streaming and intermediate results of D-Stream computations create new D-Streams and may create an intermediate state in the form of resilient distributed datasets (RDDs) [89]. Due to dividing data stream into batches, Spark applications are not well-suited for applications with latency needs below a few hundred milliseconds [146].

State management: Similar to other stream processing systems, Spark streaming supports key-value based state and different state storage options including in-memory, HDFS and Cassandra. The input data of Spark is usually kept in-memory, but when there is insufficient memory, Spark spills the data into disk which can impact the latency of streaming applications. Therefore, it is recommended to provide sufficient memory to Spark applications to avoid high latency results.

⁸<https://spark.apache.org/>

Windowing: Spark only supports time-based tumbling windows and time-based sliding windows. As window data is always kept in-memory, Spark can only support large windows if there is sufficient memory to keep all the window data in-memory.

Fault-tolerance: Spark streaming uses checkpoints and replication to support fault-tolerance. Spark replicates input data among multiple executors to handle failures with the default replication factor of two. Moreover, Spark uses two types of checkpoints: metadata checkpointing and data checkpointing. The metadata checkpointing process persists the information defining the streaming operator and the data checkpointing process persists the generated RDDs.

2.6.6 Samza

Samza ⁹ which is based on Hadoop and Kafka [147] is a stream processing engine that supports at-least-once guarantee that can result in events being processed multiple times after a failure.

State management: Samza also supports key-value based state and can efficiently handle large states. Samza handles large states by storing the state in disk when memory is insufficient and reduce the latency overhead of disk-based storage through caching mechanisms.

Fault-tolerance: Samza uses periodic checkpoints to persist the offset of the latest message processed and stores a changelog capturing changes to the state and the content of the changelogs is used to recover state after failures. Samza uses changelogs arguing that saving state updates is more efficient than checkpointing the complete state [148].

⁹<https://samza.apache.org/>

2.6.7 Apex

Apex¹⁰ is another stream processing system that supports stateful computations. Apex applications are represented as a directed acyclic graph of operators, where operators are connected by streams. Apex operators can be divided into three categories: input adapter, generic operator and output adapter.

State management: Apex supports key-value based state and also facilitates state spilling to handle large states. Users can set the maximum size of the state that can be kept in-memory and when the state size exceeds, data persisted in the state storage is removed from in-memory.

Windowing: Apex supports four different types of windows: time-based sliding, time-based tumbling, session windows and global windows. Apex provides two storage methods to save window state, in-memory windowed storage and spillable windowed storage. The in-memory windows storage is suited for small window sizes as it keeps window data in-memory and window state is only persisted periodically at the checkpointing time. The spillable windowed storage keeps state in persistent storage and can handle large states.

Fault-tolerance: Apex uses periodic checkpoints to provide fault-tolerance. At the time of the checkpoint, the operator state is saved to local disk and then the system asynchronously copies the saved state to Hadoop Distributed File System (HDFS). Asynchronous copying to HDFS is done to ensure that time taken to write to HDFS does not impact the application performance. In Apex, a checkpoint is considered as complete when the entire state is written to HDFS. However, for Apex to provide exactly-once guarantee, copying the operator state to HDFS is done synchronously blocking the operator until the state is written which increases the checkpoint cost compared to providing at-least-once guarantee.

Table 2.4 shows a comparison of stateful processing support of streaming systems.

¹⁰<https://apex.apache.org/>

2.6.8 Related Surveys

There are several surveys that compare the capabilities and limitations of state-of-the-art stream processing systems. For example Noghabi et al. [148], Carbone et al. [86] and Zaharia et al. [89] presented how stateful computations are supported in Samza, Flink and Spark respectively. Noghabi et al. [148] evaluate and compare two state storage options, local state which can either be in-memory or the disk and remote state in which case an external storage is used. Moreover, they highlight several techniques they have used to improve system performance such as partitioned state to improve scalability and incremental and asynchronous snapshotting to improve fault-tolerance. Similarly, Carbone et al. [86] highlight the differences between local state and remote state and the importance of the partitioned state in a distributed environment. In addition, Zaharia et al. [89] demonstrate how the performance of checkpoint based fault-tolerance is impacted by checkpoint interval and the number of nodes in a cluster.

Inoubli et al. [147] presented a survey on big data frameworks and To et al. [3] presented a taxonomy for state management for generic data processing focusing on state management aspects including state sharing, fault tolerance, elasticity and load balancing. They proposed four main dimensions to consider when evaluating state management solutions, namely, efficiency, ease of use/management, functionality and seamless integration, which can be used to evaluate streaming systems. Several surveys have been done on stream processing systems highlighting various aspects such as architecture, scalability, state management and fault tolerance [5, 142, 149, 150] while Heinze et al. [151] have surveyed state management and fault tolerance mechanisms used by existing stream processing systems.

2.6.9 Discussion

Most state-of-the-art stream processing systems support stateful computations but the efficiency of stateful computations can be further improved by using the theoretical models

proposed in literature which we discussed in sections 2.3, 2.4 and 2.5.

For instance, when supporting window processing, most stream processing systems support key-based partitioning to allow windows of different sub streams partitioned using a user-defined key to be processed in parallel. However, techniques like pane-based partitioning have to be implemented in stream processing systems to handle parallel processing of large windows such as global windows. Moreover, most systems support key-value based states and do not support non-keyed states or complex states like map-based states or graph-based states. Supporting complex state types similar to operator state and managed state introduced in Flink could be beneficial to other stream processing systems to support complex stateful computations.

Table 2.4: Comparison of stream processing systems.

System	State types	Window types	Fault-tolerance	State storage	Message processing guarantee
Storm	Key-value	Count-based tumbling Count-based sliding Time-based tumbling Time-based sliding Time-based with count-based slide Count-based with time-based slide	Checkpoints and tuple acknowledgement	In-memory Redis HBase	at-least once
Storm- Trident	Non-transactional Transactional Opaque transactional	Count-based tumbling Count-based sliding Time-based tumbling Time-based sliding	Persisting state and transaction id	In-memory HDFS Memcached Cassandra any persistent storage	exactly-once: by transactional, opaque transactional spouts at-most-once, at-least-once:by non-transactional spouts
Spark streaming	Key-value	Time-based tumbling Time-based sliding	Checkpoints, replication and lineage	In-memory HDFS any persistent storage	exactly-once

Table 2.4: Comparison of stream processing systems (continued).

Flink	Key-value operator	Count-based tumbling Count-based sliding Time-based tumbling Time-based sliding Session window Global window	Checkpoints	In-memory HDFS S3 any persistent storage	exactly-once at-least-once
Heron	Key-value	Count-based tumbling Count-based sliding Time-based tumbling Time-based sliding Time-based with count-based slide Count-based with time-based slide	Checkpoints and tuple acknowledgement	Local file system Zookeeper	effectively once: stateful, idempotent topologies at-most-once, at-least-once: stateful non-idempotent topologies, stateless topologies
Samza	Key-value	Time-based tumbling Time-based sliding Session window	Checkpoints and log of updates	In-memory Disk	at-least-once
Apex	Key-value	Time-based tumbling Time-based sliding Session window Global window	Checkpoints	In-memory Disk HDFS	at-least-once at-most-once exactly-once

2.7 Summary

We started this chapter with an overview of different state types and state management tasks focusing on state storing and state communication and the methodologies used to handle large states. We also discuss how window semantics are defined and different approaches proposed to improve window processing efficiency followed by an introduction to common stateful computations used in stream processing applications and the methods used by existing work to support those stateful computations. We also discussed the limitations of existing work in terms of scalability and efficiency. We provided an overview of fault-tolerance mechanisms used to restore state after failures including checkpointing, replication and optimistic recovery. We also presented the optimizations suggested to improve the discussed fault-tolerance approaches. Finally, we provided an overview of state-of-the-art stream processing systems and how these systems support stateful computations, state management and fault-tolerance.

Although there is work done to support stateful computation in stream processing and to provide fault-tolerance, still there are several areas that have to be further investigated to efficiently support complex stateful computations and to reduce the overhead of fault-tolerance approaches on system performance. For example, efficient state management for computations such as iterative computations that maintain state which frequently changes, models to define the semantics of complex stateful window computations to ensure results of window operations does not differ from one implementation to another and optimization techniques to reduce the overhead of fault-tolerance techniques in state-of-the-art stream processing systems are few of the areas that require further exploration. Hence this thesis explores how stream processing systems can be improved to support complex stateful computations focusing on state management and fault-tolerance. Our contributions are presented in the next four chapters.

Chapter 3

Complex Stateful Computational Support for Stream processing Systems

3.1 Introduction

Stream processing systems incorporate state when supporting windows and stateful operators. Real-world streaming applications require windows to partition the continuous data to subsets and stateful operators to model domain specific computations. However, both windows and stateful operators cannot efficiently model complex streaming use cases due to lack of formal semantics and communication restrictions. Therefore, this chapter provides models to efficiently support stateful processing with the focus on windowing and iterative computations which is a common stateful operation frequently used in streaming applications.

This chapter is derived from:

- Sachini Jayasekara, Shanika Karunasekera, and Aaron Harwood. "A Multi-stage Hierarchical Window Model with Application to Real-Time Graph Analysis," *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*.
- Sachini Jayasekara, Shanika Karunasekera, and Aaron Harwood. "Enhancing the Scalability and Performance of Iterative Graph Algorithms on Apache Storm," *2018 IEEE International Conference on Big Data (Big Data)*.
- Sachini Jayasekara, Xunyun Liu, Shanika Karunasekera, and Aaron Harwood. "Communication Model for Parallel Iterative Stream Processing," *2018 IEEE International Conference on Big Data (Big Data)*.

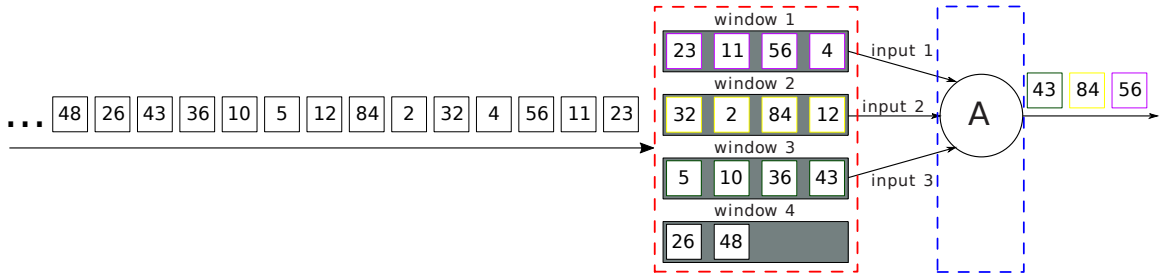


Figure 3.1: Window-based computations with a window of length 4 and operator A calculating the maximum element.

Figure 3.1 depicts a simple example of a window-based computation. The figure shows a window of length 4 and the window elements are sent to operator A to calculate the element with the maximum value. In a streaming application, a window which is highlighted in red in the figure can be defined based on different policies and windowed data could be sent to any type of operator which is highlighted in blue. The windowing models available to date do not go far enough to aid in the development of well defined complex analysis that requires multiple stages of computations. Therefore, in this chapter, we first present a hierarchical windowing model that allows defining windows at multiple levels.

Windowed data can be sent to any operator to perform computations. Iterative computations are a common stateful operation used in various stream processing use cases including graph processing and machine learning. The DAG based model that is used to represent stream processing applications, shown in Figure 3.2, however, cannot efficiently support iterative algorithms as it does not allow communication among parallelly running instances of the same operator (dashed arrows indicated in Figure 3.2); we refer to this type of communication as intra-operator communication.

One of the distinguishing characteristic of an iterative algorithm is that it requires the results of the previous iteration to perform the next iteration. Therefore, when implementing iterative algorithms using an operator, all the instances of the operator have to communicate the results of the previous iteration with each other to proceed with a new iteration. But with the communication restriction that does not allow intra-operator com-

munication, it is not possible to directly pass the intermediate results among the operator instances without additional resources facilitating state sharing among the instances. Because of the strict DAG based communication, iterative algorithms can be performed in a stream processing system in only a limited number of ways. The most common approaches that can be used to support iterative algorithms are listed below:

- Sending all the data to an operator with a single operator instance as shown in Figure 3.3a, so that the operator has all the information to perform succeeding iterations. When handling large volumes of data, using a single operator instance is not practical due to memory constraints.
- Offloading the computation part to a third-party system which is shown in Figure 3.3b. Examples of a third-party system could be a distributed storage [18, 19] or a graph processing system [21, 73]. This approach requires an additional communication channel between different systems which is inefficient and also requires maintenance of multiple independent systems.
- If iterations can be performed only with local results of the previous iteration, then we can use a DAG with multiple operator instances as shown in Figure 3.3c. However, this approach is not applicable for most iterative algorithms due to the needs of aggregation and synchronization.
- Unrolling the loop [46] as shown in Figure 3.3d, so that each iteration is performed by a separate operator. This approach is not practical when there is a large number of iterations or when the number of iterations is not fixed but determined by the processed data.

When analyzing iterative algorithms, there are algorithms that have complex communication patterns between iterations. For example, PageRank algorithm requires communication between neighboring vertices per iteration and in algorithms such as bitonic mergesort, communication pattern changes in each iteration. Therefore, having a communication flow that can support these types of complex patterns can improve the overall

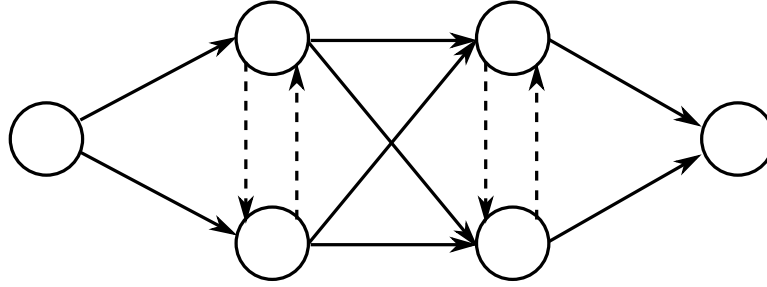


Figure 3.2: DAG of operators. Dashed arrows indicate the communication paths not allowed in a DAG.

efficiency of stream processing applications. We show that intra-operator communication can be used to facilitate these type of complex communication patterns.

Using intra-operator communication, intermediate results can be passed between operator instances without reducing the degree of parallelism of the operators or using a third-party system. However, introducing intra-operator communication is complex in a streaming context because concerns such as maintaining synchronization and handling the continuous arrival of input data have to be addressed. Therefore, we introduce a communication flow model with intra-operator communication which can improve the efficiency of iterative stream processing while guaranteeing the correctness of results. In addition, we show that the efficiency of non-iterative parallel algorithms can be further improved by our communication model. The main contributions of this chapter are summarized as follows:

- We present a multi-stage hierarchical windowing model to define complex computations on continuous streaming data.
- We propose an intra-operator communication model which can facilitate iterative computation implementations without using any third-party system.
- We implement the proposed intra-operator communication model on top of Apache Storm and provide a generic API with which iterative streaming algorithms can be modeled.
- We demonstrate the efficiency of the proposed model by comparing it to state-of-

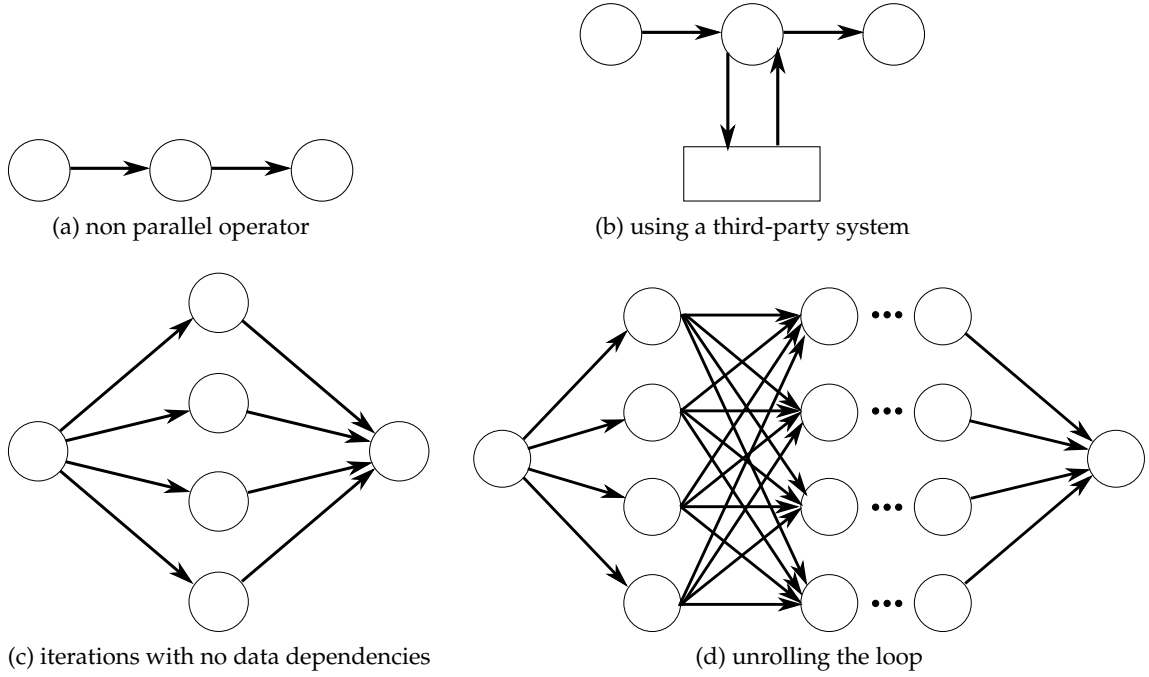


Figure 3.3: Approaches for iterative stream computation.

the-art approaches using a set of well-known iterative algorithms and real-world large-scale datasets.

3.2 Window Model

In a streaming setting, data continuously comes to the system and the computations have to incorporate the new data to produce accurate results that reflect the actual properties of the data. Windows are used in stream processing systems to handle the continuous updates and a window can be defined as a bounded, contiguous set of tuples from a stream. The work by Botan et al. [31] provides a rigorous definition of windowing *for a single windowing stage*. In our work we recognize that time windowing is useful at multiple different time scales in the one system and at different stages of processing and so we propose a generic model for stream processing that supports a hierarchical approach to windowing, extending the approach proposed by Botan et al. They suggest their model

could be extended in some way like this, as future work, in their discussion of binary operators. We also extend Botan's work to include time intervals for the application time of events. Our work has different assumptions about how the stream processing system is being used. Botan considers that users are making queries, and expects the queries to be reproducible for a given stream of data. In our work we consider the system is streaming results to a viewer and we expect that the stream result be well defined. This affords some subtle changes in semantics that allow system time based windows to be used in a rigorous way, leading to a simpler implementation. The next section describes the fundamentals of our windowing model.

3.2.1 Tuples

In our work a tuple contains three fundamental elements:

$$e = \langle x, t^{app}, t^{sys} \rangle,$$

where x is the data specific to the tuple, $t^{app} = [t^{app,min}, t^{app,max}]$ is the time interval in which the tuple was generated by an application, and t^{sys} is the time at which the tuple was received by some system for processing. We can also write x_e , t_e^{app} and t_e^{sys} to refer to the individual elements of e .

For example, consider a microblog post that occurred at time t_1 , i.e. it was generated by a microblog service at this time by a user of the service. If the post data is x then $t^{app} = [t_1, t_1]$. In other words this post is represented as occurring at a single time point. At this point the value of is not applicable but it can be set to t_1 as well, representing zero delay thus far, as we discuss further in. Let the entire post become a single tuple, let's say $e = \langle x, t^{app}, t^{sys} \rangle$ and $x_e = post$. Consider e being received by another system for processing at a later time $t_2 > t_1$ and consider that this system outputs a set of tuples, S , as a result, e.g. one tuple for each word in the post. At arrival, t_e^{sys} is set to t_2 and this value is used when processing the tuple internally. For $f \in S$, the system would set $t_f^{app} = t_e^{app}$. These tuples all have the same t^{app} as the original post tuple, maintaining

the fact that the data (words) associated with the tuples were generated by an event that occurred at time t_1 . The value of t_f^{sys} can be set to the time the tuple was emitted, but it will be set again by a downstream system to be the time the tuple arrived at that system.

3.2.1.1 Intervals:

More generally if a result tuple f is output as a result of a set of input tuples $e \in S^{input}$, i.e. f is computed based on all the tuples in S^{input} , then:

$$t_f^{app} = \left[\min_e \{t_e^{app}\}, \max_e \{t_e^{app}\} \right].$$

The utility of t^{app} and t^{sys} is that we can know for a given tuple, which may represent the result of a query, over what application time does it represent. E.g. the result may be a word frequency histogram and it may represent posts that occurred from 10:30am to 11am. Also we can determine the latency between when the original event (or set of events more generally) occurred to when a result tuple is produced that pertains to that event:

$$\Delta = t^{sys} - t^{app, max}.$$

This gives us a measure to minimize in that we want to produce results with near zero delay.

3.2.2 Streams

In our work a stream, S , is a sequence of *delimited* tuples:

$$S = e_1, e_2, \dots, D, e_j, e_{j+1}, \dots, D, \dots, e_n$$

The delimiter, D , is a special stream element that is used to “batch” or divide the tuples into windows. It serves as a synchronization timestamp for downstream operators to advance their windows, especially in the absence of tuples in the stream. Botan suggests the

use of a “dummy tuple” for this role. In other work the delimiter is called a *heartbeat* [152] or *timeout* [153]. All tuples within the batch have identical t^{app} values. We talk about a window and a batch synonymously, in that when a window of tuples is emitted from an operator we call that a batch. In an implementation, tuples for a given window may be emitted incrementally.

We use delimiters so that empty windows can be explicitly signaled and also to avoid issues synchronizing time between parallel operators that would otherwise occur due to operators being on different physical machines each with their own local clock.

3.2.3 Window

A window is a bounded, contiguous set of tuples from a stream, $W \subset S$. Following Botan there are generally two ways in which this window can be defined:

Tuple-based: $|W|$ is bounded and e.g. each time a new tuple arrives it is added to W . If $|W|$ exceeds the bound then the oldest tuple from W is removed. E.g. the window records the last 1000 tuples.

Time-based: W has a fixed window time interval, τ , and a slide increment α . Each tuple that arrives is put into W . Every α time units the tuples that occurred in the oldest α time interval within W are removed. This approach does not bound the size of W . Also the time domain may be application time or system time. Application time domain uses values of t^{app} to control the window, whereas system time uses the local system clock. The former is technically harder to implement exactly since it relies on synchronizing time between t^{app} and the local system clock, in particular in order to slide the window at precisely the correct time, whether tuples occur or not. The later is quite easy to implement but results are not reproducible since the exact tuples that fall into a given window depend on the entire state of the system at the time.

It is also understood that the signal (or “tick” as called by Botan), for updating the window, is important to consider. In the tuple-based window above the example in-

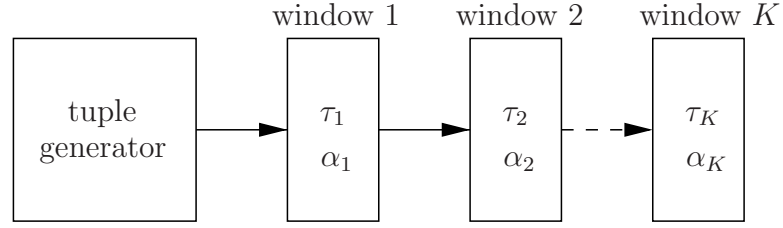


Figure 3.4: Tuple generator followed by K abstract windowing stages in a linear arrangement. Arrows represent any number of tuple streams between the stages. The source of data to the tuple generator is implicit, e.g. it may be a file or an online data source. τ and α are window length and slide length resp.

indicated that the window is updated as (and only as) tuples arrive, meaning *tuple-based signaling*. Whereas in *batch-based signaling*, it would update only after a batch of tuples had arrived and in *time-based signaling* it updates at fixed intervals of time, either in the application domain or the system domain. In our work we make use of all three in that subsequent windowing stages can make use of signaling from previous stages.

3.2.4 Multi-stage hierarchical windowing model

Figure 3.4 shows a tuple generator followed by K abstract windowing stages in a linear arrangement. Any other more complicated arrangement is possible and depends on the application requirements. The important aspect to keep in mind is that a given windowing stage depends on the windows produced by the previous stages that provide input to it. The source of data to the tuple generator is implicit, e.g. it may be a file or an online data source. Its role is to generate streams of tuples.

3.2.4.1 System time and stage 1 windowing:

The tuple generator itself does not use windowing, but it may delimit tuples. Window stage 1 uses system time domain windowing and a local clock to generate periodic signals with period τ_1 and if $\alpha_1 \neq \tau_1$ then a second signal is used as well. E.g. we might set $\tau_1 = \alpha_1 = 60s$, meaning that we are batching data into 1 minute non-overlapping, “tumbling”

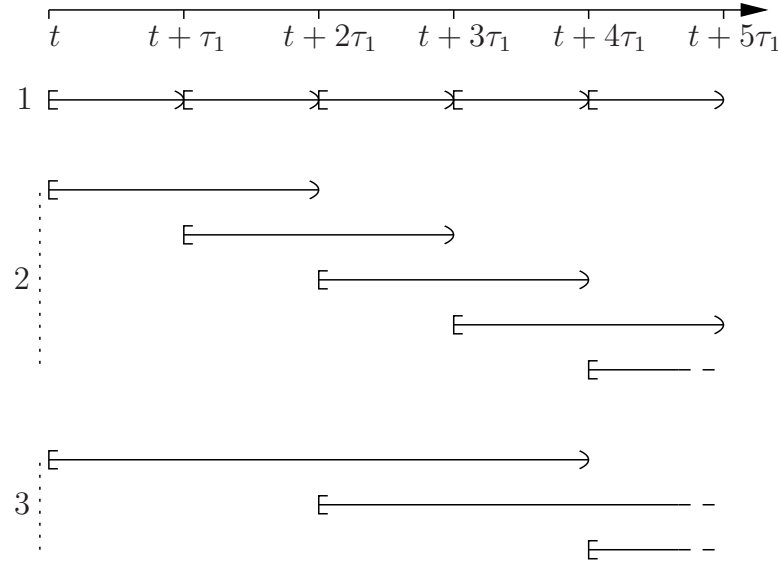


Figure 3.5: Example windowing at three stages. Stage 1 is using $\alpha_1 = \tau_1$ creating mutually exclusive, consecutive windows. Stage 2 is using $\alpha_2 = \tau_1$ and $\tau_2 = 2\tau_1$. Stage 3 is using $\alpha_3 = \tau_2$ and $\tau_3 = 2\tau_2$.

windows.

Figure 3.5 shows an example of three stages with example windows as they slide forward in time. Windowing parameters allow for a wide range of possibilities and the exact choice depends on the application requirements.

3.2.4.2 Operating on windowed data:

So far we have discussed only the notion of windowing data. Now we discuss operators that undertake calculations on the windowed data. The output from a stage is not simply from a unary operator on tuples, but rather more general calculation for a given window at that stage. E.g. if a stage is counting word frequencies, then its input may be individual tuples that represent individual words, and its output – each time the window slides – may be the counts for those words that appeared within the time window. This is a completely different stream of tuples. It may be the count *updates* for any words that have changed count from the previously emitted window batch, as a form of optimization.

Batch Processing. In a batch processing implementation, when a stage has received all of the batches that make up its current window then it processes the current windowed data, outputs the results and slides the window. While in theory this is ideal, in practice processing and transmission takes a non-negligible amount of time. One possible implementation is to clone the current window and process that while receiving data for the new window. If the new window becomes complete before the cloned window processing has completed then either the new window is dropped (which degrades results) or the new window is cloned into a pool of windows awaiting processing, which is both memory and CPU constrained.

Incremental processing. Incremental processing assumes that $\alpha < \tau$, in that data for the current window comes in smaller batches. This allows the data to be applied to the current result for that window as the data arrives. In any case, when the window is ready to slide, the current window, for which still some outstanding processing may be underway with regard to the last received batch, must be cloned to allow the processing to complete.

3.2.5 Parallel operators

In some cases an operator are parallelized using multiple operator instances. Typically the operator instances will reside on different machines. E.g. consider the case where the tuple generator is outputting words to a group of counting operators, using a *group* pattern, i.e. where each word is hashed to a given operator. Each of the operator instance maintains a window within which the word counts pertain. In a last stage, these windowed counts are emitted to an aggregation operator that outputs the top 10 most frequent words within a windowed period. The expectation is for the calculation to be identical to a non-parallel version. However the operator instances are asynchronous, with their own local clocks, and the windowed results that they emit are not necessarily synchronized. In our model the stream delimiters can be used to assist. The aggregation operator can maintain a delimiter counter for each incoming parallel stream and ensure

that each of the windows is accounted for.

The next section presents the intra-operator communication model that can be used to represent iterative computations.

3.3 Intra-Operator Communication Model

The aim of our model is to provide an efficient method for streaming iterative computations using intra-operator communication. In this section, we discuss the main components of the model, how it guarantees the correctness of the results and the API which can be used to utilize the model.

3.3.1 Operator instance

In a stream processing system, data goes through a DAG of operators. In order to handle the large volumes of data, an operator has multiple operator instances and the input is partitioned among the operator instances. An operator instance, OP_{id} has an unique id to differentiate it from the other instances that run in parallel. If an operator has x instances then the ids of the instances will be in the range of $[1 \dots x]$. Hence the n^{th} operator instance of OP is represented by OP_n .

When performing iterative algorithms using a single operator, there has to be an intra-operator communication to pass the result of iterations to the same operator as shown in operator B of Figure 3.6. Although it seems like a simple loop from a high-level perspective, having this type of a data flow in a distributed application can lead to having multiple data flow paths. Internally operator will have multiple instances, therefore, per iteration data should be able to pass from any operator instance to another. Figure 3.7 shows all the possible data flow paths for an operator with three instances. As the number of instances increases, the number of possible data flow paths increases.

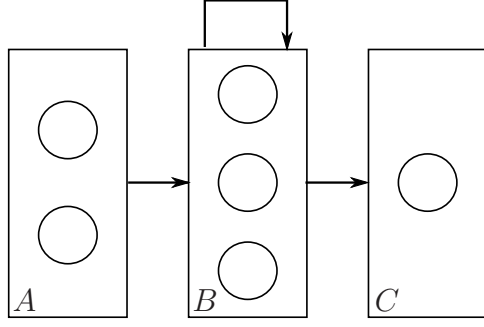


Figure 3.6: High level data flow of an iterative computation.

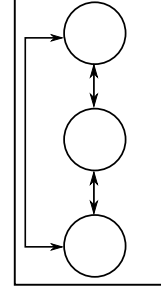


Figure 3.7: Internal data flow paths of operator instances.

3.3.2 Communication

When multiple operator instances perform a task in parallel, data is communicated with each other in different ways. These communications can be redistributions or aggregations [154]. Broadcast and scatter are examples of redistribution where data is passed between operator instances. Reduce and all reduce are examples of aggregation where the output of operator instances is combined by a reduce task. We introduce three types of communication methods to facilitate redistributions and aggregations.

One to one: The format of this communication can be of two ways,

$\langle OP_{src}, OP_{dest}, message \rangle$ or $\langle OP_{src}, dest, message \rangle$ where OP_{src}, OP_{dest} represent the id of source and destination operator instances and $dest$ represents an identification of a component that resides in one of the operator instances, e.g. vertex id. In the first format, OP_{src} instance directly sends a message to the destination instance, OP_{dest} . In the next format, OP_{dest} is determined by performing a hash function on the $dest$. This format is needed when performing graph computations. For instance, when a vertex needs to send a message to another vertex, OP_{dest} is determined by the hash of destination vertex id.

One to many: In this method there is a set of destinations instead of one. The format of the method is $\langle OP_{src}, \{OP_{dest_1}, \dots, OP_{dest_n}\}, message \rangle$ or $\langle OP_{src}, \{dest_1, \dots, dest_n\}, message \rangle$.

Broadcast: Communication format of this method contains two elements, $\langle OP_{src}, message \rangle$. Here, the message is passed to all the operator instances.

3.3.3 Dependency graph

Algorithms have different data flow paths based on their computational tasks. While some algorithms require an operator instance to pass data to all the other instances and wait for data from other instances to continue the computations, some may require an operator instance to pass data to a specific operator instance and start computations as soon as it receives an input. These different data flow paths can be represented as a tree to easily observe the dependencies between operator instances. Using the dependency graph we can observe how the intermediate data is passed between operator instances, whether the synchronization should happen globally or locally.

Figure 3.8a shows an example of three operator instance dependency graph of the first two iterations of a vertex-centric algorithm assuming that vertices reside in any of the instances have neighbors in both other instances. Dashed arrows indicate that the instances use the local results of their previous iteration for the computations of the next iteration. In iteration 2 of the figure, OP_1 has 2 incoming edges from OP_2 and OP_3 and has 2 outgoing edges to OP_2 and OP_3 . This shows that OP_1 has to initially wait for input from both other instances to output results of the next iteration. Figure 3.8b shows another example where only some instances pass messages to others. OP_2 needs input from OP_1 and OP_2 to start iteration 2 while OP_3 does not have an input for iteration 2.

A node in the dependency graph is represented by 4 elements, $\langle OP_n, i, S, D \rangle$. OP_n represents the instance id, i represents the iteration number, S represents a set of instance ids that send input for the iteration and D represents a set of instance ids to which the output of the iteration should be sent to. The nodes of graph shown in Figure 3.8a are represented as follows: $\langle OP_1, 1, \emptyset, \{2, 3\} \rangle$, $\langle OP_1, 2, \{2, 3\}, \{2, 3\} \rangle$, $\langle OP_2, 1, \emptyset, \{1, 3\} \rangle$, $\langle OP_2, 2, \{1, 3\}, \{1, 3\} \rangle$, $\langle OP_3, 1, \emptyset, \{1, 2\} \rangle$, $\langle OP_3, 2, \{1, 2\}, \{1, 2\} \rangle$. To complete a computational task, an instance needs to have access only to the dependency graph nodes relevant to itself.

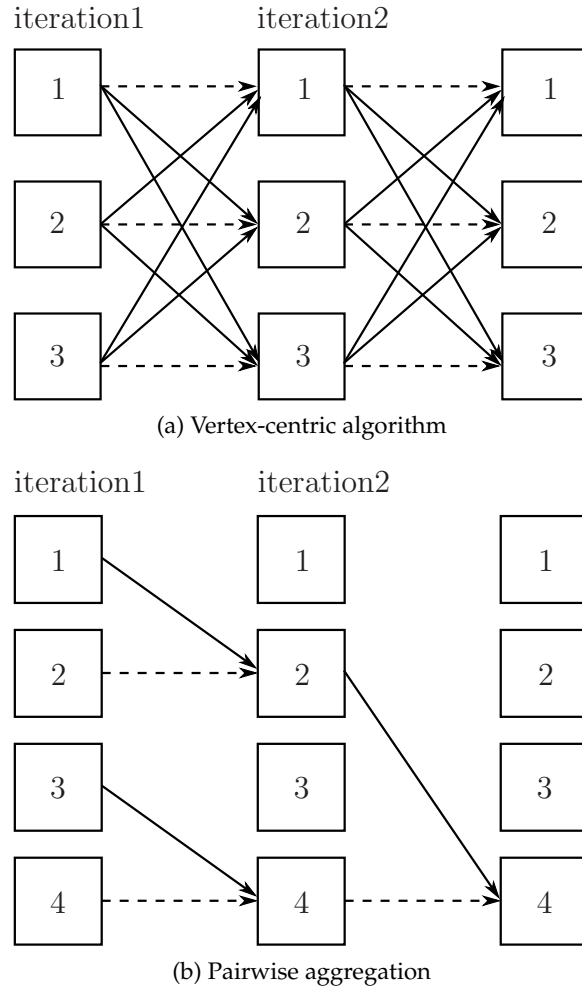


Figure 3.8: Dependency Graphs.

3.3.4 Out of order event handling and synchronization

Based on the dependency graph, there can be cases where input arrives in out of order. Hence, all the operator instances keep a local buffer to keep out of order input. In Figure 3.8b, if OP_2 finishes iteration 2 before OP_3 and OP_4 finish iteration 1, then OP_4 will receive input for iteration 3 before other inputs. Since OP_4 knows that it should receive an input from OP_2 for iteration 3 based on the dependency graph, it simply buffers the input for later use. As soon as an instance finishes an iteration, it checks the buffer for input related to the next iteration.

```

receiveInput(input) {
    if(currentIteration == input.getIteration()){
        inputsReceived++
        inputList.add(input)
        if(inputsReceived == expectedInputCount){
            performIteration(currentIteration, inputList)
        }
    }else {
        buffer(input)
    }

    if(input.acknowledgeRequired()){
        sendAck(myInstanceID, input)
    }
}

```

Figure 3.9: Operator instance handling inputs for iterative computations.

3.3.4.1 Synchronization

Synchronization between operator instances is critical to guarantee the correctness of the results. Different algorithms require different types of synchronization and we can achieve synchronization requirement using only the information in the dependency graph such as the number of inputs and the input sources for each iteration. Figure 3.9 demonstrates how an operator instance handles the input, showing that a new iteration is only performed when all the required inputs are received.

When we consider vertex-centric graph algorithms, they have a synchronization barrier to make sure a new iteration starts only after all the instances finish the previous iteration. This type of algorithms require global synchronization. Global synchronization can be achieved by operator instances broadcasting the iteration end to other instances.

Some algorithms can be executed with local synchronization where an instance can start its computations as soon as it receives the input without waiting for other instances to finish their iterations. Moreover, some algorithms may require an acknowledgement from the message receiver to resume the computations of the message sender. All these different requirements can be achieved by the information available to each operator instance as indicated in Figure 3.9.

These different cases can be further explained using Figure 3.10. Figure 3.10 shows

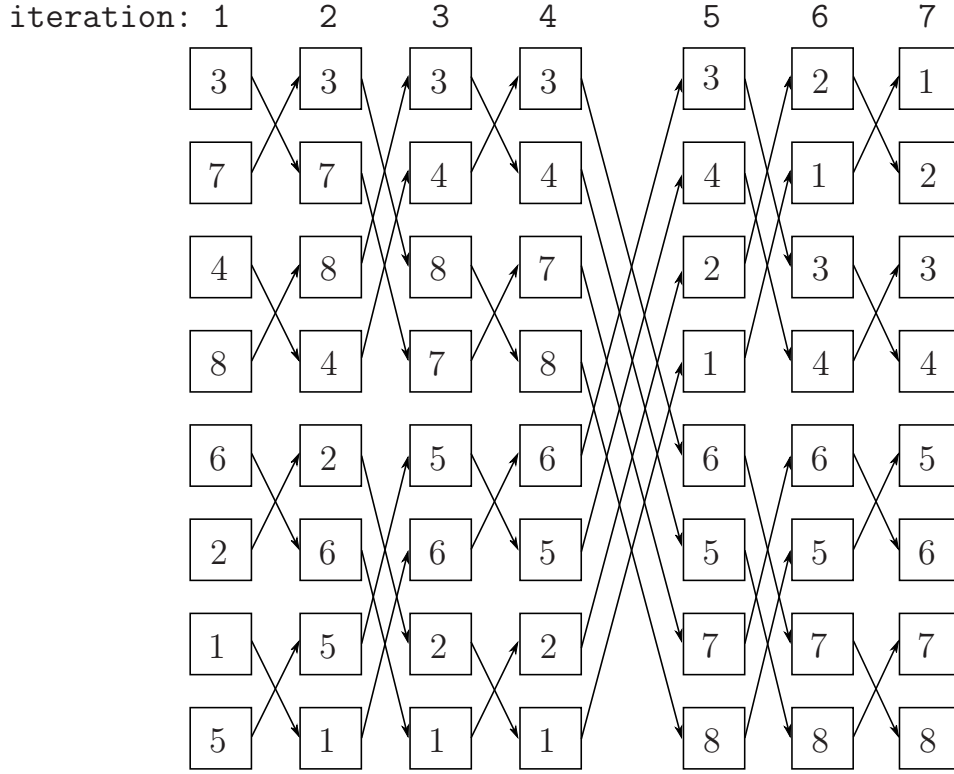


Figure 3.10: Bitonic Mergesort communication pattern.

the communication paths of bitonic mergesort when there are eight operator instances. Consider the third iteration of OP_1 , OP_2 , OP_3 and OP_4 , these four instances can start the third iteration if they have completed the second without considering the other instances. For example, when the first four instances start the third iteration, OP_5 , OP_6 , OP_7 and OP_8 could still be executing the first iteration. Out of order messages can also occur in this communication pattern. For instance, consider a case where OP_1 finishes the 4th iteration and passes the output to OP_5 and OP_5 is still performing computations of iteration 1. Then OP_5 has to buffer the message sent by OP_1 till it finishes iteration 1,2,3 and 4.

3.3.5 Apache Storm based API for application development

We implemented the proposed model using a well-known stream processing engine, Apache Storm and we provide an API on top of Storm which can be used to model parallel iterative algorithms. In stream processing systems like Storm, data comes to the system continuously and goes through a set of well-defined operators to perform com-

putations. An operator can perform tasks like filtering, windowing and aggregation. An operator can have multiple instances so that instances can run in parallel by dividing the input of the operator among the set of its instances. A typical Storm application (topology) contains a fixed number of operators, which are referred to as spouts and bolts. The data processed by a topology comes as a stream which is an unbounded sequence of tuples. In a topology, spouts are the data sources which generate continuous input streams and bolts perform the processing of the data and output the results as a stream. Spouts and bolts are interconnected by streams such that spouts emit streams and bolts emit as well as receive streams.

We provide an abstract bolt called `ParallelBolt` which provides a set of methods listed in Table 3.1 to define the computational task and to support intra-operator communication between bolt instances. The first two methods can be called from the bolt that extends the `ParallelBolt` and the last two methods have to be implemented by that bolt.

<code>broadcast(message)</code>	broadcasts a message to other bolt instances.
<code>send(message, destination)</code>	sends a message to the specified destination.
<code>receiveBroadcast(message, source)</code>	invoked by the receiving instances at the arrival of broadcast message.
<code>receive(message, source)</code>	invoked by the destination instance at the arrival of send message.

Table 3.1: Methods supported by abstract Storm bolt.

The `ParallelBolt` extends Storm `BaseRichBolt` where the `execute` method is invoked for every input tuple. In the `ParallelBolt`, `execute` method checks the stream id of the input tuple and if it is broadcast stream the `ParallelBolt` calls the `receiveBroadcast` method, if it is send stream the `receive` method is called. For other tuples which are not related to intra-operator communication, we call `executeTuple` method. For instance, in our implementation at the arrival of a new window, `executeTuple` calls a method called `compute` to start the computation.

Figures 3.11 and 3.12 show example usages of the API. Figure 3.11 shows one way of performing top- k in parallel. In this example, all the instances calculate local top- k and then send the local top- k values to the last instance to merge and get the final result. In

```

compute() {
    topk = calculate local top-k
    if(myInstanceId != n) {
        send(topk, n)
    }
}
receive(message, source) {
    receivedCount = receivedCount + 1
    receivedTopk = message
    topk = combine(topk, receivedTopK)
    if(receivedCount == n-1) {
        result = topk
    }
}

```

Figure 3.11: API usage: top- k implementation. n = total instances.

```

compute() {
    performIteration(1)
    broadcast(end, myinstanceId)
}
receiveBroadcast(message, source) {
    receivedCount = receivedCount + 1
    if(receivedCount == n-1) {
        performIteration(message.iteration+1)
    }
}

```

Figure 3.12: API usage: global barrier implementation. n = total instances.

the compute method, the instances calculate local top- k and all the instances other than the last instance send the calculated value to the last instance. The `receive` method is invoked only in the last instance because no messages are sent to other instances using the `send` method. Hence, in the `receive` method the received top- k values are merged to get the final result.

Figure 3.12 shows how global synchronization is modeled using the API. The `compute` method performs the first iteration and sends an end of iteration message to other instances using `broadcast`. The `receiveBroadcast` method starts the next iteration when it receives the end of iteration message from all the other instances. For simplicity, this example assumes that all the end of iteration messages of an iteration x are received before the end of iteration messages of iteration $x + 1$, which is not the case with

real-world applications. Buffering is used to handle these cases in the actual implementation. In addition, replication based state management can be used to achieve fault tolerance [134].

3.4 Parallel Processing Applications

In this section, using well-known examples, we demonstrate how our model can be used to support parallel algorithms and how synchronization is achieved.

3.4.1 Parallel merging

Most of the parallel algorithms have a reduce/merge step often at the end of the computation. If we consider top- k calculation, it can be performed by having multiple tasks calculating the top- k based on their local data and sending the results to a reduce task to merge and output the final top- k . In a pure DAG based environment, this is done by having one operator with multiple instances to perform the initial top- k and another operator with a single instance merging the calculated top- k values as shown in Figure 3.13a. If there are n map tasks in the trivial implementation, top- k merge operator will receive $\mathcal{O}(n)$ messages and will take $\mathcal{O}(n)$ operations to get the final result. Using the binary tree based method[155] we can have multiple instances contributing to the final merge operation. This approach partitions the merge workload among the operator instances. Further, this method reduces the number of messages received by each instance and each instance will receive at most $\mathcal{O}(\log_2(n))$ messages. This can be implemented in a pure DAG as shown in Figure 3.13b. This approach needs a new operator for each merge stage. If the number of initial top- k calculation operator instances are n , then the number of merge stage required would be $\lceil \log_2(n) \rceil$. Therefore, DAG has to be modified every time the n changes.

Figure 3.13c depicts how top- k can be implemented using our model. This approach only requires a single operator and the operator instances communicate with each other internally to perform the different merge stages. In contrast to the model in Figure 3.13b, users do not have to add/remove operators when the number of operator instances

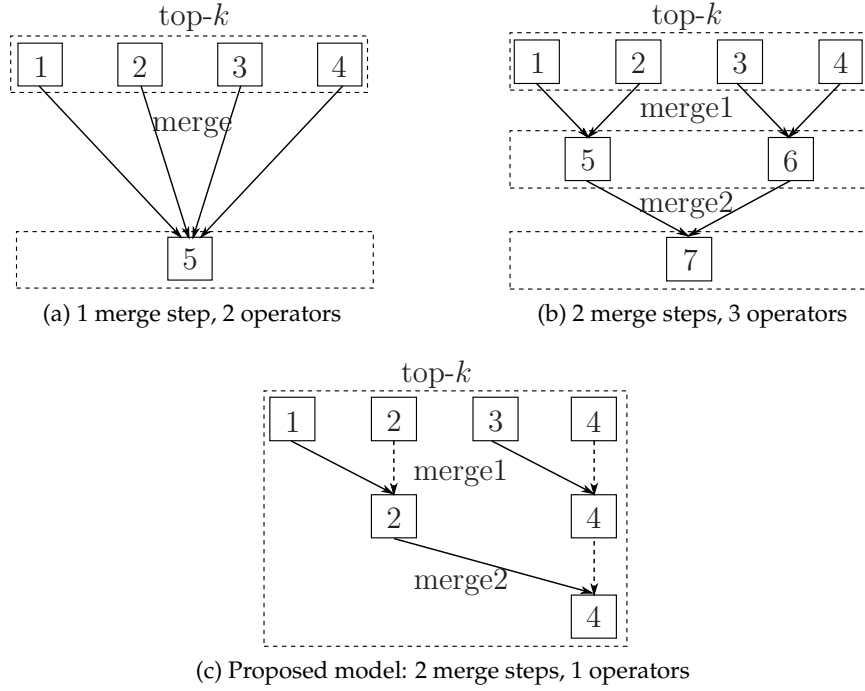


Figure 3.13: Parallel top- k computation approaches. Operators are shown using dashed lines and operator instances are shown using solid lines.

changes. Further, our model only require 1 operator for the computations while the models represented in Figures 3.13a and 3.13b require 2, 3 operators respectively. For this type of computations, global synchronization is unnecessary. Hence, we use local synchronization at ancestor level when performing the computations. This indicates that an instance can perform an operation only after all the ancestor operations are finished in the dependency graph. If we consider stage 2 merge operation of instance 4 in Figure 3.13c, its ancestor operations are stage 1 merge of instances 2 and 4.

3.4.2 Iterative parallel processing

Graph algorithms: A well-known class of iterative algorithms is graph algorithms. The vertex-centric model is adapted by many graph processing systems to support parallel graph processing. Vertex-centric algorithms require a synchronization barrier at every iteration to guarantee that a new iteration starts only after all the vertices finish the previous iteration. This algorithm category can be easily implemented by our model us-

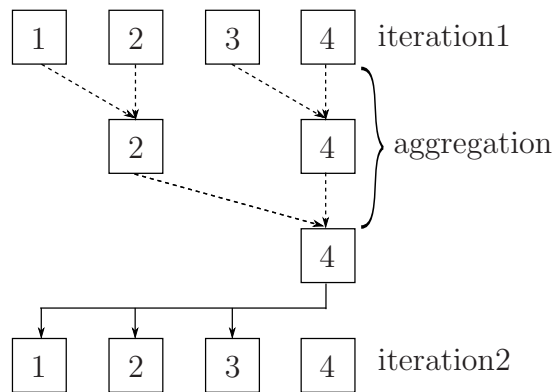


Figure 3.14: Achieving synchronization of k -means clustering.

ing broadcast communication to facilitate the global barrier. When an operator instance finishes an iteration it will broadcast an end of the iteration message to all the other instances. An instance will start a new iteration as soon as it receives the end of the iteration messages from all the other operator instances.

K-means clustering: K -means clustering represents the type of parallel iterative algorithms that have a merge stage at the end of each iteration. Unlike vertex-centric algorithms, k -means requires aggregating the results of all the operator instances to start the next iteration. Therefore, the end of iteration message needs to be sent by the instance that performs the final aggregation despite how the aggregation is done. Aggregation can be done by all the instance sending their output to one instance. However, we can use the more efficient binary tree based method described in section 3.4.1, where the instances communicate in a pairwise manner to perform the aggregation. Aggregation message path is shown in dashed arrows in Figure 3.14. As indicated by the solid arrows, the final aggregated result is calculated by the last operator instance which sends the end of iteration message to notify the start of the next iteration.

Bitonic mergesort: Bitonic mergesort is another type of parallel iterative sorting algorithm that can be modeled using a local synchronization barrier. The difference between bitonic mergesort compared to typical parallel iterative algorithms is that the communication patterns between operator instances change with every iteration. At every iteration, pairs of instances communicate with each other to compare and sort the local data. Since the communication happens in a pairwise manner, as soon as an instance sends the

data to its paired instance, receives data back from the paired instance and processes it, it can start the next iteration.

For all the above algorithms, if an instance knows who it is paired with at every iteration, it can carry on the computations without any other information. For example, when an instance finishes an iteration and if it knows the instance id that will send data for the next iteration. The instance will wait for data from that particular instance. However, if the instance receives data required for a future iteration, it will buffer the data to be processed at the correct iteration. For most algorithms, buffering needs to be done for a comparatively low number of future iterations. For instance, in vertex-centric algorithms an instance would only receive the messages for the immediate next iteration before the messages of the current iteration. Therefore, the buffer does not incur performance overhead. Nevertheless, for occasional cases where an instance has to buffer a large number of messages and the memory constraints the size of the buffer, we can flush the buffer content to a disk and read back the content at the required time.

3.5 Implementation

This section explains how the proposed intra-operator communication model is implemented in Storm using graph processing as an example and describes the alternative approaches used to compare the proposed model.

3.5.1 Graph representation

A Graph $G = (V, E)$ consists of vertices, V and edges, E . An edge, $E = (u, v)$ is represented by two vertices, where u represents the source vertex, v represents the destination vertex. The weight of an edge (u, v) , is represented by w_{uv} . In our model, all edges are represented as directed edges. An undirected edge is handled by converting it to two directed edges. For example, an undirected edge from vertex u to v is represented by two edges $e_1 = (u, v)$, $e_2 = (v, u)$. In our model, a graph is represented by a continuous set of edge insertions, deletions and updates, which we refer to as the graph stream, $\mathbb{G}$. A tuple, t in the graph stream represents an edge of a graph: $t = (u, v, w_{uv})$. Tuple, $t = (u, v, 2)$

represents a directed edge from vertex u to vertex v of weight 2. Further, an edge deletion is represented by a 0 edge weight.

3.5.2 Distributed graph processing

We used a vertex based partitioning mechanism to distribute the graph information among the graph computation bolt instances. Vertices of the graph are distributed among the bolt instances in a way such that a single vertex resides only in one bolt instance. We use a hash based method to decide to which instance a particular edge should be sent to. We use the hash based partitioning method because it doesn't require the model to maintain a global table, mapping the vertex with the instance id it resides in. For a large-scale graph having such mapping will require additional memory and accessing a global table may require additional communication over the network which is time consuming.

Partition set, $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n\}$ breaks the graph, $G = (V, E)$ into n partitions. A partition $\mathcal{P}_i$ is a subset of vertices and edges of the graph such that $\forall_{i \in \{1, 2, \dots, n\}} \mathcal{P}_i = \{V_i, E_i, D_i\}$ where $V_i \subseteq V$ and E_i and D_i are outgoing edges of V_i . For simplicity, we write V_i to mean $\mathcal{P}_i(V)$ and similarly for E_i and D_i .

The edge set, E_i is the completely contained edge set in the partition $\mathcal{P}_i$ so that source and destination vertices of all the edges in E_i are in V_i ,

$$\forall uv \in E_i \Rightarrow u, v \in V_i.$$

The edge set, D_i is the partially contained edge set in the partition $\mathcal{P}_i$ so that the source vertex of all the edges in E_i are in V_i and the destination vertices are in a different partition,

$$\forall uv \in D_i \Rightarrow (u \in V_i \wedge v \notin V_i).$$

Partitions divide the graph vertices and edges into n disjoint sets,

$$\bigcup_{i=1}^n V_i = V \text{ and } \forall_{i, j \in \{1, 2, \dots, n\}} \substack{i \neq j} V_i \cap V_j = \emptyset.$$

Since partition, $\mathcal{P}_i$ only keeps the outgoing edges of vertices in V_i , edges are not dupli-

vertex	dest	w	vertex	dest	w	vertex	dest	w
q	u	2	v	x	1	y	v	1
	v	3	w	x	3		x	2
	w	4	x	v	2		z	1
u	v	1				z	v	3
	w	1						
(a) Instance 1			(b) Instance 2			(c) Instance 3		

Table 3.2: Partitioning tables of the graph shown in Figure 3.15.

cated among partitions:

$$\bigcup_{i=1}^n (E_i \cup D_i) = E,$$

and

$$\forall_{i,j \in \{1,2,\dots,n\} \atop i \neq j} ((E_i \cup D_i) \cap (E_j \cup D_j)) = \emptyset.$$

In this approach, each bolt instance stores details of a set of vertices along with the vertex information. An instance of the bolt has an in-memory data structure called a neighborhood mapping table which stores the list of vertices and the information of its outgoing edges.

Figure 3.15 represents a graph that is generated for the tuples: $(q, u, 2)$, $(q, v, 3)$, $(q, w, 4)$, $(u, v, 1)$, $(u, w, 1)$, $(v, x, 1)$, $(x, v, 2)$, $(w, x, 3)$, $(y, v, 1)$, $(y, x, 2)$, $(y, z, 1)$, $(z, v, 3)$. It shows how the graph is distributed among three bolt instances assuming that parallelism of the bolt is 3. We can represent the three partitions as follows: $\mathcal{P}_1 = \{V_1, E_1, D_1\}$ in instance 1 where $V_1 = \{q, u\}$, $E_1 = \{(q, u)\}$, $D_1 = \{(q, w), (q, v), (u, v), (u, w)\}$, $\mathcal{P}_2 = \{V_2, E_2, D_2\}$ in instance 2 where $V_2 = \{v, w, x\}$, $E_2 = \{(w, x), (v, x), (x, v)\}$, $D_2 = \emptyset$ and $\mathcal{P}_3 = \{V_3, E_3, D_3\}$ in instance 3 where $V_3 = \{y, z\}$, $E_3 = \{(y, z)\}$, $D_3 = \{(z, v), (y, v), (y, x)\}$. Neighborhood mapping tables of instance 1, 2 and 3 are shown in Table 3.2, where w represents the edge weight. Edge details are not duplicated over multiple instances as each vertex stores only its outgoing edges in the neighborhood mapping table.

The graph stream tuples represent edge additions, modifications (edge weight change) and deletions. For edge additions, a new entry is added to the neighborhood mapping table. For modification, we change the edge weight of the table and for deletion, we

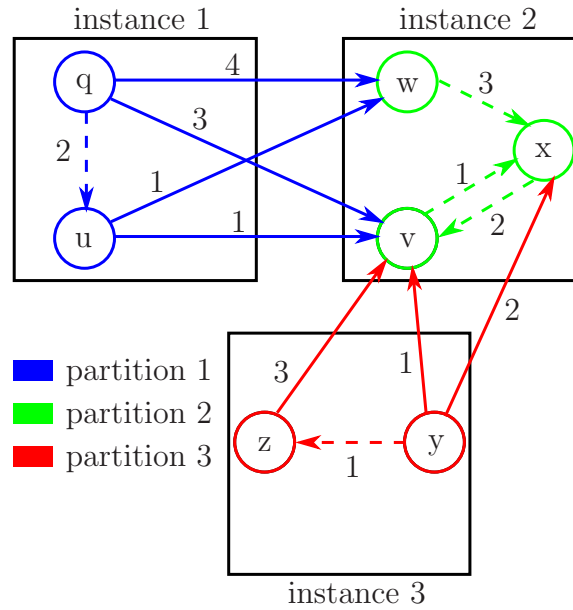


Figure 3.15: A directed graph distributed among 3 operator instances. Colors represent different partitions. Dashed lines represent internal message passing and solid lines represent external message passing.

remove the entry from the table.

3.5.3 Graph processing implementation methods

Graph algorithms usually consist of a set of iterations. At each iteration, results of the current iteration need to be communicated among graph components to use as input to the next iteration. In some graph algorithms, the number of iterations is not predefined. For instance, a vertex-centric algorithm may finish the computation when there are no more new messages to be sent. If we represent each iteration as a bolt, then we cannot represent graph algorithms where the number of iterations is not known in advance because Storm topologies have a fixed number of stages. One way to handle this is the off-loading approach [21, 73] where the graph computation is delegated to an existing graph processing system which can handle the complexities of graph algorithms.

Another approach is to use a single bolt to perform the complete algorithm. When using a single bolt, there has to be a mechanism to pass intermediate results between vertices per iteration. This is challenging because vertices are partitioned among bolt

instances which can be in different physical machines. Intermediate message passing between bolt instances can be of two ways:

- internal message passing– when sender vertex and receiver vertex are in the same bolt instance.
- external message passing– when sender vertex and receiver vertex are in different bolt instances in different machines.

Figure 3.15 shows how the result of an iteration is passed between vertices that are scattered in a set of bolt instances. In the internal message passing mode, neighboring vertices reside in the same instance. Communication between (q, u) , (w, x) , (v, x) , (x, v) , (y, z) in Figure 3.15 falls into this category. In the external message passing mode, neighbors reside in different instances. Communication between (q, v) , (q, w) , (u, v) , (u, w) , (z, v) , (y, v) , (y, x) falls to this category. Although internal message passing is trivial, external message passing requires additional effort if the bolt instances are in different physical machines. One approach that can be used for external message passing is to have a distributed data-store to communicate results between instances. We refer to this as the distributed memory approach. Another approach is to use intra-operator communication model to pass messages between bolt instances.

3.5.3.1 Off-loading model

The off-loading model, where a stream processor handles the continuous updates and a graph processing system handles the graph computation is used in several existing systems [21, 73]. In our implementation, Storm handles the continuous updates and the well-known distributed graph processing system, Giraph handles the graph processing. Giraph reads the input graph from a storage, usually from Hadoop Distributed File System (HDFS), runs the computation and writes output to the storage. In this model, for every window, Storm creates an input graph based on the neighborhood mapping table, calls Giraph to run the computation and reads back the output for further processing. Figure 3.16 shows the data flow of this model. Continuous graph updates are received by the input writer bolt as the graph stream and then the input writer bolt writes the

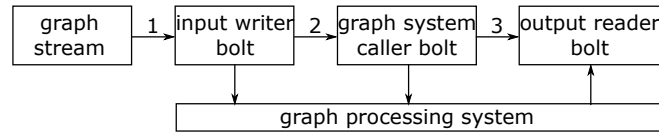


Figure 3.16: Data flow of the off-loading approach.

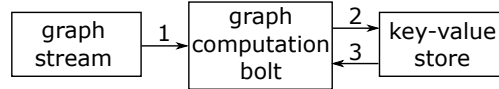


Figure 3.17: Data flow of the distributed memory approach.

input into a format required by graph system and calls the graph system. When graph computation finishes, the output is read and sent to downstream bolts for further processing. In this model, apart from handling continuous updates in real-time, Storm has to convert the input to a format recognized by the graph system and convert back the output to be used for downstream processing. These tasks can take a significant amount of time when processing large-scale graphs. Another drawback of this approach is the requirement of more resources to maintain and monitor two independent systems, Storm and Giraph.

3.5.3.2 Distributed memory model

This model utilizes a distributed memory to support intermediate communication between iterations. We use Redis , an open source in-memory data structure store in our implementation. In this model at each iteration, results are stored in Redis using destination vertex id as the key. At the start of all the iterations (other than the initial iteration), bolt instances read the results of the previous iteration from Redis to perform the next iteration. Figure 3.17 depicts the data flow of this model. Similar to the off-loading model, continuous graph updates are received by the graph computation bolt as the graph stream. At each iteration, bolt instances store the intermediate results in Redis. At the beginning of the next iteration, each instance reads the messages related to the vertices partitioned to the instance from the key-value store and performs the computation. Steps 2 and 3 in Figure 3.17 occur for every iteration.

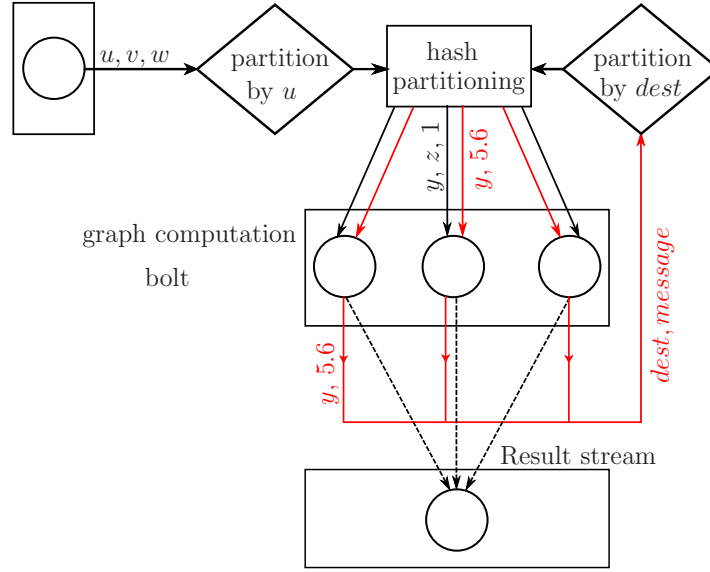


Figure 3.18: Data flow of intra-operator communication model. Boxes represent operators, circles represent operator instances, diamonds represent data stream partitioning and the arrows indicate the direction of the data flow. Black arrows represent the graph stream, the red arrows represent the intermediate result stream and the dashed arrows represent the result stream.

3.5.3.3 Intra-operator communication model

We use the proposed intra-operator communication model to facilitate intermediate result passing between bolt instances. Figure 3.18 illustrates how streams are passed between bolt instances in this approach. The graph stream is distributed among a set of bolt instance, which we refer to as the graph computation bolt. The bolt instances perform computations in parallel and communicate the results of each iteration with each other using the intermediate stream. Once all the iterations are finished, final results are sent to the downstream bolt as the result stream.

Initially, we use a hash-based partitioning on the source vertex of graph stream tuples to distribute the graph stream among graph computation bolt instances. We use the same hash partitioning on *dest* of intermediate result stream tuples when passing messages to the neighbors. *dest* of the intermediate result stream represents the vertex id to which the result should be sent to. Figure 3.18 depicts this scenario. The intra-operator communications paths are depicted in red in the figure.

In vertex-centric graph computations, global synchronization barrier is used to make

sure that a new iteration starts only after all the vertices finish the previous iteration. In the intra-operator communication model, when a bolt instance finishes an iteration, it sends a signal to other instances to indicate the end of the current iteration. When an instance receives the end iteration signal from all the other instances it starts the next iteration.

3.6 Evaluation

In this section, we select a set of iterative algorithms, namely (1) graph algorithms, (2) top- k , (3) k -means clustering, (4) bitonic mergesort, (5) FloydWarshall algorithm and show how we have implemented them using our model. We describe in detail how we support iterative computations for each of these algorithms in the following subsections. All experiments were conducted on m2.small nodes running on OpenStack. Each virtual machine has one CPU core, 4 GB of RAM with 30 GB of disk space and runs Ubuntu 15.10. Java 1.7.0_91, Storm 0.10.2, Redis 3.2.8, Spark 2.1.0, GraphX 2.1.0, Flink 1.3.3, Giraph 1.3.0 and Hadoop 2.7.3 were used for the experiments.

3.6.1 Graph analysis performance

Initially, we evaluated the three approaches we implemented for graph computations: off-loading, distributed memory and inter-operator communication model, using PageRank and Single Source Shortest Path (SSSP) algorithms. We selected these two algorithms due to their different communication and computation behavior. For instance, all the vertices are active in each iteration in PageRank and it is more communication-intensive than SSSP. In SSSP, the number of active vertices increases in early iterations and decreases in the later iterations and the communication patterns between vertices change across iterations. We used a set of real-world graph-structured datasets from the Stanford Large Network Dataset Collection [156] for the experiments. Table 3.3 shows the statistics of the datasets.

Figure 3.19 illustrates the running time of 10 iterations of PageRank and Figure 3.20 illustrates the running time of SSSP. Both figures show the running time for datasets

Table 3.3: Dataset statistics.

Name	Type	Nodes	Edges
LiveJournal	Undirected	3997962	34681189
Orkut	Undirected	3072441	117185083
Youtube	Undirected	1134890	2987624
higgs-twitter	Directed	456626	14855842

LiveJournal, *Orkut*, *Youtube* and *Twitter* respectively for different degrees of parallelism, 25, 50 75 and 100. The running time of the off-loading approach consists of the time taken to write the input graph into HDFS, run the Giraph job with checkpointing turned off and read the output from HDFS. For the distributed memory and the intra-operator communication approaches, the running time consists of the time to create the graph snapshot and perform the computation.

As shown in the figures, the performance of SSSP is better compared to PageRank due to the less communication-intensive nature of SSSP. The intra-operator communication and the off-loading approaches perform significantly better than the distributed memory approach. However, there is a notable decrease in the running time of the distributed memory approach with the increase in the degree of parallelism. The intra-operator communication and the off-loading approaches have similar running times for PageRank for the LiveJournal and the Orkut datasets. However, as the degree of parallelism increases, intra-operator approach starts performing better and the difference between the two approaches becomes more significant. For example, the observed improvement in running time of the intra-operator model is 65% over the off-loading model for the Orkut dataset for parallelism of 100. However, the intra-operator communication approach performs better in all the cases for the less communication-intensive SSSP algorithm.

Overall, the results of both algorithms indicate the intra-operator communication approach scales well compared to the off-loading approach. As the parallelism increases, communication cost increases and computation cost reduces. Based on the results of the intra-operator communication approach in Figures 3.19 and 3.20, we see that the running time decreases with the degree of parallelism, indicating the reduction of computation

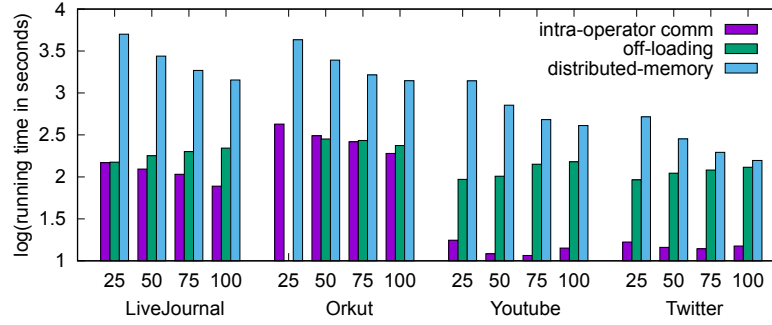


Figure 3.19: PageRank computation time of the three models.

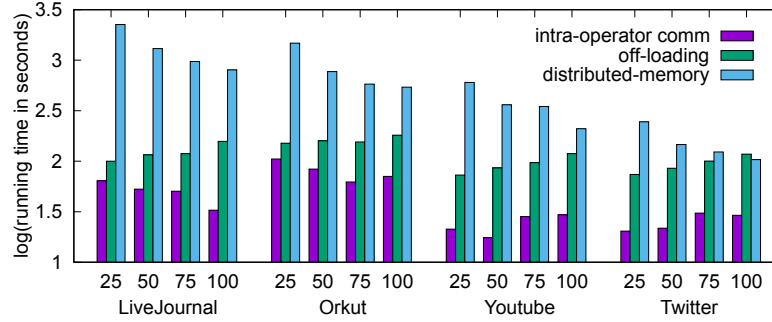


Figure 3.20: SSSP computation time of the three models.

cost is more significant than the increase in communication cost. This trend is changed if we increase the degree of parallelism to a point where communication overhead diminishes the computation time reduction. This can be seen in the Twitter dataset running times. As indicated in the PageRank results of the Orkut dataset which is the largest dataset we used, we were unable to run the Giraph job using 25 nodes due to memory limitations. This is one of the disadvantages when using multiple systems to perform the computations. In this case, resources are divided between two independent systems (Storm and Giraph), and Giraph does not have sufficient resources to complete the job.

Figure 3.21 shows the corresponding speedups of PageRank and SSSP running times shown in Figures 3.19 and 3.20. Speedups are calculated for degree of parallelism 50, 75 and 100 using the running time of parallelism 25 as the baseline, except for the PageRank speedups of Giraph for the Orkut dataset. For the Orkut dataset, PageRank speedup of the off-loading model is calculated using the running time of parallelism 50 as the baseline. For the two largest datasets, Orkut and LiveJournal, the speedup increases with the increase of the degree of parallelism for both intra-operator communication and dis-

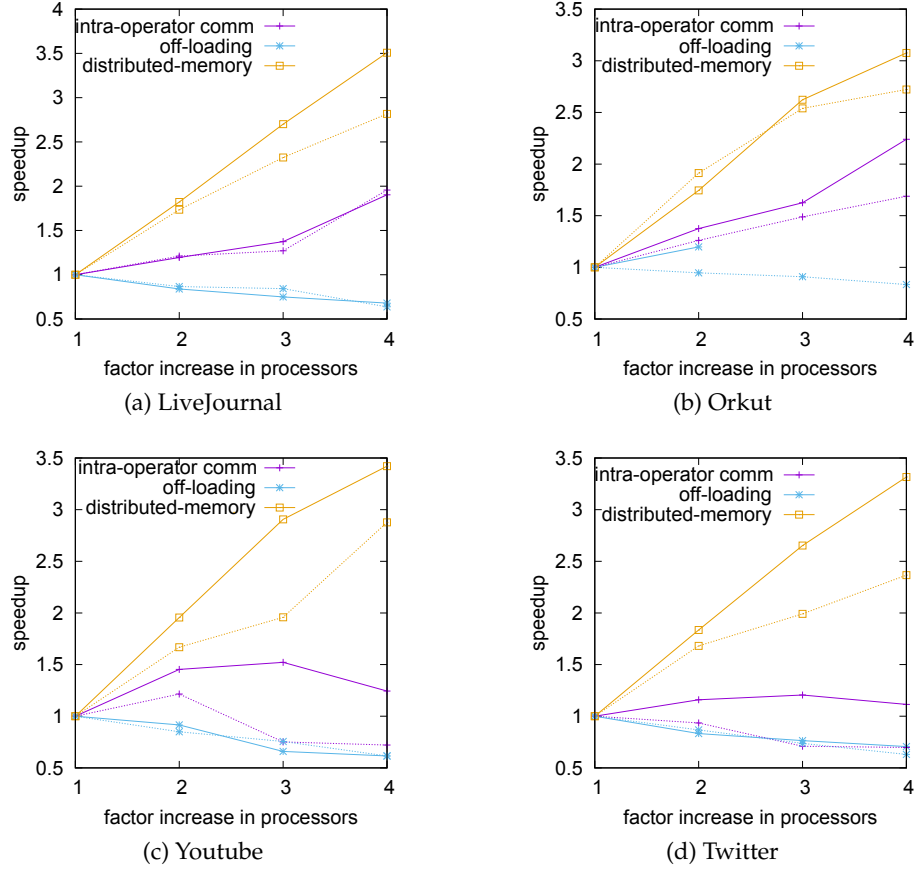


Figure 3.21: PageRank and SSSP computation speedup of the three models. Solid lines indicate PageRank speedup and dotted lines indicate SSSP speedup.

tributed memory approach. However, for the smaller datasets Youtube and Twitter, the speedup of intra-operator communication model decreases when the factor increase in processors is 4 (at parallelism of 100). A decrease in speedup with the factor increase is visible in the off-loading model as well. This is because the communication overhead is more significant than the improvement of the computation cost due to increased parallelism.

3.6.1.1 Streaming performance

Since the intra-operator communication and the off-loading models are more efficient compared to the distributed memory model, we further analyzed the two models by generating continuous graph updates and performing periodic computations on a dy-

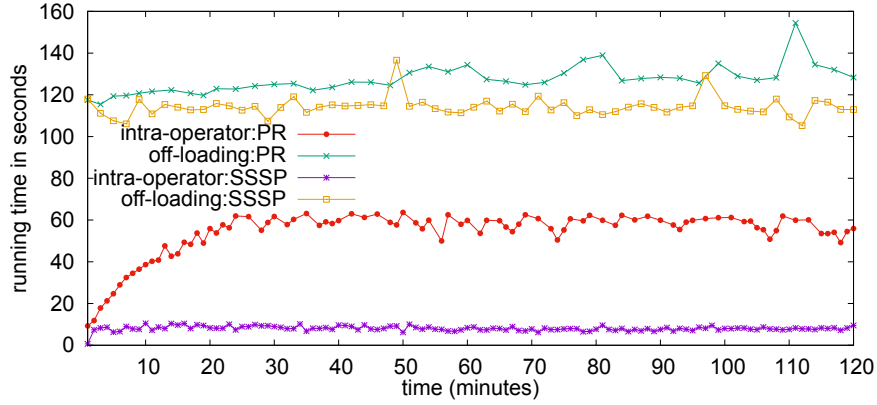


Figure 3.22: PageRank and SSSP computation time over 2 hour period for 1 minute window slide.

dynamic graph. We simulated a stream of graph updates by continuously selecting edges from a dataset and adding and deleting the selected edges. If the selected edge is already included in the current graph snapshot, we send a graph update to delete the edge else we send an edge addition update. We used the LiveJournal dataset for the experiments and we used PageRank and SSSP algorithms for the evaluation.

Figure 3.22 shows the running time of PageRank and SSSP for both intra-operator communication model and off-loading using the LiveJournal dataset and a parallelism of 50. Figure 3.23 shows the corresponding number of edges per each graph snapshot. We calculated the running times for snapshots created at every 1 minute over a 2 hour period. As shown in the Figure 3.22, intra-operator communication based model performs significantly better than off-loading throughout the 2-hours for every snapshot calculation. The running time of SSSP of intra-operator communication model is comparatively low because the number of messages passed per iteration is smaller compared to the PageRank. After several minutes, the number of edges per snapshot stabilizes and the number varies around 34.68 million. As the dynamic graph size stabilizes, the variation of the snapshot running time stabilizes as well.

In a dynamic graph processing system, computations occur while the continuous updates come to the system. We evaluated how the updates are handled by the three models by calculating the throughput. Throughput is calculated by the number of graph updates handled by the system per second. Figure 3.24 shows the average throughput of

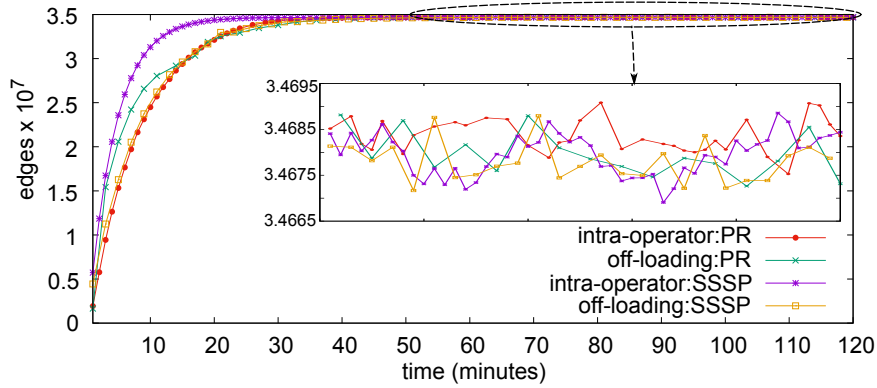


Figure 3.23: Dynamic graph edge distribution over 2 hour period for 1 minute window slide. Number of edges in last several minutes are zoomed to show the dynamic nature of the graph.

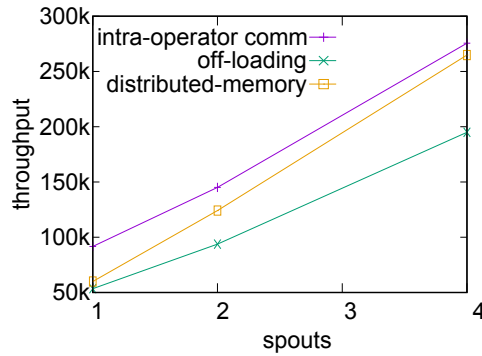


Figure 3.24: Throughput comparison of the three models.

the three models when updates are emitted by a different number of Storm spouts for calculating PageRank for 1-minute window slide using parallelism of 25. As indicated, the throughput increases with the number of spouts. The off-loading model has the lowest throughput because more resources are required to convert and write the Storm tuples to a format recognized by the third party graph processing system which reduces the update emission rate.

3.6.1.2 Communication optimizations

One of the main performance concerns of the intra-operator communication model is the communication overhead. Unlike other two approaches, in intra-operator communication model, a large number of tuples are emitted and received by the bolt instances in

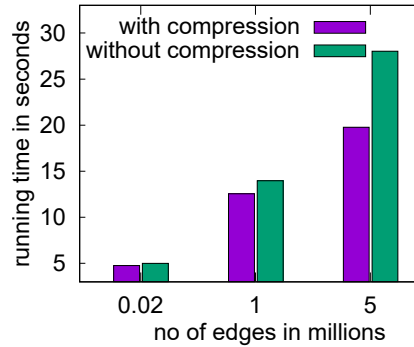


Figure 3.25: Performance comparison with and without message compression.

every iteration to communicate intermediate results. As the number of edges increases the number of messages passed over the network increases. In addition, if we distribute the same graph over more instances, the number of messages passed over the network may increase. Hence, the number of messages passed over the network depends on the graph size as well as the number of bolt instances. Using combiners suggested in Pregel is one way of reducing the number of messages. Combiners aggregate the messages which are sent to the same destination vertex before sending it to a remote worker over the network. In addition to combiners, we compress the messages before sending them. We observed that the message compression significantly improves the performance and the improvements become more evident as the graph size increases.

Figure 3.25 shows the running time of 10 PageRank iterations for different numbers of edges for a parallelism of 100. As indicated in the figure, we observed significant performance improvement for all the experimented scenarios. Moreover, the performance improvement becomes more significant as the graph size increases. For instance, for 0.02 million edges, the performance improvement is around 5% and for 5 million edges, it is nearly 30%.

3.6.1.3 Comparing with existing systems

To provide an additional perspective on Storm-based graph processing, we compared it with two well-known large-scale data processing systems, Flink and Spark's graph API, GraphX. We compared these two systems with the intra-operator communication ap-

proach since it is the most efficient one from the three approaches we investigated. The running times of all three approaches consist of the time taken to create the graph and to complete the algorithm. Figures 3.26 and 3.27 show the running time to complete 10 iterations of PageRank and SSSP for different datasets. We used off-the-shelf settings for Flink and Spark setup and we acknowledge that these settings may not be the most optimized settings for these two systems. As indicated in the figure, for the LiveJournal and the Orkut datasets, all systems scales well with the increase of the degree of parallelism and the intra-operator communication model shows better performance for smaller datasets, Youtube and Twitter. As discussed in section 3.6.1.2, reducing the communication overhead increases the performance of the intra-operator communication model significantly. Therefore, the performance difference between intra-operator communication model and the other two systems is more observable in SSSP because SSSP has less communication overhead than the PageRank algorithm. Our intention is to emphasize that Storm can be used for graph computations and it performs as well as the existing systems. Storm has been used for a wide range of applications and these applications can adopt approaches like intra-operator communication to perform graph computations without having to delegate the graph computations to a third-party system.

3.6.2 Top- k

Parallel top- k consists of 2 stages. Workers calculate their local top- k values in the first stage and in the second stage, local top- k values are merged to get the final top- k . The merge stage can be executed in parallel as discussed in section 3.4.1. We compared the performance of the pairwise approach shown in Figure 3.13c with the baseline approach implemented using Storm and Hadoop as shown in Figure 3.13a. We observed that Hadoop is highly inefficient compared to the other two approaches. For example, to calculate top-100000 the pairwise approach took around 8 seconds, the baseline approach implemented on Storm took 22 seconds while Hadoop took 84 seconds which is nearly 10 times higher than the pairwise.

Figure 3.28 shows the running times of top- k for different k values. When k value is small, the pairwise approach which is implemented based on our model does not out-

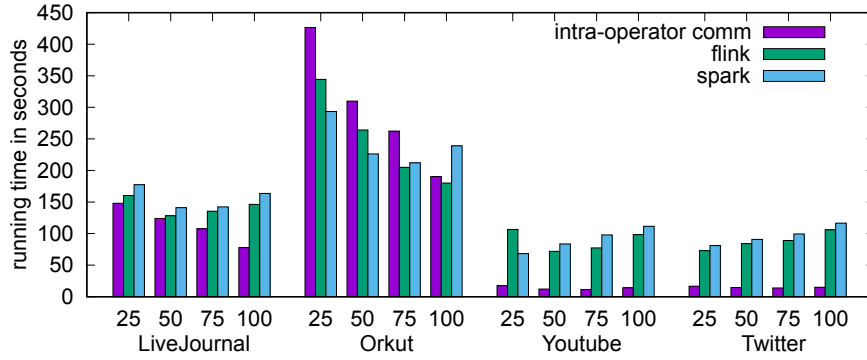


Figure 3.26: PageRank comparison with existing systems.

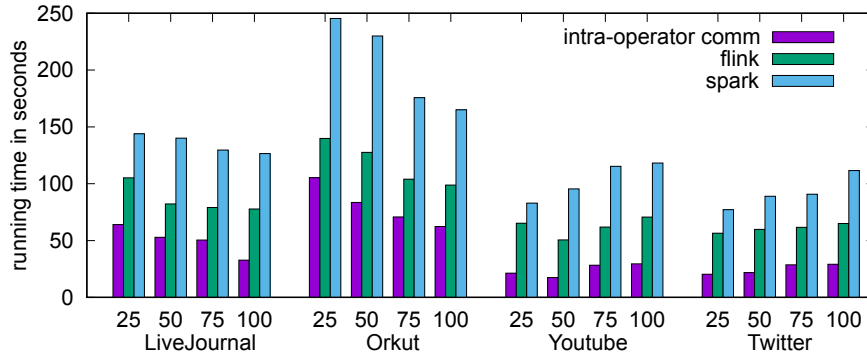
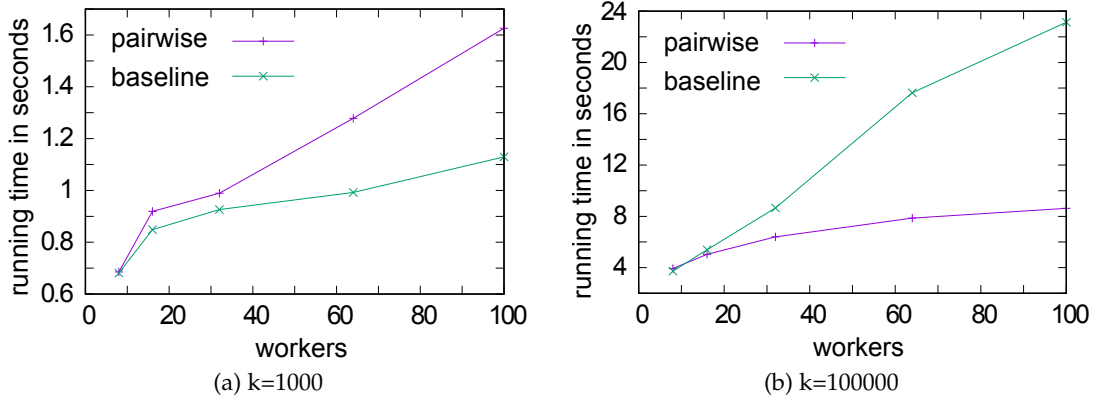


Figure 3.27: SSSP comparison with existing systems.

perform the baseline approach implemented on Storm. This is because the reduction of computation time due to parallelization is suppressed by the increased communication cost. For smaller k values, the size of the messages passed over the network is smaller and the computation time for merge operation is comparatively low. However, as k value increases the message size increases, resulting in high communication time. In addition, the computation time of the merge operation increases with the k value. Hence, the pairwise approach starts outperforming the classic top- k calculation approach. Moreover, the running time increases as the number of workers increases. This is because the number of messages passed and merge computations performed increase with the number of workers. However, for more computationally intensive tasks, increasing the workers reduces the running time because the communication cost increase is suppressed by the reduction of computation cost due to increased workers. These results show that our approach is well-suited for large-scale data as the performance improvement of our model compared to the traditional baseline approach becomes more significant with the increase of

Figure 3.28: Top- k calculation time.

the dataset size as well as with larger k values.

In our model, snapshots are created continuously over time for every window slide. Therefore, we can support incremental computations by using the results of the previous snapshot calculation to perform the next snapshot calculation without starting the computation from scratch. Figure 3.29 shows the advantages of incremental computations using the running time of top- k calculation for $k=50000$ using 100 workers for both incremental and non-incremental approaches over a 1 hour period using a window slide of 1 minute. Initially, both approaches take the same time for calculation. However, the incremental approach performs better as the window slides due to less computation cost and fewer messages passed over the network. Nevertheless, incremental computations cannot achieve performance improvements over non-incremental approaches in cases, where there is a considerable difference between consecutive snapshots due to high input rates and large window slides. Therefore, factors including the type of computation, input rate has to be considered before adopting incremental computations.

3.6.3 K-means clustering

We compared the k -means algorithm implemented on our model as shown in Figure 3.14 with k -means implementations on Spark and Hadoop. Hadoop implementation is highly inefficient for this type of algorithms. For every iteration of the k -means, a new Hadoop job has to be executed and at every iteration the intermediate results are written

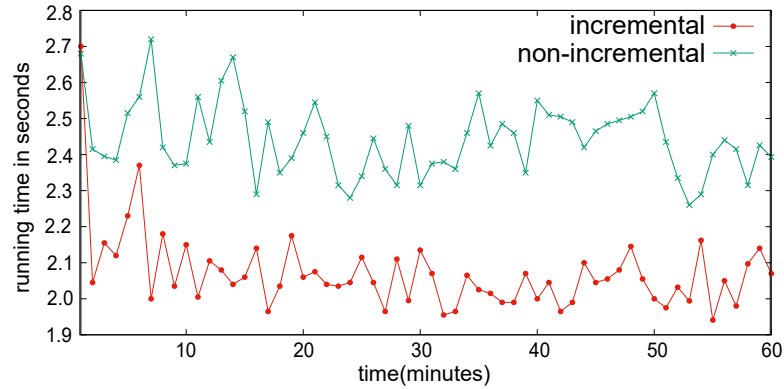


Figure 3.29: Top- k calculation time over 1 hour period for 1 minute slide.

to Hadoop distributed file system and then read back at the start of the next iteration. To perform k -means clustering when $k=10$ using 10 workers for a dataset with 33920 points, our model took around 3 seconds while Hadoop completed the computation in 880 seconds. Therefore, we only show the comparison between our model and Spark in Figure 3.30. We used Tower, a three dimensional dataset [157] which consists of 4915200 data points as our input. Figure 3.30 shows the running times of Spark and our model based on Storm for different k values. As indicated in the figure, our model outperforms Spark and scales better than Spark when the number of workers increases. The size and the number of messages processed by each worker has a significant impact on Storm performance. Hence the running time of Storm decreases significantly when we increase the number of workers. Initially in both approaches, the running time decreases as the workers increase. However, when we further increase the workers, the running time starts increasing again. For instance, the figure shows that Spark gives the lowest running time for all the k values when there are 32 workers, while our model performs best when there are 64 workers. When we increase the number of workers, the computation cost reduces but the communication cost increases because more messages need to be passed among the workers. The increase of the running time after further increasing the workers is because the reduction of computation cost is lower than the increase in communication cost due to the increased number of workers.

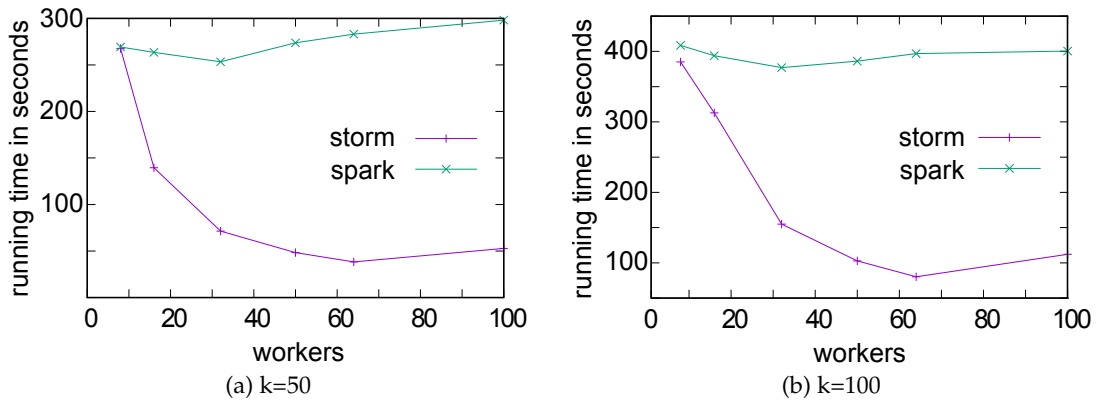


Figure 3.30: K-means calculation time.

3.6.4 Bitonic mergesort

Figure 3.31 shows the running time of bitonic mergesort to sort 10 and 20 million items using two communication patterns. Bitonic mergesort consists of a set of iterations and in each iteration, pairs of workers communicate with each other to sort their local data. Because of the flexibility of the dependency graph, we can implement different approaches to perform the sorting between a pair of workers. One approach is that both workers send their items to each other and then each worker merges the received items, sorts the merged items and then one worker keeps the first half of the sorted list and the other worker the other half. This approach is referred to as duplicate-merge. Another approach is, one worker sends its items to the other and at the arrival, the receiving worker merges the items, sorts them and keeps the first half of the sorted items and sends back the items that need to be changed at the sender side. This way the number of items passed after sorting is minimized. This is referred to as single-merge. If the number of items per worker is n , in the duplicate-merge method, $2n$ items are passed between every pair of workers per iteration. In the single-merge, $2n$ items have to be passed only in the worst case, where a worker swaps all its items with the items of the paired worker. Therefore, the single-merge approach gives better results in all the cases due to its lower communication cost. For 25 workers, single-merge improves performance by 5.33% and 9.31% for 10 and 20 million items respectively. As the dataset size increases the performance gain of single-merge approach becomes more significant.

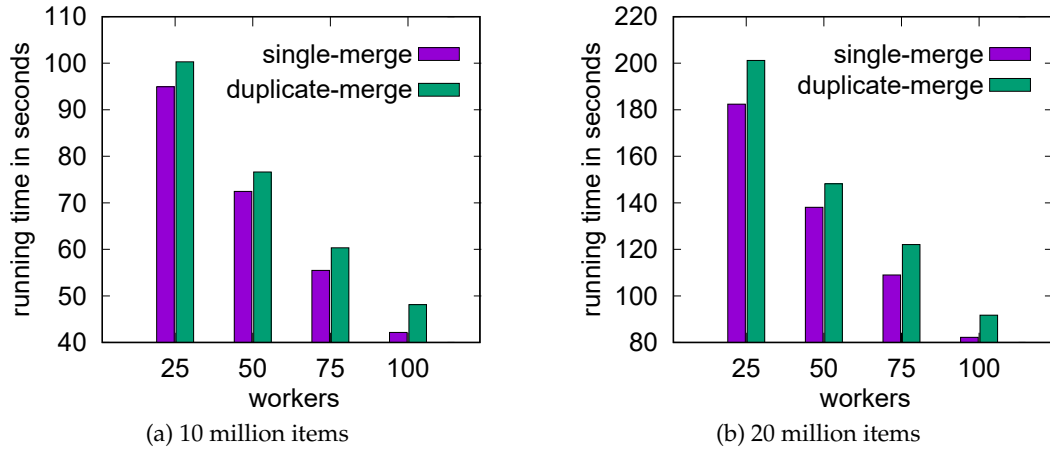


Figure 3.31: Bitonic mergesort calculation time.

3.6.5 Dynamic programming

We implemented the FloydWarshall algorithm which is an example of dynamic programming that also falls into the iterative algorithm category. Since the message size has a significant impact on the performance of Storm, we use message compression as an optimization technique in our model. We compress the messages passed to other workers before sending it over the network. Figure 3.32 shows the running time of FloydWarshall algorithm with and without the optimization technique. As shown in the figure, message compression significantly reduces the running time. For instance, message compression gains 63% performance improvement for 50 workers.

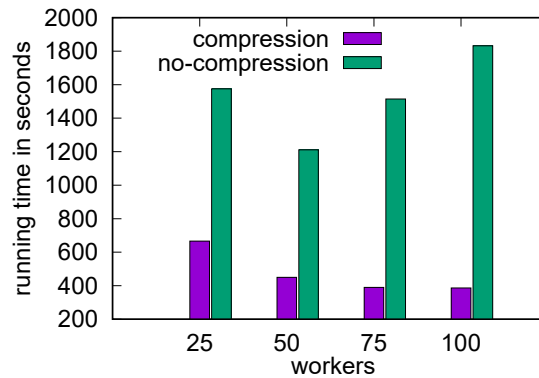


Figure 3.32: FloydWarshall calculation time.

3.7 Conclusion

In this chapter, we introduce two models to support complex stateful processing. We first present a hierarchical windowing model that can be used to define multiple stages of windows to support complex streaming use cases. The hierarchical windowing model we proposed supports parallel window computations and incremental processing and allows stage specific optimizations at each windowing stage.

Next we present the intra-operator communication model that can support iterative computations. This communication model differs from the existing approaches because none of the existing approaches explores intra-operator communication to support iterative computations. We have shown that our model can support complex computations that cannot be modeled using a classic DAG and it can perform computations that can be modeled in a DAG in a more efficient manner. Furthermore, we have shown that this model can be used to support different types of distributed iterative computations ranging from graph processing, sorting to machine learning which are common in streaming use cases. We also present an API using which iterative streaming applications can be implemented. The evaluations have demonstrated the scalability and efficiency of the proposed model for different applications. Moreover, the results suggest that the performance gain of our model becomes more significant for large datasets and computationally intensive algorithms.

After exploring how stateful processing can be efficiently represented in streaming applications through the hierarchical windowing model and intra-operator communication based state sharing, the next chapter focuses on the state recovery aspect of the stateful processing and provides a theoretical model using which we can find the optimal settings for the checkpoint-based fault-tolerance approach to minimize its overhead on system performance.

Chapter 4

A Utilization Model for Optimization of Checkpoint Intervals in Stream Processing Systems

4.1 Introduction

Fault-tolerance is an essential aspect of stateful computational support which ensures processing can continue despite failures. As the system scale increases to cope with an ever increasing big data demand, the mean time to failure (MTTF) of the stream processing system as a whole decreases [123, 158]. For instance, MTTF of Exascale systems is anticipated to be in minutes [159, 160] which arises the need for efficient fault tolerance approaches to ensure state can be recovered to continue processing even after failures. Failures expected in stream processing systems can be of different types. A failure may occur in hardware, e.g. power supplies and memory read errors, that can lead to correlated failure of all processes on a machine, or may occur in the application, e.g. as an exception, where a single process or thread on the machine fails. Moreover, failures may occur at any time, even while the system is trying to recover from a previous failure.

The latest practical deployments of distributed stream processing systems, such as *Apache Flink* and *Storm* [161, 145] use *checkpointing* [86, 64] to support fault-tolerance. The general approach employed, called global or *system-wide checkpointing*, is to only buffer

This chapter is derived from:

- Sachini Jayasekara, Aaron Harwood, and Shanika Karunasekera. “A Utilization Model for Optimization of Checkpoint Intervals in Distributed Stream Processing Systems,” *Future Generation Computer Systems*, vol. 110, pp. 68-79, 2020.

messages at the sources of the topology and to replay such messages when the entire topology rolls back to a checkpoint. This avoids the problem of buffering messages at every operator of the topology (local checkpointing) which would otherwise become infeasible with high volume data streams; though global checkpointing requires the entire topology to roll-back on failure which also has limitations. Checkpointing is argued as a more efficient alternative to state replication [162, 134, 136, 87], which becomes more evident as the state size grows.

In this chapter we address the fundamental checkpointing question: how often should checkpoints be computed? We answer this question by formulating an analytical expression for *utilization* – the fraction of total time available to do useful work – by which we can express the optimal checkpoint interval as that which maximizes utilization, in the context of a stream processing system as a whole, accurately modeling the state-of-the-art implementations. Our derivation is similar to but more appropriate for stream processing than the seminal work of Daly [94, 95], which follows from Young [93] who introduced a first order model to approximate the optimal checkpoint interval, where we also take into account higher order effects such as failure during recovery and multiple failures during a single interval. Furthermore Daly’s approach minimizes the total run-time of the system, given a bounded workload, whereas for stream processing systems the workload is unbounded and so our utilization model is more applicable. The very recent work of Zhuang et al. [103] proposes a checkpointing model for stream processing systems, also based on Daly, that is workload aware. They do not consider failure during recovery, they make some assumptions on average time to a failure within a checkpoint period (similar to Daly’s first order model) and they do not show any experiments to confirm their results.

4.1.1 Contribution

In this work, we present a theoretical framework for determining the optimal checkpoint interval for stream processing systems. This model takes into account failure rate of the system, checkpoint cost, restart cost and stream processing specific aspects such as the application depth which we elaborate on the next sections. We confirm the accuracy of

the theoretical model through a set of simulations.

4.2 Fundamental Utilization Model

In this section we provide a derivation of our utilization model from first principles, which considers an underlying abstract system, progressively refined with the salient aspects of checkpointing and recovery. In tandem we compare our utilization model with the outcomes of a stochastic simulation of the abstract system, to support the model's correctness. In Section 4.3 we extend this to a stream processing system.

At a high level, we define the *utilization*, $0 \leq U \leq 1$, of a system as the fraction of the system's time for which its resources are available to do useful work (to process load), as opposed to work done solely to maintain the system's operation, sometimes called overhead, which in our definition includes the overhead associated with loss and recovery from system failure – in fact in this chapter we assume this as the only significant source of overhead. The work done by the system to create a checkpoint, and the work done by the system from the checkpoint time to the time taken to detect an occurred failure and successfully restart from the checkpoint is not useful work under our definition and thereby detracts from the utilization. The only useful work is therefore the work done, without failure, between two consecutive checkpoints (not including the work to create the checkpoints), or between a successful restart to the next checkpoint. For systems that process streaming data, where the load varies over time, the utilization is an expression of idle time plus time doing work when load is present. If there is no load on the system then the utilization is wholly the idle time of the system.

4.2.1 Without failure

Consider a system that does checkpointing with a constant periodicity of T seconds, i.e. if a single failure were to occur then less than T seconds of work is lost. Let $0 \leq c \leq T$ be an abstract constant cost for checkpointing, here expressed without loss in generality as to how or when the checkpoint is created, in units of time, which without failure as

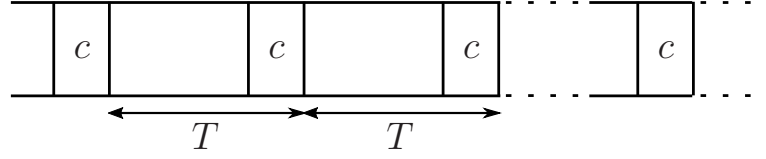


Figure 4.1: Checkpointing with period T , checkpoint overhead c and no failures occurring.

shown in Fig. 4.1 leads to an expression for utilization:

$$U = \frac{T - c}{T}. \quad (4.1)$$

In our work we depict the checkpoint as being created in the last c seconds of the period T , however mathematically it is immaterial as to exactly where in the period T the system spends time creating the checkpoint, suffice to say that it completes exactly at the end of the period. The actual value of c depends on the system itself (e.g. the performance of the disk drives for storing the checkpoint) and the application (how much data needs to be checkpointed) and we assume that this value is known or measured as needed (perhaps periodically measured if T is to be periodically updated). Therefore the question to answer is what should be the value of T : too small (close to c) and the utilization is degraded due to spending too much time doing (unnecessary) checkpointing, too large and the utilization will be degraded due to too much work lost when failure occurs, as we show next.

4.2.2 With failure and negligible restart cost

We model failure as a series of independent failure events having an exponential inter-arrival time given by random variable $\mathbf{X}$ with parameter λ , where the probability of failure at time t is $\mathbb{P}[\mathbf{X} = t] = \lambda e^{-\lambda t}$, the probability distribution for the probability of failure by time t is $\mathbb{P}[\mathbf{X} < t] = 1 - e^{-\lambda t}$, and the mean time to failure is

$$\mathbb{E}[\mathbf{X}] = \int_0^\infty t \mathbb{P}[\mathbf{X} = t] dt = \frac{1}{\lambda}.$$

Assume that the system has just completed a checkpoint period. Either no failure happens within the next time period T , with probability $e^{-\lambda T}$, in which case the system successfully computes the next checkpoint, or failure happens within time T as indicated in Fig. 4.2 and the system needs to restart from the existing checkpoint. In the later case we should ask at what point within time T did the failure happen, more generally, given that failure does happen within time T then what is the mean time to failure, $F(T)$? To do this we normalize the distribution over the interval $[0, T]$ and re-evaluate the mean:

$$\begin{aligned} F(T) &= \mathbb{E}[\mathbf{X} | \mathbf{X} < T] = \frac{\int_0^T t \mathbb{P}[\mathbf{X} = t] dt}{\mathbb{P}[\mathbf{X} < T]} \\ &= \frac{e^{T\lambda} - T\lambda - 1}{\lambda (e^{T\lambda} - 1)} < \frac{1}{\lambda}. \end{aligned} \quad (4.2)$$

The value for $F(T)$ tells us the average amount of time lost (not utilized) if a failure happens, not having restarted yet. In this section we assume the time to detect the failure and to restart is negligible. Fig. 4.2 depicts only a single failure occurring preceding a successful period. However in general, the number of consecutive failures that could occur, k , before a successful period is unbounded and selected from the set $k \in \{0, 1, 2, \dots\}$ at random with a geometric distribution, $(1 - p)^k p$, having parameter

$$p = p_T = \mathbb{P}[\mathbf{X} \geq T] = 1 - \mathbb{P}[\mathbf{X} < T],$$

giving an average number of consecutive failures $\frac{1-p_T}{p_T}$. Each consecutive failure loses on average an additional $F(T)$ time (again not including restart and recovery time). We include this lost time into our utilization model from Eq. 4.1 by expressing the *effective* period,

$$T_{eff} = T + \frac{1 - p_T}{p_T} F(T),$$

and writing:

$$U = \frac{T - c}{T_{eff}} = \frac{\lambda (T - c)}{e^{T\lambda} - 1}. \quad (4.3)$$

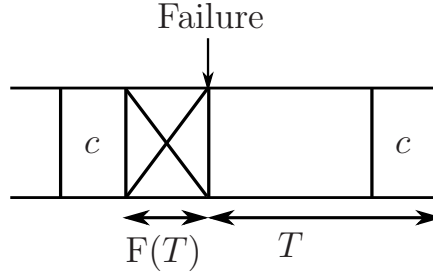


Figure 4.2: Failure resulting in lost time $F(T)$ and instantaneous recovery followed by a successful period T .

4.2.3 Including the time to detect and recover from failure

This section includes the time to detect and recover from the failure, R , as shown in Fig. 4.3. In this case it is at first glance tempting to write

$$T_{eff} = T + \frac{1 - p_T}{p_T} (F(T) + R),$$

since every failure requires a restart, however we note that failure may also occur *during* the restart, in which case we assume that the restart must itself start again. Similar to the number of consecutive failures, the number of attempts to restart, $r \in \{1, 2, \dots\}$, is selected at random using a geometric distribution (note that following a failure at least 1 restart is always required), $(1 - p)^{r-1}p$, with parameter $p = p_R = \mathbb{P}[X \geq R]$, leading to an average number of restarts $\frac{1}{p_R} \geq 1$. For any given failed restart attempt, given that we know the failure occurred within the restart time R we know from Eq. 4.2 that the average time lost is $F(R)$. Taking all of this into account leads to

$$T_{eff} = T + \frac{1 - p_T}{p_T} \left(F(T) + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right)$$

and finally:

$$U = \frac{T - c}{T_{eff}} = \frac{\lambda (T - c)}{e^{\lambda(R+T)} - e^{R\lambda}}. \quad (4.4)$$

This completes the salient features of our checkpoint and restart system model for a single process. We adapt this to a distributed stream processing system in Section 4.3.

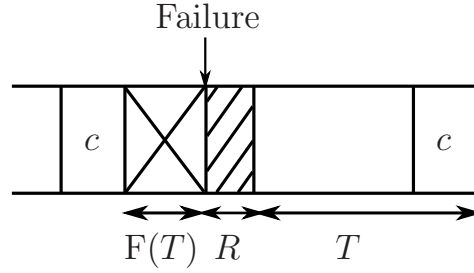


Figure 4.3: Failure resulting in lost time $F(T)$ and recovery time R followed by a successful period T .

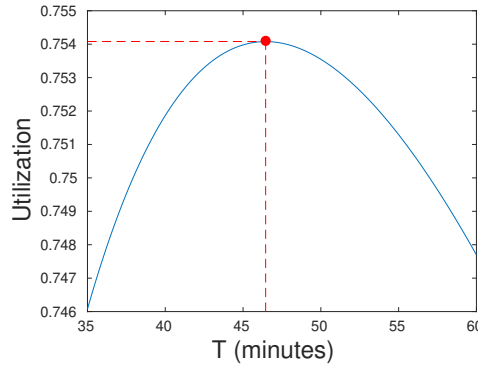


Figure 4.4: Utilization for $\lambda = 0.005$ per minute, $c = 5$ minutes, $R = 10$ minutes.

4.2.4 Optimization of utilization

The value of T that maximizes the utilization U from Eq. 4.4, T^* , is found by solving

$\frac{\partial U}{\partial T} = 0$ for T :

$$\frac{\partial U}{\partial T} = \frac{\lambda}{e^{\lambda(R+T)} - e^{R\lambda}} - \frac{\lambda^2 e^{\lambda(R+T)} (T - c)}{(e^{\lambda(R+T)} - e^{R\lambda})^2} = 0,$$

$$T^* = \frac{c\lambda + W(-e^{-c\lambda-1}) + 1}{\lambda},$$

where $W(z)$ is the Lambert W function on the principal branch. Interestingly T^* is not a function of R . For example, Fig. 4.4 shows the utilization for varying T , with harsh values of c and R , using the model expressed in Eq. 4.4. In this example, $U = 0.7541$ is maximum when $T = T^* = 46.452$ minutes.

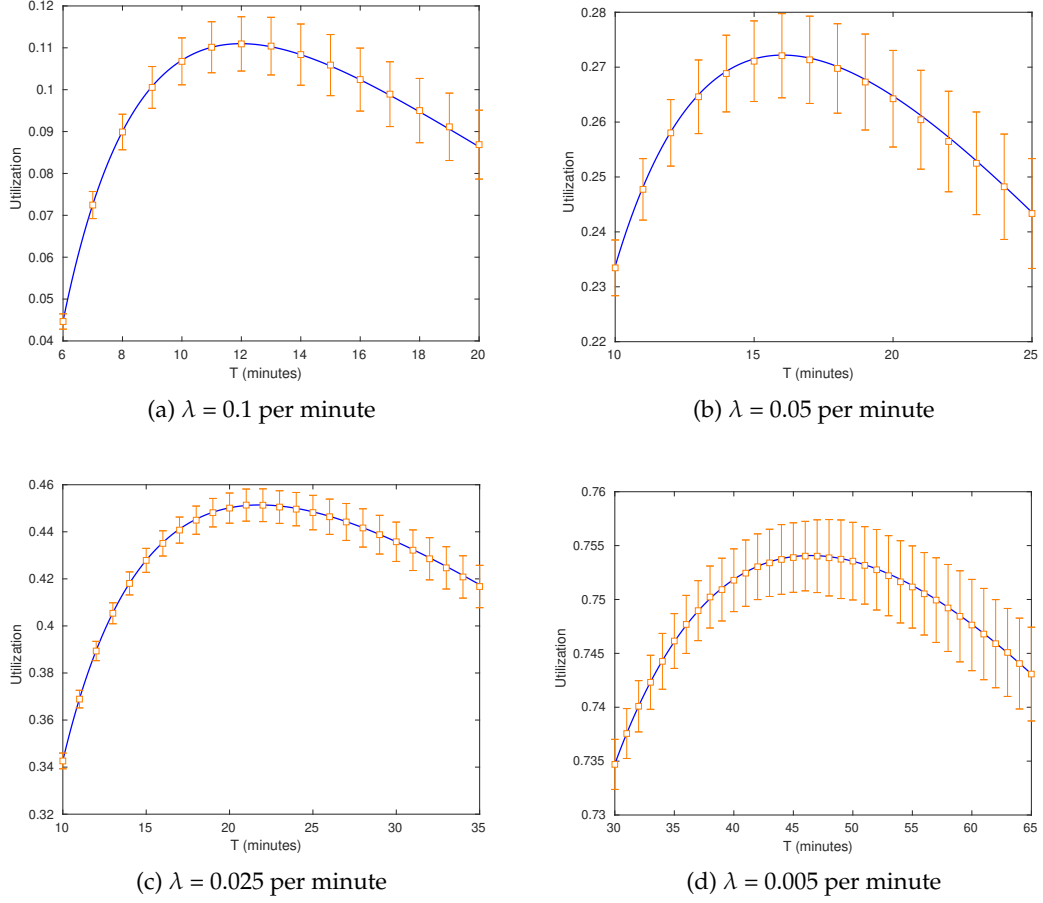


Figure 4.5: Utilization for $R = 10$ minutes $c = 5$ minutes. Solid lines are theoretical utilization and data points with error bars are the average utilization and the standard deviation of simulation observations.

4.2.5 Comparison to stochastic simulation

We developed a stochastic simulation to support the derived model. The simulation generates random failures based on an exponential distribution and calculates the utilization taking the randomly generated failures into account. Fig. 4.5 shows the utilization comparison between our model based on Eq. 4.4 and the simulation results for different failure rates. The solid lines represent theoretical utilization and the data points with error bars represent the average utilization and the standard deviation of 250 runs of the simulation with each simulation running for $\frac{2000}{\lambda}$ minutes.

4.3 Utilization of a Stream Processing System

In this section we extend the proposed model to describe a distributed stream processing system, which is typically represented by a DAG of operators. An operator performs computations on its input stream(s) and may produce one or more output streams to succeeding operators in the DAG. Operators without input streams are sources and those without output streams are sinks. Token based checkpointing is the state-of-the-art approach used in stream processing systems. In this approach, a “checkpoint token” is passed at regular checkpoint intervals from all sources of the DAG, through the system covering all operators, to the sinks. The checkpoint token signals each operator to store its state as the next checkpoint. The *system-wide* checkpoint completes when all operators have completed the checkpoint for a given token. At a failure event, all operators restore the last persisted checkpoint and continue processing. The system-wide checkpoint is the most common fault tolerance approach used in stream processing systems as it avoids the infeasible buffering of large volumes of data at every operator that would otherwise arise from high volume data streams. For example, well-known stream processing systems used in real-world deployments [163] such as Storm and Flink use this method to support state management and fault tolerance. However, system-wide checkpointing requires the entire topology to rollback for each failure.

Some distributed checkpointing schemes such as Meteor Shower [87] have each operator checkpointing independently. This type of a scheme imposes additional overhead and needs more effort to maintain a consistent global state compared to system-wide checkpoint [164, 165]. For instance, this approach requires saving the message buffers at each operator to recover from failures whereas a system-wide checkpoint saves message buffers only at the sources. For a large-scale system dealing with large volumes of data buffering data at every operator is not desirable as it requires a significant amount of additional storage.

In streaming applications, operators can be of different degrees of parallelism residing in different nodes in a cluster. In such cases, for an application level failure affecting

If the system contains cycles then it can be logically transformed to a DAG for the purposes of checkpointing via the introduction of virtual sources at upstream points [86].

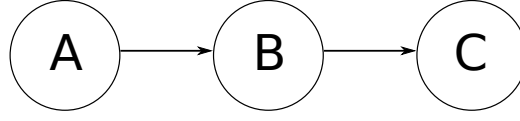


Figure 4.6: DAG of operators representing a streaming application with 3 operators.

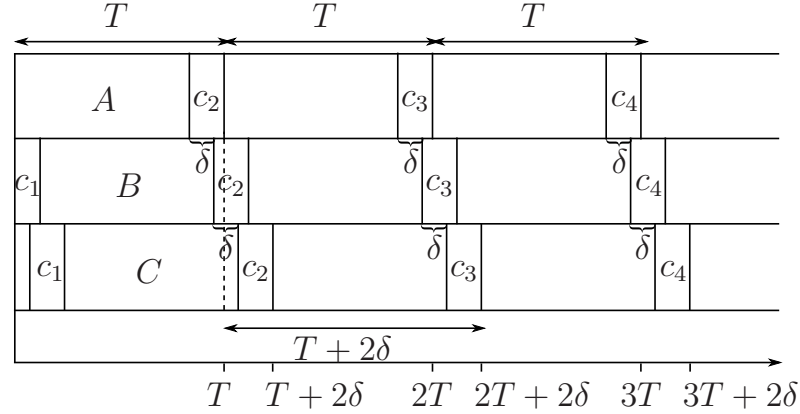


Figure 4.7: Checkpointing process of the streaming application represented in Fig. 4.6.

a single operator or for a hardware failure affecting a set of nodes in the cluster, the application undergoes the same recovery process despite the failure type. Fig. 4.6 shows a DAG of three operators, A , B and C . When determining the utilization, we assume that all operators are stateful and have identical checkpointing cost $0 \leq c \leq T$ and identical cost to detect and recover from the failure, R .

Fig. 4.7 depicts the operators from Fig. 4.6 in terms of time and the flow of a checkpoint token that takes δ time to be transmitted from one operator to the next, causing the checkpoints for a given interval (note that we have numbered checkpoints $c_1, c_2, \dots$ for convenience) to be staggered. As shown in the figure, the checkpoint token is received by operator A at time $T - c$. Then A sends the token to operator B and starts performing the checkpoint c_2 . A finishes its checkpoint at time T . The checkpoint token sent by A is then received by operator B at time $T - c + \delta$. Then B forwards the token to operator C and performs its checkpoint. Finally, operator C receives the token at $T - c + 2\delta$ and finishes its checkpoint at $T + 2\delta$. In this case the system-wide checkpoint is not complete until time $T + 2\delta$. Let the longest path between all source and sink operators (critical path) in the DAG consist of n operators including the source and the sink, then, without

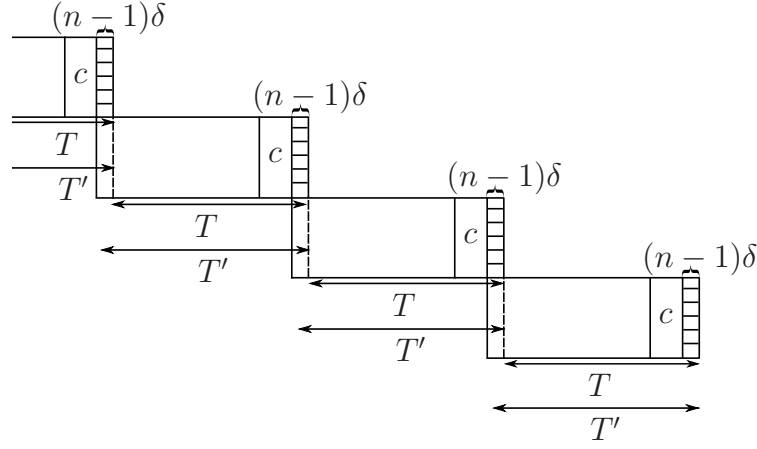


Figure 4.8: Alternative view of the checkpointing process of a streaming application showing the overlap between checkpoints and the computation.

considering failure and using Fig. 4.8, we have

$$T_{eff} = T' = T + (n-1)\delta$$

and so:

$$U = \frac{T-c}{T_{eff}} = \frac{T-c}{T+(n-1)\delta}. \quad (4.5)$$

4.3.1 Including failure and recovery cost

In our model when a failure is detected in the stream processing system, all operators restore the last system-wide completed checkpoint, i.e. checkpoint c_i for the largest i where all operators have completed c_i . As discussed in Sections 4.2.2 and 4.2.3, we include the time due to failures during T' and failures during R to improve the accuracy of T_{eff} . Similar to the number of consecutive failures, within time T for a single operator, the number of attempts to complete T' , $k' \in \{1, 2, \dots\}$, is selected at random using a geometric distribution, $(1-p)^{k'}p$, with parameter

$$p = p_{T'} = \mathbb{P}[\mathbf{X} \geq T'] = 1 - \mathbb{P}[\mathbf{X} < T'],$$

leading to an average number of consecutive failures

$$\frac{1 - p_{T'}}{p_{T'}}.$$

Each consecutive failure loses on average an additional $F(T')$ time. Similarly to the number of consecutive failures, the average number of restarts is $\frac{1}{p_R} \geq 1$ and the average time lost due to a failure during restart is $F(R)$. Taking all of this into account leads to

$$T_{eff} = T' + \frac{1 - p_{T'}}{p_{T'}} \left(F(T') + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right). \quad (4.6)$$

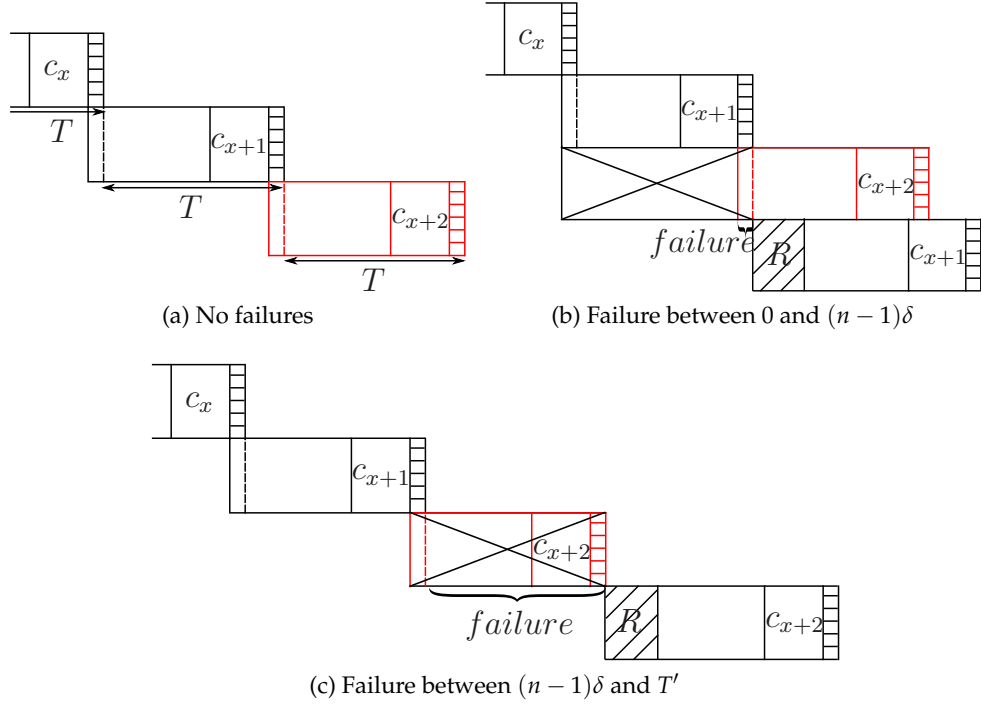
4.3.2 Including the overlap between consecutive checkpoints

During time T' , the last completely persisted checkpoint changes as the computation progresses. For example, consider the computation interval between checkpoints c_{x+1} and c_{x+2} highlighted in Fig. 4.9a. If a failure occurs during $(n-1)\delta$ as shown in Fig. 4.9b, then the system has to restore from the checkpoint c_x as checkpoint c_{x+1} is still underway. If the failure occurs after $(n-1)\delta$, then the system can restore from the checkpoint c_{x+1} as shown in Fig. 4.9c.

For T' leading up to checkpoint c_x , if a failure occurs during the first $(n-1)\delta$ time of the computation, then checkpoint restoration uses c_{x-2} . This is same as if the failure occurs during the previous T' leading up to checkpoint c_{x-1} . If a failure occurs between time $(n-1)\delta$ and T' , then the checkpoint restoration is done using the previous checkpoint c_x . Therefore, recovery due to failure during the first $(n-1)\delta$ of a T' is the same as if the failure occurs during the previous T' . The first $(n-1)\delta$ of T' is already represented by the previous T' as it overlaps with the previous T' as shown in Fig. 4.8. Hence, for one T' , we only have to consider the time between $(n-1)\delta$ and T' , ignoring the first $(n-1)\delta$ of T' .

The average number of consecutive failures before completing the first $(n-1)\delta$ of T' is

$$\frac{1 - p_{(n-1)\delta}}{p_{(n-1)\delta}},$$

Figure 4.9: Failure during T' .

where

$$p_{(n-1)\delta} = \mathbb{P}[\mathbf{X} \geq (n-1)\delta] = 1 - \mathbb{P}[\mathbf{X} < (n-1)\delta].$$

Taking all of this into account, the effective period to complete $(n-1)\delta$ is:

$$(n-1)\delta + \frac{1-p_{(n-1)\delta}}{p_{(n-1)\delta}} \left(F((n-1)\delta) + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right).$$

Subtracting this from T_{eff} in Eq. 4.6 avoids the double representation of the first $(n-1)\delta$ and leads to:

$$T_{eff} = T' + \frac{1-p_{T'}}{p_{T'}} \left(F(T') + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right) - \left((n-1)\delta + \frac{1-p_{(n-1)\delta}}{p_{(n-1)\delta}} \left(F((n-1)\delta) + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right) \right)$$

$$= T + \frac{1 - p_{T'}}{p_{T'}} \left(F(T') + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right) - \left(\frac{1 - p_{(n-1)\delta}}{p_{(n-1)\delta}} \left(F((n-1)\delta) + R + \left(\frac{1}{p_R} - 1 \right) F(R) \right) \right).$$

Finally:

$$U = \frac{T - c}{T_{eff}} = \frac{\lambda e^{\delta \lambda} (T - c)}{e^{\lambda(R+T+\delta n)} - e^{\lambda(R+\delta n)}}. \quad (4.7)$$

Remarkably the expression for utilization has a simple form that includes δ and n in a natural way. This completes the salient features of our checkpoint and restart system model for a distributed stream processing system.

4.3.3 Optimization of utilization

The value of T that maximizes the utilization U from Eq. 4.7, T^* , is found by solving $\frac{\partial U}{\partial T} = 0$ for T :

$$\frac{\partial U}{\partial T} = -\frac{\lambda e^{\delta \lambda}}{e^{\lambda(R+\delta n)} - e^{\lambda(R+T+\delta n)}} - \frac{\lambda^2 e^{\lambda(R+T+\delta n)} e^{\delta \lambda} (T - c)}{(e^{\lambda(R+\delta n)} - e^{\lambda(R+T+\delta n)})^2} = 0$$

$$T^* = \frac{c \lambda + W(-e^{-c \lambda - 1}) + 1}{\lambda},$$

where $W(z)$ is the Lambert W function on the principal branch. Interestingly T^* is identical to that of a single process, i.e. it is independent of n and δ .

Fig. 4.10 shows the utilization for different values of T , using the model expressed in Eq. 4.7. In this example, $U = 0.667$ is the maximum when $T = 46.452$ minutes. Fig. 4.11 shows the utilization of a single operator and a DAG of operators for the same λ, R, c values. As indicated in the figure, T^* is identical for both cases despite the δ and n terms introduced. However, the utilization is significantly less due to the impact of δ and n . For example, in the figure we can observe 11.6% percentage decrease in utilization for a DAG of operators with $n = 50$ compared to a single operator.

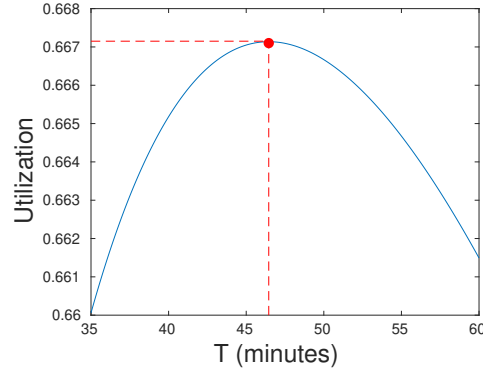


Figure 4.10: Utilization of a DAG of operators for $\lambda = 0.005$ per minute, $c = 5$ minutes, $R = 10$ minutes, $n = 50$, $\delta = 0.5$ minutes.

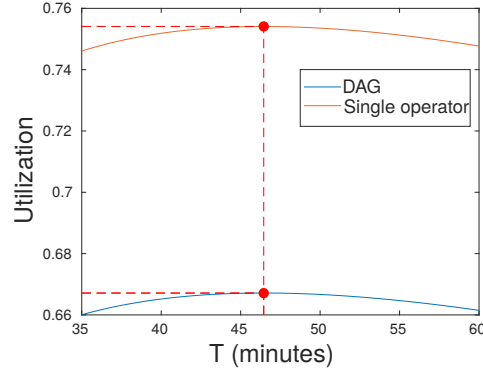


Figure 4.11: Utilization of a single operator and a DAG of operators for $\lambda = 0.005$ per minute, $c = 5$ minutes, $R = 10$ minutes, $n = 50$, $\delta = 0.5$ minutes.

4.3.4 Comparison to stochastic simulation

Fig. 4.12 shows the utilization comparison between our model based on Eq. 4.7 and the simulation results for different failure rates using DAGs with different critical path lengths. The solid lines are theoretical utilization while the data points and error bars represent the average utilization and the standard deviation observed after 250 runs of the simulation, with each simulation running for $\frac{2000}{\lambda}$ minutes. For constant error rate, utilization decreases as n increases and T^* remains unchanged.

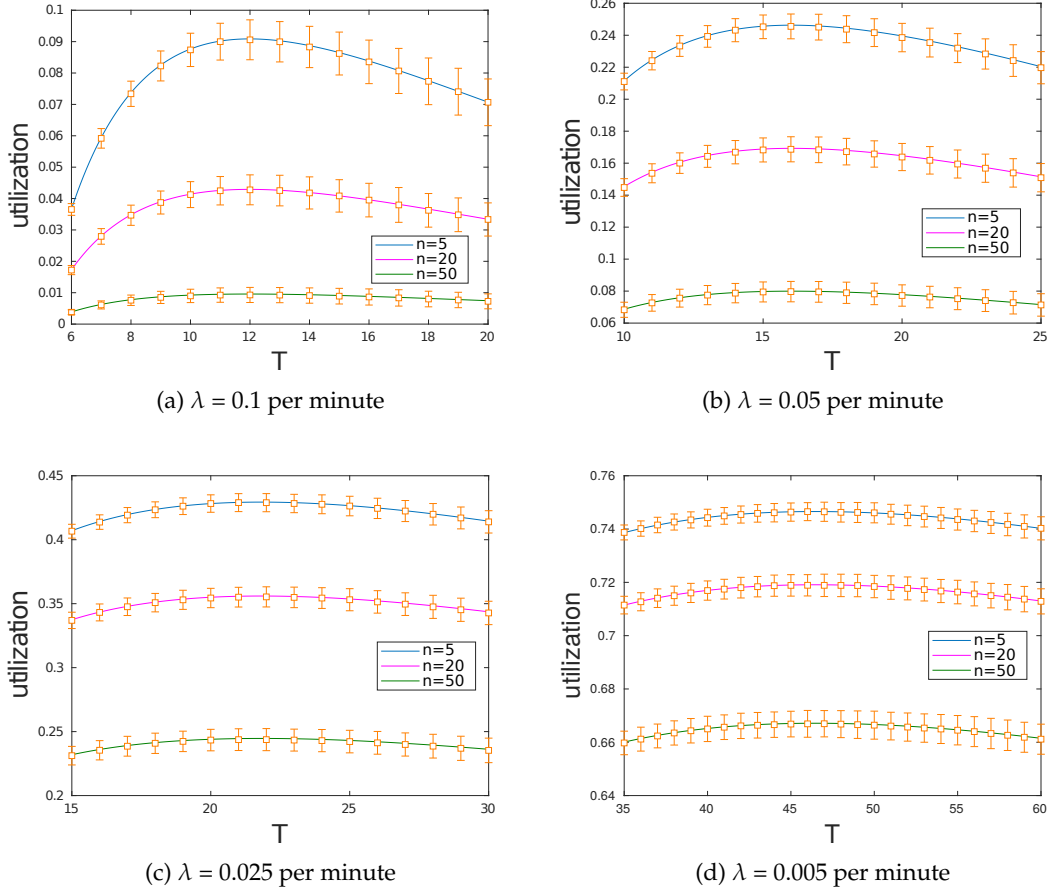


Figure 4.12: Utilization for $R = 10$ minutes $c = 5$ minutes and $\delta = 0.5$ minutes. Solid lines represent theoretical utilization and data points with error bars represent the average utilization and the standard deviation.

4.4 Evaluation

We calculated the percentage utilization increase achieved using the theoretical optimal compared to the default 30 minute checkpoint interval, when $R = 30s$, $c = 5s$, $\delta = 50ms$ and $n = 5$, for the five real-world distributed systems in [166] that have failure rates 0.8475, 0.1701, 0.135, 0.1161 and 0.0606 per hour. In this case we can achieve utilization increase of 18.91%, 2.4%, 1.73%, 1.4% and 0.5% respectively. The default 30 minute checkpoint interval gives the optimal utilization for failure rate, $\lambda = 0.0022$ per hour assuming $c = 1s$. Most distributed systems have higher failure rates than 0.0022 per hour indicative of roughly 1 failure every 19 days. Therefore, using 30 minute checkpoint interval in

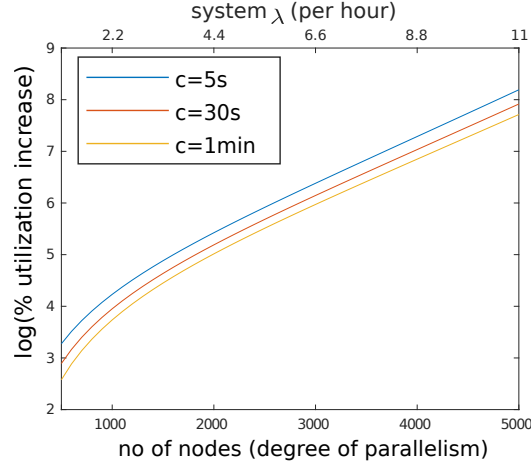


Figure 4.13: System failure rate and the utilization increase gained using T^* instead of 30 minutes for increasing number of nodes.

real-world systems can lead to inefficient system operation.

4.4.1 Comparison and scaling up

Considering the scaling up of distributed stream processing systems, the failure rate of the system increases with the number of nodes in the system. For example, in Exascale systems multiple failures are expected everyday [167, 168] and MTTF is anticipated to be in minutes [159, 160]. For systems such as Flink where a failure of a single node results in restarting the whole application from the previous checkpoint, the failure rate of the system is $\sum_{i=1}^n \lambda_i$, where λ_i is the failure rate of node i [158]. Fig. 4.13 shows how the failure rate changes with the number of nodes in the system considering the failure rate of all nodes is 0.0022 per hour. The figure also shows the percentage utilization increase we obtain using T^* instead of using 30 minute checkpoint interval for $R = 30s, c = 5s, \delta = 50ms$ and $n = 5$. As the number of nodes increases, failure rate increases and the utilization increase we obtain also increases significantly. For example, as indicated in the figure, we can achieve 68.8% utilization increase for 1000 nodes and 226.83% for 2000 nodes. Furthermore, as the checkpointing cost increases, percentage utilization increase reduces as well.

Utilization of a streaming application decreases with the increase of n . Although real-world streaming applications can be represented using a considerable low n value, for

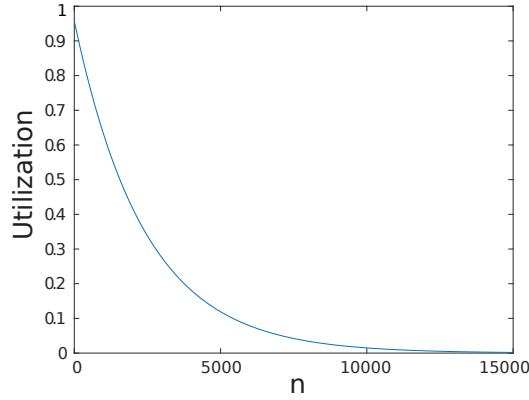


Figure 4.14: Decrease in utilization with the increase of n using T^* .

applications with a large n value, utilization will have a significant impact due to δ and n . Fig. 4.14 indicates how the utilization decreases for large n values. The figure shows the utilization using the model expressed in Eq. 4.7 for T^* , $R = 30s$, $c = 10s$, $\delta = 5s$ and $\lambda = 0.005$ per minute. As indicated in the figure for large n , the utilization comes close to 0. For example, in the figure utilization is 0.0018 for $n = 15000$. Therefore, large-scale applications with large n and δ values can incur significant overheads due to the checkpointing process.

Fig. 4.15 shows the values of T^* given by our model, Daly's first order model [95], $\sqrt{2c(\frac{1}{\lambda} + R)}$ and Zhuang et al.'s model [103], $\sqrt{2c(\frac{1}{\lambda} + R) + c^2}$. For the model of Zhuang et al., we assumed maximum processing rate and the average input rate of the system is same. All the models give near similar results for smaller c and R values as shown in Fig. 4.15a. However, as the values of c and R get bigger, our model deviates from the other models for larger λ values. This is because the assumptions made in Daly's first order model are not accurate for large λ values as stated in Daly's paper [95]. Moreover, we calculated the utilization based on Eq. 4.7 for the proposed T^* and the optimal values given by Daly and Zhuang et al. presented in Fig. 4.15b for $\delta = 30$ seconds and $n = 25$. Fig. 4.16 shows the percentage utilization increase gained using the proposed T^* compared to other models. As indicated in the figure, the utilization increase we can gain using the proposed model becomes more significant as λ increases. For example, when $\lambda = 11$, the utilization increase gained using the proposed T^* compared to the values given by Daly's model and Zhuang et al.'s model is 2.3% and 3.7% respectively.

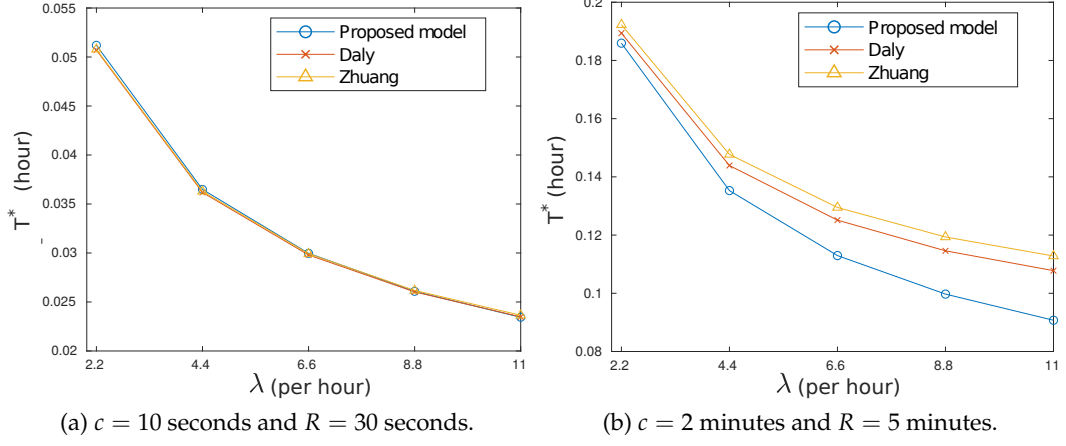


Figure 4.15: Optimal checkpoint interval comparison of the proposed T^* , Daly's model [95] and model of Zhuang et al. [103].

4.4.2 Discussion

In stream processing systems, the input rate can vary widely which can affect the checkpoint cost, e.g. if no data is observed for some time the checkpoint cost may decrease significantly. As T^* depends only on c and λ , in each checkpoint interval we may measure the c of completed checkpoint and λ , and update T^* dynamically for the next checkpoint interval, to adapt to changing workloads. Figure 4.17 shows how T^* changes with c for different failure rates. As indicated in the figure T^* increases with c . We also performed simulations to investigate the impact of dynamically adjusting T based on the checkpoint cost. Figure 4.18 shows the observed utilization of the simulations for applications that can have checkpoint costs in different ranges with and without changing the T based on previous checkpoint costs. For static simulation, we set T as the optimal checkpoint interval for the average value of the used checkpoint cost range and for the dynamic case, T starts with T^* for the average checkpoint cost of the range and T is changed after every checkpoint based on the average of the last ten completed checkpoints. As shown in the figure, as the range of the checkpoint cost increases in an application, changing T dynamically does not result in utilization improvement. Moreover, the utilization difference between two cases is not significant. For instance, the percentage difference in utilization is 0.0021% and 2.5033% for cases 1 and 4 respectively.

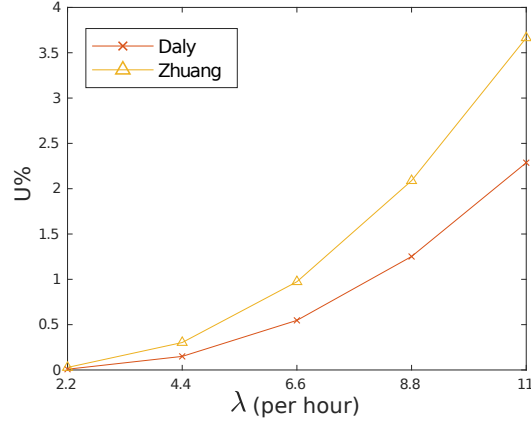


Figure 4.16: Percentage increase in utilization using the proposed T^* instead of Daly's model [95] and model of Zhuang et al. [103] for $c = 2$ minutes, $R = 5$ minutes, $\delta = 30$ seconds and $n = 25$.

The input rate also affects the time required to redo the work lost due to failures. We can further improve the model by taking this factor into account. $F(T)$ indicates the time lost if a failure happens and the amount of rework to be done which was lost during $F(T)$ depends on the input rate, i and processing rate, p of the system. For example, if the input rate is very low between the last checkpoint and the failure, then the time taken to redo the lost computations could be much less than, $F(T)$. Therefore, we can improve the rework time as $F(T)i/p$.

4.5 Conclusion

We provided a rigorous analytical expression for the utilization of a distributed stream processing system that allows optimization of the checkpoint interval through maximizing utilization. The optimal checkpoint interval is seen to be dependent only on checkpoint cost and failure rate. Our analytical formulation provides a solid theoretical basis for the analysis and optimization of more elaborate checkpointing approaches such as *multi-level checkpointing* [123]; where Moody et al. show that considering a hierarchy of faults, or multi-level failure model, provides improvement.

The next chapter validates the proposed theoretical model using practical experimentation in a distributed setting and demonstrates the benefits of using the optimal check-

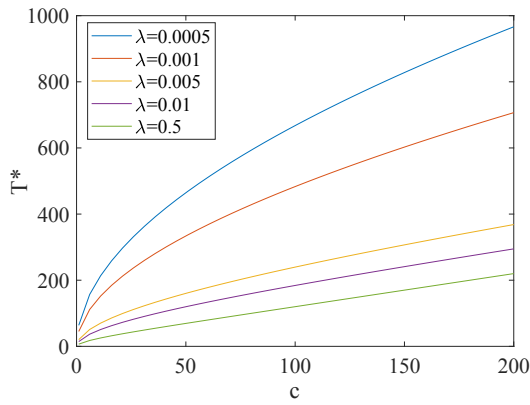


Figure 4.17: Checkpoint cost and optimal checkpoint interval comparison.

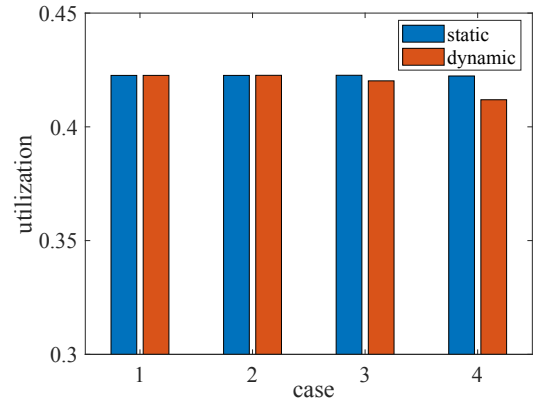


Figure 4.18: Utilization of static and dynamic T for different c ranges. c range-case 1: 49-51, case 2: 45-55, case 3: 25-75, case 4: 1-99 seconds.

point interval compared to nominal values.

Chapter 5

Optimizing Checkpoint-based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice

5.1 Introduction

Checkpointing is the most common fault-tolerance approach used in exiting stream processing systems to support state recovery. Although checkpointing is vital to ensure that the system does not need to reprocess all the data from the beginning to obtain the correct state, it degrades systems performance as the system has to allocate time and resources to perform the checkpoints. Therefore, determining how often checkpoints should be performed is a critical factor that can affect system performance. We derived a theoretical expression for the optimal checkpoint interval and validated the model through simulations in Chapter 4 and in this chapter, we evaluate the efficiency and practicality of the proposed optimal checkpoint interval when used in real-world distributed stream processing applications.

In this chapter, we demonstrate how the optimal checkpoint interval we proposed can achieve performance improvements using practical examples implemented on Apache Flink, which has a built-in checkpointing mechanism to support fault-tolerance. We used Flink for our experiments because of its superiority in handling the checkpointing pro-

This chapter is derived from:

- Sachini Jayasekara, Shanika Karunasekera, and Aaron Harwood. "Optimizing Checkpoint-based Fault-Tolerance in Distributed Stream Processing Systems: Theory to Practice," *Software: Practice and Experience*, (Under review).

cess such that the application state will reflect every received event exactly once despite failures. We first use a set of common benchmark streaming applications and demonstrate how checkpointing influences application latency, throughput and utilization and show how optimal checkpoint interval can improve the application performance compared to a nominal checkpoint interval. Then we extend our evaluations to indicate how the performance of well-known algorithms is affected by checkpointing and show that significant performance improvements can be gained using the optimal checkpoint interval. Furthermore, we look into the factors that influence the value of the optimal checkpoint interval and the application utilization such as checkpoint cost, restart cost and show what aspects in practical stream processing systems determine the values of these costs and how we can reduce these different costs to improve application performance.

5.1.1 Contribution

This chapter validates the theoretical framework proposed in Chapter 4 for determining the optimal checkpoint interval in a global checkpointing system, that is applicable to distributed stream processing through extensive experiments using a real-world stream processing system in a cloud setting. In contrast to existing work, the proposed optimal checkpoint interval is derived specifically for stream processing systems considering factors such as the use of a global checkpoint interval for all of the operators in an application and message transfer delay between two operators in an application. Furthermore, we investigate how checkpointing influences performance parameters such as utilization, latency and throughput and show how well the negative impact of checkpointing can be reduced using the optimal checkpointing interval.

5.2 Evaluation Methodology

This section describes the system setup we used to evaluate the impact of the optimal checkpoint interval on a practical system. Although the accuracy of the optimal checkpoint interval, presented in Chapter 4, has been verified using simulations, the degree to which it can improve the efficiency of a practical system has to be further investigated,

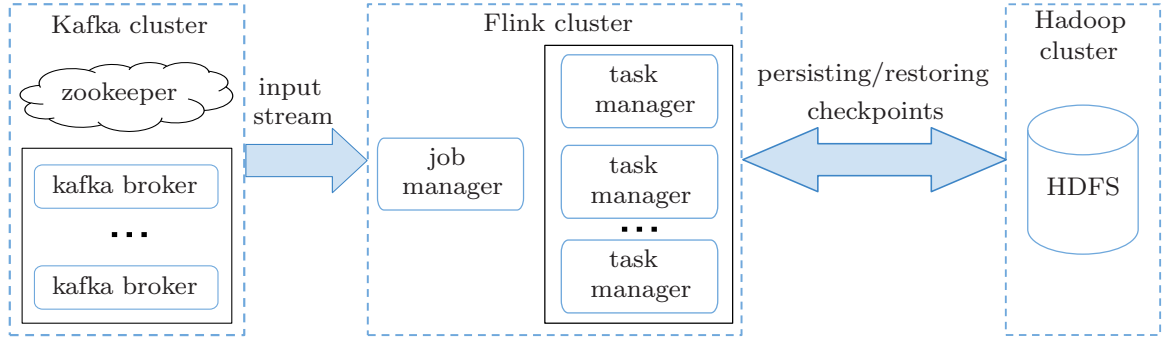


Figure 5.1: Overview of the system.

as theoretical models cannot take into account all of the different aspects in a real-world system, such as network delays, caching and garbage collection. Therefore, using several stream processing use cases we show how our proposed optimal checkpoint interval can improve the performance of stream processing applications in terms of utilization, latency and throughput compared to using a nominal checkpoint interval.

We use Apache Flink as the stream processing system to evaluate the influence of checkpointing in streaming applications and to demonstrate the benefits of the proposed model by comparing the application performance in terms of utilization, latency and throughput when checkpoint interval is configured to the proposed optimal checkpoint interval and to the widely used 30 minute checkpoint interval. Figure 5.1 depicts the high-level overview of the system setup used in our analysis.

5.2.1 Experimental setup

All experiments were conducted using *m2.medium* nodes running on *OpenStack*. Each virtual machine had two CPU cores, 6 GB of RAM, 30 GB of disk space and ran Ubuntu 15.10, Java 1.7.0_91, and Flink 1.3.3. *Hadoop Distributed File System* 2.7.3 was used as the state backend and *Apache Kafka* 0.10.0.0 consumer was used as the source of the applications. The next section briefly explains the different systems used in the experimental setup.

5.2.1.1 Apache Flink

Apache Flink is a well-known stream processing system that supports state management and fault-tolerance through checkpointing. Flink performs periodic checkpoints by sending a checkpoint token from the sources of the application through the graph of operators to the sinks. At the arrival of the token, all of the operators persist the state. The checkpointing process is coordinated and managed by Flink job manager which is the master process of the Flink cluster. Task managers are the worker processors that perform the actual checkpointing. In cases where an operator has multiple inputs from different upstream operators, the receiver performs the checkpoint when it receives checkpoint tokens from all of the upstream operators. When a failure occurs, the job manager notifies all of the worker processors to restore the last completed checkpoint and the data processed after the last checkpoint is reprocessed by the application.

To facilitate the checkpointing process, Flink requires two additional resources, a persistent data source and a persistent storage [169]. The persistent data source should be able to replay events that were processed before a failure to provide exactly-once processing semantic and the persistent storage is used to store the checkpoints. The most common persistent data source used in Flink is Apache Kafka, which we use in our setup. Flink provides a Kafka consumer that can read from Kafka topics and the user only has to setup and configure the Kafka cluster and provide the cluster information to Flink's Kafka consumer. For the persistent storage, we use Hadoop Distributed File System (HDFS).

In our evaluations, we used Flink logs and Flink REST API to measure the checkpointing parameters required in our theoretical model. The checkpoint cost of a Flink application is included in the Flink job manager logs that contain the checkpoint cost of each completed checkpoint in the format: *org.apache.flink.runtime.checkpoint.CheckpointCoordinator - Completed checkpoint 19 (5659789 bytes in 805 ms)*. In addition, we can get more details using the REST API such as total completed checkpoints and details about each checkpoint like the checkpoint cost and state size of individual operators of the application. We take the checkpoint cost of the operator that has the highest checkpoint cost as the value of c .

The restart cost can also be measured using the logs of the Flink job manager. Flink applications can be in one of eleven different states: created, running, failing, failed, canceling, canceled, finished, restarting, suspending, suspended or reconciling. The logs include the time of the state changes of a Flink application such as the time an application changes to the running state after the application creation, or the time an application moves to the fail state from the running state. We can calculate the restart cost using logs that contain the time of the failure and the time the application restarts after a failure. These logs take the format: *org.apache.flink.runtime.executiongraph.ExecutionGraph - Job X switched from state RUNNING to FAILING* and *org.apache.flink.runtime.executiongraph.ExecutionGraph - Job X switched from state CREATED to RUNNING*.

The last measured parameter, the time taken to pass the checkpoint token, δ can also be measured using the REST API. Unlike checkpoint cost, the value of δ is not directly given by the API but can be calculated using the other checkpointing properties directly exposed by the API [169]: $\delta = end_to_end_duration - synchronous_duration - asynchronous_duration$. Theoretical utilization and the optimal checkpoint interval can be calculated with these measured parameters and the enforced failure rate.

5.2.1.2 Hadoop

Hadoop Distributed File System (HDFS) is the primary data storage of Hadoop applications. HDFS is a fault-tolerant file system that maintains multiple copies of the data over several nodes in the Hadoop cluster which makes it an excellent backend to store checkpointed states of streaming applications. Moreover, persistent file systems like HDFS are recommended to be used as the state backend in Flink applications with large states and windows.

5.2.1.3 Apache Kafka

Apache Kafka is an open-source stream-processing software platform that acts as a message queue by publishing and subscribing to streams of records. Zookeeper is used to coordinate the Kafka cluster. Data can be published to a topic in Kafka and data can

be consumed by giving the topic name. Kafka facilitates fault-tolerance for the streams of records through replication. Moreover, the records published to Kafka topics have a sequential id referred to as the offset to uniquely identify a record.

Apache Flink provides a Kafka consumer that can read from Kafka topics which are often used as the data source for Flink checkpoint enabled streaming applications. When Flink applications perform periodic checkpoints, the offset of consumed Kafka records is persisted as the state. Hence, after a failure, the records after the persisted offset will be reconsumed to guarantee all of the records are processed without losing any. Figure 5.2 shows an example where at the time of the checkpoint, the consumer has processed the record that has the offset 4 and waiting for the next record with offset 5. Therefore, offset 4 is persisted as the state. If a failure happens before the next checkpoint all of the records after offset 4 will be reconsumed from Kafka.

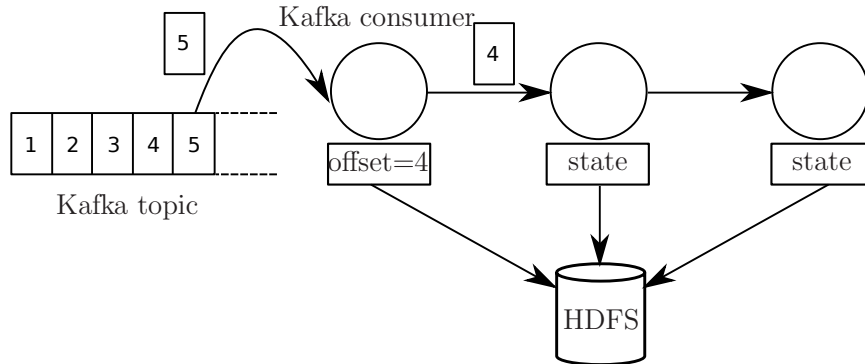


Figure 5.2: Flink checkpointing process at the source operator.

The next section evaluates how the derived optimal checkpoint interval can improve the performance of practical stream processing applications using different measures such as utilization, latency and throughput. Furthermore, we look into different factors that impact the checkpoint cost, c and restart cost, R of stream processing systems.

5.3 Experiments

To evaluate the accuracy and the efficacy of our model, we experimented with various Flink applications, with differing values of application depths n . We simulate random failures by killing one of the running Flink task managers based on an exponential distri-

λ	failure rate
n	depth of the streaming application
$\tilde{c}$	observed checkpoint cost
$\tilde{R}$	observed restart cost
$\tilde{\delta}$	observed checkpoint token communication time
T^*	the theoretical optimal T for the given failure rate and observed parameters
$\tilde{U}$	observed utilization
U	theoretical utilization

Table 5.1: Notations used in the evaluation.

bution at precomputed failure event times using different λ values. Notations used in the evaluation are listed in Table 5.1. For each experiment, we compared $\tilde{U}$ of the application with U given by the theoretical model, using parameters $\tilde{c}$, $\tilde{R}$ and $\tilde{\delta}$ measured from Flink logs as inputs. Then, using the observed checkpoint costs we determined the optimal interval, T^* , based on our model and rerun the application using T^* as the checkpoint interval and observed the achieved utilization. In addition, we compared the latency and throughput of the applications with and without checkpoints.

5.3.1 Use cases

We implemented three common stream processing stateful applications: word count, fraud detection and tweet analysis, and also a set of fundamental algorithms, and compared the performance using the optimal checkpoint interval and the widely used 30 minute checkpoint interval.

5.3.1.1 Word count

Word count is a common streaming application used for benchmarking streaming applications. In our experiment, a Kafka producer continuously reads data from a set of input files and creates a Kafka record for each line in the file. Flink application depicted in Figure 5.3 consists of a Kafka consumer that reads the lines from the predefined Kafka topic. Then the consumed lines are divided into different words by the line splitter operator.

Each word generated by the line splitter is saved in a sliding time window that computes the count of each unique word. At the end of each window, counts are emitted as the output of the application. Contents of log files were sent continuously as input to this application.

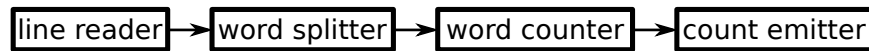


Figure 5.3: Word count use case.

5.3.1.2 Fraud detection

Fraud detection is a common time-critical streaming use case that generates alerts if the events of the input stream follow a given set of patterns or rules. For example in credit-card payments, a set of small credit-card payments made within a few hours could be flagged as possible fraud. We use a similar set of patterns in our application and when the application receives input events that follow the given patterns, it outputs an alert that consists of the event information that followed the pattern as depicted in Figure 5.4. We generated a synthetic set of events that includes faulty events that follow the pattern checked by the application to trigger alerts.

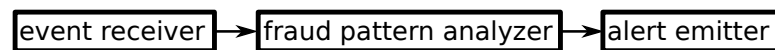


Figure 5.4: Fraud detection use case.

5.3.1.3 Tweet analysis

Tweet analysis is a common application used for various scenarios including finding trending topics, finding influential users, disaster monitoring and sentiment analysis. In our application, depicted in Figure 5.5, the source continuously receives tweets which then go through an operator that extracts hashtags and user information from each tweet. Then the extracted hashtags and user information are stored in time windows and the hashtag window outputs the set of top- k most occurring hashtags. The popular hashtags are then emitted as the application output. In addition, the application joins the popular

hashtags with the user information window to select the users who used these popular hashtags and output the set of users along with the associated hashtag. We used a set of tweets collected from Twitter from 6th March to 23rd April 2012 as the input to the application.

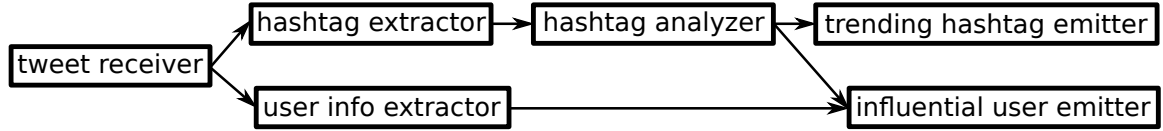


Figure 5.5: Tweet analysis use case.

5.3.1.4 Algorithms

We also selected a set of well-known algorithms for the evaluation; connected components, top- k and k -spanner, which is used in many domains [170, 171, 172]. For the graph algorithms, connected components and k -spanner, LiveJournal dataset [156] was used as the input and for the top- k algorithm, content of log files was used to compute the top- k most frequent words. In all the experiments, algorithms were performed on time windows and input was sent to the application continuously to simulate a stream of input data.

5.3.2 Evaluation

This section demonstrates how utilization, latency and throughput can be improved using the optimal checkpoint interval.

5.3.2.1 Impact on utilization

Table 5.2 shows the λ settings used for the three applications with $\tilde{c}$, $\tilde{R}$, $\tilde{\delta}$, utilization values $\tilde{U}$ and U , when $T = 30$ minutes, calculated T^* , and utilization values $\tilde{U}$ and U , when $T = T^*$. The last column gives the percentage increase in utilization using T^* over the default $T = 30$ minutes. We ran each experiment five times, each running for 48 hours using a five node Flink cluster with one node as the master and rest of the nodes as slaves and the average values of the 5 runs are shown in the table.

Table 5.2: Experimental results of the three applications using a five node Flink cluster.

(a) Settings and observations

Usecase	n	λ	$\tilde{c}$ (ms)	$\tilde{R}$ (s)	$\tilde{\delta}$ (ms)
Word count	4	0.0075	.156 $\pm$.02	18.97 $\pm$ 0.94	11.8 $\pm$ 0.4
		0.05	.242 $\pm$.052	18.15 $\pm$ 0.16	20.27 $\pm$ 2.2
Fraud detection	3	0.005	.098 $\pm$.007	12.89 $\pm$ 0.7	14.06 $\pm$ 2.8
		0.025	.165 $\pm$.029	11.95 $\pm$ 1.12	20.29 $\pm$ 7.1
Tweet analysis	4	0.01	.184 $\pm$.017	18.03 $\pm$ 1.04	14.24 $\pm$ 3.5
		0.075	.341 $\pm$.041	17.59 $\pm$ 0.01	42.73 $\pm$ 2.6

(b) Utilization comparison of $T = 30$ minutes and $T = T^*$

Usecase	$T = 30$ minutes		T^* (min)	$T = T^*$		%U increase
	$\tilde{U}$	U		$\tilde{U}$	U	
Word count	.8564 $\pm$.03	.8895	0.8398	.9904 $\pm$.001	.9927	15.65
	.4319 $\pm$.16	.4243	0.3678	.9802 $\pm$.004	.972	127
Fraud detection	.9106 $\pm$.012	.9258	0.8076	.9955 $\pm$.001	.9959	9.32
	.6541 $\pm$.026	.6680	0.47035	.9855 $\pm$.002	.9874	50.67
Tweet analysis	.8234 $\pm$.071	.8548	0.7942	.9881 $\pm$.004	.9903	20.00
	.3157 $\pm$.063	.2592	0.3916	.9508 $\pm$.009	.9572	201

The results shown in Table 5.2 demonstrates that utilization can be improved using the optimal interval for all of the failure rates that we tested. Moreover, as the failure rate increases, the utilization improvement becomes more significant. Hence in large scale systems where failure rates are high, we can achieve significant performance improvements using the optimal interval. For the failure rate of 0.0075 per minute synonymous with roughly 1 failure every 2.22 hours, we can achieve 15.65% increase in utilization. Hence by using the optimal interval, we can achieve even more increase in utilization for exascale systems that expect failures every few minutes. For instance, when $\lambda = 0.075$ we observed 201% increase in utilization using the optimal value.

Table 5.3 depicts experimental results of the selected set of algorithms using a fifty

node Flink cluster with one node as the master and rest of the nodes as slaves. The table shows the λ settings used for the experiments with $\tilde{c}$, $\tilde{R}$, $\tilde{\delta}$, utilization values $\tilde{U}$ and U , when $T = 30$ minutes, calculated T^* , and utilization values $\tilde{U}$ and U , when $T = T^*$. The last column gives the percentage increase in utilization using T^* over the default $T = 30$ minutes. We ran each experiment five times, each running for 48 hours.

Similar to the results of the small-scale cluster, the results show that utilization can always be improved using the theoretical T^* for all used λ values, compared to the 30 minute checkpoint interval. For instance, for the lowest λ , 0.01, we can gain 13.36% utilization increase and for the highest λ , 0.06 we observed 164.1% increase in the utilization. Furthermore, the actual utilization of all the algorithms compares well with the utilization given by the theoretical model for the given parameters. For example, the observed utilization of the connected components algorithm for $\lambda = 0.01$ is 0.8598 and the utilization predicted by our theoretical model is 0.8545. From the two tables, it is evident that the optimal interval has increased the system efficiency in terms of utilization for small-scale as well as large-scale systems.

5.3.2.2 Impact on latency and throughput

Latency and throughput are two major indicators of the performance of streaming applications. Figures 5.6 and 5.7 indicate the throughput and latency of the use cases we tested using different failure rates compared to the throughput and latency when checkpoints are disabled. When checkpoints are disabled, Flink application stops as soon as the first failure occurs and doesn't restart unlike when checkpoints are enabled. Therefore, all the inputs after the first failure will be lost when no fault-tolerance mechanism is deployed.

Figure 5.6 depicts the average throughput of the use cases for 30 minute checkpoint interval and the optimal checkpoint interval using different λ values and the throughput when checkpoints are disabled. Throughput is calculated by the number of unique events processed per second by the application which ensures replayed events due to failures are not measured multiple times. As shown for all of the use cases, the throughput of the checkpoint disabled case is the highest as the system spends the entire time on computations. However, for real-world systems where failures are inevitable dis-

Table 5.3: Experimental results of selected algorithms using a fifty node Flink cluster.

(a) Settings and observations

Usecase	λ	$\tilde{c}$ (ms)	$\tilde{R}$ (s)	$\tilde{\delta}$ (ms)
connected components	0.01	$1.093 \pm .04$	17.11 ± 1.1	11.24 ± 2.2
	0.04	$1.064 \pm .48$	18.91 ± 1.1	19.13 ± 7.6
top-k	0.03	$.764 \pm .029$	$17.84 \pm .36$	13.47 ± 2.7
	0.06	$.893 \pm .022$	$18.18 \pm .61$	19.82 ± 4.1
k-spanner	0.02	$1.112 \pm .61$	$18.74 \pm .64$	18.96 ± 2.1
	0.05	$.937 \pm .099$	$18.42 \pm .57$	24.15 ± 2.0

(b) Utilization comparison of $T = 30$ minutes and $T = T^*$

Usecase	$T = 30$ minutes		T^* (min)	$T = T^*$		%U increase
	$\tilde{U}$	U		$\tilde{U}$	U	
connected components	$.8598 \pm .05$	.8545	1.9149	$.9747 \pm .009$	.9783	13.36
	$.5251 \pm .02$	.5311	0.9479	$.9477 \pm .009$	.9513	80.48
top-k	$.6099 \pm .06$	.6108	0.926	$.963 \pm .002$	.9631	57.89
	$.3562 \pm .12$	.3498	0.7097	$.9407 \pm .009$	.9419	164.1
k-spanner	$.7173 \pm .02$	.7248	1.3680	$.9633 \pm .01$	.9692	34.29
	$.4272 \pm .11$	.4240	0.7626	$.9449 \pm .01$	.9484	121.2

abling checkpointing is not an option. When comparing the throughput of 30 minutes and the optimal interval, it can be observed that throughput is higher when the optimal checkpoint is used. Furthermore, when the failure rate increases, the increase in throughput using optimal checkpoint interval becomes more significant. For instance, in twitter analysis we observed 19.81% throughput increase when $\lambda = 0.01$ and 53.91% throughput increase when $\lambda = 0.075$.

Figure 5.7 depicts the average latency of the use cases when checkpoints are disabled and for 30 minute checkpoint interval and optimal checkpoint interval using different λ values when checkpoints are enabled. The latency is calculated as the time taken to process an event by the application: the difference between the time an event is received

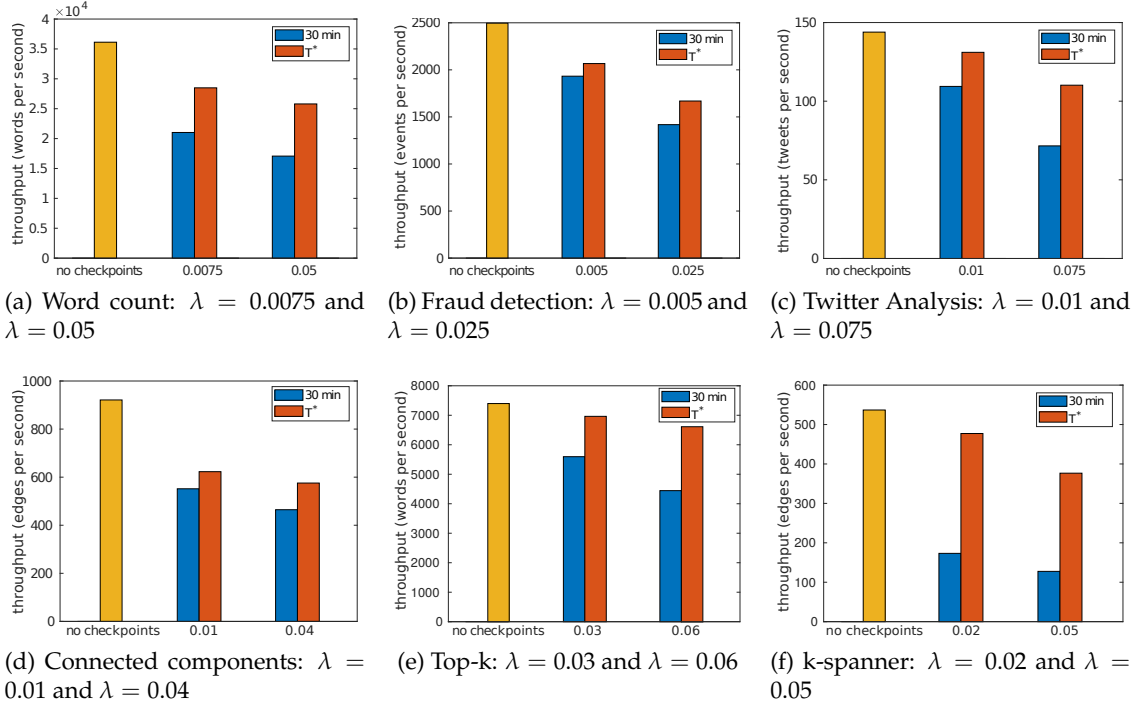


Figure 5.6: Throughput comparison: $T = 30$ minutes, $T = T^*$ and without checkpoints.

by Kafka and the corresponding output is generated by the sinks of the streaming application. In word count, tweet analysis and algorithm use cases, we use time window operators resulting in higher latency values as tuples reside in the window for several minutes/hours before the corresponding output is generated at the end of the time window. However, in the fraud detection application, an alert is generated as soon as an event that follows the faulty pattern is received by the application resulting in lower latency compared to the streaming applications that use window operators. As indicated in the figure, latency is the lowest when checkpoints are disabled. When comparing the latency of all the use cases, latency becomes quite high for 30 minute checkpoint interval, but when the optimal checkpoint is used we can observe a significant decrease in the latency. For instance, in the word count application we observed 32.75% latency decrease when $\lambda = 0.0075$ and 58.19% latency decrease when $\lambda = 0.05$. For all the cases, the latency of the optimal checkpoint interval is quite close to the latency when checkpoints are disabled. For instance, the average latency of the fraud detection application when checkpoints are disabled is around 10.0 seconds and when the optimal checkpoint is used

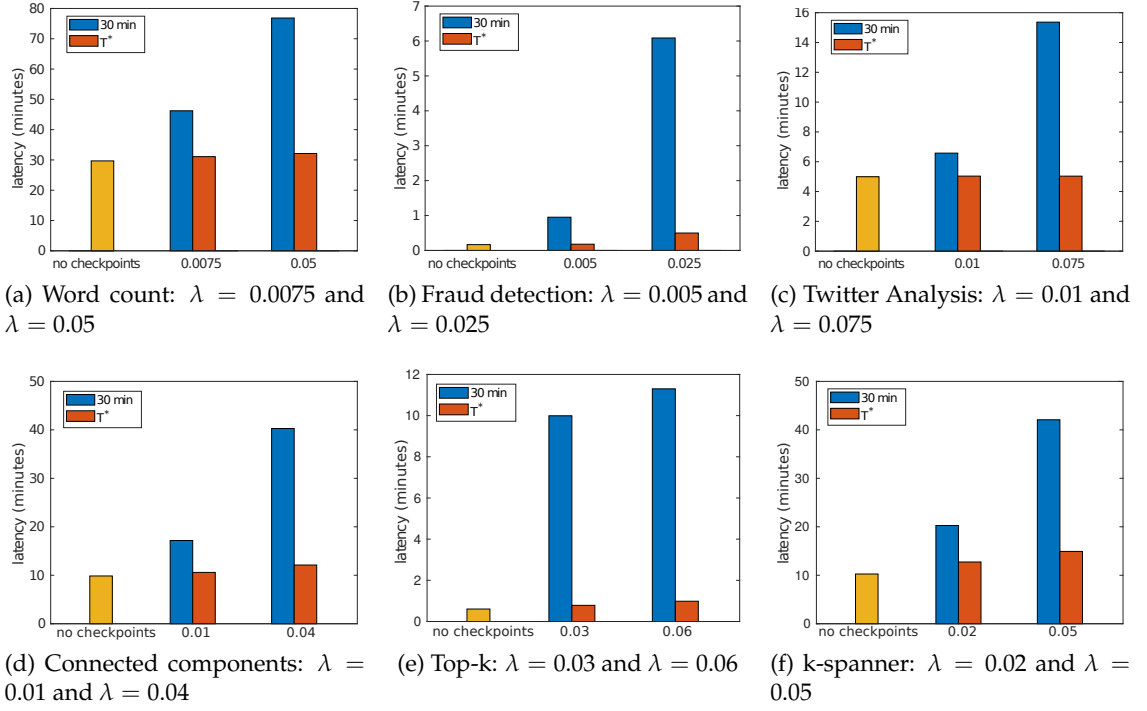


Figure 5.7: Latency comparison: $T = 30$ minutes, $T = T^*$ and without checkpoints.

latency becomes 10.7 and 30 seconds for failure rates 0.005 and 0.025 respectively. From our observation, it is clear that using the optimal checkpoint interval not only increases application utilization but also results in higher throughput and lower latency.

5.3.3 Parameters influencing checkpoint cost and restart cost

This section focuses on checkpoint costs and restart costs which directly influence the utilization of streaming applications and explores the factors that determine the values of the checkpoint and restart costs.

5.3.3.1 State size

Checkpoint cost is the main factor that determines the impact of the checkpointing on system performance. Figure 5.8a shows how utilization decreases with increasing checkpoint costs. Checkpoint cost depends on the state size and the write speed of the back-end storage system used to persist the checkpointed state. Therefore, large states result

in higher checkpoint costs. The state size varies based on the computations defined in the streaming application. For instance, if an application's task is to count the number of records processed, the state that needs to be checkpointed is the count which is relatively a small state having a size of few bytes. However, if an application has large windows such as a one day window, then the content of the whole window has to be checkpointed. Depending on the window size and the content of the tuples residing in the window, the state size could grow up to gigabytes. Figures 5.8b and 5.8c show how the checkpoint cost and state size increase as the window size grows.

The figure shows the checkpoint costs of Flink applications that use different window sizes. The state size grows with the window size and this results in increased checkpoint cost. For the window based Flink applications we tested, the throughput of the window operator is on average 55000 tuples per minute. Since we maintained a constant throughput for all different window sizes, the state size grows as the window size increases. However, it is important to note that window size depends on the input rate as well. Even if the window size is large, the state size can be small if the number of tuples residing in the window is very small due to low input rates.

As indicated in the figure, large states result in high checkpoint costs which degrades the utilization. Although state size depends on the computations operators are performing, optimizations can be done to reduce the state size. For example, Flink window operators receiving the same input can have different state sizes based on the function performed on the window tuples [169]. Window functions such as *reduce*, aggregates incoming tuples to one tuple per window and functions like *processWindow* keep individual tuples resulting in large states. If we consider a word count example, it can be performed using a *reduce* function that increment the count as input tuple arrives to the window without storing individual tuples or a *processWindow* function that keeps all the tuples and calculates the count at the end of the window. Therefore, carefully choosing the functions to use for a particular computation can reduce the checkpointing overhead.

Another approach stream processing applications can adopt to deal with large states is incremental checkpointing [111, 112] which can reduce the checkpoint cost. In incremental checkpoints, instead of persisting the complete state at the time of the checkpoint,

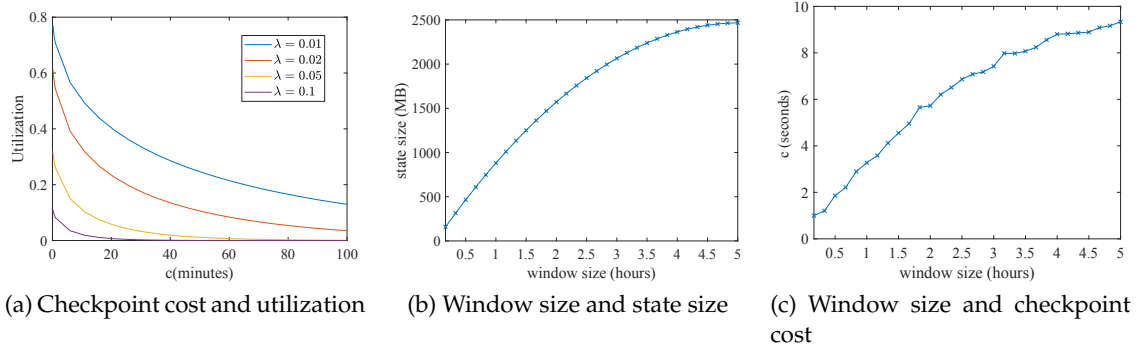


Figure 5.8: Checkpoint cost analysis.

only the changes to the previous checkpointed state is persisted. This can significantly reduce state size. For instance, in a window operator if only a small number of tuples arrived after the last checkpoint, then the difference in the state is small resulting in a small state size difference. Hence, incremental checkpoints can be more efficient when dealing with applications that have large states such as windows, large graphs. However, more effort needs to be put into creating the completed state by combining the checkpointed state differences in order to get the complete state to restore the state when a failure happens. Moreover, the incremental checkpointing approach will not have a significant performance improvement for applications where state changes often.

5.3.3.2 Restart cost

The restart cost, R is another factor that determines the utilization of a stream processing application. Figure 5.9a depicts how utilization changes with different R values. Although the optimal interval is independent of R , utilization decreases with increased R values. Therefore, having a lower R value is beneficial to the system performance.

The restart cost depends on several system specific parameters such as the time to detect a failure and the time for the system to read a completed checkpoint and restore the state. For instance, when an application fails, Flink waits for 10 seconds before restarting the application which is referred to as “fixed delay restart strategy” [169]. This delay is used to ensure that the connections or transactions with external systems can reach a timeout before the state is restored and the application starts processing. Therefore,

despite the failure type, a Flink application will at least take 10 seconds to restart. Furthermore, in real-world systems, the time to detect failures depends on the type of failure. In Flink, the time to restore after a task manager failure depends on the heartbeat interval of the task manager. However, application level exceptions are detected almost instantaneously, therefore the application only waits for the predefined 10 seconds before restarting. For instance, when the heartbeat interval is set to default 10 seconds along with the 10 seconds of delay for a restart after a failure, the restart cost we observed after a task manager failure is around 18 seconds and the restart cost after an application level exception is around 10.3 seconds.

In order to reduce the restart cost, users can update the system specific parameters such as decreasing the heartbeat interval. However, reducing the heartbeat interval can result in Flink task managers wrongly detected as killed due to delayed heartbeat messages. Therefore, system specific parameters have to be changed based on the application requirements and system settings such as cluster size and network delay. Figure 5.9b indicates how the restart cost varies with the heartbeat interval. The figure shows the minimum, average and maximum restart costs observed for different heartbeat intervals. As shown in the figure the average restart cost increases with the heartbeat interval and the maximum restart cost observed for each interval is similar to the value of the heartbeat interval. As indicated in the figure, the minimum restart cost could be comparatively low which is due to the heartbeat signal being sent soon after a failure occurs. The restart cost becomes the highest when a failure occurs soon after a heartbeat interval is sent. In that case, the system has to wait till the time the next heartbeat should be received to detect the failure.

5.3.4 Discussion

In practice, the checkpoint interval is often configured to nominal values such as 30 minutes or 1 hour [92]. Using these values as the checkpoint interval results in lower system utilization as these values are not based on application specific factors such as checkpoint cost. Therefore, the optimal checkpoint interval we derived based on the salient aspects of the checkpoint process can provide better system utilization. From the evaluations,

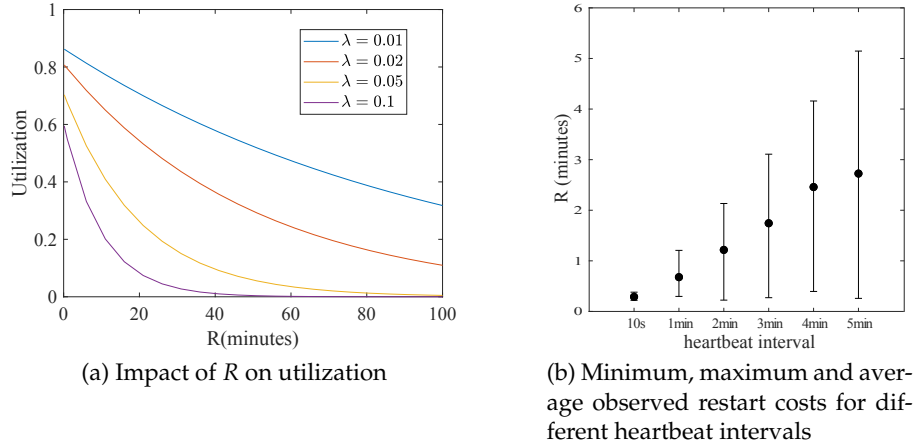


Figure 5.9: Restart cost analysis.

it is evident that we can always gain utilization improvements using the optimal checkpoint interval instead of using a nominal value. Furthermore, the theoretical utilization predicted by the model and the actual utilization of the experiments compares well, confirming the accuracy of our model when applied to practical systems. We also measured the latency and throughput of the tested applications as high throughput and low latency are common requirements of streaming applications. Our observations show that, in addition to the improved utilization, we can also achieve better latency and throughput using the optimal interval.

5.4 Conclusion

Checkpoint-based fault-tolerance is the most common fault-tolerance mechanism used in state-of-the-art stream processing systems. Although the efficiency of streaming applications is affected by the checkpoint periodicity, systems are often configured to perform checkpoints using a nominal value such as 30-minutes, resulting in performance degradation. To address this issue we have proposed a theoretical optimal interval that minimizes the overhead of checkpointing and this chapter investigates the impact of the optimal interval on application performance when applied to real-world systems. We presented an empirical study that confirms the benefits of using the optimal checkpoint interval, based on application specific parameters over the use of a nominal value. With

practical experimentation on Flink using common use cases, we have shown that the efficiency of streaming applications can be improved using the optimal interval in terms of latency, throughput as well as utilization. Furthermore, the observations confirm the accuracy of the theoretical utilization model used to determine the optimal checkpoint interval. The experiments have demonstrated the practical importance of selecting a checkpoint interval based on application specific parameters to minimize the overhead imposed by the checkpointing process and to satisfy low latency and high throughput requirements of stream processing applications.

Although the overhead of checkpointing can be minimized using the optimal checkpoint interval, checkpointing becomes too inefficient for systems with high failure rates. The next chapter looks into multi-level checkpointing that can be used to address the inefficiencies of the single-level checkpointing process.

Chapter 6

An Optimization for Interval-based Multi-level Checkpointing in Stream Processing Systems

6.1 Introduction

The checkpointing approach discussed in Chapters 4 and 5 can be referred to as single-level checkpointing because there is only one kind of checkpoint used over the entire system. Hence, the checkpoints should be able to recover from any failure types at any level of the system, from threads and processes to individual machines, groups of machines attached to a single power supply, and even entire racks, depicted in Figure 6.1. For example, failures caused by unhandled exceptions, out of memory and buffer overflow issues can affect several, but not all, processes of a machine while hardware failures affecting several machines can have more severe impact. In some high performance computing (HPC) systems, sets of machines are combined to form a mid-plane and a rack is formed by a set of mid-planes. Failures affecting this hardware can have varying forms of impacts. For instance, a mid-plane switch failure can affect all of the machines in the mid-plane while the rack's power supply failure can affect all of the machines in the rack [173, 174]. When a component fails, we assume that all of the state maintained by that component is lost. While the failure rate of any single component can be quite low,

This chapter is derived from:

- Sachini Jayasekara, Aaron Harwood, and Shanika Karunasekera. "Optimal Multi-Level Global Checkpointing for Exascale Stream Processing Systems," *IEEE Transactions on Parallel and Distributed Systems*, (Under review).

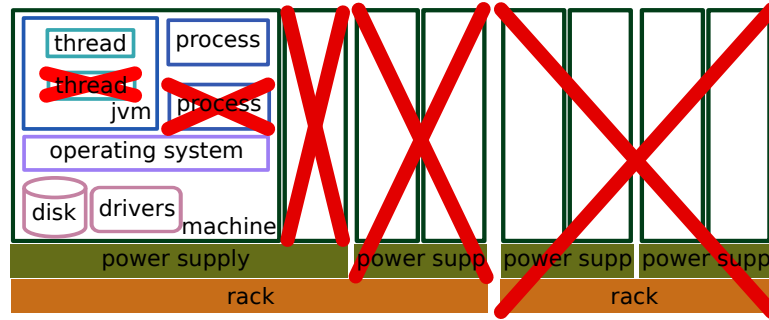


Figure 6.1: Different types of failure levels could include, from right to left: rack failure, power supply failure, machine failure (including e.g. disk drive failure), process failure, thread failure. Failures at the rack and power supply level are expected to happen (very much) less frequently compared to failures at the process and thread levels.

the aggregate failure rate readily becomes overwhelming.

Some HPC researchers have recently provided experimental results that show how checkpointing can be improved by considering different “levels” of failure, especially in Exascale systems that are vulnerable to several different failure types [123, 175, 125]. Each level has its own failure rate, checkpoint cost and restart cost. In this chapter, we propose a theoretical model to define optimal parameters for stream processing systems that use multiple levels of checkpoints to support fault-tolerance.

Multi-level checkpoints can be performed in several ways. One approach is, each level independently performing checkpoints at its own unique checkpoint interval. This approach is considered challenging to implement in real-world systems [176, 125] and the main issue is the occurrence of overlapping checkpoints [125], where checkpoints of different levels have to be performed at the same time. Another approach is to define a repeating pattern of checkpoints over the levels [125], e.g. for 2 levels: 1, 1, 2, 1, 1, 2, 1, 1, 2, . . . etc. In this approach only the pattern is repeated periodically. However, inside the pattern, different types of checkpoints can happen one after another with no computations in-between leading to aperiodic checkpoint intervals. Unlike the pattern-based model, the model we are proposing is for systems that use periodic checkpoints, where all the checkpoints are performed periodically based on a unique checkpoint interval. Our model is similar to the single-level periodic checkpointing approach used in existing systems with the exception of having different levels of checkpoints instead of one.

In this chapter, we talk about *low level* checkpointing being a checkpoint that can recover from thread and process failure, while *high level* checkpointing is one which can recover from more severe failure such as power supply and rack failure. A high level checkpoint can be used to recover from any of the failures at its own level or below, but this is not the case in the opposite direction. E.g. a thread or process can checkpoint to an in-memory checkpoint manager: checkpointing and recovery can be quite cheap but failure of the machine will need a higher level checkpoint to recover. Higher level checkpointing is more costly both in the time to save the checkpoint data and in the recovery, but higher level failures happen less often than lower level failures. Therefore the natural question in a multi-level checkpoint scheme is how often should we checkpoint at each level?

To date there is no multi-variable optimization solution that optimizes both the checkpoint selection, i.e. which checkpoint level to choose at each checkpoint interval, and the checkpoint interval. All the existing multi-level approaches focus on pattern-based checkpointing [176] and the optimal parameters given by the theoretical models of the pattern-based solutions have to be rounded to integer values to be used in practical systems which reduces the precision of the solution whereas the optimal values given by our interval-based model can be used in practical systems as it is.

6.1.1 Contribution

This chapter presents a theoretical framework for determining optimal parameter settings in a multi-level global checkpointing system that uses a single periodic checkpoint interval, applicable to Exascale stream processing. Our approach is stochastic, where at a given checkpoint interval, a level $l \in 1, 2, \dots, L$ is selected with some probability p_l for checkpointing, where $\sum_{l=1}^L p_l = 1$. We derive the optimal checkpoint interval, T^* , and associated optimal checkpoint probabilities, $p_1^*, p_2^*, \dots, p_L^*$ for a multi-level system, that considers as input: failure rates, $\lambda_1 > \lambda_2 > \dots > \lambda_L$, checkpoint costs $c_1 < c_2 < \dots < c_L$ and restart costs, $r_1 < r_2 < \dots < r_L$. Our derivation takes into account higher order effects such as failure during recovery and multiple failures during a single interval. Existing multi-level checkpointing models, such as proposed by Di et al. [124] for HPC

applications, minimize the total runtime of an application, given a bounded workload. In stream processing systems this is not applicable. To overcome this, we maximize *utilization*, $0 < U < 1$, of the system - the fraction of total time available to do useful work, which is applicable to an unbounded workload. We confirm our results with stochastic simulation and practical experimentation.

6.2 Probabilistic Checkpointing

In a single-level checkpointing process, the checkpointing cost and the restart cost remain the same and all failure types are recoverable through the stored checkpoints. For example, failures causing a process to die and failures causing an entire rack to go down are recovered from the same type of checkpoint. Checkpoints are usually stored in an external storage system as checkpoints have to be accessible after any type of a failure. This results in unnecessary recovery cost for less severe failures which could recover quickly using checkpoints persisted in-memory or machine local storage.

However, multi-level checkpointing consists of different types of checkpoints with varying checkpoint and restart costs. Different types of checkpoints are performed considering different types of failures and their failure rates. In this approach, as the checkpoint level goes up, the failure rate of the level becomes lower and the checkpoint and restart become more expensive. Existing work focuses on pattern-based multi-level checkpointing, where a pattern is repeated periodically and not necessarily every checkpoint being performed periodically.

In our work, we propose a probabilistic approach of multi-level checkpointing which can easily be adopted by existing stream processing systems. In the probabilistic approach, the system has one global checkpoint interval, while the checkpoint level that should be performed at the checkpoint time is defined by a discrete probability distribution over the levels. We formulate an analytical expression to determine the optimal checkpoint interval and the optimal probability distribution.

For a multi-level checkpointing process with L levels, let p_l be the probability of performing a level- $l \in \{1, 2, \dots, L\}$ checkpoint, c_l be the checkpointing cost and r_l be the

restart cost of level- l . Fig. 6.2 shows a multi-level checkpointing process with a checkpointing periodicity of T using three types of checkpoints with checkpointing probabilities, $p_1 = \frac{2}{3}$, $p_2 = \frac{2}{9}$, $p_3 = \frac{1}{9}$ and level-1 with c_1 checkpointing cost, level-2 with c_2 ($c_1 < c_2$) and level-3 with c_3 ($c_2 < c_3$). In this model, failures of any level can be recovered from checkpoints of the same level or any of the higher levels. For a level-3 checkpoint process, level-1 failures can be recovered from any checkpoint, level-2 failures can be recovered from a level-2 or a level-3 checkpoint while level-3 failures can only be recovered from a level-3 checkpoint. As lower level failures can be recovered from the same or higher level checkpoints, if a level- l failure occurs, we assume that the system restores from the nearest completed checkpoint which is of level l or higher. For instance, assume a case where the system performs a level-3 checkpoint and a level-1 failure occurring after the checkpoint. Since the last completed level-3 checkpoint can recover from level-1 failures, the system uses that checkpoint to recover and does not go back to the last completed level-1 checkpoint.

We model utilization of a system to determine the optimal checkpointing interval and optimal probabilities. Utilization, $0 < U < 1$, of a system is defined as the fraction of the systems time for which its resources are available to do useful work (to process load), as opposed to work done solely to maintain the systems operation, sometimes called overhead, which in our definition includes the overhead associated with loss and recovery from system failure. The work done by the system to create a checkpoint, and the work done by the system from the checkpoint time to the time taken to detect an occurred failure and successfully restart from the checkpoint is not useful work under our definition and thereby detracts from the utilization. The only useful work is therefore the work done, without failure, between two consecutive checkpoints (not including the work to create the checkpoints), or between a successful restart to the next checkpoint. In the next section, we start with the simplest form of multi-level checkpointing, 2-level checkpointing and subsequently consider L -level checkpointing in later sections.

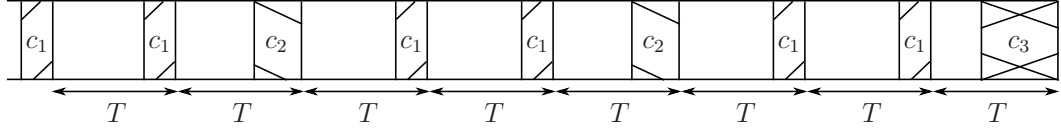


Figure 6.2: Multi-level checkpointing with 3 levels ($c_1 < c_2 < c_3$). The cost of the checkpoint operation is shown as a hatched area, the remaining time is useful time available for utilization.

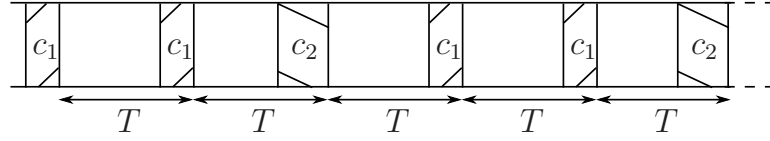


Figure 6.3: 2-level checkpointing with period T , $p_1 = \frac{2}{3}$, overhead c_1 , $p_2 = \frac{1}{3}$, overhead c_2 and no failures occurring.

6.3 2-level Checkpointing Utilization Model

In this section, we provide a derivation of our utilization model for 2-level checkpointing from first principles. A 2-level checkpointing process consists of two levels with each level having an independent failure rate, checkpoint cost and restart cost. Level-1 failures can recover from a level-1 or a level-2 checkpoint while level-2 failures can recover only from a level-2 checkpoint. Therefore, we assume that in case of a level-1 failure the system restarts from the last completed checkpoint which could be a level-1 or a level-2 and in case of a level-2 failure, the system restarts from the last completed level-2 checkpoint. For example, a system can maintain level-1 checkpoints as static in-memory objects which can be used to restore after a thread failure and use an external storage to persist level-2 checkpoints which can recover from more severe failures. In this approach checkpoints stored in both external storage and in-memory can be used to recover from a thread failure and if a machine fails then the state can only be restored from the external storage.

6.3.1 Without failure

Consider a system that performs two-level checkpointing with a constant periodicity of T seconds where the probability of performing a level-1 checkpoint is p_1 and the probability of level-2 checkpoint is p_2 . Let $0 < c_1 < c_2 < T$ be abstract constant costs for level-1 and

level-2 checkpointing, here expressed without loss in generality as to how or when the checkpoint is created, in units of time, which without failure as shown in Fig. 6.3 leads to an expression for expected utilization:

$$U = \frac{T - p_1 c_1 - p_2 c_2}{T}. \quad (6.1)$$

The value of U is what we would expect to observe given a sufficiently large number of checkpoint intervals. In our work we depict the checkpoint as being created in the last c_1 or c_2 seconds of the period T based on the checkpoint level.

6.3.2 With failure and negligible restart cost

Similar to prior studies [176, 125, 120, 124, 94], we model failure of level $l \in \{1, 2\}$ as a series of independent failure events having an exponential inter-arrival time, given by failure rate λ_l , where the probability of a level- l failure at time t is $\lambda_l e^{-\lambda_l t}$, the probability of failure by time t is $1 - e^{-\lambda_l t}$. Let $\Lambda = \lambda_1 + \lambda_2$ be the combined failure rate of both failure levels, in which case the probability of failure considering failures of both levels at time t is $\mathbb{P}[\mathbf{X} = t; \Lambda] = \Lambda e^{-\Lambda t}$, the probability of failure by time t is $\mathbb{P}[\mathbf{X} < t; \Lambda] = 1 - e^{-\Lambda t}$, and the mean time to failure is $\mathbb{E}[\mathbf{X}] = \int_0^\infty t \mathbb{P}[\mathbf{X} = t; \Lambda] dt = \frac{1}{\Lambda}$.

Assume that the system has just completed a checkpoint period of either level. Either no failure happens within the next time period T , with probability $e^{-\Lambda T}$, in which case the system successfully computes the next checkpoint, or failure of either level happens within time T as indicated in Fig. 6.4, 6.5 and the system needs to restart from an existing checkpoint. Given that a failure of level-1 or level-2 does happen within time T , then the mean time to failure, $F_\Lambda(T)$ is:

$$\begin{aligned} F_\Lambda(T) = \mathbb{E}[\mathbf{X} | \mathbf{X} < T] &= \frac{\int_0^T t \mathbb{P}[\mathbf{X} = t; \Lambda] dt}{\mathbb{P}[\mathbf{X} < T; \Lambda]} \\ &= \frac{e^{T\Lambda} - T\Lambda - 1}{\Lambda(e^{T\Lambda} - 1)}. \end{aligned} \quad (6.2)$$

The value for $F_\Lambda(T)$ gives the average amount of time lost from the last completed checkpoint if a failure happens, not having restarted yet. In this section, for now, we

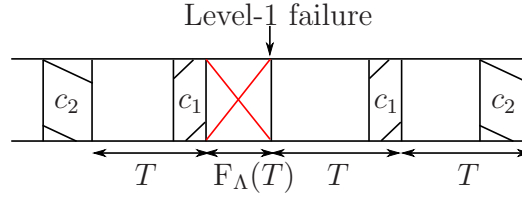


Figure 6.4: Level-1 failure resulting in lost time $F_{\Lambda}(T)$ and instantaneous recovery followed by a successful period T .

assume the time to detect the failure and to restart is negligible. We include this lost time due to a single failure into our utilization model from (6.1) by expressing the effective period, $T_{eff} = T + F_{\Lambda}(T)$, and writing:

$$U = \frac{T - p_1 c_1 - p_2 c_2}{T_{eff}}.$$

Although failure of either level results in $F_{\Lambda}(T)$ lost time, level-2 failures can result in additional lost time as depicted in Fig. 6.5. As level-2 failures can only recover from level-2 checkpoints, the time lost due to a level-2 failure includes the time from the completion of the last level-2 checkpoint and the occurrence of the level-2 failure. This lost time can include zero or more level-1 checkpoints. For instance in Fig. 6.3, if a level-2 failure occurs in f_1, f_2, f_3 then 0, 1, 2 completed level-1 checkpoints are lost respectively in addition to $F_{\Lambda}(T)$. No checkpoints would be lost if the last completed checkpoint is a level-2 checkpoint which can happen with probability p_2 , the probability of losing a single level-1 checkpoint is $p_1 p_2$ and the probability of losing i consecutive level-1 checkpoints is $p_1^i p_2$, which results in an average $\frac{p_1}{p_2} = \frac{p_1}{(1-p_1)}$ lost completed checkpoints. Therefore, apart from the $F_{\Lambda}(T)$ lost due to any failure, level 2 failures result in additional $T_{eff} \frac{p_1}{1-p_1}$ lost time, where T_{eff} indicates the average time taken to complete a lost completed level-1 checkpoint.

Figs. 6.4 and 6.5 depict only a single failure occurrence followed by a successful period. In general, the number of consecutive failures that could occur, k , before a successful period, is unbounded and selected from the set $k \in \{0, 1, 2, \dots\}$ at random with a geometric distribution, $(1-q)^k q$, having parameter $q = q_{T,\Lambda} = \mathbb{P}[\mathbf{X} \geq T; \Lambda] = 1 - \mathbb{P}[\mathbf{X} < T; \Lambda]$, giving an average number of consecutive failures of any of the two levels, $\frac{1-q_{T,\Lambda}}{q_{T,\Lambda}}$. From

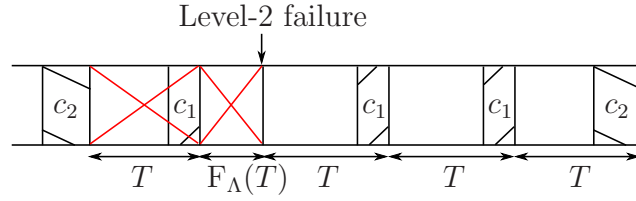


Figure 6.5: Level-2 failure resulting in lost time $T + F_{\Lambda}(T)$ and followed by a successful period T .

the average number of consecutive failures, the proportion of level-1 failures is $\frac{\lambda_1}{\Lambda}$ and the proportion of level-2 failures is $\frac{\lambda_2}{\Lambda}$. We can include this lost time into our utilization model by expressing the effective period:

$$\begin{aligned}
 T_{eff} &= T + \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \frac{\lambda_2}{\Lambda} \left(\frac{T_{eff} p_1}{1 - p_1} \right) \right) \\
 &= \frac{T + \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} F_{\Lambda}(T)}{1 - \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} \left(\frac{\lambda_2 p_1}{\Lambda(1 - p_1)} \right)}
 \end{aligned}$$

6.3.3 Including the time to detect and recover from failure

This section includes the time to detect and recover from a failure as shown in Fig. 6.6 and 6.7. Every failure requires a restart and the restart cost depends on the failure type. Restart cost r_1 is the cost of restarting from a level-1 checkpoint and r_2 is the cost of restarting from a level-2 checkpoint. As level-1 failures can recover from either a level-1 or level-2 checkpoint, the restart cost depends on the last completed checkpoint type before the failure. For example, if a level-1 failure occurs after a level-2 checkpoint similar to the first failure shown in Fig. 6.6, then the restart cost is r_2 . If a level-1 failure occurs after a level-1 checkpoint as the second failure shown in Fig. 6.6, then the restart cost is r_1 . Since the probability of the last completed checkpoint before a level-1 failure being a level-1 is p_1 , we can write the expected recovery cost of a level-1 failure, $R_1 = p_1 r_1 + p_2 r_2$. However, for level-2 failures the restart cost is always r_2 as level-2 failures can only

recover from a level-2 checkpoint which is depicted in Fig. 6.7. This leads to:

$$T_{eff} = T + \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \frac{\lambda_1(p_1 r_1 + p_2 r_2)}{\Lambda} + \frac{\lambda_2}{\Lambda} \left(\frac{T_{eff} p_1}{1-p_1} + r_2 \right) \right)$$

$$= \frac{T + \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \frac{\lambda_1(p_1 r_1 + p_2 r_2)}{\Lambda} + \frac{\lambda_2 r_2}{\Lambda} \right)}{1 - \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(\frac{\lambda_2 p_1}{\Lambda(1-p_1)} \right)}.$$

However, we note that failure may also occur *during* the restart, in which case we assume that the restart must itself start again. Similarly to the number of consecutive failures, the number of attempts to restart, $r \in \{1, 2, \dots\}$, is selected at random using a geometric distribution (note that following a failure at least 1 restart is always required), $(1-q)^{r-1}q$, with parameter q assuming that the failures of same level, l or lower levels can occur during r_l as restart costs are small and the failure rates of higher levels are very low resulting in almost no higher level failures during r_l . Let $q = q_{r_l, \Lambda_l} = \mathbb{P}[\mathbf{X} \geq r_l; \Lambda_l]$, leading to an average number of restarts $\frac{1}{q_{r_l, \Lambda_l}} \geq 1$, where $\Lambda_l = \sum_{i=1}^l \lambda_i$. For any given failed restart attempt, given that we know the failure occurred within the restart time r_l we know from (6.2) that the average time lost is $F_{\Lambda_l}(r_l)$. Therefore, the average number of restarts during r_1 is $\frac{1}{q_{r_1, \Lambda_1}}$ with an average time lost of $F_{\Lambda_1}(r_1)$, and the average number of restarts during r_2 is $\frac{1}{q_{r_1, \Lambda_1 + \Lambda_2}} = \frac{1}{q_{r_1, \Lambda}}$ with an average time loss of $F_{\Lambda_1 + \Lambda_2}(r_2) = F_{\Lambda}(r_2)$. Therefore, the recovery cost of a level-1 failure, R_1 and the recovery cost of a level-2 failure, R_2 can be written as:

$$R_1 = p_1 \left(r_1 + \left(\frac{1}{q_{r_1, \Lambda_1}} - 1 \right) F_{\Lambda_1}(r_1) \right) + p_2 \left(r_2 + \left(\frac{1}{q_{r_2, \Lambda}} - 1 \right) F_{\Lambda}(r_2) \right),$$

$$R_2 = r_2 + \left(\frac{1}{q_{r_2, \Lambda}} - 1 \right) F_{\Lambda}(r_2).$$

This leads to:

$$T_{eff} = T + \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \frac{\lambda_1}{\Lambda} R_1 + \frac{\lambda_2}{\Lambda} \left(\frac{T_{eff} p_1}{1-p_1} + R_2 \right) \right)$$

$$= \frac{T + \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \frac{\lambda_1 R_1 + \lambda_2 R_2}{\Lambda} \right)}{1 - \frac{1-q_{T,\Lambda}}{q_{T,\Lambda}} \left(\frac{\lambda_2 p_1}{\Lambda(1-p_1)} \right)},$$

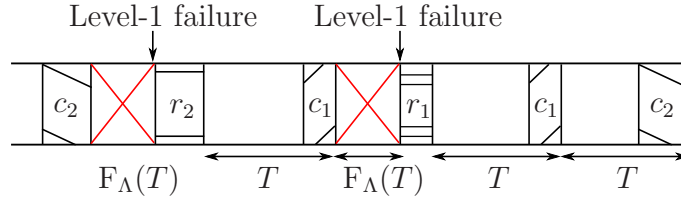


Figure 6.6: Level-1 failure resulting in lost time $F_\Lambda(T)$ and recovery time r_1 or r_2 followed by a successful period T .

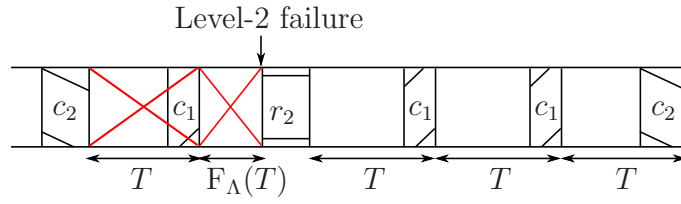


Figure 6.7: Level-2 failure resulting in lost time $T + F_\Lambda(T)$ and recovery time r_2 followed by a successful period T .

and finally:

$$\begin{aligned}
 U &= \frac{T - p_1 c_1 - p_2 c_2}{T_{eff}} \\
 &= \frac{\Lambda (\lambda_2 (p_1 e^{T\Lambda} - 1) - \lambda_1 p_2) (T - c_1 p_1 - c_2 p_2)}{(1 - e^{T\Lambda}) p_2 (e^{r_2 \Lambda} (\lambda_2 + \lambda_1 p_2) - \lambda_2 p_1 + \Lambda p_1 e^{\lambda_1 r_1})} \quad (6.3)
 \end{aligned}$$

This completes the salient features of our 2-level checkpoint and restart system model for a single process. We adapt this to a distributed stream processing system in Section 6.4.

6.3.4 Optimization of utilization

Unlike the single level case, that involves just T as the free parameter, deriving closed form equations for the optimal values of multi-level checkpointing that maximize U , i.e. for 2-level case by solving $\frac{\partial U}{\partial T} = 0$ and $\frac{\partial U}{\partial p_1} = 0$, becomes intractable as we involve more levels due to the non-linearities present in the expression for U . The optimal values, T^* and p_1^* can also be found by using nonlinear optimization functions such as *fmincon* in MATLAB. Fig. 6.8 shows the plot of partial derivatives for $\lambda_1 = 24, \lambda_2 = 0.4$ per day, $c_1 = R_1 = 10$ and $c_2 = R_2 = 30$ seconds. As indicated in the figure, there is only one intersection point which gives the optimal parameters. For a two-level checkpointing

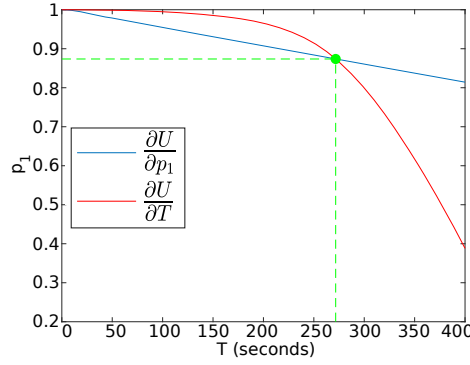


Figure 6.8: $\frac{\partial U}{\partial T} = 0, \frac{\partial U}{\partial p_1} = 0$ for $\lambda_1 = 24, \lambda_2 = 0.4$ per day, $c_1 = r_1 = 10, c_2 = r_2 = 30$ seconds. Intersection point shows T^* and p_1^* .

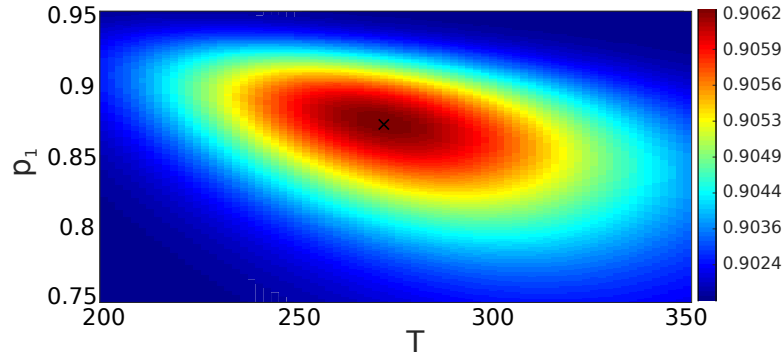


Figure 6.9: The optimal U sits at the peak of a nearly flat plateau which is shaped by failure rate, checkpoint cost and restart cost ($\lambda_1 = 24, \lambda_2 = 0.4$ per day, $c_1 = r_1 = 10$ and $c_2 = r_2 = 30$ seconds).

process with these parameters, optimal values are; $T^* = 271.6709$ seconds and $p_1^* = 0.8737$.

Fig. 6.9 shows how U changes with T and p_1 for $\lambda_1 = 24, \lambda_2 = 0.4$ per day, $c_1 = r_1 = 10$ and $c_2 = r_2 = 30$ seconds. Utilization is highest in the dark red colored area and the highest utilization is marked by $\times$. However, investigating how T^* and p^* vary with varying shape parameters λ_1 and λ_2 , as shown in Fig. 6.10, reveals a regime change when we consider $\lambda_1 \gg \lambda_2$. Similar findings appear with respect to shape parameters r_1 and r_2 , i.e. we can consider negligible restart cost. In this regime, which is indeed intuitively the more likely operating regime for a system, we can simplify (6.3) and obtain through

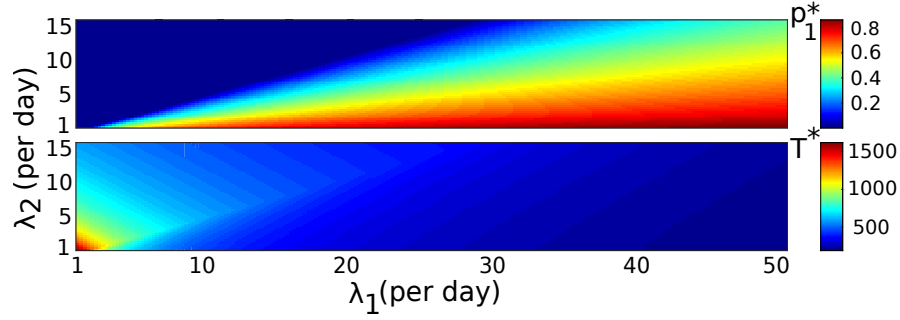


Figure 6.10: T^* and p_1^* for $c_1 = r_1 = 10$ and $c_2 = r_2 = 30$ seconds.

differentiation:

$$T^* \approx c_1 p_1 + c_2(1 - p_1) + \frac{W(-e^{c_2 \lambda_1 p_1 - c_1 \lambda_1 p_1 - c_2 \lambda_1 - 1}) + 1}{\lambda_1} \quad (6.4)$$

$$p_1^* \approx 1 - \sqrt{\frac{\lambda_2 (T - c_1) (e^{T \Lambda} - 1)}{(c_2 - c_1) (\lambda_1 + \lambda_2 e^{T \Lambda})}} \quad (6.5)$$

where $W(z)$ is the Lambert W function on the principal branch. These approximations appear to be quite robust, where for all the values of λ_1, λ_2 values shown in Fig. 6.10, the maximum difference of the utilization using the actual optimal values and the approximate values given by (6.4) and (6.5) is 0.0046.

6.3.5 Comparison to stochastic simulation

We developed a stochastic simulation to support the derived model. The simulation generates random failures based on an exponential distribution for both levels and calculates the utilization taking the randomly generated failures into account. Fig. 6.11 shows the utilization comparison between our model based on (6.3) and the simulation results for different failure rates, checkpointing costs and restart costs. The solid lines represent theoretical utilization and the data points with error bars represent the average utilization and the standard deviation of 100 runs of the simulation with each simulation running for $\frac{1000}{\lambda_2}$ days.

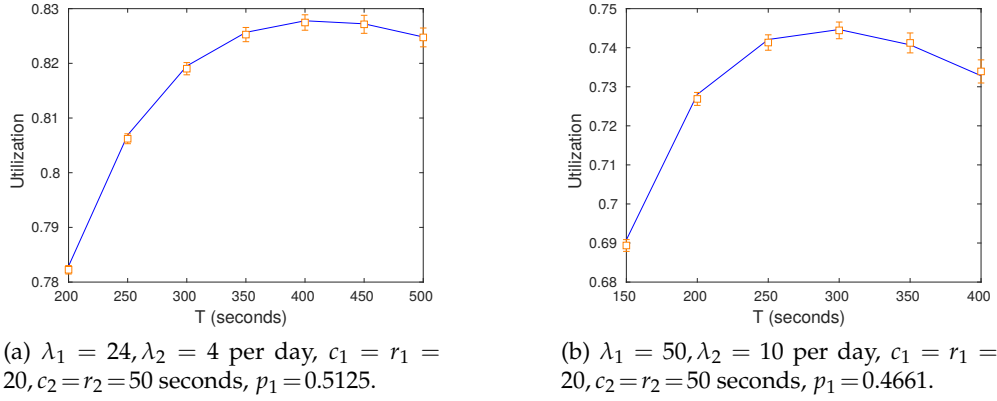


Figure 6.11: Utilization of 2-level checkpointing. Solid lines represent theoretical utilization and data points with error bars represent the average utilization and the standard deviation.

6.3.6 Comparison to 1-level checkpointing

Fig. 6.12 shows the maximum utilization that can be achieved by a 2-level checkpointing process for different T values indicated in blue solid lines and the utilization of a 1-level checkpointing process indicated in red dashed lines. In the 1-level process, $p_1 = 0$, failure rate is $\lambda_1 + \lambda_2$ and the system only performs level-2 checkpoints. As shown, using 1-level checkpointing for a system with the same failure rate ($\lambda_1 + \lambda_2$) and performing only a single checkpoint type instead of performing 2 levels of checkpoints leads to lower utilization. Furthermore, if λ_2 is low and c_2, r_2 values are comparatively higher than the costs of level-1, then having 2 levels gives noticeable improvements in the optimal utilization as shown in Fig. 6.12b.

Moreover, optimal p_1 decreases with T and for large T values optimal p_1 comes closer to 0. Therefore, the utilization of 2-level checkpointing becomes closer to the utilization of 1-level checkpointing as shown in the figure.

Table 6.1 shows p_1^*, T^*, U for optimal values, maximum U for 1-level where $p_1 = 0$ and the percentage increase in utilization $\%U$ when using 2-level over 1-level for $\lambda_1 = 50$ per day, $c_1 = r_1 = 20, c_2 = r_2 = 50$ seconds and different λ_2 values. As indicated, p_1 decreases as λ_2 increases, and the achievable utilization increases using 2-level checkpointing decreases as the values of λ_2 and λ_1 become closer.

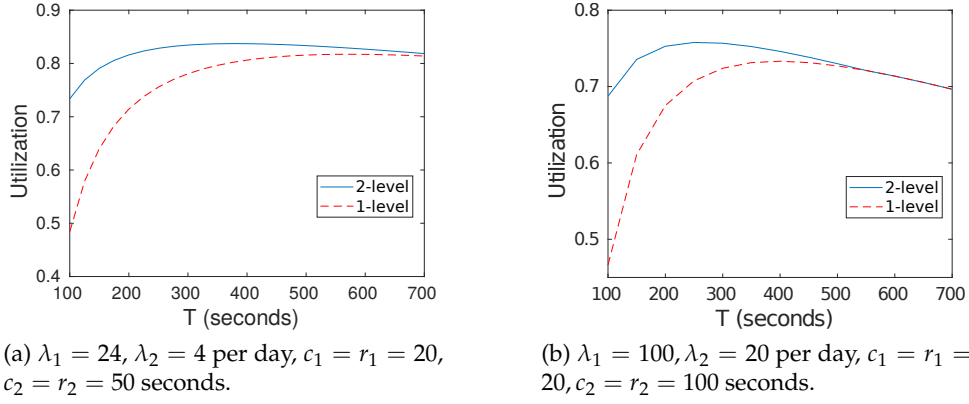


Figure 6.12: Comparison of 2-level and 1-level checkpointing. Blue solid lines are the utilization of 2-level checkpointing and red dashed lines are the utilization of 1-level checkpointing.

Table 6.1: Impact of λ_2 to p_1^* , T^* , and U for $\lambda_1 = 50$ per day, $c_1 = r_1 = 20$, $c_2 = r_2 = 50$ seconds.

λ_2	p_1^*	T^*	U	U single-level	% U increase
5	0.6408	276.0128	0.7712	0.7444	3.6088
1	0.8439	268.3256	0.8106	0.7537	7.5449
0.5	0.8897	268.0672	0.8206	0.7549	8.6943
0.1	0.9506	268.5856	0.8342	0.7559	10.3654
0.01	0.9844	269.3067	0.8420	0.7561	11.3593
0.001	0.9950	269.6005	0.8445	0.7561	11.6842

6.4 2-level Checkpointing for Stream Processing

In this section, we extend the model to work with a distributed stream processing system. Stream processing applications are usually represented as a directed acyclic graph (DAG) of operators where each operator performs computations on the incoming data streams and outputs the results as an output stream to subsequent operators in the DAG. An operator can have multiple operator instances which perform computations on the input data in parallel. Existing stream processing systems use a token based approach to perform checkpointing where a token is sent periodically from source operators till

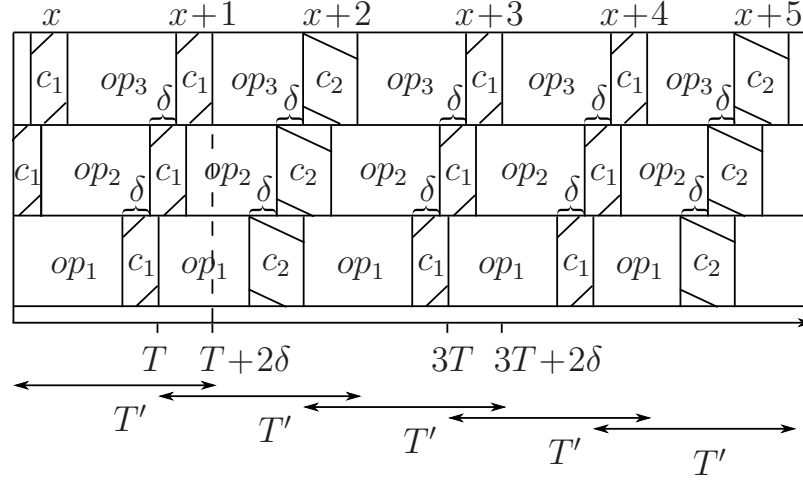


Figure 6.13: 2-level checkpointing of a streaming application with 3 operators. x to $x+5$ are the ids of completed checkpoints.

it reaches sink operators. At the arrival of the token, each operator starts performing the checkpoint and passes the token to the next operator or operators. The checkpoint is fully completed once all the operators in the DAG complete the checkpoint. In case of a failure, all the operators restore the last fully completed checkpoint and if there is a partially completed checkpoint, i.e. a checkpoint completed by some of the operators in the DAG is discarded. Furthermore, failure recovery process is the same for any type of failure which could occur in a single operator instance, all of the instances of the same operator or a set of operators. In our model for stream processing systems, we assume that all operators are stateful and each level of all the operators have same checkpointing costs $0 < c_1 < c_2 < T$ and same restart costs, $r_1 < r_2$.

Fig. 6.13 shows 2-level checkpointing for a stream processing application with 3 operators. δ is the time between sending the token from one operator and receiving the token by the next operator in the DAG. As this DAG consists of 3 operators, the time taken from starting the checkpoint by first operator and completion of the checkpoint by the last operator is $c_1 + 2\delta$ or $c_2 + 2\delta$. Therefore, the time to fully complete a checkpoint from the start of computation is $T + 2\delta$. If the longest path between all source and sink operators (critical path) in the DAG consists of n operators including the source and the sink, then, without considering failure, we have $T_{eff} = T' = T + (n - 1)\delta$ and we can

write expected utilization as:

$$U = \frac{T - p_1c_1 - p_2c_2}{T_{eff}} = \frac{T - p_1c_1 - p_2c_2}{T + (n-1)\delta}. \quad (6.6)$$

6.4.1 Including failure and recovery cost

In our streaming model, when a level-1 failure occurs then all of the operators are restored from the last fully completed checkpoint which could be a level-1 or a level-2 checkpoint. When a level-2 failure occurs then all of the operators restore from the last fully completed level-2 checkpoint. If at least one of the operators in the DAG is still completing a checkpoint at a failure occurrence, that checkpoint is discarded and the system is restored from a fully completed checkpoint.

Similar to sections 6.3.2 and 6.3.3, we include the time due to failures during T' and failures during recovery (r_1, r_2) to improve the accuracy of T_{eff} . Similar to the number of consecutive failures, within time T for a single operator the number of attempts to finish T' , $k' \in \{1, 2, \dots\}$, is selected at random using a geometric distribution, $(1-q)^{k'}q$, with parameter $q = q_{T', \Lambda} = \mathbb{P}[\mathbf{X} \geq T'; \Lambda] = 1 - \mathbb{P}[\mathbf{X} < T'; \Lambda]$, leading to an average number of consecutive failures $\frac{1-q_{T', \Lambda}}{q_{T', \Lambda}}$. Each consecutive failure loses on average an additional $F_{\Lambda}(T')$ time. Similarly to the number of consecutive failures, the average number of restarts for recovering from a level-1 checkpoint is $\frac{1}{q_{r_1, \Lambda_1}} \geq 1$ and the average time lost due to a failure during restart is $F_{\Lambda_1}(r_1)$. And for recovering from a level-2 checkpoint, the average number of restarts is $\frac{1}{q_{r_2, \Lambda_1 + \Lambda_2}} \geq 1$ and the average time lost due to a failure during restart is $F_{\Lambda_1 + \Lambda_2}(r_2)$. Therefore, the recovery cost of a level-1 failure, R_1 and the recovery cost of a level-2 failure, R_2 can be written as:

$$R_1 = p_1 \left(r_1 + \left(\frac{1}{q_{r_1, \Lambda_1}} - 1 \right) F_{\Lambda_1}(r_1) \right) + p_2 \left(r_2 + \left(\frac{1}{q_{r_2, \Lambda}} - 1 \right) F_{\Lambda}(r_2) \right),$$

$$R_2 = r_2 + \left(\frac{1}{q_{r_2, \Lambda}} - 1 \right) F_{\Lambda}(r_2).$$

This leads to:

$$\begin{aligned}
 T_{eff} &= T' + \frac{1 - q_{T',\Lambda}}{q_{T',\Lambda}} \left(F_{\Lambda}(T') + \frac{\lambda_1}{\Lambda} R_1 + \frac{\lambda_2}{\Lambda} \left(\frac{T_{eff} p_1}{1 - p_1} + R_2 \right) \right) \\
 &= \frac{T' + \frac{1 - q_{T',\Lambda}}{q_{T',\Lambda}} \left(F_{\Lambda}(T') + \frac{\lambda_1 R_1 + \lambda_2 R_2}{\Lambda} \right)}{1 - \frac{1 - q_{T',\Lambda}}{q_{T',\Lambda}} \left(\frac{\lambda_2 p_1}{\Lambda(1 - p_1)} \right)}. \tag{6.7}
 \end{aligned}$$

6.4.2 Including the overlap between consecutive checkpoints

Even though the first operator of a DAG completes its checkpoint at T , it takes $T' = T + (n - 1)\delta$ time for the whole DAG to complete a checkpoint. Therefore, even if the first operator of the DAG starts computation after it finishes its checkpoint, if a failure occurs between the start of the computation and $(n - 1)\delta$, then the DAG has to restart from the checkpoint completed by all the operators, not the checkpoint completed only by the first operator. For instance in Fig. 6.13, if a level-1 failure occurs during T and $T + 2\delta$ then the DAG has to restore from checkpoint x not from $x + 1$ as op_2, op_3 are not finished with checkpoint $x + 1$. However, if a level-1 failure happens after $T + 2\delta$, then the DAG can restore its state from $x + 1$ checkpoint.

During the first $(n - 1)\delta$ period of T' leading to checkpoint $x + 1$, some operators of the DAG are still performing checkpoint x . Therefore, recovery due to failure during the first $(n - 1)\delta$ of a T' is the same as if the failure occurs during the previous T' . As there is an overlap between the first $(n - 1)\delta$ of T' with the previous T' as shown in Fig. 6.13, for one T' , we only have to consider the time between $(n - 1)\delta$ and T' , ignoring the first $(n - 1)\delta$ of T' which is represented by the previous T' . The average number of consecutive failures before completing the first $(n - 1)\delta$ of T' is $\frac{1 - q_{(n-1)\delta,\Lambda}}{q_{(n-1)\delta,\Lambda}}$, where $q = q_{(n-1)\delta,\Lambda} = \mathbb{P}[\mathbf{X} \geq (n - 1)\delta; \Lambda]$. Taking all of this into account, the effective period to complete $(n - 1)\delta$ is:

$$(n - 1)\delta + \frac{1 - q_{(n-1)\delta,\Lambda}}{q_{(n-1)\delta,\Lambda}} \left(F_{\Lambda}((n - 1)\delta) + \frac{\lambda_1}{\Lambda} R_1 + \frac{\lambda_2}{\Lambda} \left(\frac{T_{eff} p_1}{1 - p_1} + R_2 \right) \right).$$

Subtracting this from T_{eff} in (6.7) to avoid the duplicate representation of the overlapping

time between two consecutive T' leads to:

$$U = \frac{T - p_1 c_1 - p_2 c_2}{T_{eff}} = e^{-\delta \Lambda (n-1)} \Lambda (T - c_1 p_1 - c_2 p_2) \left(\frac{\Lambda - \lambda_1 p_1 - \lambda_2 p_1 (e^{\Lambda(T+\delta(n-1))} - e^{\delta \Lambda (n-1)} + 1)}{p_2 (e^{T\Lambda} - 1) (e^{r_2 \Lambda} (\lambda_2 + \lambda_1 p_2) - \lambda_2 p_1 + \Lambda p_1 e^{\lambda_1 r_1})} \right). \quad (6.8)$$

This completes the salient features of 2-level checkpointing model for a distributed stream processing system.

6.4.3 Optimization of utilization

The values of T and p_1 that maximize the utilization U from (6.8), T^* and p_1^* can be found by using nonlinear optimization functions such as *fmincon* in MATLAB. Similar to the single process, investigating how T^* and p_1^* vary with varying shape parameters λ_1 and λ_2 reveals a regime change when we consider $\lambda_1 \gg \lambda_2$. Similar findings appear with respect to shape parameters r_1 and r_2 , i.e. we can consider negligible restart cost. In this regime, we can simplify (6.8) and obtain through differentiation:

$$T^* \approx c_1 p_1 + c_2 (1 - p_1) + \frac{W(-e^{c_2 \lambda_1 p_1 - c_1 \lambda_1 p_1 - c_2 \lambda_1 - 1}) + 1}{\lambda_1} \quad (6.9)$$

$$p_1^* \approx 1 - e^{\frac{\delta n \Lambda}{2}} \sqrt{\frac{\lambda_2 (T - c_1) (e^{T\Lambda} - 1)}{(c_2 - c_1)(\Lambda e^{\delta \Lambda} - \lambda_2 e^{\delta n \Lambda} + \lambda_2 e^{\Lambda(T+\delta n)})}} \quad (6.10)$$

where $W(z)$ is the Lambert W function on the principal branch. Interestingly approximate T^* is identical to that of a single process, but the approximate p_1^* is dependent of n and δ .

We can also consider T^* and p_1^* values of a single operator as approximations for T^* and p_1^* of a streaming application with same parameters as δ and n values have a very low influence on T^* and p_1^* . For example T^* and p_1^* for a single operator with parameters $\lambda_1=24, \lambda_2=0.4$ per day, $c_1=r_1=10, c_2=r_2=30$ seconds is $[271.6709, 0.8737]$ and for same parameters with $\delta = 0.5$ seconds and $n = 5, n = 50, n = 500$, T^* and p_1^* values

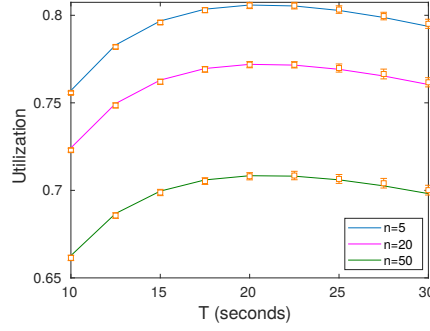


Figure 6.14: Utilization of 2-level checkpointing for a streaming application for $\lambda_1 = 432$, $\lambda_2 = 43.2$ per day, $c_1 = r_1 = 1$, $c_2 = r_2 = 5$, $\delta = 0.5$ seconds, $p_1 = p_1^*$ and different n values.

are $[271.6892, 0.8737]$, $[271.6934, 0.8733]$ and $[271.9213, 0.8691]$ respectively.

6.4.4 Comparison to stochastic simulation

Fig. 6.14 shows the utilization comparison between our model based on (6.8) and the simulation results for 2-level checkpointing using DAGs with different critical path lengths. The solid lines are theoretical utilization while the data points and error bars represent the average utilization and the standard deviation observed after 100 runs of the simulation, with each simulation running for $\frac{1000}{\lambda_2}$ days. As indicated in the figure, for constant level-1 and level-2 error rates, utilization decreases with the value of n .

6.5 L -level checkpointing Utilization Model

We now consider the more general L -level checkpointing process, that consists of L failure levels with each level having an independent failure rate, checkpoint cost and restart cost. As the level goes up from 1 to L , failure rate goes down ($\lambda_1 > \lambda_2 > \dots > \lambda_L$) and checkpoint cost and restart cost goes up ($c_1 < c_2 < \dots < c_L, r_1 < r_2 < \dots < r_L$). In this checkpointing approach, level- l failures can be recovered from checkpoints from level- l to level- L . The model assumes that level- l failures are recovered from the latest completed level- l or higher level checkpoint. For example, any level-1 failure can be recovered using any of the checkpoints and therefore, uses the last completed checkpoint to recover from failure. Similar to two-level checkpointing, in L -level checkpointing, check-

points are performed with a constant periodicity of T , and probability of performing a level- l checkpoint being p_l . Hence without failures we can write the expected utilization as:

$$U = \frac{T - \sum_{l=1}^L p_l c_l}{T}$$

6.5.1 Including failure and recovery cost

Similar to sections 6.3.2 and 6.3.3, we include the time due to failures during T and failures during recovery to improve the accuracy of T_{eff} . The average number of consecutive failures is $\frac{1 - q_{T', \Lambda}}{p_{T', \Lambda}}$, where $\Lambda = \sum_{l=1}^L \lambda_l$. Each consecutive failure loses on average an additional $F_{\Lambda}(T)$ time. Similarly to the number of consecutive failures, the average number of restarts for recovering from a level- l checkpoint is $\frac{1}{q_{r_l, \sum_{i=1}^l \lambda_i}} \geq 1$, and the average time lost due to a failure during r_l is $F_{\sum_{i=1}^l \lambda_i}(r_l)$. Therefore, the recovery cost to recover from a level- l failure is:

$$R_l = \sum_{i=l}^L \frac{p_i}{\sum_{j=l}^L p_j} \left(r_i + \left(\frac{1}{q_{r_i, \sum_{j=1}^i \lambda_j}} - 1 \right) F_{\sum_{j=1}^i \lambda_j}(r_i) \right)$$

where $\frac{p_i}{\sum_{j=l}^L p_j}$ indicates the probability of last recoverable checkpoint type being a level- i ($i \geq l$) checkpoint to recover from a level- l failure. This leads to:

$$T_{eff} = T + \frac{1 - q_{T, \Lambda}}{q_{T, \Lambda}} \left(F_{\Lambda}(T) + \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} R_l \right) \right),$$

where $\frac{\lambda_l}{\Lambda}$ indicates the proportion of level- l failures from total failures.

Although a failure of any level results in $F_{\Lambda}(T)$ lost time, failures other than level-1 can result in additional lost time as depicted in Fig. 6.5. As level- l failures can only recover from a level- l or a higher level checkpoint, the time lost due to a level- l failure includes the time from the completion of the nearest level- l or higher checkpoint and the occurrence of the level- l failure. This lost time can include several lower level checkpoints, i.e. checkpoints from level-1 to level- $(l-1)$. Since the probability of losing i level- $(l-1)$ or a lower level checkpoints is $(\sum_{j=1}^{l-1} p_j)^i (1 - \sum_{j=1}^{l-1} p_j)$, on average the num-

ber of lost completed checkpoints is:

$$\frac{\sum_{i=0}^{l-1} p_i}{1 - \sum_{i=0}^{l-1} p_i} = \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i}.$$

This leads to:

$$\begin{aligned} T_{eff} &= T + \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} (T_{eff} \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} + R_l) \right) \right) \\ &= \frac{T + \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} \left(F_{\Lambda}(T) + \sum_{l=1}^L \frac{\lambda_l R_l}{\Lambda} \right)}{1 - \frac{1 - q_{T,\Lambda}}{q_{T,\Lambda}} \left(\sum_{l=1}^L \frac{\lambda_l}{\Lambda} \left(\frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} \right) \right)}. \end{aligned}$$

and finally:

$$U = \frac{T - \sum_{l=1}^L p_l c_l}{T_{eff}}. \quad (6.11)$$

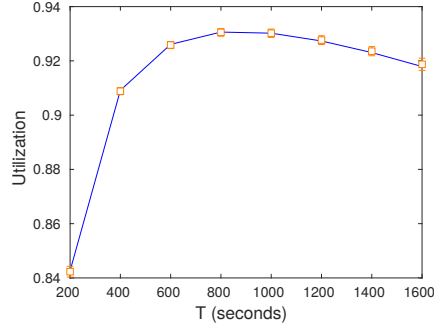
This completes the salient features of our L -level checkpoint and restart system model for a single process.

6.5.2 Comparison to stochastic simulation

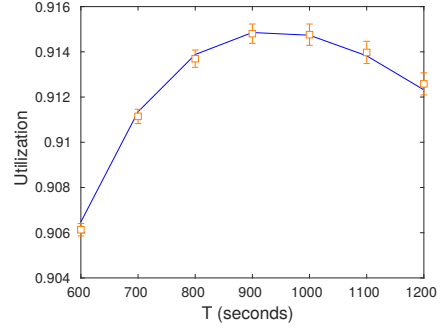
Fig. 6.15a and 6.15b show the utilization comparison between our model based on (6.11) and the simulation results for 3-level and 4-level checkpointing. The solid lines are theoretical utilization while the data points and error bars represent the average utilization and the standard deviation observed after 100 runs, with each 3-level and 4-level simulation running for $\frac{1000}{\lambda_3}, \frac{1000}{\lambda_4}$ days respectively.

6.6 L -level Checkpointing for Stream Processing

In this section, we extend the L -level checkpointing model to a distributed stream processing system. As explained in section 6.4, if δ is the time between sending the token from one operator and receiving the token at the next operator in the DAG, and the critical path in the DAG consist of n operators including the source and the sink, then without considering failure, we have $T_{eff} = T' = T + (n - 1)\delta$ and we can write the expected uti-



(a) 3-level checkpointing for $p_1 = 0.5600$, $p_2 = 0.3044$



(b) 4-level checkpointing for $p_1 = 0.3306$, $p_2 = 0.1273$, $p_3 = 0.3902$

Figure 6.15: Utilization for $\lambda_1=2.4, \lambda_2=1.2, \lambda_3=0.6, \lambda_4=0.12$ per day, $c_1=r_1=3, c_2=r_2=30, c_3=r_3=50, c_4=r_4=100$ seconds.

lization as:

$$U = \frac{T - \sum_{l=1}^L p_l c_l}{T_{eff}} = \frac{T - \sum_{l=1}^L p_l c_l}{T + (n-1)\delta}.$$

6.6.1 Including failure and recovery cost and the overlap between consecutive checkpoints

As explained in section 6.4.1, we include the time due to failures during $T' = T + (n-1)\delta$ and failures during recovery to improve the accuracy of T_{eff} . This leads to:

$$\begin{aligned} T_{eff} &= T' + \frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} \left(F_{\Lambda}(T') + \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} (T_{eff} \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} + R_l) \right) \right) \\ &= \frac{T' + \frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} \left(F_{\Lambda}(T') + \sum_{l=1}^L \frac{\lambda_l R_l}{\Lambda} \right)}{1 - \frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} \left(\sum_{l=1}^L \frac{\lambda_l}{\Lambda} \left(\frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} \right) \right)}. \end{aligned} \quad (6.12)$$

where $\frac{\lambda_l}{\Lambda}$ indicates the proportion of level- l failures from total failures and $\frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i}$ indicates the average number of checkpoints lost due to a level- l failure and R_l indicates the recovery cost of a level- l failure.

Similar to section 6.4.2 we have to take into account the overlap between the first $(n-1)\delta$ of T' with the previous T' as shown in Fig. 6.13. The effective period to complete

$(n-1)\delta$ is:

$$(n-1)\delta + \frac{1 - q_{(n-1)\delta, \Lambda}}{q_{(n-1)\delta, \Lambda}} \left(F_{\Lambda}((n-1)\delta) + \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} (T_{eff} \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} + R_l) \right) \right)$$

We can improve T_{eff} by subtracting the effective period to complete $(n-1)\delta$ from T_{eff} in (6.12). This leads to:

$$\begin{aligned} T_{eff} &= T + \frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} F_{\Lambda}(T') - \frac{1 - q_{(n-1)\delta, \Lambda}}{q_{(n-1)\delta, \Lambda}} F_{\Lambda}((n-1)\delta) \\ &+ \left(\frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} - \frac{1 - q_{(n-1)\delta, \Lambda}}{q_{(n-1)\delta, \Lambda}} \right) \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} (T_{eff} \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} + R_l) \right) \\ &= \frac{T + \frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} \left(F_{\Lambda}(T') + \sum_{l=1}^L \frac{\lambda_l R_l}{\Lambda} \right) - \frac{1 - q_{(n-1)\delta, \Lambda}}{q_{(n-1)\delta, \Lambda}} \left(F_{\Lambda}((n-1)\delta) + \sum_{l=1}^L \frac{\lambda_l R_l}{\Lambda} \right)}{1 - \left(\frac{1 - q_{T', \Lambda}}{q_{T', \Lambda}} - \frac{1 - q_{(n-1)\delta, \Lambda}}{q_{(n-1)\delta, \Lambda}} \right) \sum_{l=1}^L \left(\frac{\lambda_l}{\Lambda} (T_{eff} \frac{\sum_{i=0}^{l-1} p_i}{\sum_{i=l}^L p_i} + R_l) \right)} \end{aligned}$$

and finally:

$$U = \frac{T - \sum_{l=1}^L p_l c_l}{T_{eff}}. \quad (6.13)$$

This completes the salient features of our L -level checkpoint and restart system model for a stream processing system.

6.6.2 Comparison to stochastic simulation

Fig. 6.16 shows the utilization comparison between our model based on (6.13) and the simulation results for 3-level checkpointing using DAGs with different critical path lengths. The solid lines are theoretical utilization while the data points and error bars represent the average utilization and the standard deviation observed after 100 runs, with each simulation running for $\frac{1000}{\lambda_3}$ days. Similarly to 2-level checkpointing, the utilization decreases with the value of n as shown in the figure.

Table 6.2 shows T^* , p_1^* , p_2^* , and U for the optimal values for 3-level checkpointing for $\lambda_1 = 20$, $\lambda_2 = 5$ per day, $c_1 = r_1 = 10$, $c_2 = r_2 = 20$, $c_3 = r_3 = 100$, $\delta = 0.5$ seconds and different λ_3 values along with optimal values and U of 2-level checkpointing for

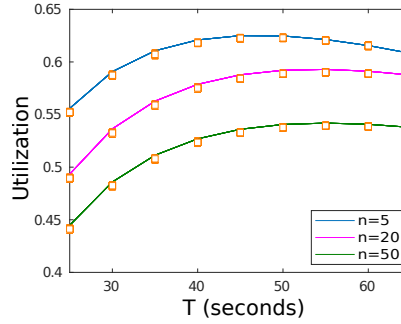


Figure 6.16: Utilization of 3-level checkpointing for a streaming application for $\lambda_1 = 432$, $\lambda_2 = 43.2$, $\lambda_3 = 8.64$ per day, $c_1 = r_1 = 5$, $c_2 = r_2 = 10$, $c_3 = r_3 = 20$, $\delta = 0.5$ seconds and different n values.

the same parameters and failure rates using $p_2 = 0$. As indicated in the table, optimal values change slightly as n changes when other parameters are constant. For all the cases level-3 gives better utilization than level-2 checkpointing. The utilization increase we can achieve by 3 levels instead of 2 levels becomes more significant when the difference of λ_3 and λ_2 increases or when n increases.

6.7 Experimental Results with Apache Flink

We used Apache Flink, a state-of-the-art stream processing framework to evaluate the efficacy of our model. Since Flink does not support multi-level checkpointing, the experiments are for single-level checkpointing. We experimented with various Flink application instances that undertake word counting, a common example used for benchmarking streaming applications, with differing values of topology depth n . All experiments were conducted using *m2.medium* nodes running on OpenStack. Each virtual machine had 2 CPU cores, 6 GB of RAM, 30 GB of disk space and ran Ubuntu 15.10, Java 1.7.0_91, and Flink 1.3.3. We used a fifty node cluster with one node as master and forty nine nodes as slaves. The word count application was loaded with a continuous stream of data and word counting was performed in a sliding window with a set of stateful operators to keep statistics of word counts in each window. Hadoop Distributed File System was used as the state backend for the experiments and Apache Kafka consumer was used as the source. Two methods were used to simulate two levels of random failures. Level-1,

Table 6.2: Comparison of 3-level and 2-level checkpointing.

λ_3	n	3-level				2-level			%U increase
		T^*	p_1^*	p_2^*	U	T^*	p_1^*	U	
1	5	338.95	0.201	0.675	0.833	329.85	0.719	0.784	6.2
	20	339.25	0.199	0.676	0.831	330.11	0.718	0.783	6.2
	50	339.83	0.196	0.679	0.827	330.63	0.717	0.780	6.2
0.1	5	336.01	0.215	0.745	0.873	322.01	0.747	0.795	9.7
	20	336.24	0.214	0.746	0.871	322.21	0.746	0.793	9.7
	50	336.37	0.226	0.743	0.866	322.6	0.746	0.789	9.8
0.01	5	335.56	0.214	0.772	0.885	321.27	0.75	0.796	11.2
	20	335.76	0.213	0.774	0.883	321.46	0.749	0.794	11.2
	50	336.04	0.213	0.774	0.88	321.84	0.748	0.791	11.2

throwing an application level exception and level-2, killing a task manager. Throwing an exception and killing a task manager were done based on two exponential distributions at precomputed failure event times with failure rates λ_1, λ_2 respectively. Using different λ_1, λ_2 values we ran each experiment five times, each running for 24 hours. These λ values are artificially large so as to indicate results that would be seen at a scale that we cannot experiment with due to lack of resources.

We compared the observed utilization, $\tilde{U}$, obtained using the default Flink parameters for checkpoint interval 30 minutes, with our theoretical prediction of utilization, U using measured parameters $\tilde{c}$, $\tilde{r}_1$, $\tilde{r}_2$ and $\tilde{\delta}$ from Flink's logs as inputs. Although Flink performs single-level checkpoints, we observed that restart cost of the two types of failures we enforced are different. The restart cost from an application level exception ($\tilde{r}_1$) is smaller compared to cost of recovering from a killed task manager ($\tilde{r}_2$) as the later requires more time to restart the task manager. We also computed the optimal utilization we can achieve if Flink used 2-levels of checkpoints using our theoretical model, taking the cost of level-2 checkpoints as the measured value $\tilde{c}$ from logs and assuming the cost of level-1 checkpoints is half the cost of level-2 checkpoints as measured $\tilde{r}_1$ is approximately half the value of measured $\tilde{r}_2$. Table 6.3 shows the settings of our experiments

Table 6.3: Experimental results using Apache Flink.

(a) Settings and observations

λ_1	λ_2	n	$\tilde{c}$ (ms)	$\tilde{r}_1$ (s)	$\tilde{r}_2$ (s)	$\tilde{\delta}$ (ms)
.02	.008	5	946.39 $\pm$ 21	10.09 $\pm$ 0.29	17.03 $\pm$ 1.76	17.01 $\pm$ 5.35
		7	1194.7 $\pm$ 18	10.19 $\pm$ 0.20	19.07 $\pm$ 0.75	17.04 $\pm$ 4.75
.03	.008	5	922.07 $\pm$ 20	10.20 $\pm$.27	17.57 $\pm$ 0.29	16.29 $\pm$ 6.06
		7	1187.3 $\pm$ 11	10.18 $\pm$.07	18.96 $\pm$ 1.08	18.33 $\pm$ 6.57
.04	.01	5	964.04 $\pm$ 17	10.21 $\pm$.16	18.28 $\pm$ 0.99	21.89 $\pm$ 5.08
		7	1195 $\pm$ 29	10.22 $\pm$.02	18.53 $\pm$ 0.47	18.31 $\pm$ 2.96

(b) Optimal parameters, observed utilization and theoretical utilization

$T = 30$ minutes		T^* (min)	p_1^*	$T = T^*, p_1 = p_1^*$	
$\tilde{U}$	U			U	% U increase
0.6256 $\pm$ 0.04	0.6342	0.9997	0.1131	0.9629	53.92%
0.6245 $\pm$ 0.06	0.6339	1.1072	0.1371	0.9585	53.48%
0.5283 $\pm$ 0.04	0.5318	0.7407	0.3493	0.9565	81.03%
0.5268 $\pm$ 0.02	0.5316	0.8373	0.3554	0.9514	80.59%
0.4358 $\pm$ 0.05	0.4264	0.6390	0.4016	0.9476	117.46%
0.4343 $\pm$ 0.02	0.4263	0.7183	0.3895	0.9430	117.09%

and observations: λ_1, λ_2, n , observed $\tilde{c}, \tilde{r}_1, \tilde{r}_2$ and $\tilde{\delta}$, the observed utilization, $\tilde{U}$, and theoretical utilization, U , when $T = 30$ minutes, the theoretical optimal, T^*, p_1^* , for the given settings and observed parameters assuming $c_2 = \tilde{c}, c_1 = \frac{c_2}{2}$, theoretical utilization when using the theoretical optimal, $T = T^*, p_1 = p_1^*$, and the percentage increase in utilization % U over the default $T = 30$ minutes if 2-level checkpointing is used. As shown in the table, theoretical predictions of utilization compare well to the observed utilization and we can always achieve significant utilization increase if 2-level checkpointing is used instead of 1-level based on the theoretical model. In the table T^* changes with a change in n because c changes with n , as a result of the windowing sizes changing with a deeper topology which increases the checkpoint cost.

Table 6.4: Parameters used for comparison from Di et al. [120].

case	c_1	r_1	c_2	r_2	λ_1	λ_2
1	20	20	50	50	24	4
2	20	20	50	50	50	10
3	20	20	100	100	100	20
4	10	10	40	40	100	20
5	10	10	40	40	200	40
6	10	10	100	100	200	40
7	40	40	200	200	300	60
8	50	50	300	300	400	60

6.7.1 Comparison

We compared our model with existing solutions suggested for optimizing single-level checkpointing by Daly [95] and Young [93]. Table 6.4 shows the settings used for the comparisons which are chosen from the parameters used by Di et al. [120] in their work. All the checkpoint costs and restart costs are in seconds and failure rates are in failures per day. We calculated the average utilization for the optimal parameters determined by our model, Daly’s model and Young’s model. Figure 6.17 depicts the results including the utilization given by our theoretical model and the average utilization observed after 100 runs of the simulation using the theoretical optimal values.

As shown in the figure, simulation results compare well with our theoretical model. For all the cases, our solution performs better than the single level optimizations. For example, our model achieves 11.9%, 12.8% utilization increase for case 3 and 32.9%, 35.7% utilization increase for case 6 compared to the models of Daly and Young. Although our model uses a single checkpointing interval similar to Daly and Young, the probabilistic approach we use to facilitate multi-level checkpointing provides improvements over single-level checkpointing.

We also compared the difference between the utilization predicted by the theoretical model and the utilization observed in simulations of our model and existing pattern-based models for the parameters shown in Table 6.4, which is depicted in Figure 6.18. For pattern-based models proposed by Benoit et al. [125] and Di et al. [120], we used simulation and theoretical results published by Benoit et al. [125] in their work. As shown in the figure, the utilization difference of our model is comparatively lower than the differ-

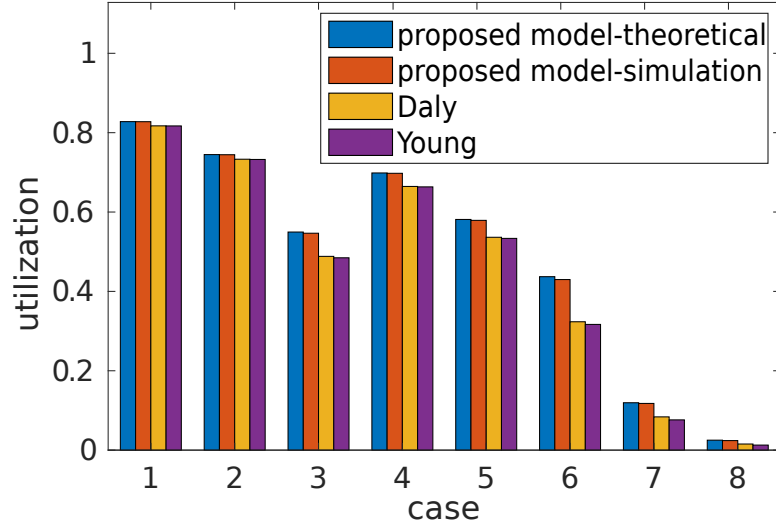


Figure 6.17: Performance comparison of different models using parameters in Table 6.4.

ence of pattern-based models. For example, the percentage difference for case 8 is 152.6% and 44.4% for Benoit et al. and Di et al. models, while the difference of our model is 4.5%. The higher difference between the simulation results and values predicted by the theoretical model for pattern-based checkpointing can be attributed to the assumptions made in these models such as no failures during checkpoints and recovery.

6.8 Conclusion

We presented an analytical expression for the utilization of a distributed stream processing system that uses periodic multi-level checkpoints. This model allows optimization of the checkpoint interval and the probabilities of performing different level checkpoints through maximizing utilization. We have shown the correctness of the model using a stochastic simulation for different levels of checkpoints. With practical experimentation using Flink combined with the theoretical model, we have shown how multi-level checkpointing could improve system efficiency instead of using single-level checkpointing used by well-known stream processing systems.

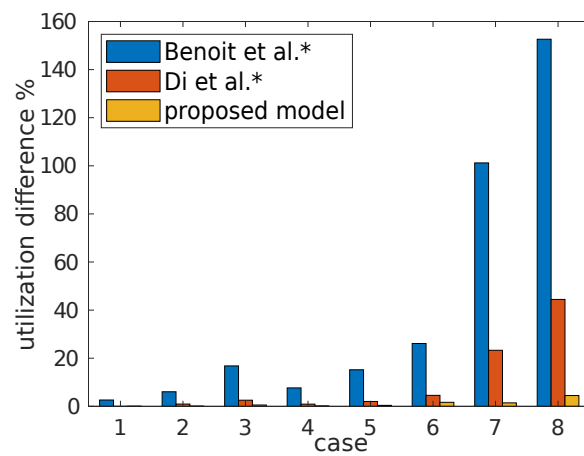


Figure 6.18: Comparison between theoretical and simulation utilization of different multi-level models using parameters in Table 6.4. *Using results published by Benoit et al. [125].

Chapter 7

Conclusions and Future Work

Focus towards stream processing has been increasing in past years due to the need to process large volumes of data to provide near real-time results to make time-sensitive decisions. While stream processing systems were initially focused on providing support for stateless computations, the increased use of stream processing in various domains has made the support for stateful computations an expected functionality of stream processing systems.

When supporting complex stateful computations, stream processing systems have to ensure that computations can be performed in a distributed environment in a scalable manner so that large volumes of data can be handled while ensuring synchronization and consistency requirements to guarantee correctness of the results. To achieve efficient and scalable stateful computations, systems have to carefully decide several aspects of state management including where to store state, how to distribute large state and how to share and communicate distributed state. Furthermore, fault-tolerance approaches to recover state after failures is another main part of supporting stateful computations. However, the overhead of fault-tolerance approaches can have a significant impact on system performance, especially when dealing with large-scale systems where the failure rate increases with the system size. Therefore, efficient state management and fault-tolerance mechanisms are important to provide the low latency requirements of streaming applications.

There are several challenges that need to be addressed in order to improve the scalability and efficiency of distributed stateful computations in stream processing systems to achieve high throughput and low latency requirements while ensuring integrity and

reliability. In this thesis, we have proposed methods to improve the efficiency of complex stateful computations and minimize the impact of fault-tolerance approaches used to recover state after failures. We summarize our contributions in the next sections.

7.1 Complex Stateful Computational Support for Stream processing Systems

Streaming applications are used in various domains and some streaming use cases have complex computations that involve multiple windowing stages. The existing window semantics are only defined for a single windowing stage and cannot be used to understand the behavior of applications that have a hierarchy of window stages. Therefore, we propose a multi-stage hierarchical window model that can aid in rigorous understanding of complicated real-time results of streaming applications with multiple windowing stages.

The second focus of this work is to provide a scalable model to efficiently represent iterative algorithms in stream processing systems. In general, streaming applications are formulated as a fixed number of inter-connected operators, where the operator graph indicates the sequence of computations that apply to the streaming data in motion. Typically an iterative operation in a streaming application is done by embedding the iteration in a single operator, which excludes multiple operator instances and hence not scalable or by using a sequence of operators, one for each iteration, which requires knowledge of the number of iterations a priori. However, the number of iterations of some applications such as k-means clustering cannot be determined at the creation of the application as it depends on the properties of the forthcoming data received at runtime and the convergence criterion. Hence only a limited number of iterative computations can be executed using this approach.

We propose a communication model to support intra-operator communication so that iterative computations, including those with an arbitrary number of iterations, can be efficiently executed in streaming applications. We show that the proposed model can support different iterative algorithms that have complex communication patterns. Moreover, this model can support complex computations that cannot be modeled using a classic di-

rected acyclic graph of operators and it can perform computations that can be modeled in a DAG in a more efficient manner. Furthermore, we implement this model using Apache Storm, a well-known stream processing system and present an API using which iterative streaming applications can be implemented.

Finally, through evaluating a number of parallel iterative algorithms using large-scale datasets, we demonstrate the scalability and performance of our proposed communication model and compare it to the existing approaches used for constructing iterative streaming applications. For example, we show how iterative graphs algorithms can be executed efficiently in a distributed manner for scalability using this model and show that our approach outperforms alternative approaches that can be used for parallel-iterative computations in Storm, both in terms of scalability and performance. Moreover, we show the efficiency and scalability of the model by comparing it with existing graph processing systems. Not only for graph algorithms, but the evaluations have also demonstrated the scalability and efficiency of the proposed model for different applications. The results suggest that the performance gain of our model becomes more significant for large data sets and computationally intensive algorithms.

7.2 Optimization of Checkpoint Intervals

With the introduction of stateful computations emerged the challenge of providing fault-tolerance to guarantee that state can be recovered and the data analysis can continue even after failures. Checkpointing is the fault-tolerance approach used by many stream processing systems to recover state of stateful computations. Checkpointing is essential to ensure that the system does not need to start computations from scratch after a failure. Therefore, state-of-the-art distributed stream processing systems such as Apache Flink and Storm have recently included checkpointing to provide fault-tolerance for stateful applications. This is a necessary eventuality as these systems head into the Exascale regime, and is evidently more efficient than replication as state size grows.

However, current systems use a nominal value for the checkpoint interval such as 30 minutes, that does not take into account the salient aspects of the checkpoint process, nor

the system scale, which can readily lead to inefficient system operation. In this work, we focus on addressing this shortcoming, we provide a rigorous derivation of utilization – the fraction of total time available for the system to do useful work – that incorporates checkpoint interval, failure rate, checkpoint cost, failure detection and restart cost, depth of the streaming application and message delay. Our model yields an elegant expression for utilization and provides an optimal checkpoint interval given these parameters, interestingly showing it to be dependent only on checkpoint cost and failure rate. We confirm the accuracy and efficacy of our model through stochastic simulation. We observe that the improvements in system utilization gained by the use of optimal checkpoint interval become more significant as the system size increases. This model provides a solid theoretical basis for the analysis and optimization of more elaborate checkpointing approaches.

7.3 Impact of Optimal Checkpoint Interval on Real-world Stream Processing Applications

State-of-the-art stream processing systems use checkpointing to support fault-tolerance for stateful computations where the state of the computations is periodically persisted. In practice, systems are often configured to perform checkpoints based on crude values like 30 minutes or 1 hour, ignoring factors such as checkpoint and restart costs, leading to suboptimal performance. Therefore, we can use a theoretical optimal checkpoint interval that maximizes the system utilization for stream processing systems to minimize the impact of checkpointing on system performance.

In this work, we investigate the practical benefits of our proposed theoretical optimal by conducting experiments in a real-world cloud setting using different streaming applications; we use Apache Flink, a well-known stream processing system for our experiments. The experiment results demonstrate that an optimal interval can achieve better utilization, confirming the practicality of the theoretical model when applied to real-world applications. We observed utilization improvements from 10% - 200% for a range of failure rates from 0.3 failures per hour to 0.075 failures per minute. Moreover, we ex-

plore how performance measures: latency and throughput are affected by the optimal interval. Our observations demonstrate that significant improvements can be achieved using the optimal interval for both latency and throughput.

7.4 Optimization for Multi-level Checkpointing

State-of-the-art stream processing platforms make use of checkpointing to support fault tolerance, where a checkpoint tuple flows through the topology to all operators, indicating a checkpoint and triggering a checkpoint operation. The checkpoint will enable recovering from any kind of failure, be it as localized as a process fault or as widespread as power supply loss to an entire rack of machines. As we move towards Exascale computing, it is becoming clear that this kind of single-level checkpointing is too inefficient to scale. Some HPC researchers are now investigating multi-level checkpointing, where checkpoint operations at each level are tailored to specific kinds of failure to address the inefficiencies of single-level checkpointing. Multi-level checkpointing has been shown in practice to be superior, giving greater efficiency in operation over single-level checkpointing. However, to date there is no theoretical basis that provides optimal parameter settings for a multi-level checkpointing approach in a stream processing system.

In this work, we focus on presenting a theoretical framework for determining optimal parameter settings in a multi-level global checkpointing system, that is applicable to stream processing. Our approach is stochastic, where at a given checkpoint interval, a level is selected with some probability for checkpointing. We derive the optimal checkpoint interval and associated optimal checkpoint probabilities for a multi-level checkpointing system, that considers failure rates, checkpoint costs, restart costs and possible failure during restarting, at every level. We confirm our results with stochastic simulation and practical experimentation and demonstrate the efficiency of multi-level checkpointing compared to single-level checkpointing used in current stream processing systems.

7.5 Future Work

This section discusses several future directions yet to be further explored to address the gaps and challenges in stateful computations in distributed stream processing systems.

7.5.1 State management

Most of the existing systems only support key-value based states. However, some operators can have different types of states such as maps or graphs. Therefore, future work could look into providing support to maintain different types of complex operator states in terms of state storage and state sharing.

Work has been done to reduce the overhead of checkpointing the state by using compression techniques [117, 118, 119]. Not only focusing on the checkpoints but extending this work to minimize the memory requirements of maintaining large states and reduce the communication overhead of sharing states also is another future direction that is worth exploring.

Optimization techniques such as slicing where the operator state is divided into small parts called slices are used to increase the efficiency of stateful operators when dealing with varying workloads [22]. However, the slicing technique is limited to dividing computationally independent states and therefore has limited applicability to common stateful computations such as iterative computations. Hence extending these types of different optimization techniques to support all types of operator states in general is another promising future direction.

When performing parallel stateful computations, the highest overhead comes from communicating state or intermediate results between parallel running operator instances. Partition techniques such as hashing and key-based are mostly used to divide the workloads in parallel systems which can result in poor workload balance and high communications overhead. Therefore, future work could look into partitioning techniques that can divide the workload such that the communication between parallel instances is minimal. These techniques can reduce the communications costs while ensuring workload is evenly distributed among parallel instances.

We evaluated our proposed models using common stateful computations such as iterative graph algorithms and machine learning algorithms. Extending the evaluations to assess the impact of our models in different settings that have different characteristics such as IoT use cases like healthcare where data arrival rates are relatively high is another interesting future direction.

Since input data is continuous and changes over time in streaming applications, static partitioning techniques alone cannot balance state between instances and minimize communications overhead. Future work could address the limitations in static partitioning schemes by looking into dynamic load-balancing and re-partitioning techniques to handle continuous data streams.

7.5.2 Fault-tolerance

In our work proposed for optimizing single-level checkpointing of stream processing applications, the optimal checkpoint interval depends only on checkpoint cost and system failure rate. Although we assume that the checkpoint cost is constant throughout the application running time, it can vary based on the input rate of the streaming application. It would be interesting to update the checkpoint model to consider varying checkpoint costs based on input rates. Furthermore, investigating the trade-offs between the computational cost of dynamically determining the checkpoint interval to adapt to changing workload and the performance increase that can be achieved with a dynamic checkpoint interval compared to a static one is a good extension to our work. Moreover, the models we proposed for checkpointing assume that failures follow an exponential distribution. Generalizing the proposed models to use distributions such as Weibull distribution that can model increasing failure rates with the system age is another possible future direction.

Uncoordinated checkpointing is another option that can be used to provide fault-tolerance in streaming applications [102]. In this approach, each operator has an independent checkpoint interval and therefore each operator has to buffer the input data and output data to recover from failures. Additional space requirement to buffer the data is the main disadvantage of this approach. Another future direction would be to theoretic-

cally model the uncoordinated checkpointing approach taking into account factors such as maximum space available for buffering. With the use of such a theoretical model, one can determine whether using uncoordinated checkpointing is a viable option with the available memory and if so what are the optimal checkpoint intervals for each operator that can work with the available memory.

Incremental checkpointing is another optimization of the checkpointing process that is used to reduce the checkpointing overhead [111, 112, 113, 114]. Extending our model to express the behavior of incremental checkpointing in a stream processing system is a possible future direction. This extension can be used to determine the optimal checkpoint interval of incremental checkpointing for unbounded workloads. However, incremental checkpointing is not well-suited for computations where state changes often. Therefore, investigating a checkpointing approach that can dynamically determine when to use incremental checkpoints based on factors such as workload and the number of state updates in between consecutive checkpoints can be beneficial for future stream processing applications.

For multi-level checkpoints, our work is done under the assumption that for a failure of a particular level, recovery is done using the nearest completed checkpoint which is of the same level or a higher level. However, in some cases it could be less expensive to go back to the last completed checkpoint of the same level ignoring all the checkpoints of higher levels as recovery cost increases with the failure level. Instead of pre-defining which checkpoint should be used to recover from a failure, it would be interesting to dynamically select the checkpoint that can guarantee the lowest recovery cost and re-work cost to recover from a failure based on historical data. Exploring this approach and investigating the benefits compared to our proposed model is another possible future direction.

This thesis mostly focuses on checkpointing-based fault-tolerance to recover state after failures. However, checkpointing becomes inefficient for systems with high failure rates, especially for exascale systems that expect failures in every few minutes. Nevertheless, other most common fault-tolerance approach, replication also is not practical in such cases as it requires a substantial amount of additional resources to replicate the com-

putations. Therefore, looking into novel hybrid fault-tolerance techniques that combine existing approaches such as checkpointing and replication to handle high failure rates is a potential future research direction.

Bibliography

- [1] D. Reinsel, J. Gantz, and J. Rydning. “Data age 2025: the digitization of the world from edge to core”. In: *Seagate*, <https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf> (2018).
- [2] T. De Matteis and G. Mencagli. “Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach”. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 382–401.
- [3] Q.-C. To, J. Soto, and V. Markl. “A Survey of State Management in Big Data Processing Systems”. In: *The VLDB Journal* 27.6 (Dec. 2018), pp. 847–872.
- [4] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García. “Paving the way towards high-level parallel pattern interfaces for data stream processing”. In: *Future Generation Computer Systems* 87 (2018), pp. 228–241.
- [5] M. D. de Assuno, A. da Silva Veith, and R. Buyya. “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions”. In: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17.
- [6] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. “Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: ACM, 2013, pp. 725–736.
- [7] B. Shao, H. Wang, and Y. Li. “Trinity: A Distributed Graph Engine on a Memory Cloud”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: ACM, 2013, pp. 505–516.
- [8] R. Cheng et al. “Kineograph: Taking the Pulse of a Fast-changing and Connected World”. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys ’12. Bern, Switzerland: ACM, 2012, pp. 85–98.
- [9] A. Quamar, A. Deshpande, and J. Lin. “NScale: Neighborhood-centric Large-scale Graph Analytics in the Cloud”. In: *The VLDB Journal* 25.2 (Apr. 2016), pp. 125–150.
- [10] Y. Kwon, M. Balazinska, and A. Greenberg. “Fault-tolerant Stream Processing Using a Distributed, Replicated File System”. In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 574–585.

- [11] A. Brito, C. Fetzer, H. Sturzhelm, and P. Felber. "Speculative Out-of-order Event Processing with Software Transaction Memory". In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS '08. Rome, Italy: ACM, 2008, pp. 265–275.
- [12] A. Roy, I. Mihailovic, and W. Zwaenepoel. "X-Stream: Edge-centric Graph Processing Using Streaming Partitions". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farmington, Pennsylvania: ACM, 2013, pp. 472–488.
- [13] A. Kyrola, G. Blelloch, and C. Guestrin. "GraphChi: Large-scale Graph Computation on Just a PC". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46.
- [14] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. "Chaos: Scale-out Graph Processing from Secondary Storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 410–424.
- [15] B. Liu, Y. Zhu, and E. Rundensteiner. "Run-time Operator State Spilling for Memory Intensive Long-running Queries". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 347–358.
- [16] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. "Load Shedding in a Data Stream Manager". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 309–320.
- [17] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. "A Catalog of Stream Processing Optimizations". In: *ACM Comput. Surv.* 46.4 (Mar. 2014).
- [18] J. Ekanayake et al. "Twister: A Runtime for Iterative MapReduce". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. Chicago, Illinois: ACM, 2010, pp. 810–818.
- [19] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. "HaLoop: Efficient Iterative Data Processing on Large Clusters". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 285–296.
- [20] X. Shi, B. Cui, Y. Shao, and Y. Tong. "Tornado: A System For Real-Time Iterative Analysis Over Evolving Data". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 417–430.
- [21] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel. "GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 309–316.

- [22] Y. Wu and K. Tan. "ChronoStream: Elastic stateful stream computation in the cloud". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 723–734.
- [23] L. Fegaras. "Incremental Query Processing on Big Data Streams". In: *IEEE Transactions on Knowledge and Data Engineering* 28.11 (2016), pp. 2998–3012.
- [24] J. Meehan et al. "S-Store: Streaming Meets Transaction Processing". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), pp. 2134–2145.
- [25] H. Röger and R. Mayer. "A Comprehensive Survey on Parallelization and Elasticity in Stream Processing". In: *ACM Comput. Surv.* 52.2 (Apr. 2019), 36:1–36:37.
- [26] J. Ding et al. "Optimal Operator State Migration for Elastic Data Stream Processing". In: *ArXiv abs/1501.03619* (2015).
- [27] J. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann. "Dynamic Resource Management in Energy Constrained Heterogeneous Computing Systems Using Voltage Scaling". In: *IEEE Transactions on Parallel and Distributed Systems* 19.11 (2008), pp. 1445–1457.
- [28] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara. "Online Scheduling and Interference Alleviation for Low-Latency, High-Throughput Processing of Data Streams". In: *IEEE Transactions on Parallel and Distributed Systems* 28.12 (2017), pp. 3553–3569.
- [29] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. "Providing Resiliency to Load Variations in Distributed Stream Processing". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, 775786.
- [30] B. Gedik. "Generic windowing support for extensible stream processing systems". In: *Software: Practice and Experience* 44.9 (2014), pp. 1105–1128.
- [31] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. "SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 232–243.
- [32] K. Patroumpas and T. Sellis. "Window specification over data streams". In: *International Conference on Extending Database Technology*. Springer. 2006, pp. 445–464.
- [33] J. Krämer. *Continuous queries over data streams—Semantics and implementation*. Gesellschaft für Informatik eV, 2009.
- [34] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. "Semantics and evaluation techniques for window aggregates in data streams". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005, pp. 311–322.
- [35] M. Ivanova and T. Risch. "Customizable parallel execution of scientific stream queries". In: *VLDB*. Vol. 5. 2005, pp. 157–168.
- [36] J. Traub et al. "Scotty: Efficient window aggregation for out-of-order stream processing". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1300–1303.

- [37] K. G. S. Madsen, Y. Zhou, and L. Su. "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*. 2016, pp. 37–48.
- [38] G. Mencagli, M. Torquati, F. Lucattini, S. Cuomo, and M. Aldinucci. "Harnessing sliding-window execution semantics for parallel stream processing". In: *Journal of Parallel and Distributed Computing* 116 (2018), pp. 74–88.
- [39] N. Backman, R. Fonseca, and U. Çetintemel. "Managing parallelism for stream processing in the cloud". In: *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*. 2012, pp. 1–5.
- [40] C. Balkesen and N. Tatbul. "Scalable data partitioning techniques for parallel sliding window processing over data streams". In: *International Workshop on Data Management for Sensor Networks (DMSN)*. 2011.
- [41] A. Kolios, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. "Saber: Window-based hybrid stream processing for heterogeneous architectures". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 555–569.
- [42] R. Mayer, M. A. Tariq, and K. Rothermel. "Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 54–65.
- [43] L. Golab and M. T. Özsu. "Issues in Data Stream Management". In: *SIGMOD Rec.* 32.2 (June 2003), pp. 5–14.
- [44] K. Patroumpas and T. Sellis. "Window update patterns in stream operators". In: *East European Conference on Advances in Databases and Information Systems*. Springer. 2009, pp. 118–132.
- [45] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. "Slider : Incremental Sliding-Window Computations for Large-Scale Data Analysis". In: 2012.
- [46] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. "Spinning Fast Iterative Data Flows". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1268–1279.
- [47] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. "Incremental, Iterative Data Processing with Timely Dataflow". In: *Commun. ACM* 59.10 (Sept. 2016), 7583.
- [48] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farmington, Pennsylvania: ACM, 2013, pp. 439–455.
- [49] K. Qi, Z. Zhao, J. Fang, and Y. Han. "MapReduce-Based Data Stream Processing over Large History Data". In: *Service-Oriented Computing*. Ed. by C. Liu, H. Ludwig, F. Toumani, and Q. Yu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 718–732.

- [50] A. M. Aly et al. "M3: Stream Processing on Main-Memory MapReduce". In: *2012 IEEE 28th International Conference on Data Engineering*. 2012, pp. 1253–1256.
- [51] J. Ekanayake, S. Pallickara, and G. Fox. "MapReduce for Data Intensive Scientific Analyses". In: *2008 IEEE Fourth International Conference on eScience*. 2008, pp. 277–284.
- [52] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama. "Asynchronous Algorithms in MapReduce". In: *2010 IEEE International Conference on Cluster Computing*. 2010, pp. 245–254.
- [53] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. "CIEL: A Universal Execution Engine for Distributed Data-flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 113–126.
- [54] Q. Ke, M. Isard, and Y. Yu. "Optimus: A Dynamic Rewriting Framework for Data-parallel Execution Plans". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 15–28.
- [55] M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger. "Composing Algorithmic Skeletons to Express High-Performance Scientific Applications". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 415–424.
- [56] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. "A Comparison of Big Data Frameworks on a Layered Dataflow Model". In: *Parallel Processing Letters* 27.01 (2017), p. 1740003.
- [57] G. Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146.
- [58] L. G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111.
- [59] S. Salihoglu and J. Widom. "GPS: A Graph Processing System". In: *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. SSDBM. Baltimore, Maryland, USA: ACM, 2013, 22:1–22:12.
- [60] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. "GraphX: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. New York, New York: ACM, 2013, 2:1–2:6.
- [61] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-scale". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1804–1815.
- [62] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 169–182.

- [63] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. "Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine". In: *Proc. VLDB Endow.* 8.2 (Oct. 2014), pp. 161–172.
- [64] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727.
- [65] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. "From "Think Like a Vertex" to "Think Like a Graph"". In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 193–204.
- [66] D. Yan, J. Cheng, Y. Lu, and W. Ng. "Blogel: A Block-centric Framework for Distributed Computation on Real-world Graphs". In: *Proc. VLDB Endow.* 7.14 (Oct. 2014), pp. 1981–1992.
- [67] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12.* Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30.
- [68] D. Sengupta et al. "GraphIn: An Online High Performance Incremental Graph Processing Framework". In: *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833.* New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 319–333.
- [69] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. "STINGER: High performance data structure for streaming graphs". In: *2012 IEEE Conference on High Performance Extreme Computing.* 2012, pp. 1–5.
- [70] T. Suzumura, S. Nishii, and M. Ganse. "Towards Large-scale Graph Stream Processing Platform". In: *Proceedings of the 23rd International Conference on World Wide Web. WWW '14 Companion.* Seoul, Korea: ACM, 2014, pp. 1321–1326.
- [71] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. "Time-evolving Graph Processing at Scale". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems. GRADES '16.* Redwood Shores, California: ACM, 2016, 5:1–5:6.
- [72] T. Abughofa and F. Zulkernine. "Towards online graph processing with spark streaming". In: *2017 IEEE International Conference on Big Data (Big Data).* 2017, pp. 2787–2794.
- [73] C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmiss, and V. K. Prasanna. "Real-Time Analytics for Fast Evolving Social Graphs". In: *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on.* 2015, pp. 829–834.
- [74] W. Han et al. "Chronos: A Graph Engine for Temporal Graph Analysis". In: *Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14.* Amsterdam, The Netherlands: ACM, 2014, 1:1–1:14.
- [75] H. Kim and R. Anderson. "Temporal node centrality in complex networks". In: *Phys. Rev. E* 85 (2 2012), p. 026107.

- [76] W. D. Vijitbenjaronk, J. Lee, T. Suzumura, and G. Tanase. "Scalable time-versioning support for property graph databases". In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, pp. 1580–1589.
- [77] U. Khurana and A. Deshpande. "Efficient snapshot retrieval over historical graph data". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 997–1008.
- [78] D. Sengupta and S. L. Song. "EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU". In: *High Performance Computing*. Ed. by J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes. Cham: Springer International Publishing, 2017, pp. 97–119.
- [79] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. "GTS: A Fast and Scalable Graph Processing Method Based on Streaming Topology to GPUs". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 447–461.
- [80] Z. Fu, M. Personick, and B. Thompson. "MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs". In: *Proceedings of Workshop on GRAPh Data Management Experiences and Systems*. GRADES'14. Snowbird, UT, USA: ACM, 2014, 2:1–2:6.
- [81] J. Rosen et al. *Iterative MapReduce for Large Scale Machine Learning*. 2013. arXiv: 1303.3517 [cs.DC].
- [82] H. Tamano, S. Nakadai, and T. Araki. "Optimizing Multiple Machine Learning Jobs on MapReduce". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. 2011, pp. 59–66.
- [83] C.-T. Chu et al. "Map-Reduce for Machine Learning on Multicore". In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*. NIPS06. Canada: MIT Press, 2006, 281288.
- [84] A. Avizienis, J. Laprie, and B. Randell. "Fundamental Concepts of Dependability". In: 2000.
- [85] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.
- [86] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. "State Management in Apache Flink&Reg: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729.
- [87] H. Wang, L. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. "Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 1180–1191.
- [88] T. Akidau et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1033–1044.
- [89] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farmington, Pennsylvania: ACM, 2013, pp. 423–438.

- [90] C. Xu, M. Holzemer, M. Kaul, J. Soto, and V. Markl. "On Fault Tolerance for Distributed Iterative Dataflow Processing". In: *IEEE Transactions on Knowledge and Data Engineering* 29.8 (2017), pp. 1709–1722.
- [91] G. Jacques-Silva et al. "Consistent Regions: Guaranteed Tuple Processing in IBM Streams". In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1341–1352.
- [92] N. El-Sayed and B. Schroeder. "Checkpoint/restart in practice: When simple is better". In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 2014, pp. 84–92.
- [93] J. W. Young. "A First Order Approximation to the Optimum Checkpoint Interval". In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531.
- [94] J. T. Daly. "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps". In: *Future Gener. Comput. Syst.* 22.3 (Feb. 2006), pp. 303–312.
- [95] J. Daly. "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps". In: *Proceedings of the 2003 International Conference on Computational Science*. ICCS'03. Melbourne, Australia: Springer-Verlag, 2003, pp. 3–12.
- [96] H. Jin, Y. Chen, H. Zhu, and X. Sun. "Optimizing HPC Fault-Tolerant Environment: An Analytical Approach". In: *2010 39th International Conference on Parallel Processing*. 2010, pp. 525–534.
- [97] N. Naksinehaboon, M. Paun, R. Nassar, C. Box Leangsuksun, and S. Scott. "High Performance Computing Systems with Various Checkpointing Schemes". In: *International Journal of Computers, Communications & Control (IJCCC)* IV (Oct. 2009), pp. 386–400.
- [98] Y. Ling, J. Mi, and X. Lin. "A variational calculus approach to optimal checkpoint placement". In: *IEEE Transactions on Computers* 50.7 (2001), pp. 699–708.
- [99] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio. "Distribution-free checkpoint placement algorithms based on min-max principle". In: *IEEE Transactions on Dependable and Secure Computing* 3.2 (2006), pp. 130–140.
- [100] K. Tang et al. "Power-Capping Aware Checkpointing: On the Interplay Among Power-Capping, Temperature, Reliability, Performance, and Energy". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016, pp. 311–322.
- [101] M. T. Rahman, H. Nguyen, J. Subhlok, and G. Pandurangan. "Checkpointing to Minimize Completion Time for Inter-Dependent Parallel Processes on Volunteer Grids". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2016, pp. 331–335.
- [102] L. Fialho, D. Rexachs, and E. Luque. "What is Missing in Current Checkpoint Interval Models?" In: *2011 31st International Conference on Distributed Computing Systems*. 2011, pp. 322–332.
- [103] Y. Zhuang, X. Wei, H. Li, Y. Wang, and X. He. "An Optimal Checkpointing Model with Online OCI Adjustment for Stream Processing Applications". In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 2018, pp. 1–9.

- [104] S. M. A. Akber, H. Chen, Y. Wang, and H. Jin. "Minimizing Overheads of Checkpoints in Distributed Stream Processing Systems". In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)* (2018), pp. 1–4.
- [105] M. Lotfi, S. A. Motamedi, and M. Bandarabadi. "Proactive blocking coordinated checkpointing with dynamic intervals". In: *2009 41st Southeastern Symposium on System Theory*. 2009, pp. 118–121.
- [106] Z. Xu, C. Men, W. Li, and X. Li. "Checkpoint scheduling model for optimality". In: *Information Processing Letters* 111.19 (2011), pp. 979–984.
- [107] Yudan Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott. "An optimal checkpoint/restart model for a large scale high performance computing system". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–9.
- [108] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott. "A reliability-aware approach for an optimal checkpoint/restart model in HPC environments". In: *2007 IEEE International Conference on Cluster Computing*. 2007 IEEE International Conference on Cluster Computing. 2007, pp. 452–457.
- [109] M. Shastry and K. Venkatesh. "A New Optimal Checkpoint Restart Model". In: 2011.
- [110] L. Zhu, J. Gu, Y. Wang, and T. Zhao. "Research on Optimum Checkpoint Interval for Hybrid Fault Tolerance". In: *Advanced Parallel Processing Technologies*. Ed. by C. Wu and A. Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 367–380.
- [111] M. Paun, N. Naksinehaboon, R. Nassar, C. Leangsuksun, S. L. Scott, and N. Taerat. "Incremental Checkpoint Schemes for Weibull Failure Distribution". In: *Int. J. Found. Comput. Sci.* 21 (2010), pp. 329–344.
- [112] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments". In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. 2008, pp. 783–788.
- [113] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee. "Compiler-enhanced Incremental Checkpointing for OpenMP Applications". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '08. Salt Lake City, UT, USA: ACM, 2008, pp. 275–276.
- [114] G. Jacques-Silva, B. Gedik, H. Andrade, and K. Wu. "Language level checkpointing support for stream processing applications". In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 2009, pp. 145–154.
- [115] Z. Sebeopou and K. Magoutis. "CEC: Continuous eventual checkpointing for data stream processing operators". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 2011, pp. 145–156.
- [116] G. Zheng, Xiang Ni, and L. V. Kal. "A scalable double in-memory checkpoint and restart scheme towards exascale". In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. 2012, pp. 1–6.

- [117] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka. "Exploration of Lossy Compression for Application-Level Checkpoint/Restart". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 914–922.
- [118] D. Ibtesham, K. B. Ferreira, and D. Arnold. "A checkpoint compression study for high-performance computing systems". In: *The International Journal of High Performance Computing Applications* 29.4 (2015), pp. 387–402.
- [119] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges. "On the Viability of Checkpoint Compression for Extreme Scale Fault Tolerance". In: *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2. Euro-Par'11*. Bordeaux, France: Springer-Verlag, 2012, pp. 302–311.
- [120] S. Di, Y. Robert, F. Vivien, and F. Cappello. "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 244–259.
- [121] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski. "Detailed Modeling and Evaluation of a Scalable Multilevel Checkpointing System". In: *IEEE Transactions on Parallel and Distributed Systems* 25.9 (2014), pp. 2255–2263.
- [122] S. Di, L. Bautista-Gome, and F. Cappello. "Optimization of a Multilevel Checkpoint Model with Uncertain Execution Scales". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 907–918.
- [123] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11.
- [124] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. "Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1181–1190.
- [125] A. Benoit, A. Cavelan, V. Le Fvre, Y. Robert, and H. Sun. "Towards Optimal Multi-Level Checkpointing". In: *IEEE Transactions on Computers* 66.7 (2017), pp. 1212–1226.
- [126] Q. Huang and P. P. C. Lee. "Toward High-Performance Distributed Stream Processing via Approximate Fault Tolerance". In: *Proc. VLDB Endow.* 10.3 (Nov. 2016), 7384.
- [127] Z. Cheng, Q. Huang, and P. P. Lee. "On the performance and convergence of distributed stream processing via approximate fault tolerance". In: *The VLDB Journal* 28.5 (2019), pp. 821–846.
- [128] G. Chang, P. Li, and G. Xu. "A Highly-Efficient Fault Tolerance Method for a Scalable Stream Processing System". In: *2017 2nd International Conference on Control, Automation and Artificial Intelligence (CAAI 2017)*. Atlantis Press. 2017.

- [129] L. Su and Y. Zhou. "Query-Centric Failure Recovery for Distributed Stream Processing Engines". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1276–1279.
- [130] M. A. U. Nasir. "Fault Tolerance for Stream Processing Engines". In: *ArXiv abs/1605.00928* (2016).
- [131] F. J. Cangialosi et al. "The Design of the Borealis Stream Processing Engine". In: *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. Asilomar, CA, 2005.
- [132] M. A. Shah, J. M. Hellerstein, and E. Brewer. "Highly Available, Fault-tolerant, Parallel Dataflows". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 827–838.
- [133] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. "Stormy: An Elastic and Highly Available Streaming Service in the Cloud". In: *Proceedings of the 2012 Joint EDBT/ICDT Workshops*. EDBT-ICDT '12. Berlin, Germany: ACM, 2012, pp. 55–60.
- [134] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya. "E-Storm: Replication-Based State Management in Distributed Stream Processing Systems". In: *2017 46th International Conference on Parallel Processing (ICPP)*. 2017, pp. 571–580.
- [135] A. Brito, C. Fetzer, and P. Felber. "Multithreading-Enabled Active Replication for Event Stream Processing Operators". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. 2009, pp. 22–31.
- [136] A. Kumbhare, M. Frincu, Y. Simmhan, and V. K. Prasanna. "Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. 2015, pp. 328–338.
- [137] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. "StreamScope: Continuous Reliable Distributed Processing of Big Data Streams". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 439–453.
- [138] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale. "Adaptive Fault-Tolerance for Dynamic Resource Provisioning in Distributed Stream Processing Systems." In: *EDBT*. 2014, pp. 85–96.
- [139] J. Fang, P. Chao, R. Zhang, and X. Zhou. "Integrating workload balancing and fault tolerance in distributed stream processing system". In: *World Wide Web* 22.6 (2019), pp. 2471–2496.
- [140] Y. Feng, N. Huang, and Y. Wu. "Efficient and Adaptive Stateful Replication for Stream Processing Engines in High-Availability Cluster". In: *IEEE Transactions on Parallel and Distributed Systems* 22.11 (2011), pp. 1788–1796.
- [141] L. Su and Y. Zhou. "Passive and Partially Active Fault Tolerance for Massively Parallel Stream Processing Engines". In: *IEEE Transactions on Knowledge and Data Engineering* 31.1 (2019), pp. 32–45.

- [142] S. Kamburugamuve, G. Fox, D. Leake, and J. Qiu. "Survey of distributed stream processing for large stream sources". In: *Grids Ucs Indiana Edu 2* (2013), pp. 1–16.
- [143] S. Dudoladov et al. "Optimistic Recovery for Iterative Dataflows in Action". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1439–1443.
- [144] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. ""All Roads Lead to Rome": Optimistic Recovery for Distributed Iterative Data Processing". In: *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*. CIKM '13. San Francisco, California, USA: ACM, 2013, pp. 1919–1928.
- [145] A. Toshniwal et al. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156.
- [146] M. Fu et al. "Twitter Heron: Towards Extensible Streaming Engines". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 1165–1172.
- [147] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo. "An experimental survey on big data frameworks". In: *Future Generation Computer Systems* 86 (2018), pp. 546–564.
- [148] S. A. Noghabi et al. "Samza: Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645.
- [149] G. Hesse and M. Lorenz. "Conceptual Survey on Data Stream Processing Systems". In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 2015, pp. 797–802.
- [150] D. Namiot. "On Big Data Stream Processing". In: *International Journal of Open Information Technologies* 3 (Jan. 2015), pp. 48–51.
- [151] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak. "Cloud-based Data Stream Processing". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: ACM, 2014, pp. 238–245.
- [152] U. Srivastava and J. Widom. "Flexible Time Management in Data Stream Systems". In: *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '04. Paris, France: ACM, 2004, pp. 263–274.
- [153] *StreamBase Complex Event Processing Overview*. <http://www.tibco.com/products/event-processing/complex-event-processing/streambase-complex-event-processing>.
- [154] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. "Collective Communication: Theory, Practice, and Experience: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 19.13 (Sept. 2007), pp. 1749–1783.
- [155] W. D. Hillis and G. L. Steele Jr. "Data Parallel Algorithms". In: *Commun. ACM* 29.12 (Dec. 1986), pp. 1170–1183.
- [156] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. <http://snap.stanford.edu/data>.

- [157] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. "StreamKM++: A Clustering Algorithm for Data Streams". In: *J. Exp. Algorithmics* 17 (May 2012), 2.4:2.1–2.4:2.30.
- [158] F. Petrini, K. Davis, and J. C. Sancho. "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 209–.
- [159] O. Subasi. "Reliability for exascale computing : system modelling and error mitigation for task-parallel HPC applications". PhD thesis. Universitat Politècnica de Catalunya: UPC, Departament d'Arquitectura de Computadors, 2016.
- [160] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience". In: *The International Journal of High Performance Computing Applications* 23.4 (2009), pp. 374–388. eprint: <https://doi.org/10.1177/1094342009347767>.
- [161] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38 (2015), pp. 28–38.
- [162] R. Chen et al. "Replication-Based Fault-Tolerance for Large-Scale Graph Processing". In: *IEEE Transactions on Parallel and Distributed Systems* 29.7 (2018), pp. 1621–1635.
- [163] L. Belcastro, F. Marozzo, and D. Talia. "Programming models and systems for Big Data analysis". In: *International Journal of Parallel, Emergent and Distributed Systems* 34.6 (2019), pp. 632–652.
- [164] A. Gradwohl, H. Senger, L. Arantes, and P. Sens. "Comparing Distributed Online Stream Processing Systems Considering Fault Tolerance Issues". In: *Journal of Emerging Technologies in Web Intelligence* 6 (May 2014), pp. 174–179.
- [165] G. Brettlecker, H. Schuldt, and H.-J. Schek. "Efficient and Coordinated Checkpointing for Reliable Distributed Data Stream Management". In: *Advances in Databases and Information Systems*. Ed. by Y. Manolopoulos, J. Pokorný, and T. K. Sellis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 296–312.
- [166] S. M. A. Akber, H. Chen, Y. Wang, and H. Jin. "Minimizing Overheads of Checkpoints in Distributed Stream Processing Systems". In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. 2018, pp. 1–4.
- [167] X. Yang, Z. Wang, J. Xue, and Y. Zhou. "The Reliability Wall for Exascale Supercomputing". In: *IEEE Transactions on Computers* 61.6 (2012), pp. 767–779.
- [168] R. Bianchini and T. A. El-Ghazawi. "System Resilience at Extreme Scale White Paper". In: 2009.
- [169] *Apache Flink Documentation*. <https://ci.apache.org/projects/flink/flink-docs-stable/>.
- [170] M. A. U. Nasir, G. De Francisci Morales, D. Garca-Soriano, N. Kourtellis, and M. Serafini. "The power of both choices: Practical load balancing for distributed stream processing engines". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015 IEEE 31st International Conference on Data Engineering. 2015, pp. 137–148.

- [171] F. Riaz and K. M. Ali. "Applications of Graph Theory in Computer Science". In: *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*. 2011 Third International Conference on Computational Intelligence, Communication Systems and Networks. 2011, pp. 142–145.
- [172] C. T. Johnston and D. G. Bailey. "FPGA implementation of a Single Pass Connected Components Algorithm". In: *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. 4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008). 2008, pp. 228–231.
- [173] M. Snir et al. "Addressing Failures in Exascale Computing". In: *Int. J. High Perform. Comput. Appl.* 28.2 (May 2014), pp. 129–173.
- [174] T. Herault and Y. Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st. Springer Publishing Company, Incorporated, 2015.
- [175] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. "Optimizing Checkpoints Using NVM as Virtual Memory". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 29–40.
- [176] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. "An Analysis of Multilevel Checkpoint Performance Models". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 783–792.