Delft University of Technology

Adaptivity for Streaming Dataflow Engines

Siachamis, G.

DOI
10.4233/uuid:7d364f56-d84a-4cb0-84cb-4c317d275373

Publication date
2024

Document Version
Final published version

Citation (APA)
Siachamis, G. (2024). *Adaptivity for Streaming Dataflow Engines*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:7d364f56-d84a-4cb0-84cb-4c317d275373

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Adaptivity for Streaming Dataflow Engines

Adaptivity for Streaming Dataflow Engines

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology,
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates,
to be defended publicly on
Monday 26 November 2024 at 12.30 o'clock

by

Georgios SIACHAMIS

Master of Engineering in Electrical and Computer Engineering,
National Technical University of Athens, Griekenland,
born in Marousi, Greece.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. A. van Deursen,	Delft University of Technology, Promotor
Prof. dr. ir. G.J.P.M. Houben,	Delft University of Technology, Promotor
Dr. A. Katsifodimos,	Delft University of Technology, Copromotor

Independent members:

Prof. dr. M. Garofalakis,	Technical University of Crete, Greece
Dr. V. Kalavri,	Boston University, USA
Prof. dr. G. Smaragdakis,	Delft University of Technology
Prof. dr. Y. Zhou,	University of Copenhagen, Denmark
Prof. dr. K.G. Langendoen,	Delft University of Technology, reserve member

This research was partially supported by AI for Fintech Research, member of the ICAI Labs.

SIKS Dissertation Series No. 2024-41
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Keywords: stream processing, adaptivity, similarity joins, load balancing, autoscaling, checkpointing, fault tolerance

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

Copyright © 2024 by Georgios Siachamis
ISBN 978-94-6366-953-5

The measure of intelligence is the ability to change.

Albert Einstein

Contents

Summary **xi**

Samenvatting **xiii**

Acknowledgments **xv**

1 Introduction **1**
1.1 Stream processing in the Cloud . . . 3
1.1.1 Adapting to statistical changes . . . 5
1.1.2 Adapting to infrastructure failures . . . 8
1.1.3 Adapting to input rate changes . . . 9
1.2 Main Research Questions . . . 12
1.3 Methodology . . . 13
1.4 Contributions . . . 14
1.5 Thesis Origins . . . 15

2 Adaptive Distributed Streaming Similarity Joins **17**
2.1 Introduction . . . 18
2.2 Preliminaries . . . 19
2.3 Problem Statement . . . 20
2.4 Related Work . . . 22
2.5 Approach Overview . . . 24
2.6 Space Partitioning . . . 26
2.6.1 Space partitions . . . 26
2.6.2 Selecting Centroids . . . 26
2.6.3 Avoiding duplicate comparisons . . . 26
2.7 Workset Formulation . . . 27
2.7.1 Step 1: Deciding Inner vs. Outer Partition . . . 27
2.7.2 Step 2: Assign to an Inner Set . . . 28
2.7.3 Step 3: Creating New Worksets . . . 28
2.7.4 Step 4: Labeling Outliers . . . 28
2.7.5 Step 5: Assign to Outer Sets . . . 29
2.7.6 Set Boundaries in Metric Space . . . 29
2.7.7 Similarity Computations . . . 31
2.8 Adaptive Workset Balancing . . . 31
2.8.1 Migrating Worksets W/O Repartitioning . . . 31
2.8.2 Workset-Balancing vs. Job-Scheduling . . . 32
2.8.3 The Workset Balancing Algorithm . . . 32

2.9 Experiments . . . 34
2.9.1 Performance Metrics . . . 34
2.9.2 Datasets . . . 35
2.9.3 Experimental Setup . . . 35
2.9.4 Baseline: ClusterJoin . . . 35
2.9.5 Partitioning Performance. . . . 35
2.9.6 Benefits of Load Balancing . . . 38
2.9.7 Summary of experiments. . . . 39
2.10 Conclusions . . . 40

3 Evaluating Checkpointing Protocols for Streaming Dataflows 41
3.1 Introduction . . . 42
3.2 Preliminaries . . . 43
3.2.1 Processing Semantics. . . . 44
3.2.2 Consistency of Global State . . . 46
3.3 Checkpointing Protocols . . . 47
3.3.1 Coordinated Aligned Checkpointing (COOR). . . . 47
3.3.2 Uncoordinated Checkpointing (UNC) . . . 49
3.3.3 Communication-induced Checkpointing (CIC) . . . 51
3.4 Testbed System. . . . 52
3.5 Metrics. . . . 52
3.6 Streaming Query Workload . . . 53
3.7 Experimental Evaluation . . . 55
3.7.1 Evaluation setup . . . 55
3.7.2 Results . . . 55
3.8 Related Work. . . . 64
3.9 Conclusions . . . 64

4 Evaluating Stream Processing Autoscalers 65
4.1 Introduction . . . 66
4.2 Background . . . 67
4.2.1 Autoscaling Process . . . 67
4.2.2 Common notions. . . . 68
4.3 Related Work. . . . 68
4.4 Control-based Autoscalers . . . 70
4.4.1 Dhalion . . . 70
4.4.2 DS2 . . . 71
4.4.3 HPA . . . 71
4.4.4 HPA-Varga . . . 72
4.5 Evaluation Components . . . 73
4.5.1 Performance Evaluation Metrics . . . 73
4.5.2 Queries. . . . 73
4.5.3 Workloads . . . 74
4.5.4 Discussion . . . 76

4.6 Experimental Evaluation . . . 77
4.6.1 Experimental Setup . . . 77
4.6.2 Workload Comparison . . . 78
4.6.3 Query Comparison . . . 82
4.6.4 Convergence comparison . . . 83
4.6.5 Summary of findings . . . 85
4.7 Conclusion . . . 87

5 Conclusion 89
5.1 Main Findings . . . 90
5.1.1 Adaptivity to statistical changes for streaming similarity joins . . . 90
5.1.2 Recovering from infrastructure failures using checkpoints . . . 91
5.1.3 Adapting to input rate changes using automated solutions . . . 92
5.2 Limitations . . . 93
5.3 Future Research Directions . . . 94
5.3.1 Evaluating MapReduce solutions in a streaming environment . . . 94
5.3.2 Learned Partitioning for Streaming Similarity Joins . . . 94
5.3.3 Hybrid Checkpointing for Streaming Dataflows . . . 95
5.3.4 Tackling Adaptivity Problems with One Stone . . . 95
5.3.5 Rethinking Stream Processing Benchmarking . . . 95

Bibliography 97

Curriculum Vitæ 117

List of Publications 119

SIKS Dissertation Series 121

Summary

Data processing has heavily evolved in the last two decades, from single-node processing to distributed processing and from the MapReduce paradigm to the stream processing paradigm. At the same time, cloud computing has emerged as the primary means of deploying and operating a data processing system. In the cloud era, flexible resource allocation combined with flexible pricing schemes have brought forward new opportunities and have democratized access to computing resources. However, streaming dataflow or stream processing engines were originally designed for in-house clusters of fixed resources with limited needs for adaptivity. Therefore, they lack the mechanisms to adapt to unexpected changes in the needs of the processing workload. When solutions have been proposed in the literature, their experimental evaluation is limited hindering the progress of the field. The same applies to the native fault tolerance mechanisms that virtually every stream processing engine employs. In this thesis, we study the problem of adaptivity for streaming dataflow engines, and we focus on three major adaptivity subproblems: adaptivity to *i*) statistical changes, *ii*) infrastructure failures, and *iii*) input rate changes.

In Chapter 2, we study adaptivity to statistical changes through the important task of streaming similarity joins that is heavily affected by imbalanced loads, a by-product of statistical changes. We propose S^3J; the first adaptive distributed streaming similarity joins method in the general metric space that employs a two-layered adaptive partitioning scheme to reduce unnecessary similarity computations and distribute the load to the available workers. Our partitioning scheme is paired with an efficient load balancing scheme that leverages the existing partitioning in order to rebalance any imbalanced load. Our results show that S^3J outperforms the employed baseline, inspired by a MapReduce method, in terms of partitioning efficiency. Additionally, our experiments show that the load balancing scheme can gradually defuse the imbalanced load and involve all the available workers in the processing.

The majority of the stream processing engines employ a checkpoint-based fault tolerance mechanism. In Chapter 3, we look at the adaptivity to infrastructure failures through the existing checkpointing protocols. We propose CheckMate, a principled experimental framework for evaluating checkpointing protocols for streaming dataflows. First, we summarize all the essential preliminaries required to study checkpoint-based fault tolerance. Then, we discuss in detail, implement, and evaluate in different scenarios the three main checkpointing protocols. Our evaluation shows that when the load is uniformly distributed, the implemented by most stream processing engines coordinated checkpointing protocol outperforms the alternatives. However, the uncoordinated prevails in the presence of skew, while it shows no domino effect when cyclic queries are employed.

Finally, in Chapter 4, we address the problem of adaptivity to input rate changes. Although multiple solutions have been proposed, their experimental evaluation is shallow and does not include detailed comparisons with other solutions. We propose a principled evaluation framework for stream processing autoscalers. We establish important metrics,

queries, and workloads in order to provide guidelines for the evaluation of autoscaling solutions for stream processing. We discuss the state-of-the-art control-based autoscalers, and we evaluate them using the proposed framework. Our results show that, for complex queries, none of the evaluated autoscalers can adapt efficiently, while for simple stateless queries, a simple generic autoscaler outperforms the solutions tailored to stream processing.

We conclude this thesis by summarizing our main findings and discussing the limitations of our work. Based on the valuable insights we gained while designing and implementing the research work included in this thesis, we propose a series of interesting and important future research directions that are not limited to adaptivity problems but address stream processing in general.

Samenvatting

Gegevensverwerking is de afgelopen twee decennia sterk geëvolueerd, van verwerking op een enkele node tot gedistribueerde verwerking en van het MapReduce-paradigma tot het streamverwerkingsparadigma. Tegelijkertijd is cloud computing naar voren gekomen als het primaire middel voor het implementeren en opereren van een gegevensverwerkingssysteem. In het cloudtijdperk hebben flexibele toewijzing van middelen, gecombineerd met flexibele prijsschema's, nieuwe kansen gebracht en de toegang tot rekenmiddelen gedemocratiseerd. Echter, streaming dataflow of streamverwerkingsengines zijn oorspronkelijk ontworpen voor interne clusters van vaste middelen met beperkte behoefte aan adaptiviteit. Daarom missen ze de mechanismen om zich aan te passen aan onverwachte veranderingen in de behoeften van de verwerkingswerkbelasting. Wanneer oplossingen zijn voorgesteld in de literatuur, is hun experimentele evaluatie beperkt, wat de vooruitgang van het veld belemmert. Hetzelfde geldt voor de native fouttolerantiemechanismen die vrijwel elke streamverwerkingsengine hanteert. In deze scriptie bestuderen we het probleem van adaptiviteit voor streaming dataflow-engines, en richten we ons op drie belangrijke adaptiviteitssubproblemen: adaptiviteit aan *i*) statistische veranderingen, *ii*) infrastructuurstoringen en *iii*) veranderingen in de inputrate.

In Hoofdstuk 2 bestuderen we adaptiviteit aan statistische veranderingen door de belangrijke taak van streaming gelijkenisjoins, die sterk worden beïnvloed door onevenwichtige belastingen, een bijproduct van statistische veranderingen. We stellen S^3J voor; de eerste adaptieve gedistribueerde streaming gelijkenisjoins-methode in de algemene metrische ruimte die een tweeledig adaptief partitioneringsschema toepast om onnodige gelijkenisberekeningen te verminderen en de belasting te verdelen over de beschikbare werkers. Ons partitioneringsschema wordt gecombineerd met een efficiënt load-balancing schema dat gebruik maakt van de bestaande partitionering om elke onevenwichtige belasting opnieuw in evenwicht te brengen. Onze resultaten laten zien dat S^3J de gebruikte baseline, geïnspireerd door een MapReduce-methode, overtreft in termen van partitioneringsefficiëntie. Daarnaast tonen onze experimenten aan dat het load-balancing schema geleidelijk de onevenwichtige belasting kan verminderen en alle beschikbare werkers kan betrekken bij de verwerking.

De meeste streamverwerkingsengines gebruiken een checkpoint-gebaseerd fouttolerantiemechanisme. In Hoofdstuk 3 bekijken we de adaptiviteit aan infrastructuurstoringen door de bestaande checkpointing-protocollen. We stellen CheckMate voor, een principieel experimenteel raamwerk voor het evalueren van checkpointing-protocollen voor streaming dataflows. Eerst vatten we alle essentiële basisprincipes samen die nodig zijn om fouttolerantie op basis van checkpoints te bestuderen. Vervolgens bespreken we gedetailleerd, implementeren we, en evalueren we in verschillende scenario's de drie belangrijkste checkpointing-protocollen. Onze evaluatie toont aan dat wanneer de belasting gelijkmatig verdeeld is, het door de meeste streamverwerkingsengines geïmplementeerde gecoördineerde checkpointing-protocol de alternatieven overtreft. Echter, het ongecoördineerde

protocol blijkt beter te presteren in de aanwezigheid van scheefheid en vertoont geen domino-effect wanneer cyclische queries worden gebruikt.

Ten slotte behandelen we in Hoofdstuk 4 het probleem van adaptiviteit aan veranderingen in de inputrate. Hoewel er meerdere oplossingen zijn voorgesteld, is hun experimentele evaluatie oppervlakkig en omvat het geen gedetailleerde vergelijkingen met andere oplossingen. We stellen een fundamenteel evaluatiekader voor autoscalers bij streamverwerking voor. We stellen belangrijke metrics, queries en workloads vast om richtlijnen te bieden voor de evaluatie van autoscaling-oplossingen voor streamverwerking. We bespreken de meest geavanceerde op controle gebaseerde autoscalers en evalueren ze met behulp van het voorgestelde evaluatiekader. Onze resultaten tonen aan dat geen van de geëvalueerde autoscalers efficiënt kan aanpassen aan complexe queries, terwijl voor eenvoudige stateless queries een eenvoudige generieke autoscaler beter presteert dan de oplossingen die zijn toegespitst op streamverwerking.

We sluiten deze scriptie af door onze belangrijkste bevindingen samen te vatten en de beperkingen van ons werk te bespreken. Op basis van de waardevolle inzichten die we hebben verkregen tijdens het ontwerpen en implementeren van het onderzoekswerk dat in deze scriptie is opgenomen, stellen we een reeks interessante en belangrijke toekomstige onderzoekrichtingen voor die niet beperkt zijn tot adaptiviteitsproblemen, maar streamverwerking in het algemeen aanpakken.

Acknowledgments

This dissertation is a collection of the important scientific results and insights gained during the short four-year period of my Ph.D. trajectory. It results from hard work and endurance but only represents a fraction of the lessons learned and personal growth during this academic journey. Becoming an independent researcher and, most importantly, a better person requires personal effort. Still, it is also owed to those who inspired, enabled, and supported me throughout this challenging journey. Therefore, I would like to extend my gratitude to all the people who, in their way, made it possible.

To my family: my parents, my grandparents, and my brother, for all their sacrifices toward my education and for equipping me with principles that proved invaluable toward this Ph.D. and helped me become a better person.

To my close friends, who always supported me and believed in my abilities despite my periodical lack of responsiveness to their calls and messages.

To Gogo, who endured the most and never stopped believing in me, who was always there in the highs and the lows, eager to listen, discuss my fears and troubles, and congratulate me on my successes.

To Kyriakos, Christos, and Agathe, who made the last four years memorable and fun. I learned a lot from all of you and am grateful to have been your friend. This is not the end, just another beginning.

To George, with whom I had countless discussions about basketball and academia despite our short overlap, and who I wish I had the chance to work with more extensively.

To Garrett, Lorenzo, Alisa, Ujwal, David, Ziyu, Andra, Mireia, Gaole, Lijun, Jurek, Sara, Kostas, and Katerina, with whom I share great memories from Delft and who have helped me feel like home.

To all my old and new colleagues in WIS and AFR, with whom I had countless exciting conversations and who have helped me broaden my horizons with their research and their stories.

To my promotors, Arie and Geert-Jan, who did their best to provide everything I needed in this journey. Our discussions have always been valuable lessons for me.

To Asterios, who was not only a supervisor but also a mentor and a friend and who has influenced me in ways that I still have not realized.

To Marios, who was always there to help and guide me during this Ph.D.

To my doctoral committee members who have honored me with their presence and their reviews.

To Katerina, Stamatis, and Manos, who inspired me to pursue a Ph.D. title.

With this dissertation, I am concluding an exciting academic journey toward the Ph.D. title. A journey that would not have been possible and successful without all of you.

Georgios
Delft, November 2024

1 Introduction

Processing data efficiently has been a major problem since the very beginning of the digitization of everyday life and the introduction of personal computers. As computing power increases, digital content and computer applications take over traditional everyday tasks. For example, traditional newspapers are losing more and more ground to online news websites and digital newspapers. This increasing digitization has led to ever-increasing volumes of data, leading to what is known as the era of Big Data [140, 104, 118]. The *Big Data* era is characterized by data of high *volume*, *variety* and *velocity* [118, 140]. In simple terms, *volume* refers to the massive size of the accumulated data. Modern enterprises create and collect an abundance of data daily that is hard to process by traditional means. At the same time, these data can come from various sources and in different formats, resulting in high *variety*. Finally, not only are data produced fast, but they also need to be processed fast to enable fast-paced decision-making to adhere to the needs of modern enterprises. This requirement for fast-paced processing embodies the *velocity* aspect of *Big Data*.

To tackle the rising challenges of *Big Data* and provide efficient data processing, the research community and pioneering partners from industry have devised a plethora of different approaches: new data structures and techniques have been introduced to speed up data processing. Indices tailored for specific use cases and data profiles [58, 79, 105, 80, 100] have been employed in order to track the location of data and reduce the time spent in accessing data and discarding irrelevant data. In addition, different types of synopses [45, 117, 133, 34] have been proposed to improve the efficiency of data processing tasks by approximating the computations without significantly compromising the quality of the results.

Another approach to solving the problem of efficient data processing involves rethinking the employed processing paradigms. To leverage the multiprocessing capabilities of powerful servers, the message passing interface (MPI) was employed to scale up data processing [153], while distributed data processing was introduced [98] in order to surpass the computing limitations of single machines. However, the most impactful processing paradigm that revolutionized data analytics over the last two decades is the MapReduce paradigm [50, 51]. MapReduce divides processing into three stages: the *Map*, the *Shuffle*, and the *Reduce* stage. During the *Map* stage, the data is split into small chunks and shipped

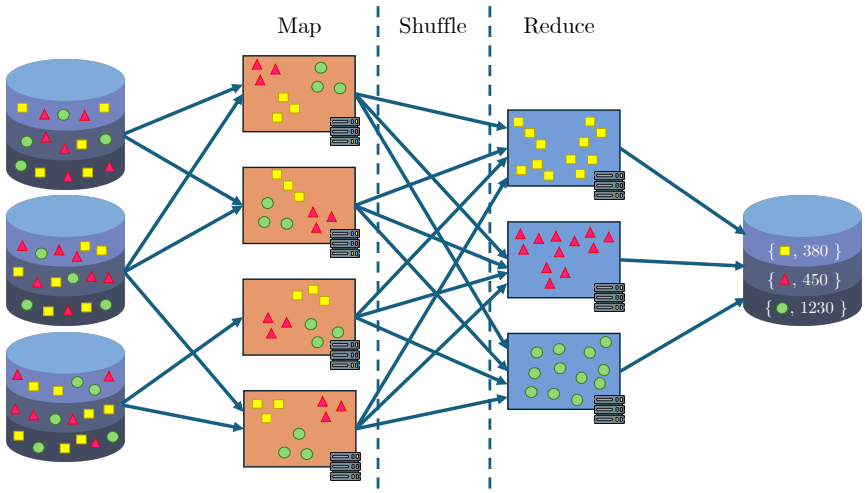


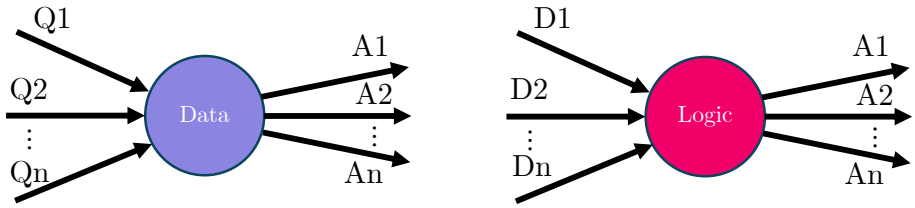
Figure 1.1: An example application using the MapReduce paradigm.

to workers called Mappers, where a mapping function is applied to all data of every chunk. Then, the intermediate output of the Mappers is sorted, combined, and shuffled based on the output keys provided by the Mappers so that all items with the same key will be grouped and shipped to the same downstream worker. In the *Reduce* stage, the downstream workers, called Reducers, apply a reduce function on the combined data of the shuffle stage and output the final result to the durable storage or feed it as input to another MapReduce pipeline.

A typical MapReduce processing pipeline is illustrated in figure 1.1, which counts the number of instances of each shape. The shapes are ingested from two sources. The input is split into small chunks and loaded to the Mappers, which distinguish the instances of different shapes and assign a unique output key to each shape category. Then, in the shuffle stage, the shapes are grouped and routed to the Reducer responsible for the specific key. The reducer counts the instances of each shape and outputs a single tuple containing the shape and its counter. Although our example employs shapes as input, the applicability of MapReduce is not limited to a specific input, as MapReduce can handle a variety of input types.

Despite the huge adoption of the MapReduce processing paradigm in the 2010s, it still imposed limitations. Although it can effectively deal with the *volume* and *variety* of *Big Data*, it cannot tackle sufficiently the *velocity* aspect of *Big Data*. Modern enterprises rely on fast, real-time data processing to make executive decisions on the fly. At the same time, they handle time-sensitive data that need to be processed as soon as they are produced in order to be able to extract any business value. Although the MapReduce paradigm can speed up data processing tasks by efficiently parallelizing any task, it cannot provide real-time results. That is mainly because of its design; subsequent stages can only start when the previous stage is finished.

The stream processing paradigm has re-emerged in the last decade to tackle the *velocity* of Big Data and enable real-time data processing and real-time results. Stream processing



(a) The Controlflow model

(b) The Dataflow model

Figure 1.2: Prevailing processing models.

is a paradigm that first appeared in the 1960s and historically came in different formalizations and flavors [154]. However, the *Dataflow* processing model has been the prevailing formalization for the stream processing paradigm in the last two decades [66]. In contrast to traditional processing models, like *Controlflow* (figure 1.2a), that focus on executing queries or instructions on top of static data, *Dataflow* (figure 1.2b) has data as its first-class citizen, focusing on the movement of the data throughout the processing pipeline and the transformation of them through the static operations. Most modern engines dedicated to stream processing [31, 73, 35, 46] implement the *Dataflow* model.

Figure 1.3 showcases the same example of counting the instances of different shapes, previously introduced for the MapReduce paradigm, using the stream processing paradigm. Our stream processing pipeline consists of interconnected operators, the fundamental unit of processing, that can reside in any available worker and are responsible for a single specific operation. An operator can have parallel instances that perform the same operation. For example, the map operator M in figure 1.3 has two parallel instances, M1 and M2, performing the same mapping operation. In contrast to MapReduce, where the data are bulk loaded to the workers of each stage, in the stream processing paradigm, data are ingested one by one and flow through the different operators of the pipeline. Once a map operator processes an ingested shape, the tuple is forwarded to the responsible parallel instance of the aggregating operator, which updates the running counter and outputs the current value of the current for the corresponding shape.

1.1 Stream processing in the Cloud

Regardless of the processing paradigm employed, an abundance of resources is required for efficient processing in the era of *Big Data*. Whether the processing requires a single high-end multi-core machine or is distributed to multiple commodity machines, the overall resources required are expensive and difficult to manage. Although large high-tech companies can build and maintain big clusters of high-end machines that can accommodate all their processing needs, small and medium-sized businesses (SMBs) struggle to secure and manage the required resources and, therefore, compete and prosper.

A solution to this disparity of access to computing resources is *cloud computing*. Recognizing the difficulty of acquiring and managing your own computing resources, the industry has decided to productize its infrastructure and provide hosting services by providing a variety of service models, such as the Infrastructure-as-a-Service (IaaS), Software-as-a-Service

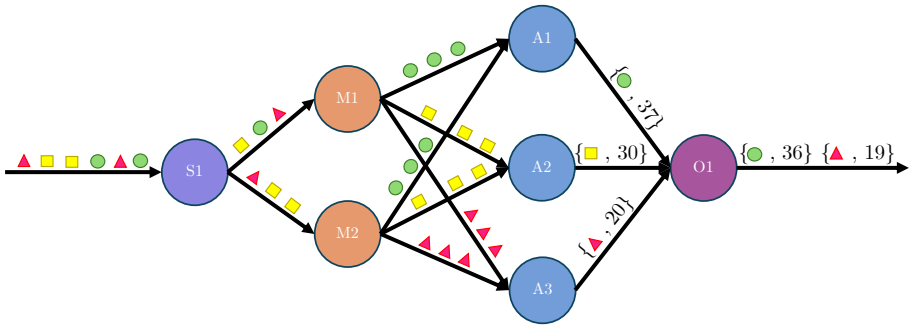


Figure 1.3: A stream processing pipeline. S1 is the source operator, while O1 is the output operator of the pipeline. M1 and M2 are parallel instances of the map operator. Similarly, A{1-3} are parallel instances of the aggregate operator.

(SaaS), or Function-as-a-Service (FaaS) models. Turning themselves to cloud providers, the field-leading high-tech companies not only have found a new source of profit but have also helped considerably with democratizing access to computing resources. *Cloud computing* offers features that can significantly alleviate the hurdles of managing on-premises infrastructure and provide opportunities for significant cost reduction. The different service models allow small or medium-sized businesses to outsource their infrastructure, reducing operational costs and optimizing their hardware according to their processing needs. The flexible pricing schemes the cloud providers introduce allow for more opportunities to save on operational expenses. The popular pay-as-you-go pricing scheme allows cloud users, including small and medium-sized businesses, to acquire and release resources according to their needs dynamically and only pay for their current use.

Although *cloud computing* provides multiple opportunities for cutting costs, leveraging these opportunities remains difficult. Without the proper tools for dynamically managing cloud resources, operational teams comprised of highly skilled experts are needed. However, forming such a team can be cumbersome and expensive for SMBs that cannot compete with big firms in acquiring very specialized and experienced staff. Therefore, despite the easiness and flexibility of resource allocation in the cloud, significant portions of the cloud processing budgets are wasted due to inefficient resource allocation. The problem is aggravated when stream processing is considered. Stream processing requires additional expertise from operations teams, but more importantly, it encompasses dynamic real-time workloads that can be affected by frequent shifts or variations in the statistical properties of the input load, such as load distribution and input rate, leading to inadequate performance and service-level agreement violations.

In theory, the dynamic nature of stream processing makes *cloud computing* a good fit. The flexibility of resource allocation that the cloud provides matches the need to adapt to the frequent changes in the workload of a stream processing pipeline. However, modern stream processing engines are designed primarily for in-house clusters of fixed resources and lack native tools and mechanisms to adapt to unexpected changes in workloads and failures automatically. Therefore, they cannot leverage the opportunities raised by *cloud computing*. In this thesis, we focus on three main problems of adaptivity for stream processing. More

specifically, we discuss adaptivity to *i*) *statistical changes*, *ii*) *infrastructure failures*, and *iii*) *input rate changes*.

1.1.1 Adapting to statistical changes

A major challenge in data processing is distributing the load of a keyed query equally to available workers to leverage the available resources optimally. The real-time nature of stream processing does not allow for predefined solutions for handling the data or pre-processing the data to create an optimized processing plan. Historical data can be leveraged to decide on an optimized initial configuration. However, they are often not available or are obsolete in terms of the current input properties. Additionally, historical data can provide little information on the unpredictable changes that may occur during execution and, therefore, cannot be leveraged effectively in order to provide adaptivity during execution. Static distribution strategies relying on fixed load distribution to the available resources are bound to suffer from statistical changes in the input load. Such statistical changes include distribution and heavy-hitter shifts. In the former case, the input pattern shifts completely to a new distribution pattern, for example, from a uniform distribution to a Zipfian distribution. In the latter case, there is a shift in the keys that are more frequently appearing in the input, also known as heavy hitters. These types of shifts are common in real-world applications, such as analyzing the traffic of a webshop before and during a sales period and especially when dealing with social media data.

Failing to adapt to these statistical changes can be detrimental to the performance of a stream processing system. The imbalanced load can lead to straggler nodes that cannot keep up with the input load, resulting in high latency and, therefore, failing to adhere to the real-time requirements of stream processing applications. The existence of stragglers leaves the system in an unhealthy state that can, in turn, cause other issues, like hardware failures or quality issues. Additionally, in the presence of the flexible pricing schemes of *cloud computing*, the imbalanced load results in idle nodes that are not efficiently used but are still paid for. In fact, idle nodes [61, 77, 10, 62] are reported to be a significant factor in the wasted budget allocated to cloud infrastructure and computations.

Not all data processing tasks are equally influenced by an imbalanced load. Simple, low computational complexity tasks, such as filtering on exact values or map functions, are more robust to load imbalances, while tasks with high computational complexity are severely affected. Due to their computational complexity, similarity-based tasks belong to the latter, and their performance significantly deteriorated in the presence of skewed input loads. Therefore, similarity-based tasks are a prime application to study with respect to adaptivity to statistical changes. We distinguish two main similarity-based tasks: *similarity search* and *similarity joins*.

Similarity search is the task of identifying the top k most similar items to a given item given a similarity metric. The number of retrieved similar items k is application-related and user-provided. Usually, the number of retrieved items k is relatively low as users require only a handful of similar items. Multiple applications require such a *similarity search* task [55, 99, 130]. Similarity search comes in many flavors and has been studied extensively in the literature. For applications that require retrieving the exact top k similar matches, exact approaches have been proposed [92, 177, 161]. When reduced runtime is preferred, and accuracy constraints can be relaxed, approximate approaches can be

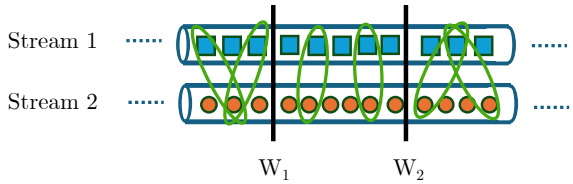


Figure 1.4: An example of a windowed streaming similarity join between two streams.

employed [75, 88, 16]. The scalability of the task has also been investigated, especially under the MapReduce paradigm [160, 156, 96, 143]. Similarity search has also been studied in the context of time series [169, 144, 134], and stream processing [157, 145, 26].

A *similarity join* is the task of identifying all similar pairs between the items of two datasets given a similarity metric and a similarity threshold. Two items are considered similar if and only if their similarity is above the user-defined threshold. Similarity joins constitute a basic and important task in many data processing pipelines, such as entity resolution pipelines [129, 128], detecting coalition in advertising [122], and data cleaning [40]. Similarly to *similarity search*, *similarity joins* have also been studied for approximate [101, 142] and exact [20, 168] results. Due to the high complexity of the task, a significant effort has been invested in scaling similarity joins, primarily through the MapReduce paradigm. Multiple solutions have been proposed for efficient and adaptive MapReduce-based similarity joins [64, 167, 53, 48]. These solutions reduce the number of similarity computations performed without affecting the completeness of the join results by efficiently partitioning the data. Some of these works also consider the problem of skewed data and perform a repartitioning step after the *map* stage. Additionally, there are MapReduce solutions that explicitly target load balancing either specifically for the similarity join problem [8, 9, 23] or general MapReduce jobs [103]. Limited work has also targeted the stream processing environment.

Similar to traditional *similarity joins*, *streaming similarity joins* (figure 1.4) try to identify all similar pairs of items between two streams of data given a similarity metric and a similarity threshold. A streaming similarity join can either be applied to recurring time windows of specific intervals or to the entirety of the ingested input, in which case we refer to the join as a full-history join or a global window join. Following the main principles of stream processing, real-time results are required, and each operator can only process the same data once. The efficiency and adaptivity to the data shifts of the streaming similarity join task are crucial in order to facilitate the streaming transformation of important business tasks and enable real-time decisions.

In this thesis, we are specifically interested in *exact* streaming similarity joins and how this task can be optimized for distributed stream processing execution while allowing it to adapt to unknown input distributions and distribution shifts. Exact streaming similarity joins are important when the high-level use case cannot tolerate missing pairs. Such use cases include monitoring crucial infrastructure, online trading, or online health monitoring. Figure 1.5 illustrates a monitoring application that involves streaming similarity joins. Within a large organization, different teams are responsible for developing, operating, and monitoring their own applications. These teams use different monitoring tools that may represent the same company assets differently. A centralized incident response team

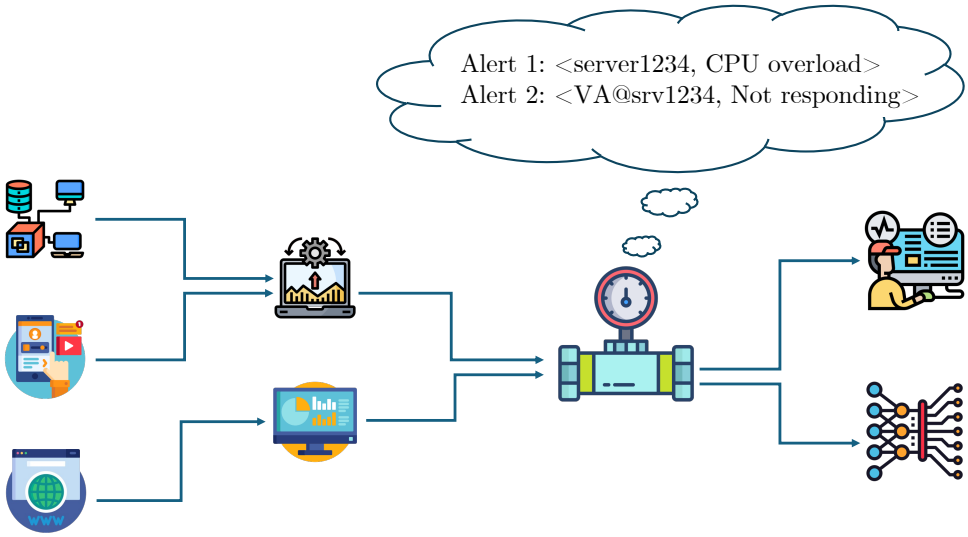
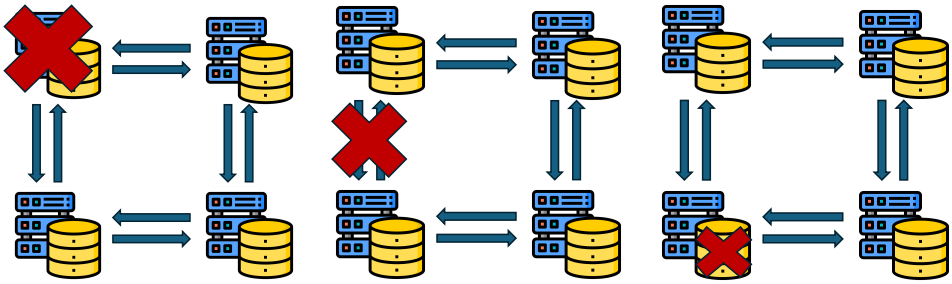


Figure 1.5: Streaming similarity joins employed in an infrastructure monitoring application. Within a large organization, different teams use different monitoring tools to monitor their assets (hardware or software). These different monitoring tools may represent the same assets differently. However, a major incident response team wants to match the produced alerts that contain relevant information in order to enrich the information provided to downstream tasks, such as root cause analysis or incident prediction model training.

collects all the alerts and monitoring messages produced by each team's monitoring tools and uses the collected information to respond to major incidents, train incident prediction machine learning models, and perform root cause analysis. Combining information from different teams that refer to the same asset directly or indirectly can heavily improve the performance of the downstream tasks. For example, as we see in figure 1.5, alert 1 contains information about a physical server, while alert 2 contains information about a video application running on this server. Combining these two alerts can help the incident team understand better the reasons behind the unexpected performance of the video application. However, manually retrieving all the related information is difficult and time-consuming. On the scale of a large organization, thousands of monitoring messages and alerts are produced every second, and the occurring incidents require immediate responses. Therefore, the ability to combine in real-time all the relevant information is of utmost importance. Providing efficient and adaptive streaming similarity joins will enable such crucial applications.

To this end, several solutions have been proposed to enable *streaming similarity joins*. In [49], the authors address the problem of streaming similarity self-joins on a single node by introducing the notion of a time-dependent similarity and employing time-based indices. Other works [60, 138, 78] target the load balancing aspect of the general streaming joins problem. However, they do not optimize for similarity joins, and they do not attempt to reduce the number of unnecessary computations but rather focus only on the load balancing aspects. The only solution addressing all the relevant aspects of our problem is proposed by [174]. The authors propose a distributed streaming set-similarity join method



(a) Worker Failure (b) Network Failure (c) Storage Failure

Figure 1.6: Types of failures in distributed stream processing.

tailored for sets of words. To reduce the number of unnecessary computations and scale up the computation, they propose a length-based filter and a hierarchical structure of bundles. However, the employed length-based partitioning has several limitations: *i*) it is only suitable for the specific problem of set-similarity, *ii*) it has limited scaling and load balancing capabilities, and *iii*) it cannot be generalized to apply to a broader set of application settings.

1.1.2 Adapting to infrastructure failures

An imbalanced load not only affects the system's performance but also has a big impact on its health. An unhealthy system is more prone to hardware failures. Additionally, the transition to cloud deployments makes handling failures from stream processing engines even more important. In order to maximize their profit and the availability of their services, cloud providers have moved from specialized hardware to commodity hardware that is more prone to failures but less expensive and of higher availability. We can distinguish three main types of failures from which cloud clusters may suffer (figure 1.6). A common failure type relates to malfunctions in the hardware or the software of a worker (figure 1.6a). Network failures (figure 1.6b) occur when the network connection between different workers becomes unavailable, disrupting the communication between the different components of the deployed system or application. Last but not least, despite the advancements of hard disk technology, storage failures (figure 1.6c) are still frequent in a public cloud of commodity hardware [132, 97]. However, regardless of its type, a failure renders parts of the processing pipeline inaccessible and leads to processing state inconsistencies due to atomicity violations or non-persisted states that are lost. Therefore, it is crucial for a data processing system to be able to adapt to such failures and quickly recover and continue processing from a consistent state. We refer to the ability to adapt to failures as *fault tolerance*.

Typical cloud computing workloads consist of stateless components that access data persisted to external storage. Therefore, it is relatively easy to restore execution by restarting the failed component to a different worker and replaying the affected computations. In contrast, most stream processing engines opt for collocating the state and the computation execution to ensure low-latency processing. Additionally, unlike typical cloud applications that involve short-lived functions or queries, stream processing applications usually em-

ploy long-running analytical queries that require actively maintaining a consistent state in memory. The collocation of state and computation execution, as well as the nature of the stream processing workloads, render fault tolerance challenging for stream processing engines. Different approaches have been proposed to provide fault tolerance to stream processing workflows.

Log-based solutions [22, 43], inspired by traditional database systems, propose the use of write-ahead logging to capture the messages sent by each operator to downstream operators and, in the case of a failure, recover by replaying these messages. When stateful operators are employed, the logs cannot be trimmed and must be kept until the operators' state no longer depends on the contents of the logs. Although windowed operators usually have short-lived states, that is not the case for all stateful operators. Therefore, logs can endlessly grow in the presence of stateful operators that require full history. For high throughput stream processing, log-based solutions can result in maintaining huge write-ahead logs that are costly and, in the case of a failure, have to be replayed from the start in order to resume processing.

Due to the inefficiencies and the high maintenance cost of logging, most stream processing engines implement *checkpoint-based* solutions in order to achieve fault tolerance [30, 73]. Checkpoint-based fault tolerance leverages snapshotting to ensure consistency. A *checkpoint* is a snapshot of an operator's state, while the *global state* of a stream processing application is the set containing a checkpoint of each operator. When a checkpoint is captured depends on the implemented checkpoint protocol. We can distinguish three main checkpointing protocols for fault tolerance in stream processing: the *coordinated* [30], the *uncoordinated* [172], and the *communication-induced* checkpointing protocols. All checkpointing protocols have the same goal of finding the most recent set of checkpoints that can reconstruct a consistent global state from where the system can recover and continue processing. However, each protocol uses different mechanisms to decide when to take a snapshot and how to find a recent and consistent global state for the system to recover from. Additionally, it requires different mechanisms to provide *exactly-once processing guarantees*, i.e., to ensure that the changes to the state that occur from processing a specific item are only reflected once to the state.

Other approaches [152, 52] use replicas to ensure fault tolerance and high availability. Clonos [152] employs passive stand-by operators that replicate the main pipeline's state. In case of a failure, Clonos can quickly recover by switching to the replicated pipeline. Rhino [52] proposes a state migration mechanism that combines checkpointing and replicas. Rhino leverages the checkpointing mechanism of the underlying stream processing engine and asynchronously replicates the captured checkpoints to multiple workers. When a failure occurs, the system can quickly recover from the replicated checkpoints. However, replication comes with additional costs in storage and computation resources, and thus, it is avoided when high availability is not a concern.

1.1.3 Adapting to input rate changes

Modern cloud providers allow for flexible resource allocation and pricing schemes. Cloud users no longer need to reserve their computing resources statically and pay for them while they are considerably underutilized. Instead, they can dynamically adapt their computing resources based on the workload they are serving, foresee, or can afford and leverage

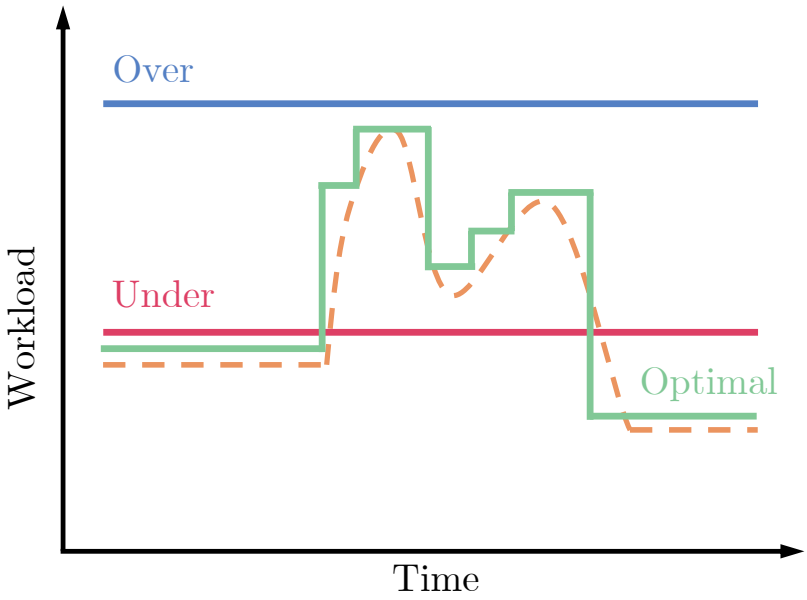


Figure 1.7: Different ways of provisioning resources.

flexible pricing schemes, such as a pay-as-you-go scheme, to only pay for resources that they are actually using. This flexibility in resource allocation and pricing enables the digital transformation for small and medium-sized businesses and allows them to remain competitive and prosper by reducing their infrastructure costs.

At first glance, cloud computing and stream processing seem a good fit. Cloud computing provides the resource allocation flexibility that can accommodate the dynamic nature of stream processing workloads. However, stream processing engines cannot currently leverage effectively the features delivered by the cloud providers. In practice, most stream processing engines are designed for on-premise clusters of fixed resources. Therefore, they lack the necessary mechanisms to adapt to changes in the input workload: acquire more resources in periods of high load, or release idle or underutilized resources when the input rate is reduced.

Currently, the configuration options for a stream processing engine are limited. Assume that a company would like to perform online streaming analytics over the traffic of its webshop in order to dynamically improve the experience of a user visiting the webshop. Figure 1.7 describes the analytics task's input rate observed over time. Usually, under normal operation, the streaming operations team of the company can expect a certain input rate. However, spikes or periods of uncommon increased input rate can be observed under certain circumstances, such as during Black Friday or a sales period. The operations team of the company has virtually two options. It can overprovision resources to the stream processing engine in order to anticipate spikes and high input rate periods. In this case, the

task will retain the desired performance even when the input rate is temporarily increased. However, the resources will be underutilized under normal operation, where a lower input rate is observed. Underutilizing the resources for the longer periods of normal operation renders part of the cloud budget wasted, as it is not actually needed to provide better performance. The other available approach is to provision the stream processing engine with the exact resources it requires to meet the service level agreements (SLAs) during normal operation. Although this strategy will refrain from wasting budget on underutilized resources, it will also make it impossible for the stream processing engine to handle spikes and periods of increased traffic, possibly leading to multiple SLA violations.

Ideally, in order to optimize costs and performance, the provisioned resources should follow the observed workload. The operations team must adapt the resources to the input rate changes by providing the minimal resources needed to ensure the desired performance under the SLAs. However, this is a difficult and laborious task. The operations team has to constantly monitor the observed input rate, decide on the optimal resources for the observed rate, and manually adjust the resources of the stream processing engine. At the same time, the operations team must consist of highly skilled and experienced people who are difficult to find and expensive.

Employing such a team can be infeasible for small and medium-sized businesses. To tackle this problem and facilitate businesses to leverage the flexibility of the cloud, several *autoscaling* techniques have been proposed that specifically target stream processing engines. Similar to autoscaling solutions targeting web services [114], stream processing *autoscalers* can also be categorized as *proactive* or *reactive*, or even more fine-grained as *threshold-based* [81, 87], *reinforcement learning* [112, 54], *queue-based* [67, 111], *control-based* [93, 65] and *time series forecasting* [113, 21]. *Threshold-based* solutions employ user-defined thresholds to decide on scaling actions based on the values of the monitored metrics. Although these solutions are simple and easily deployed, they require the user to have deep knowledge of the workload and the underlying system as they require multiple parameters to express different scenarios. *Reinforcement learning* techniques employ reinforcement learning models to learn the optimal configuration for a specific input rate. Although they require minimal intervention from the user, these techniques require training periods that might be infeasible for a real-world streaming application, as the models can take hours to days in order to converge to an acceptable predictive performance. *Queue-based* autoscalers employ queuing theory to model the streaming application and estimate the system's performance for the observed input rate under a specific configuration. However, not all streaming topologies can be modeled using queuing theory, and therefore, the applicability of the solutions is impacted. *Control-based* autoscalers handle autoscaling as a control theory problem. Given a metric and a target value for this metric, control-based autoscalers attempt to maintain the metric to the target value. All scaling decisions are tailored to this goal. Although control-based autoscalers require one or more user-provided target values for the monitored metrics, they require less user intervention than threshold-based solutions. They are versatile and can serve any streaming topology while they are easily deployed. Finally, *time series forecasting* approaches attempt to forecast the input rate in the immediate future and decide on scaling actions based on this prediction. They are usually complementary and combined with other autoscaling techniques. Regardless of their type, autoscalers can be crucial for fulfilling the SLAs of an application. Further

research on the topic can provide more reliable solutions and allow every company to harness the full potential of cloud-deployed streaming applications.

1.2 Main Research Questions

In this thesis, we study the adaptivity capabilities of modern stream processing engines and the solutions proposed in the literature to provide such adaptivity capabilities. We focus on three important types of adaptivity, namely, adaptivity to statistical changes, infrastructure failures, and input rate changes.

Statistical changes. Driven by our discussion in section 1.1.1, we recognize that a plethora of solutions have targeted adaptive and scalable similarity joins on the MapReduce paradigm. However, these solutions are not directly applicable to a streaming environment, and adapting them to the stream processing paradigm is not trivial. Limited work addresses the problem of streaming similarity joins. These works focus on optimizing the task for single node execution [49], or load balancing without reducing the unnecessary computations [60]. The solution proposed by Yang et al. [174] is the only distributed streaming solution that reduces unnecessary computations while providing adaptivity to statistical changes through load balancing. However, it is tailored for similarity joins on sets of strings and cannot be generalized to the broader problem of general streaming similarity joins. In this thesis, we argue for the necessity of an efficient general solution that can accommodate different types of data and various similarity metrics while providing an efficient load balancing mechanism in order to be able to adapt to statistical input changes. We explore such a solution through the following research question:

> **M-RQ1:** How can we perform streaming similarity joins on multidimensional streams in a distributed fashion, even when distribution changes, achieving low latency?

Infrastructure failures. Regarding adaptivity to infrastructure failures, which we introduce in section 1.1.2, to the best of our knowledge, most modern stream processing engines implement a *checkpoint-based fault tolerance* mechanism. More specifically, most of the modern stream processing engines [31, 73] opt for a flavor of the coordinated checkpointing protocol without any clear empirical and experimental evidence supporting this choice. At the same time, through mailing lists, blogs, and forums, practitioners frequently report inefficiencies of the employed coordinated checkpointing protocol and suggest ad-hoc workarounds for better performance. Although the checkpointing protocol is the central component of a fault tolerance mechanism, to this date, there is not any clear, comprehensive, and principled evaluation of the three main checkpointing protocols. Such an evaluation can shed some light on the reasons behind the preference for the coordinated approach and provide experimental proof of its prevalence, highlight cases where alternatives might perform better, and, first and foremost, provide a comprehensive guide on checkpoint-based fault tolerance that will inspire further research on the topic and will introduce practitioners to the specifics of fault tolerance for stream processing engines. To this end, in this thesis, we consider the following research question:

> **M-RQ2:** How do checkpointing protocols for streaming dataflows perform in different workloads and input data distributions?

Input rate changes. As discussed in section 1.1.3, multiple autoscaling solutions that follow different principles have been proposed in the literature. However, the experimental evaluation of these solutions is insufficient. Original works perform limited experiments, including simple workloads and scenarios that do not sufficiently resemble real-world applications and their workloads. Additionally, most of the original works do not include comparisons with other autoscaling solutions. When solutions are compared, the evaluation is narrow and tailored only to specific scenarios and objectives. To the best of our knowledge, there is no experimental work that establishes specific metrics from the literature and workloads that should guide a principled experimental evaluation of autoscaling solutions for stream processing. Therefore, it remains unclear how well existing solutions perform under heavily dynamic workloads, i.e., the current challenges and the open problems remain unknown. This lack of principled, detailed experimental evaluation of the autoscaling techniques hinders valuable future research on the topic. We summarize our concerns in our third research question:

> **M-RQ3:** How well can existing stream processing autoscalers perform? How can we effectively evaluate them, and what are their inefficiencies?

We drive our study of adaptivity for stream processing engines on these three main research questions. In Chapter 2, we address adaptivity to statistical changes through the important task of similarity joins and attempt to answer **M-RQ1**. In Chapter 3, we revisit fault tolerance as a means of adaptivity to infrastructure failures and answer **M-RQ2**. Finally, in Chapter 4, we perform an experimental evaluation of control-based autoscalers in an attempt to address **M-RQ3**.

1.3 Methodology

In this section, we discuss some important aspects of the research methodology we used in this thesis in order to address our main research questions.

Evaluation frameworks. This thesis focuses largely on providing guidelines and frameworks for evaluating different adaptivity mechanisms for stream processing. Although works on novel solutions to a problem are usually the primary focus of our research community, the importance of evaluation frameworks should not be underestimated. In this thesis, we study the relevant literature in order to establish a set of metrics, queries, and scenarios that can effectively describe and evaluate the underlying problem. We base our frameworks on NexMark, the most widely adopted open-source benchmark by the stream processing community. We use the NexMark queries and data, and we extend NexMark to include the relevant scenarios and metrics. Doing so, we provide reliable evaluation frameworks that our research community is currently missing and which can provide valuable insights into the current state of the studied problems, as well as guidelines on proper and fair comparison among proposed solutions.

Code & data availability. In this thesis, we embrace the concepts of open science and ensure that all the tools, software, and datasets employed for our implementations and experimental evaluations are publicly available and, most of the time, open-source. Additionally, we strive to use tools and software that are considered state-of-the-art and are widely adopted in practice. Therefore, we implement the solutions and the frameworks

proposed in this thesis using the state-of-the-art open-source stream processing engine Apache Flink. However, in Chapter 3, we opt for using an in-house stream processing system that allows for isolated evaluation of the checkpointing protocols without the interference of other mechanisms in order to ensure the reliability of our results. All datasets and all of our software artifacts, including our in-house stream processing system, are publicly available through GitHub. Each chapter contains its own link to a GitHub repository. We invite all interested readers to visit our repositories for more details on our implementations.

1.4 Contributions

The main contributions of this thesis are summarized as follows:

- We design and propose an adaptive distributed streaming solution for similarity joins in the general metric space. Our solution employs a stream partitioning scheme that provides fine-grained partitions, which ensure the completeness of the results while reducing the number of unnecessary computations. We pair the stream partitioning scheme with a load-balancing scheme that exploits existing partitions to alleviate heavy-loaded nodes with minimum migration costs. (Chapter 2)

- We revisit fault tolerance for streaming dataflows through a comprehensive survey of the main checkpointing approaches and the conditions under which they can guarantee exactly-once processing. We discuss in detail the main concepts of each protocol and recount the theoretical advantages and drawbacks of each approach. (Chapter 3)

- We provide an open-source streaming dataflow testbed that allows for accurate and isolated comparison of different checkpointing protocols and can be easily extended to compare other stream processing mechanisms. (Chapter 3)

- We propose a principled evaluation framework for checkpointing protocols by establishing relevant metrics, queries, and workloads. We performed the first experimental evaluation of the three main checkpointing protocols. Our findings partially justify the clear preference of stream processing engines toward the coordinated approach while highlighting the competitiveness of the uncoordinated approach and showcasing scenarios where uncoordinated checkpointing greatly outperforms coordinated checkpointing. (Chapter 3)

- We propose the first principled evaluation framework for stream processing autoscalers. We establish the most relevant to the task metrics that are found in the literature. We extend the evaluations of the state-of-the-art control-based autoscalers with heavily dynamic workloads. We implement the state-of-the-art control-based autoscalers, ensuring per-operator autoscaling, and evaluate them over diverse queries. Our findings show the inefficiencies of the evaluated solutions and motivate further research on the topic. (Chapter 4)

1.5 Thesis Origins

This thesis consists of three main chapters. In Chapter 2, we focus on the complex problem of similarity joins in a highly dynamic streaming environment, and we present a novel solution that can naturally adapt partitioning to the distribution and domain shifts, enabling real-time processing. Chapter 3 revisits fault tolerance in stream processing by focusing on checkpoint-based recovery, summarizing its core concepts, discussing the main checkpointing protocols, introducing a principled way of evaluating them, and, finally, comparing them extensively. Chapter 4 engages the problem of the elastic allocation of computing resources for stream processing engines by performing a principled experimental evaluation of autoscaling solutions tailored to the specifics of stream processing, extending existing evaluations with dynamic workloads, and providing an extensible evaluation framework that enables a thorough evaluation of future work.

Chapter 2 is based on the following paper:

> *George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Odysseas Papapetrou, Arie van Deursen and Asterios Katsifodimos. 2023. Adaptive Distributed Streaming Similarity Joins. In DEBS. Association for Computing Machinery, 25–36.* [150]

Chapter 3 is based on the following paper:

> *George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In ICDE. IEEE, 4030–4043.* [151]

Chapter 4 is based on the following papers:

> *George Siachamis, Job Kanis, Wybe Koper, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen and Asterios Katsifodimos. 2023. Towards Evaluating Stream Processing Autoscalers. In SMDB (ICDEW). IEEE, 95-99.* [149]

> *George Siachamis, George Christodoulou, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen and Asterios Katsifodimos. 2024. Evaluating Stream Processing Autoscalers. In DEBS. Association for Computing Machinery, 25–36.* [148]

2

Adaptive Distributed Streaming Similarity Joins

How can we perform similarity joins of multi-dimensional streams in a distributed fashion, achieving low latency? Can we adaptively repartition those streams in order to retain high performance under concept drifts? Current approaches to similarity joins are either restricted to single-node deployments or focus on set-similarity joins, failing to cover the ubiquitous case of metric-space similarity joins. In this chapter, we propose the first adaptive distributed streaming similarity join approach that gracefully scales with variable velocity and distribution of multi-dimensional data streams. Our approach can adaptively rebalance the load of nodes in the case of concept drifts, allowing for similarity computations in the general metric space. We implement our approach on top of Apache Flink and evaluate its data partitioning and load balancing schemes on a set of synthetic datasets in terms of latency, comparisons ratio, and data duplication ratio.

This chapter is based on the following paper:

George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Odysseas Papapetrou, Arie Van Deursen and Asterios Katsifodimos. 2023. Adaptive Distributed Streaming Similarity Joins. In DEBS. Association for Computing Machinery, 25–36. [150]

2.1 Introduction

Similarity join is the task of identifying all pairs of similar records that reside in two or more datasets according to a similarity function. Similarity joins play an important role in *data integration, data cleaning, recommender systems* and many other domains. In fact, nowadays, with data in motion becoming more ubiquitous, many of the use cases requiring a similarity join have to be performed in a streaming fashion.

Performing similarity joins on data streams is challenging and computationally expensive. The brute force approach – even for the case of static datasets – has to compare all records of the first dataset against all records on the second, leading to quadratic time complexity, $\mathcal{O}(n^2)$ where n is the number of records. As the number of records increases, brute-force solutions become infeasible. At the same time, the unbounded nature of data streams means that the complete set of records is not available in full prior to their processing. Moreover, since data streams are continuous, their statistical properties may change over time. Depending on how frequently those changes occur (also known as concept drift [162, 70]), they can have a significant impact in optimizing the similarity join operation. To recap, streaming similarity joins entail solutions that are *i*) efficient, *ii*) scalable, and *iii*) can adapt to concept drift.

Although equality joins on streams have been studied extensively, scant attention has been paid to streaming similarity joins. Existing works provide solutions specifically for set-similarity joins [174], optimizing a self-join operation in a single machine [49], and scaling out the cross-product comparisons in multiple machines [60]. However, none of these solutions target the general metric space or support efficient load balancing that can adapt the load distribution to the concept drift that frequently occurs (Table 2.1). At the same time, a significant research body targets batch similarity joins targeting the MapReduce [48, 171, 41] framework. Although these works address scalability and efficiency, their core techniques do not adhere to the properties of streaming data (unboundedness and concept drift). Thus, they cannot be applied on streams, and they do not provide load-balancing capabilities.

Streaming similarity joins can be performed in either exact or approximate fashion. Approximate similarity join algorithms, such as [90, 89] are interesting for use cases where applications can sacrifice completeness of results. In this work, we explicitly focus on exact algorithms, which are necessary for scenarios where complete answers are required. All available similarity join approaches share a common strategy: they group similar data to reduce the number of unnecessary comparisons. To this end, they either employ optimized indices [49] in a centralized setting or data partitioning schemes [174, 48, 171] in a distributed setting. We have already argued about the necessity of a distributed approach in order to handle the massive volumes of modern data streams. However, the distributed batch-based solutions require multiple passes over the data [48, 171], and they only provide means of splitting huge partitions into smaller ones rather than balancing the load. Additionally, the only distributed streaming similarity joins approach [174] targets only the specific sub-problem of set-similarity joins instead of providing an applicable solution for the general problem. The simple load-balancing scheme which the authors propose has limited scaling capabilities and can only provide balancing for scenarios where the input has varying lengths. All in all, existing solutions fail to provide efficient load balancing that can keep up with concept drift and harness the available processing power.

Table 2.1: Related work comparison.

	Method Characteristics		*Applicability*		
Work/Feature	**Candidate Pruning**	**Load Balancing**	**Mode**	**Environment**	**Problem**
Morales et al.[49]	X		Streaming	Centralized	Similarity self-joins on a vector stream
Yang et al. [174]	X	X	Streaming	Distributed	Set similarity join on sets of various lengths
ClusterJoin [48]	X	X	Batch	Distributed	Similarity joins on general metric space
ElSeidy et al. [60]		X	Streaming	Distributed	Cross-product joins to multiple machines
Proposed Solution	X	X	Streaming	Distributed	Similarity joins on general metric space

In this chapter, we propose S^3J, an approach for adaptive distributed streaming similarity joins[1] based on a load balancing scheme that leverages the underlying partitioning to redistribute the load across our group of workers. We pair our balancing scheme with a partitioning scheme that adheres to the properties required to facilitate it. The employed partitioning scheme is suitable for the general flavor of the similarity joins problem over a metric space and extends existing work by leveraging two levels of data partitioning to prune candidate pairs and scale out the similarity computations to multiple nodes.

Our contributions can be summarized as follows:

- The proposed solution is the first algorithm that solves the distributed streaming similarity join problem in the general metric space (see Table 2.1). Our algorithm is also the first to properly address the issue of load imbalance, thereby permitting better scalability and responsiveness.

- We propose a stream partitioning scheme for similarity joins in the general metric space, providing tight and fine-grained partitions, ensuring the completeness of results while reducing the number of computations. Our partitioning scheme has all required properties to effectively support our load-balancing scheme (Section 2.7).

- We show how to map the load balancing problem to the classic job rescheduling problem and propose a novel algorithm tailored to a partitioning/work imbalance measure (Section 2.8) .

- We propose a load balancing scheme to alleviate heavily loaded nodes while minimizing migration costs. In contrast to existing load-balancing solutions, we refrain from repartitioning the data, which is prohibitively expensive in streaming scenarios, and instead, we exploit the existing partitions to perform load balancing (Section 2.8).

- We conduct a detailed experimental evaluation of our solution using synthetic datasets in order to evaluate the efficiency of our method under various scenarios (Section 3.7).

2.2 Preliminaries

This section provides a discussion of existing concepts and techniques on the foundations of similarity joins and our partitioning scheme.

[1]Code available in:
`https://github.com/delftdata/s3j-adaptive-similarity-joins`
`https://zenodo.org/doi/10.5281/zenodo.11652793`

The Inner-Outer Partitioning Paradigm. The state-of-the-art MapReduce solutions for the *general metric space* [48, 171] define the inner-outer partitioning paradigm. Specifically, for each worker, one centroid is randomly selected, and a pair P of inner and outer partitions is assigned to it. Inner partitions are disjoint, i.e., they have no common records, while *outer partitions can overlap.* We provide these definitions below.

Definition 2.1 (Inner Partition). The inner partition I_i of centroid c_i contains all records for which centroid c_i is the closest centroid among all available centroids, i.e.,

$$I_i = \{r \mid \forall c_j \in C, dist(r, c_i) \leq dist(r, c_j)\}, \tag{2.1}$$

where $dist()$ is the employed distance metric.

To decide whether a record is included in an outer partition, ClusterJoin [48] proposes the following membership criterion:

Criterion 2.1 (Outer Partition Membership Criterion)**.** *Let r be an incoming record and c_i be the closest centroid to r. Then r belongs to the outer partition of $c_j, \forall c_j \in C, c_j \neq c_i$, if and only if*

$$dist(r, c_j) \leq dist(r, c_i) + 2 \times t, \tag{2.2}$$

where dist() is the employed distance metric and t is the provided distance threshold.

Based on Criterion 2.1, Definition 2.2 describes an outer partition.

Definition 2.2 (Outer Partition). The outer partition O_i of centroid c_i contains all records that do not belong to the inner partition I_i and satisfy Criterion 2.1, i.e.,

$$O_i = \{r \mid \forall r \notin I_i \wedge Criterion1(r, c_i)\} \tag{2.3}$$

Similarity Computations. A solution based on the inner-outer partitioning paradigm proceeds in performing the similarity comparisons based on the formed pairs of inner and outer partitions (Figure 2.2(a)). In short, the records of an inner partition are compared against the records of the same inner partition as well as the records of the corresponding outer partition, whereas records from an outer partition are compared only against records from the corresponding inner partition. All records with a similarity higher than a threshold are returned to the user. It can be easily shown that this algorithm retrieves all results [48, 171].

In this work, we extend the inner-outer partitioning paradigm to encapsulate fine-grained sets of data involved in computations as a group. This formulation enables us to perform load balancing of distributed streaming similarity joins with high adaptation to variable data velocity and distribution.

2.3 Problem Statement

A streaming similarity join operation identifies all pairs of records that belong to one or more data streams, arrive in the same time window, and have a similarity that exceeds a user-defined threshold. Consider a set of streams $\mathcal{S} = \{S_1, S_2, ...\}$, each containing records.

Let $sim(r_i, r_j)$ denote the user-defined similarity function between two records r_i and r_j, which takes values within $[0,1]$. [2]

Definition 2.3 (Matching records within a time window). For a given similarity threshold θ_{sim}, two records are considered a match when their similarity exceeds θ_{sim}, and they both arrive within the same time window. Formally:

$$(r_i, r_j) \text{ is a match} \Leftrightarrow sim(r_i, r_j) \geq \theta_{sim} \; and \; t_i, t_j \in W_k, \tag{2.4}$$

where t_i, t_j are the ingestion timestamps of r_i, r_j and W_k the k_{th} window. [3]

The above definition can trivially be rewritten using distances, where:

$$dist(r_i, r_j) = 1 - sim(r_i, r_j), \text{ and } \theta_{dist} = 1 - \theta_{sim}.$$

Notice that similarity join (even over static data) is often a computationally expensive operation whose complexity increases with the increase of the dimensionality of the data. The streaming context further aggravates this issue due to the high velocity of incoming records in data streams and the need for quick answers. The only way to efficiently sustain the overall computational burden is by scaling out the processing to a distributed stream processing system. This practice, however, brings forward a new set of challenges, most importantly the partitioning of the data and the load balancing across the cluster.

Definition 2.4 (Partitioning for streaming similarity joins). Assume a set of streams $\mathcal{S} = \{S_1, S_2, ...\}$ that contain records consisting of a timestamp, an id, and a (potentially high-dimensional) value, i.e., $r = (timestamp, id, value)$. Consider a set of records $R_k = \{r \mid r \in S_i, S_i \in \mathcal{S}\}$, ingested within a time window W_k, a set of worker nodes N, and a given threshold θ. Partition the records in R_k in $|N|$ partitions, such that i) each pair of matched records based on the similarity threshold θ is contained in the same partition and ii) the computation load across the worker nodes is balanced.

Unfortunately, the partitioning of the records is not guaranteed to be stable in the lifetime of a streaming workload. Even if we could efficiently decide on an optimal partitioning of the data, this remains unknown since the data is not available when creating the partitions. Furthermore, statistical changes in the incoming streaming data, i.e., a possible concept drift, create skews in data partitions. Consequently, these aspects entail the consideration of computational load and accompanying challenges. We define the load of a worker node in a set of workers as follows.

Definition 2.5 (Load). Given a partitioning scheme PS, the load L_n^{ps} of a worker n in a set of workers N is equivalent to the number of similarity comparisons it needs to perform based on the partition of records assigned to it.

[2] To simplify exposition, hereafter, we expect that $sim(r_i, r_j)$ returns values between 0 (no similarity) and 1 (identical records). If this is not the case, we can define a metric-specific function $f(sim(r_i, r_j))$ that bounds the similarity metric within $[0,1]$.

[3] Hereafter, without loss of generality, we refer to tumbling windows only to simplify the presentation of our approach with respect to time window semantics. Our work is applicable to any type of time window.

As new data arrive, the existing partitioning of the data in the cluster's workers may no longer provide a balanced workload across the cluster, thereby leading to the load balancing problem.

Definition 2.6 (Load balancing). Assume a set $R_k = \{r | r \in S_i, S_i \in \mathcal{S}, and\ r.timestamp \in W_k\}$ of streaming records ingested within a time window W_k, a set of the previously defined partitions PS that contain the streaming records R_k, and a set of workers N. Each worker n is assigned a partition P and has an estimated load L_n^{ps} based on partition P. Find a new optimal partitioning scheme $OPS(R, N)$ that minimizes how much the load of each worker differs from the desired average load, i.e.,

$$OPS(R, N) \text{ so that } min(\sum | L_n^{ops} - L_{avg}^{ops} |), \forall n \in N. \tag{2.5}$$

2.4 Related Work

In this section, we discuss the works that are most relevant to the problem of adaptive distributed similarity joins.

Distributed Stream Equi-Joins. Equality joins have been investigated thoroughly in the literature. Najafi et al. [125] propose SplitJoin, a novel stream join architecture that achieves high scalability by dividing the join operation into independent storing and processing, and employing adjustable join output ordering guarantees. Dossinger et al. [56] introduce MultiStream, a novel multi-way stream join operator that leverages tuple routing and exploits a materialization vs. network cost to perform stream join optimization. Najafi et al. [126] propose a circular multi-way join operator that benefits from hardware and a parallel multi-way join operator that reduces computation time.

None of these works deal with similarity joins.

Single-node Similarity Joins on Streams. Contrary to equality joins, research on similarity joins on data streams is limited. Morales et al. [49] propose a solution to streaming similarity self-joins. However, this work does not discuss a parallel distributed solution, and their approach cannot be trivially scaled out. Similarly, [108] introduces an operator to tackle similarity joins on uncertain data streams, but their solution cannot be trivially scaled out. In addition, none of these works considers load (re-)balancing, which is necessary to retain high performance in the case of concept drift.

To the best of our knowledge, no existing work in stream processing can provide a general solution over the metric space, which can scale out to multiple machines and provide load balancing capabilities to tackle concept drift.

Distributed Similarity Joins on Streams. Closest to the spirit of our work is the work by Yang et al. [174]. The authors propose a distributed streaming similarity join framework that employs a simple length-based filter to distribute the data across the cluster. Such a filter cannot be used for metric-space similarity computations, which we consider in this work. Instead, [174] focuses on the *set similarity* problem.

In contrast to [174], in this work, we aim for metric-space similarity computations, which are required in state-of-the-art approaches in similarity joins, like keyless joins [159].

k-Nearest Neighbours on Streams . In contrast to the exact similarity join problem on streams, streaming kNN queries attempt to identify and retrieve a specific number, k, of results. Koudas et al. [99] introduce an error-bounded variation of the problem and propose DISC that can answer error-bounded kNN queries over sliding windows. Sundaram et al. [158] propose PLSH, a fast distributed variation of LSH, that supports approximate nearest neighbours queries over high throughput streams. ADS-kNN [145] overlaps communication and computation stages and leverages an adaptive partitioning that keeps the load balanced in order to improve performance.

Streaming kNN employs similarity computations and requires efficient real-time results. However, it does not retrieve all existing pairs, and the results are not threshold bounded. Therefore, its solutions cannot be applied in our context.

Distributed, Batch Similarity Joins in MapReduce. There are two main approaches that MapReduce methods usually follow: Filter & Verification and General Metric Space [64]. The Filter & Verification methods [167, 53, 123] rely on prefixes and signatures, which they leverage to scale out the similarity computations and filter unnecessary comparisons. On the other hand, General Metric space methods [171, 48, 41, 173] divide the metric space into partitions to which similar objects are grouped.

None of the MapReduce solutions is applicable to streaming similarity joins, as they require multiple passes over the given dataset to gather statistics, as well as additional pre-processing steps.

Dynamic reconfiguration for stream processing. Load balancing is a native concern in distributed stream processing environments. Zhou et al. [178] formalize the problem of operator placement and propose heuristics that provide load balancing with minimum data movement across nodes that execute multiple queries. Pietzuch et al. [131] propose an intermediate layer between the physical network and the stream processing engine that provides load optimizations through operator placement. Madsen et al. [119] propose ALBIC, a stream processing optimizer, that unifies reconfiguration problems, such as load balancing and operation placement, and addresses it as a mixed integer linear program optimization. Finally, Cardellini et al. [32] propose a two-layered hierarchical architecture, EDF, that enhances a stream processing engine with autoscaling capabilities.

These works address reconfiguration problems over a cluster of nodes where multiple streaming queries run. However, they only focus on cluster level reconfigurations and do not deal with imbalanced parallel operators of a specific query due to skewed workloads.

Load-balancing for joining streams. A significant mass of work focuses specifically on join operations. Both [60] and [78] propose new dataflow multi-way join operators. Gu et al. [78] employs two routing algorithms that achieve load balancing without affecting the completeness of the results through data replication. On the other hand, ElSeidy et

al. [60] focus on providing minimal state relocation costs while keeping at balance the trade-off between migration costs and the costs of not having optimal data distribution. Using a different architecture, [170] describes a ring model of multi-way window-based join operators that is based on time slicing and record propagation. Qiu et al. [138] introduces a streaming variation of the HyperCube algorithm [8] for static multi-way joins. BiStream [109] leverages a new model based on managing the computational cluster as a bipartite graph to scale out or down depending on the current workload.

In summary, these works do not optimize unnecessary comparisons and load balancing for the special case of similarity joins.

2.5 Approach Overview

In this chapter, we propose S^3J, an adaptive method that enables efficient similarity joins over streams in a distributed share-nothing environment through a novel partition-aware load balancing paired with a stream partitioning scheme that can tackle the general metric space streaming similarity join problem. Figure 2.1 presents the workflow of our proposed approach, assuming two input streams.
The *similarity join* pipeline employs a workflow of four key operators, executed in a loop (also depicted in Figure 2.1):

(a) *space partitioning* (Section 2.6), where the ingested data (the yellow squares and the green circles, corresponding, in this example, to two streams) is partitioned to two partitions (the highlighted blue and highlighted orange region), each assigned to one worker;

(b) *workset formulation* (Section 2.7), which divides the previously created local partitioning of the data at each worker into smaller sub-partitions (the worksets), to facilitate a more efficient load balancing;

(c) *similarity computation* (Section 2.7.7), where the workers leverage the formed sub-partitions to independently compute the results, without further coordination, and, finally;

(d) *load balancing* (Section 2.8), which relies on statistics collected from the previous steps to re-partition the data by reassigning some worksets to different workers, in order to reduce load imbalance at the workers.

The key statistics driving the load balancing policy include the join output and the side outputs from the workset formulation operators. Particularly, when load imbalance exceeds a threshold configured by the application, the load balancing scheme computes a new distribution of the existing worksets based on the existing distribution and a migration minimization policy and employs the new distribution to the available workers without the need to create new partitions or worksets. For example, in Figure 2.1, the distribution of the load is imbalanced between workers 1 and 2. Thus, the load balancing strategy exchanges workset W2 from worker 1 with workset W1 from worker 2 to balance their load. The new distribution of the worksets is communicated to the workset formulation operators to ensure the correct routing of future records.

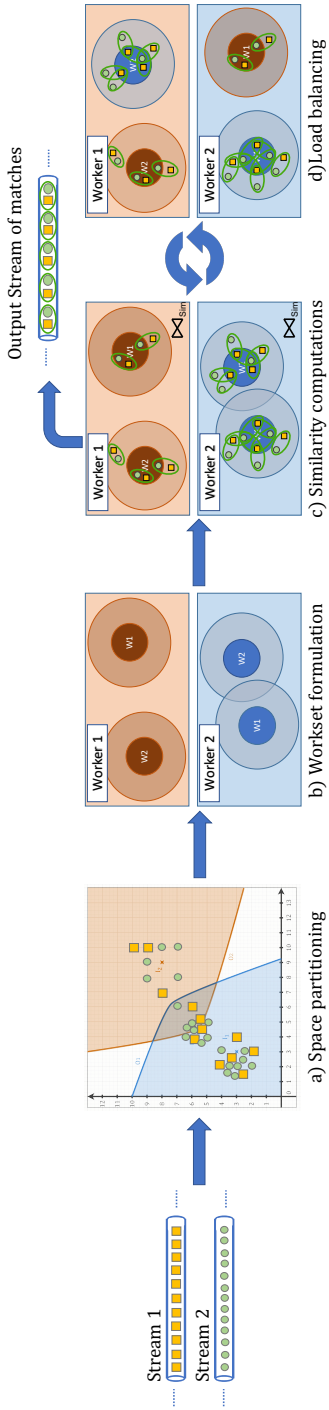


Figure 2.1: Overview of proposed solution's workflow

2.6 Space Partitioning

Our partitioning scheme aims to distribute the data among the available workers so that: (a) the computational load is evenly distributed across the workers, and (b) data duplication is reduced. In this work, we focus on providing adaptivity to streaming similarity joins through an efficient load-balancing scheme and a fine-grained partitioning scheme. Therefore, we adapt and extend the previously-proposed inner-outer partitioning paradigm, described in Section 2.2. In particular, we introduce two layers of partitioning: (a) partitioning the data into coarse partitions, and (b) breaking each partition into smaller worksets, which can be independently handled by each worker. In this section, we describe shortly our first partitioning layer, space partitioning.

2.6.1 Space partitions

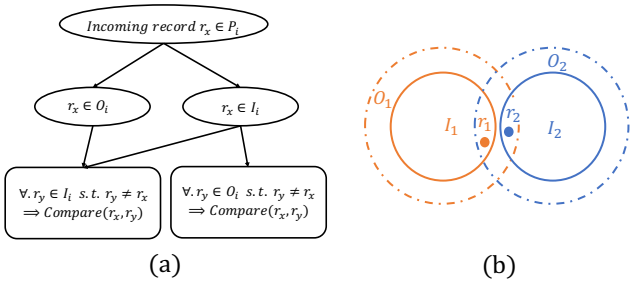
To create our fine-grained worksets, we need to distribute our incoming records to our workers. However, we cannot divide the load arbitrarily. We need to ensure that possibly similar records are co-located to the same worker so that we will not have two workers creating similar worksets. In such a case, we would risk losing matches or duplicating every record belonging to these worksets. Therefore, we opt to divide the incoming records into inner and outer partitions, adapting the inner-outer paradigm, in order to create *space partitions*. Formally, the space partition SP_i of a centroid c_i is the pair of inner partition I_i and outer partition O_i of c_i.

2.6.2 Selecting Centroids

Selecting the right centroids for the space partitions is not a trivial task. A common approach employs a random sampling of the data under processing to select partition centroids [48, 171]. However, none of these techniques is applicable in the case of streams, as they all require an extra pass over the data. In addition, historical streaming data could serve as a source of candidate centroids, but those would be obsolete in the case of concept drift. In this work, we opt for a simple approach: we randomly generate our partition centroids based on the expected space coverage. More specifically, after randomly selecting a set of partition centroids, we initialize our space partitioning instances by providing each one a copy of the available centroids. Each centroid is assigned to a downstream worker of our set of workers. For each incoming record, our partitioner instances calculate the distances to the centroids. Based on the provided Definitions 2.1 & 2.2, an incoming record is assigned to the inner partition of the closest centroid and to the outer partitions of the centroids that satisfy the Criterion 2.1.

2.6.3 Avoiding duplicate comparisons

According to the inner-outer partitioning paradigm, records that belong to neighbouring inner partitions can potentially be members of the corresponding outer partitions. Depending on how similarity comparisons are resolved, such a case would lead to the same candidate pair of records being evaluated twice, i.e., potentially by two different nodes. Figure 2.2(b) shows such an example. Record r_1 belongs to the inner partition I_1 and to the outer partition O_2, while record r_2 belongs to the inner partition I_2 and to the outer partition O_1. Therefore, the pair (r_1, r_2) is evaluated for both pair of partitions P_1 and



(a) (b)

Figure 2.2: (a) Paradigm's similarity computations workflow. (b) Example with a duplicate evaluation of a candidate pair.

P_2 (P_1 : $compare(r_1 \in I_1, r_2 \in O_1), P_2$: $compare(r_1 \in O_2, r_2 \in I_2)$). To avoid these redundant comparisons and maintain duplicate-free results, we adopt the same routing criterion employed by ClusterJoin[48] and MR-MAPSS[171] to decide whether a record should be included in a neighbouring outer partition of a space partition or not.

2.7 Workset Formulation

After dividing the incoming records based on their position in the input space, the workset formulation operation takes place in each of the responsible workers. In this step, we attempt to create self-contained, minimal worksets on top of which we will perform all our similarity comparisons. We define a workset as follows:

Definition 2.7. The ith workset $W_{j,i}$ of a space partition P_j has a centroid $c_{j,i}$ assigned to it, and it consists of an inner set $IS_{j,i}$, an outer set $OS_{j,i}$, and a set of outliers $Outliers_{j,i}$.

Similarly to the inner and outer partitions of the previous stage, inner sets are disjoint while outer sets can overlap with outer sets of other worksets, and they can contain records from inner sets or outliers' sets of multiple other worksets. The outliers' sets are also disjoint. In the following, we discuss the workflow of the workset formulation operator. We present how an incoming record is handled, and we provide all necessary definitions. We explain how we select new centroids and the concept of outliers.

2.7.1 Step 1: Deciding Inner vs. Outer Partition

For each incoming record received from workset formulation operator, we first compute the distances from all existing workset centroids in order to use them in the following steps. Then we need to specify if the record belongs to the inner or the outer partition of the space partition worker is responsible for. Records that belong to the outer partition can only participate in the outer sets of our worksets, while inner records must also be assigned to an inner set of a workset. If the received record is an outer record, we can move directly to step 5 (Section 2.7.5).

2.7.2 Step 2: Assign to an Inner Set

For each incoming record that belongs to the inner partition, we first need to identify the workset whose inner set will contain it. Each record is assigned to at most one inner set. We decide whether a record should be assigned to the inner set of a workset based on the following definition of inner sets.

Definition 2.8 (Inner Set). The inner set $IS_{j,i}$ of the workset centroid $c_{j,i}$ contains all records which are in a distance less than half the provided threshold from $c_{j,i}$, i.e.,

$$IS_{j,i} = \{x | dist(x, c_{j,i}) \leq t/2\}, \tag{2.6}$$

where $dist()$ is the employed distance metric, and t is the provided threshold.

If we manage to assign the incoming record to an existing workset, we can move to step 5. Otherwise, if the incoming record cannot be assigned to the inner set of any of the existing worksets, we check in step 3 if we can create a new workset to assign it to.

2.7.3 Step 3: Creating New Worksets

In order to create a new workset, we first need to select an appropriate centroid for it.

Selecting workset centroids. To select the workset centroids, our strategy differs significantly from the random selection of a fixed number of centroids employed in the space partitioning stage. Without knowing the exact distribution of incoming records in the future, it is very difficult to cover all input space with a fixed number of centroids. Therefore, we opt to select centroids on the fly as we process the data. In this way, we can naturally adapt to occurring concept drifts.

In more details, only records that belong to the inner partition of each space partition can be selected as workset centroids. If an incoming record s cannot be assigned to the inner set of any of the existing worksets, we check if we can create a new workset with s as its centroid. Since the inner sets of all worksets must be disjoint, s must be at a greater distance than the provided threshold from any other centroid. Otherwise, there would be at least two worksets whose inner sets would have some overlap. Therefore, to select an incoming record s as a new centroid, it must satisfy the following criterion:

Criterion 2.2 (Workset Centroid Selection)**.** *An incoming record s is selected as a workset centroid if and only if it is in distance greater than the provided threshold from all existing centroids, i.e., s is a centroid $\iff dist(s, c_i) > t, \forall c_i \in C$, where $dist()$ is the employed distance metric, t is the provided threshold, and C is the set of existing workset centroids.*

2.7.4 Step 4: Labeling Outliers

If an incoming record s neither belongs to an existing group nor can be selected as a centroid itself, it is labeled as an outlier.

Definition 2.9 (Outliers). We define as outliers *Outliers* the set of incoming records that are within greater than half the provided threshold but less than a threshold distance from all existing centroids, i.e.,

$$Outliers = \{o | t/2 < dist(o, c_i) \leq t, \forall c_i \in C\}, \tag{2.7}$$

where $dist()$ is the employed distance metric, t is the provided threshold, and C is the set of existing group centroids.

We assign each outlier to an existing workset based on a proximity criterion, i.e., each outlier is assigned to the outlier set of the workset whose centroid is the closest to it.

2.7.5 Step 5: Assign to Outer Sets

All incoming records are assigned to none, one or more outer sets of existing worksets based on the provided similarity threshold. More specifically, we decide if an existing record should be included to an outer set based on the following definition.

Definition 2.10 (Outer Set). The outer set $OS_{i,j}$ of centroid $c_{i,j}$ contains all records which are within a distance greater than half the provided threshold and less than twice the provided threshold, i.e.,

$$OS_{j,i} = \{x | t/2 < dist(x, c_{j,i}) \leq 2 \times t\} \tag{2.8}$$

where $dist()$ is the employed distance metric, and t is the provided threshold.

To ensure the completeness of our final output, all outer partition records are stored to be compared to new occurring centroids when a new workset is created.

Routing criterion. Similarly to the space partitioning stage, in workset formulation we also employ a routing criterion to decide on routing records to the outer sets of worksets. The reasoning behind this decision is again to avoid considering the same pairs of records in two different worksets. However, due to the dynamic way of creating our worksets we cannot employ the same criterion as in the space partitioning stage. That is because the aforementioned criterion considers all worksets known a priori. Therefore we opt for the simplest solution of routing records to the outer sets of a workset only if the id of the workset is smaller than the id of the workset whose inner set contains the record. Of course, as also discussed in [171], this routing scheme results in a skewed distribution of the computation load to the worksets with the smaller ids. We try to compensate for our decision through our load balancing scheme discussed later on.

Although the worksets are created within a space partition that is assigned to a specific worker, they can be distributed to any of the available workers for the downstream operation of the similarity computations. As a policy, we assign every newly created workset to the same worker that it was created, in order to avoid shuffling overhead. However, the ability to later re-assign a workset to any available worker is crucial for our load balancing scheme discussed later.

2.7.6 Set Boundaries in Metric Space

Based on the metric space properties, the bounds for the inner and outer sets of a workset are carefully chosen to ensure the correctness of the final output and, at the same time, avoid as much as possible unnecessary similarity comparisons. For our inner sets, a bound of half the provided threshold is the maximum bound which can ensure that all records participating in an inner set are at a distance of at most the desired threshold. Thus it allows us to avoid performing the actual comparisons to determine those matches. On the other hand, the selected bounds for outer sets consider the existence of outliers and

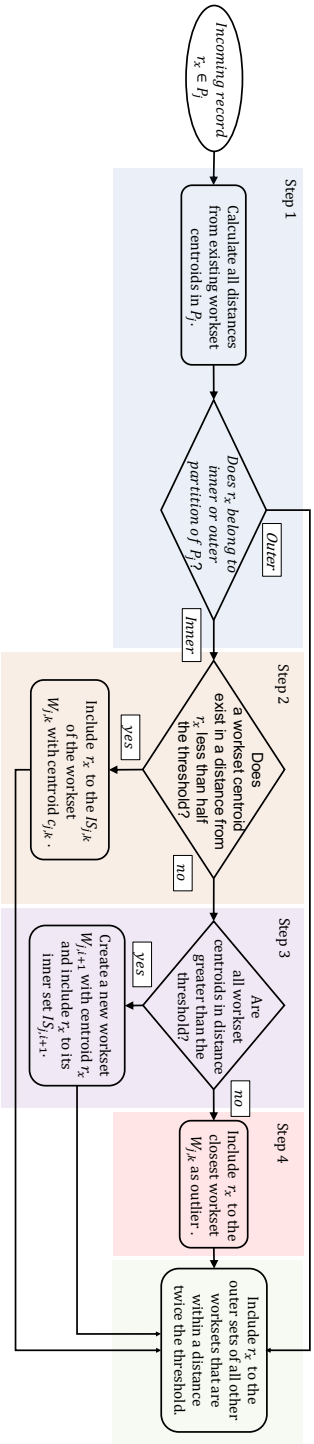


Figure 2.3: Workset Formulation Workflow

ensure the desired completeness of the final output while attempting to keep the number of computations as low as possible. Both bounds can be trivially proven using the metric space's triangle inequality property.

2.7.7 Similarity Computations

By creating worksets inside each space partition, we confine our comparisons and minimize the number of similarity computations performed. We can decide which similarity computations to perform by considering the type of the incoming records, e.g., an inner, outer, or outlier record.

When the incoming records belong to the inner set, we can immediately emit as matches all possible pairs of our incoming records and the other records of this inner set in our state. Yet, we still need to perform the comparisons with the records in the outer set of the workset as well as with the outliers assigned to workset. In the case of an incoming record that belongs to the outer set of a workset, this record needs to be compared against all records in the inner set of the workset and the outliers' set assigned to it. The case of an incoming outlier record is the most expensive since it needs to be compared against all other records of the workset.

2.8 Adaptive Workset Balancing

In the streaming context, it is important that the algorithm adapts to the incoming streams such that it continuously maintains good load-balancing properties. In this section, we discuss our approach to adaptively load-balance the worksets at runtime. Four main factors make the load balancing for similarity joins challenging: *i*) the quadratic complexity of the similarity join problem, *ii*) low latency requirements, *iii*) the zero knowledge of data distributions before execution, and *iv*) the volatile nature of streams that can lead to concept drift.

Similarity Joins are CPU-bound. Our early experiments showed that the similarity computation is the heaviest task of our pipeline, i.e., similarity joins are CPU-bound. In the rest of this section, we propose an approach that takes into account the existing partitioning scheme and reduces the load imbalance by reassigning worksets to similarity computation operators. More specifically, the goal of the balancing of worksets is to load balance the similarity computations across a set of workers.

2.8.1 Migrating Worksets W/O Repartitioning

The workset formulation algorithm (Section 2.7) aims at forming self-contained worksets in each partition, i.e., the worksets are the unit of computation, and any given workset is sufficient to produce the similar pairs of the records assigned to those worksets. We opted for creating self-contained worksets in order to be able to move them across workers without very complex state migration procedures: intuitively, a given workset that incurs very high computation cost can be moved to a worker that is less loaded.

As a result, the worksets can be easily redistributed to the available workers to reduce load imbalance without influencing the completeness and correctness of the join results. At the same time, balancing through reassigning only specific worksets ensures that we have to migrate only specific parts of an *operator's state*. In short, by minimizing load imbalance, we minimize state migration and network costs. This allows us to achieve low response

latency while adapting to streaming load spikes and stream concept drifts. In what follows, we define the problem of load balancing by redistributing worksets on the fly across the available workers.

Definition 2.11 (Load balancing based on worksets). Assume the set $R_{t_1} = \{r | r \in A \ or \ r \in B, \ and \ r = (timestamp, record)\}$ of streaming records received until the timestamp t_1, a set of worksets WS_{t_1} that contain the streaming records R_{t_1} and a set of workers N. Each worker is assigned a subset of WS_{t_1}, e.g. the worker n is assigned the subset $WS^n_{t_1}$. Let $L^n_{t_1}(WS^n_{t_1})$ be the workload of worker n at timestamp t_1 based on its currently assigned subset of worksets $WS^n_{t_1}$. Let $DI_{t_1} = \sum | L^n_{t_1}(WS^n_{t_1}) - L^{avg}_{t_1} |$ be the degree of imbalance regarding the distribution of workload on our set of workers N. Find a new optimal distribution of worksets to workers that minimizes the degree of imbalance DI and the migration cost to reach this optimal distribution from the current distribution.

2.8.2 Workset-Balancing vs. Job-Scheduling

Recall that our worksets compose our most fine-grained data partitions and, at the same time, self-contained computational units. Therefore, it is possible to think of a workset as a computation job over a certain period of time. This observation allows us to link our load-balancing problem to the classic job-scheduling problem across multiple processors. However, the classic definition of job scheduling cannot be directly applied to a streaming setting.

There is a multitude of work on job scheduling for computational grids and multi-processor settings [71, 74, 175, 146, 39]. Specifically, the *job rescheduling* flavor of the problem [42, 11] could be adapted to our workset load balancing problem. This can be achieved as follows: each workset $W_{j,i}$ can be seen as a job $J_{j,i}$ with specific load $L_{W_{j,i}}$ and migration cost based on its size in bytes $M_{W_{j,i}}$, i.e., $W_{j,i} \equiv J_i(L_{W_{j,i}}, M_{W_{j,i}})$. Every similarity computation operator can be seen as a processor, with the primary objective becoming the *minimization of the degree of imbalance*. Note that the job rescheduling problem, and thus our load balancing problem as well, is NP-hard [11, 42]. Due to its NP-hardness, all existing algorithms for the job-rescheduling problem are approximations. Similarly, in the following, we devise a greedy workset balancing algorithm.

2.8.3 The Workset Balancing Algorithm

Our workset balancing algorithm (listed in Algorithm 1) optimizes for the desired load imbalance measure. The algorithm takes as input a set of *overloaded* workers, a set of *underloaded* workers, and the *average load*, i.e., the target load across all workers. The algorithm starts by going over all overloaded workers. For each of them, if it contains one or more worksets with a load higher than the average worker load, we flag the workset with the highest load as *irremovable* (line 5). Since these worksets have a greater load than the average load of workers, moving them will not alleviate the load imbalance.

For all worksets in overloaded workers that are not flagged as irremovable, we calculate the benefit of removing the workset from the worker it currently resides in. If the benefit is positive, we add the workset to a priority queue, sorted descending on benefit. After processing all worksets, we pick the workset with the maximum benefit from the priority queue that is not yet included in the ignore list (a list initialized as empty and populated during the algorithm) and flag it as the best workset (Lines 8-17). The next step is to find an underloaded worker that can accept this workset. Therefore, for each underloaded worker,

Algorithm 1 Workset Balancing Algorithm

Require: set of overloaded workers O, set of underloaded workers U, average load L_{avg}
Ensure: load balanced distribution D_{new} of worksets to workers

```
1:  ignore_list ← []
2:  best ← null
3:  over_benefits ← priorityQueue(priority : benefit)
4:  under_benefits ← priorityQueue(priority : benefit)
5:  irremovables ← find_irremovables(O, L_avg)
6:  repeat
7:     for o ∈ O do
8:        for workset w ∈ W_o do
9:           benefit ← calculate_removal_benefit(w, W_o)
10:          if benefit > 0 and w ∉ irremovables then
11:             over_benefits.put({benefit, w})
12:          end if
13:       end for
14:    end for
15:    repeat
16:       best ← over_benefits.pop()
17:    until best ∉ ignore_list or best is null
18:    if best is not null then
19:       for u ∈ U do
20:          benefit ← calculate_addition_benefit(w, W_u)
21:          under_benefits.put({benefit, u})
22:       end for
23:       repeat
24:          optimal ← under_benefits.pop()
25:       until compute_load(optimal, u) < L_avg
                 or optimal is null
26:       if optimal is not null then
27:          assign_workset(best, optimal)
28:       else
29:          ignore_list.append(best)
30:       end if
31:    end if
32: until best is null
```

we calculate the benefit of adding the chosen workset to that worker and assign it to the worker that brings the maximum benefit. If there is no worker that has a positive benefit, we append the chosen workset in an ignore list (Lines 18-30). This procedure is repeated until there is no candidate workset left to be removed from the overloaded workers (Line 32).

Migration Costs. There have been efforts to formulate a concrete migration cost model [131, 178]. In the context of streaming similarity joins, several factors affect the migration cost: a)

stopping and restarting the streaming job, b) calculating the new partitioning, c) updating the existing state, and d) moving the repartitioned data to the workers based on the new partitioning. In this work, we only consider c) and d) and leave a) and b) for future work. Since our workers share the same resources, only the size of the worksets we move affects the migration cost. The benefit function calculates the benefit as the difference in the degree of imbalance (DI), defined in Definition 2.11, between the current workset distribution and the one occurring after a workset move. In order to include migration cost in our algorithm's model, we subtract from the calculated benefit the workset size multiplied by a user-provided factor to negatively affect the calculated benefit that will be taken into account.

Gathering Statistics. Notice that our workset balancing algorithm requires as input a collection of statistics. By monitoring the main pipeline, we measure the load and the latency for each worker, and the size and the load of the worksets assigned to each worker. All statistics are collected over an application-specified monitoring window. The algorithm requires an extra step of categorizing the workers as underloaded or overloaded based on the desired average load of a worker, which, however, takes negligible time, even for networks involving tens of thousands of workers.

2.9 Experiments

2.9.1 Performance Metrics

Previous works [174, 49, 108] in streaming similarity joins try to adapt existing metrics to stream processing use cases. However, to accurately evaluate the performance of streaming solutions, we need to employ performance metrics that adhere to the requirements of streaming workloads. For example, using the total runtime duration of a streaming similarity join operation as in [174, 49], is not suitable for a practical application involving unbounded data streams. While [108] provides a more interesting per-timestamp runtime measurement of the actual similarity join computation, this does not include an end-to-end performance measurement of the pipeline.

We argue that tuple latency is a more suitable metric since it is strongly connected to the natural requirement of stream processing for real-time results. We employ as tuple latency the processing-time latency in windowed join operators from [94]. The *processing-time latency* of a joined pair of tuples in a streaming similarity join is defined as the interval between the *maximum* ingestion time of the involved tuples and its emission time from the output sink.

A common metric in the existing literature is the *duplication ratio* of data partitioning. The *duplication ratio* is defined as the *average number of times* a tuple is duplicated across the available partitions. The duplication ratio shows the impact that the partitioning scheme has on the input size. Therefore it also provides valuable insights into the additional memory and storage resources needed to apply our partitioning. The higher the duplication ratio is, the more redundant information is transmitted and stored.

Since the goal of this line of work is to reduce the number of computations performed, we also employ a *comparisons ratio*. We define the *comparisons ratio* as the *total number of performed similarity computations* over the *number of joined pairs*. This metric allows for efficiency comparisons between the solutions.

2.9.2 Datasets

In order to evaluate our proposed solution thoroughly and to understand its limitations, we perform an experimental evaluation over synthetic datasets of various configurations.

Synthetic stream generators. We employ synthetic data to investigate in depth the performance of both our partitioning and load balancing scheme under fully-controlled conditions. In particular, we implemented a set of configurable stream generators to provide streams of different velocities with records of varying dimensionality that follow different probability distributions.

2.9.3 Experimental Setup

The experiments are conducted on a 3-node Kubernetes cluster with AMD EPYC 7H12 2.60GHz CPUs. We configure an Apache Flink cluster with a single job manager and a dynamic set of task managers based on the parallelism of the running job. The job manager and the task managers are deployed with 2 CPUs and 16GB of memory each. Apache Kafka is used as the source and the sink of the Flink job. All generators feed the similarity join job through Kafka. Minio is used as a state backend for Flink and as file storage for complementary data. We employ vectors as input values and angular distance as our metric for all experiments. We evaluate our load balancing and partitioning scheme based on the aforementioned metrics. Since Flink does not provide any online mechanism for state migration, to perform our load balancing, we stop the job with a savepoint, alter the savepoint based on the new workset distribution using the state processor API, and restart the job with the new savepoint.

2.9.4 Baseline: ClusterJoin

There is no native stream processing solution that performs similarity joins. Therefore, we opt to adapt ClusterJoin to a streaming environment, and we include it in our experiments as a baseline. ClusterJoin follows the inner-outer-paradigms and resembles our space partitioning layer. However, it partitions the data into multiple virtual partitions, which it later assigns to workers, in contrast to our one space partition per worker design. For our experiments, we configure ClusterJoin to use 500 virtual partitions as suggested in the original work [48]. For these virtual partitions, we select centroids like we select centroids for our space partitioning layer.

2.9.5 Partitioning Performance

In what follows, we present S^3J's performance over synthetic data streams with various properties. We first investigate how our stream partitioning performs against join queries of different selectivities. Then, we experiment with different levels of parallelism. For both experiments, we use synthetic data streams whose records follow a uniform distribution. In this set of experiments, we do not impose any load balancing.

Varying selectivities. In this first series of experiments, we focus on the correlation between the performance of our stream partitioning and the imposed query's selectivity. Although most of the existing literature targets various similarity thresholds, we opt for join selectivity since it better describes the properties of the join query.

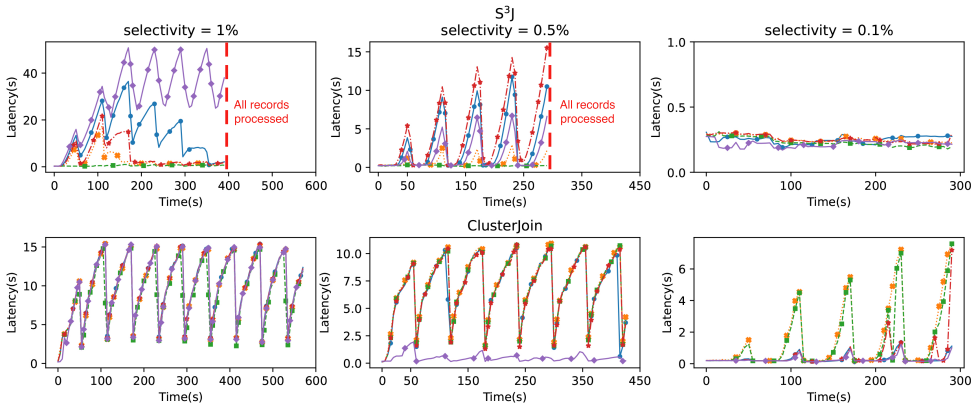


Figure 2.4: 99% latency percentile per worker for varying selectivities. Each line represents a single worker (in this case, 5 workers total). Incoming ratio 4000 records per second, Parallelism: 5, Uniform distribution.

We vary the selectivity through the similarity threshold while keeping the rest of the parameters the same. For the experiment depicted in Figure 2.4, we consider two streams of records of 2D values that follow the uniform distribution. Both streams have a rate of 2000 incoming records per second, which results in 4000 records per second total input rate, and a duration of five minutes. A tumbling window of 1 minute of processing time is employed, and the similarity job has a parallelism of 5. In this low parallelism setting, the partitioning scheme struggles to partition the data evenly across the nodes for high-selectivity queries, while for low-selectivity queries, it provides real-time latency results.

In comparison to our baseline, ClusterJoin, we observe a significant performance improvement both in high and low selectivities. In the case of the 1% selectivity experiment, the input rate is not sustainable for either approach. However, S^3J outperforms ClusterJoin significantly. First of all, we manage to retain a higher processing throughput of 6000 records per second on average, while ClusterJoin is throttled to only 4000 records per second on average. As a result, we finish processing all records 200 seconds earlier than ClusterJoin. S^3J reaches max latency after more than 40 seconds while ClusterJoin maxes out at 15 seconds. Notice, however, that this is a side effect of using the ingestion time in order to measure latency and Flink's backpressure mechanism. Although the load is unsustainable, backpressure does not reach the source operators for S^3J, and the ingestion rate matches the input rate. As a result, records remain in S^3J's pipeline longer. On the other hand, the backpressure is much higher in ClusterJoin and reaches the source operators, resulting in a drop in the ingestion rate. The same also holds for the 0.5% selectivity experiment. In the experiment of low selectivity of 0.1%, in contrast to ClusterJoin, S^3J manages to retain a sub-second latency and provide real-time results.

For these experiments, we also measure the duplication ratio. Table 2.2 summarises our findings. S^3J manages to keep the duplication ratio around 2x for all experiments and selectivities. On the other hand, ClusterJoin's duplication ratio grows fast as the selectivity is increased. S^3J's fine-grained worksets manage to partition the data more efficiently than the random virtual partitions of the adapted ClusterJoin. The lower duplication ratio

Table 2.2: Effects of selectivity on duplication ratio and comparisons reduction.

	Duplication Ratio		*Comparisons ratio*	
Selectivity	**S^3J**	**ClusterJoin**	**S^3J**	**ClusterJoin**
10%	1,96	Timeout	1,58	Timeout
5%	1,96	Timeout	1,65	Timeout
1%	1,94	6,17	1,77	9,70
0.5%	1,97	3,53	1,95	11,16
0.1%	1,92	1,49	5,82	24,51

results in less network traffic and fewer comparisons to be performed.

The comparisons ratio showcases the efficiency of S^3J. As Table 2.2 shows, S^3J for high to medium selectivities manages to keep the numbers of performed similarity computations below 2x the number of joined pairs. Compared to ClusterJoin, for all selectivities S^3J has a better comparisons ratio by 4x to 5x.

Varying parallelism. The parallelism of the job, i.e., the number of available workers, is another important parameter. We consider again a pair of 2D streams whose values follow the uniform distribution. However, we choose a low selectivity query and a higher incoming ratio of 4000 records per stream per second (8000 records per second in total). The streams have a duration of 5 minutes, and the processing happens in windows of 5 minutes. Figure 2.5 presents the effect of parallelism on the performance of our partitioning solution. The results show that our partitioning can leverage higher parallelism effectively and provide real-time latency results.

Compared to ClusterJoin, S^3J manages to scale much more efficiently. Even with the lowest parallelism, it manages to keep relatively low latencies and handle the entire load without a trailing lag. On the other hand, ClusterJoin needs almost double the time to process the entire load and has significantly worse latency performance. As we increase parallelism, in contrast to ClusterJoin, S^3J manages to harness the additional resources to achieve low latency near real-time results.

The measurement of the duplication ratio suggests that higher parallelism leads to higher record duplication for S^3J. Based on Table 2.3, the duplication ratio of S^3J grows slowly as we add more workers. This growth is mainly attributed to the increase in space partitions as more workers are added. Actually, our second layer of partitioning, i.e., the workset formulation, manages to even reduce some of the duplicate records produced from the space partitioning layer. On the other hand, ClusterJoin's duplication ratio is unaffected by the increase in parallelism. This is due to the fact that ClusterJoin partitions the incoming records based on its virtual partitions and not the number of available workers.

The comparisons ratio showcases that S^3J also benefits from parallelism in terms of comparisons performed. As Table 2.3 shows, S^3J effectively reduces the number of performed comparisons as the parallelism increases. This behaviour is related to the number of the workset centroids assigned to each space partition. As parallelism increases, more space partitions are employed, resulting in fewer worksets per space partition and, thus, fewer unnecessary comparisons per incoming record.

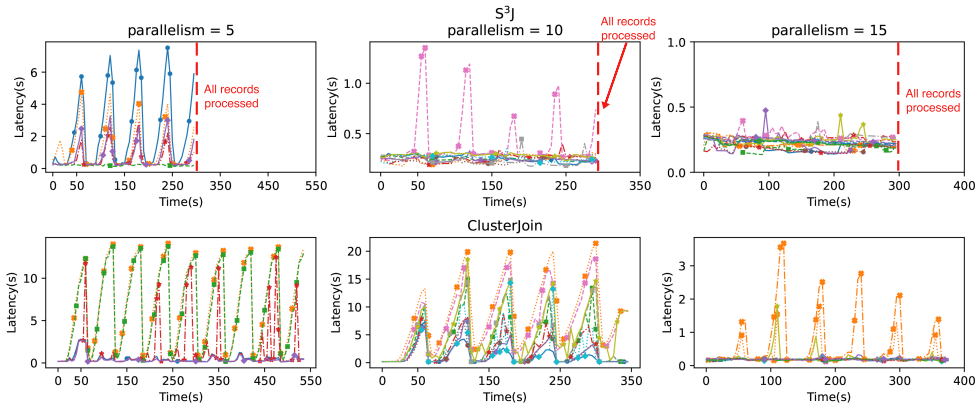


Figure 2.5: 99% latency percentile per worker for varying parallelism. Each line represents a single worker (in this case, 5, 10, and 15 workers accordingly). Incoming ratio 8000 records per second, Selectivity: 0.1%, Uniform distribution.

Table 2.3: Effects of parallelism to duplication ratio and comparisons reduction.

	Duplication Ratio		*Comparisons ratio*	
Parallelism	**S³J**	**ClusterJoin**	**S³J**	**ClusterJoin**
5	1,96	1,49	4,48	20,36
10	2,12	1,49	3,16	20,36
15	2,37	1,49	2,93	20,36
20	2,62	1,49	2,65	20,36

2.9.6 Benefits of Load Balancing

Our experiments on the effectiveness of our partitioning scheme (Figures 2.4 & 2.5) showcased that the performance of S^3J can be improved by effectively balancing the load. In Section 2.8, we propose a novel approach that addresses the balancing problem as a workset balancing problem. In Figure 2.6, we show how this load balancing approach, on top of our partitioning scheme, can benefit the performance of the similarity join task. The experiment involves two streams following a uniform distribution with a total input rate of 8000 records per second. The selectivity is set to 0.1%, and for the simplicity of the presentation, a parallelism of 5 is selected. The processing is happening in windows of 1 minute. This configuration is similar to the experiment (top left) with a parallelism of 5 from Figure 2.5. We perform load balancing at the beginning of each window.

Our load balancing scheme positively affects the performance of the similarity join job. First of all, the load balancing scheme progressively manages to reduce the maximum latency within each window. Within 3 windows of processing, the load balancing scheme reduces the maximum latency from 3 seconds in the first window (0-60s) and 3.5 seconds in the second window (60-120s) to roughly 2 seconds in the last window (180-240s). At the same time, the load balancing scheme successfully involves all available workers in the processing. In Figure 2.5 (top left), where no load balancing is employed, there is a worker (green line) that roughly receives any load throughout the experiment. We can identify the

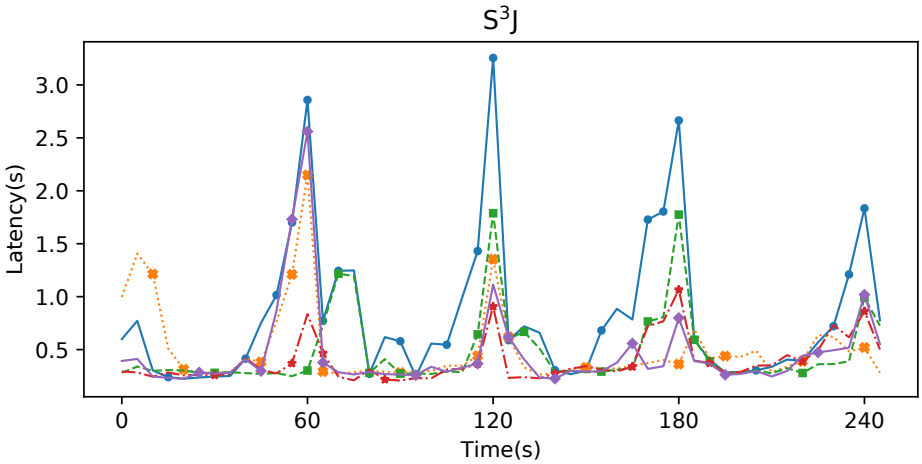


Figure 2.6: 99% latency percentile per worker. Each line represents a single worker (in this case, 5 workers in total). Uniform distribution, Incoming ratio 8000 records per sec, Selectivity: 0.1%, Parallelism: 5.

same behavior during the first window (0-60s) of the load balancing as well. However, after the first balancing action, the worker receives worksets that have an evidently high load and participates actively in the processing. Of course, we can also identify some limitations of our approach. Although the load balancing scheme manages to bring 4 out of 5 workers to similar loads, a worker still has a higher load than the rest (blue line). Responsible for this behavior is a big, heavily loaded workset. As we describe in section 2.8, we flag big worksets with a load higher than the average load of the workers as irremovable, and we do not consider them for the load balancing. In future work, we plan to divide these big worksets into smaller ones which we can then consider for load balancing.

2.9.7 Summary of experiments

Our experiments show that the partitioning scheme of S^3J can retain sub-second latency for low selectivities even with low parallelism. S^3J's partitioning can handle high selectivities significantly more efficiently than the baseline and retain a higher processing throughput for unsustainable input rates. It can scale efficiently with increasing parallelism and leverages better than the baseline the available resources. In terms of duplication, S^3J retains an almost constant ratio of 2x as the selectivity increases, in contrast to the baseline. As parallelism increases, the duplication ratio of S^3J increases, but at a slower rate. As far as comparisons reduction is concerned, S^3J manages to drastically reduce the performed comparisons, primarily thanks to its workset concept (Section 2.7) that allows S^3J to emit pairs of records belonging to the same inner set without actually performing the similarity computation. The load balancing scheme manages to redistribute well the worksets and their load to the workers. This results in gradually reducing the maximum latency as well as equally involving all workers in the processing of the load, increasing their utilization.

2.10 Conclusions

Current approaches for distributed streaming similarity joins are tailored solutions to specific problems and are unable to adapt to concept drift or load imbalance. We presented S^3J, a generic approach for distributed streaming similarity joins that tackles the problem in the general metric space and applies load balancing to adapt to load imbalances while reducing the number of computations through smart partitioning that enables the load balancing technique. Our empirical evaluation suggests that S^3J can adapt efficiently to load imbalances, scales effectively as the parallelism increases without enforcing high duplication overhead, reduces the unnecessary similarity computations, and enables low latency similarity join results for low selectivity queries.

3

Evaluating Checkpointing Protocols for Streaming Dataflows

Stream processing in the last decade has seen broad adoption in both commercial and research settings. One key element for this success is the ability of modern stream processors to handle failures while ensuring exactly-once processing guarantees. At the moment of writing, virtually all stream processors that guarantee exactly-once processing implement a variant of Apache Flink's coordinated checkpoints – an extension of the original Chandy-Lamport checkpoints from 1985. However, the reasons behind this prevalence of the coordinated approach remain anecdotal, as reported by practitioners of the stream processing community. At the same time, common checkpointing approaches, such as the uncoordinated and the communication-induced ones, remain largely unexplored.

This chapter addresses this gap by i) shedding light on why practitioners have favored the coordinated approach and ii) investigating whether there are viable alternatives. To this end, we implement three checkpointing approaches that we surveyed and adapted for the distinct needs of streaming dataflows. Our analysis shows that the coordinated approach outperforms the uncoordinated and communication-induced protocols under uniformly distributed workloads. To our surprise, however, the uncoordinated approach is not only competitive to the coordinated one in uniformly distributed workloads, but it also outperforms the coordinated approach in skewed workloads. We conclude that rather than blindly employing coordinated checkpointing, research should focus on optimizing the very promising uncoordinated approach, as it can address issues with skew and support prevalent cyclic queries. We believe that our findings can trigger further research into checkpointing mechanisms.

This chapter is based on the following paper:

George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In ICDE. IEEE. [151]

3.1 Introduction

Streaming queries constitute a crucial component of cloud applications, such as online advertising, fraud detection, real-time analytics, and Internet of Things (IoT) use cases. Streaming queries are commonly executed within multi-tenant distributed environments, subject to varying service level agreements (SLAs) regarding fault-tolerance, processing guarantees (e.g., at-least/exactly-once processing), and uptime.

The first generations of streaming engines delegated the responsibility of correctness mechanisms to the application programmers [7, 17, 37]. With the advent of cloud computing, modern streaming engines, such as Apache Flink [31], Google Millwheel [12], SEEP [63], IBM Streams [46], Hazelcast Jet [73], and Microsoft Trill [35] have adopted more advanced fault tolerance mechanisms, that achieve exactly-once processing guarantees [30, 152], without the need for programmers to change the business logic to cater for failures.

At the moment of writing, there is consensus in the use of the classic coordinated checkpointing protocol [38] and its variants for rollback recovery across production-grade stream processing engines, following its initial undertaking in Apache Flink [30]. Coordinated checkpointing protocols leverage special messages, known as markers, to capture a consistent checkpoint of the distributed global state in a coordinated fashion. Once a failure occurs, a streaming pipeline can recover by rolling all operators back to their latest checkpoint and resuming processing from an offset of the streaming input.

Despite its wide adoption, the coordinated approach has been criticized for two main drawbacks. The first is that, in large deployments, the coordination can block operators with a large number of inputs (e.g., joins or aggregates) during the marker alignment phase [6, 2]. The second issue is that in case of backpressure [1, 5], the markers cannot travel through the dataflow graph, and the checkpointing mechanism stalls, eventually halting the processing of new messages.

At the same time, multiple approaches have been proposed in the past, stemming from the original uncoordinated [24, 172] and communication-induced [15, 28, 83] checkpoints (CIC). Uncoordinated protocols allow processes to take checkpoints independently, without coordination via markers, but *i*) they require storing logs of in-flight messages, *ii*) they need to execute a recovery-line algorithm before recovery, and *ii*) the number of messages that need to be replayed upon recovery can be substantially large (depending on the recovery line found). To alleviate these issues, the communication-induced family of protocols can limit the rollback propagation during recovery by breaking the patterns that lead to invalid checkpoints with forced checkpoints during normal execution.

Despite this convergence of the stream processing engines to the coordinated checkpointing protocol, no substantial experimental evidence currently supports this system design decision against other options (e.g., uncoordinated and communication-induced checkpoints). This lack of experimental evidence can lead future streaming engines to adopt the predominant coordinated protocol along with its drawbacks, while alternative options that could behave better are ignored. Therefore, further investigation is crucial to facilitate both research and practice toward classifying checkpointing protocols and reasoning about the protocol choices that meet the needs of different workloads.

In addressing these gaps, this work is the first to revisit checkpointing for stream processing from its first principles. First, we present and analyze the theoretical cost of existing approaches. We then experimentally evaluate the three prominent checkpointing

protocol families by implementing them in a testbed system built for the needs of this evaluation. We push the protocols to their limits on diverse workloads, resulting in various topologies and processing needs, including a cyclic query. Finally, we measure the performance and the impact of the protocols both on failure-free execution as well as under failure in both uniform and skewed workloads.
In summary, this chapter makes the following contributions:

- A comprehensive survey of three families of checkpointing approaches and the conditions under which they can guarantee exactly-once processing.
- A theoretical account of the advantages and drawbacks of those three checkpointing approaches in streaming dataflows.
- An open-source streaming dataflow testbed system that enables accurate and isolated comparison of different checkpointing protocols.
- The first experimental evaluation of three checkpointing approaches on different workloads using NexMark queries [163] and a custom query that causes cycles in the dataflow graph.
- The first experimental evidence showing that:
  - Under *uniformly* distributed workloads, the coordinated approach outperforms all other approaches;
  - Under *skewed* workloads, the uncoordinated approach outperforms the coordinated one despite its expensive in-flight message logging;
  - The uncoordinated approach in practice does not suffer from the (theoretical) domino effect [59] in any of our experiments.
  - The communication-induced approach is not competitive in any scenario due to its large message overhead that it requires to avoid the (improbable, in our experiments) domino effect.

The rest of the chapter is structured as follows. In Section 3.2, we summarise all the necessary background knowledge required to understand checkpointing. Then, we discuss in detail the benchmarked protocols (Section 3.3) and the system used for the benchmarking (Section 3.4). In Section 3.7, we describe the experimental setup and present and comment on our results. In Section 3.8, we discuss related existing works. Section 3.9 concludes this chapter.
The code of CheckMate can be found online:
`https://github.com/delftdata/checkmate`
`https://zenodo.org/doi/10.5281/zenodo.11652522`

3.2 Preliminaries

In what follows, we discuss all the necessary concepts to understand better and evaluate the checkpointing protocols, particularly processing semantics and consistency, in the face of failures.

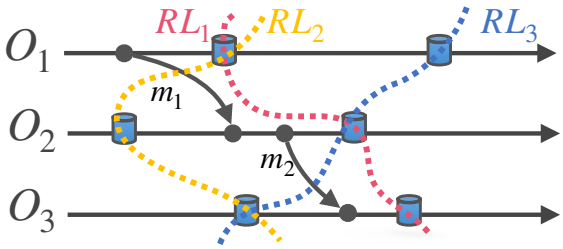


Figure 3.1: Examples of valid recovery lines when in-flight messages are included in the global state.

3.2.1 Processing Semantics

Different applications have different processing needs. Stream processing engines and their fault tolerance mechanisms provide specific processing semantics to accommodate these needs even when a failure occurs. A recent survey [66] identifies three predominant semantics with regard to processing: *at-most-once*, *at-least-once*, and *exactly-once*.

For data analytics, monitoring, or other applications that can tolerate incomplete data, a stream processing engine that provides *at-most-once semantics* is sufficient. We define *at-most-once* semantics as follows:

Definition 3.1 (At-most-once). A stream processing engine provides *at-most-once* processing semantics when it ensures that each streaming operator will process each ingested record once or not at all.

At-most-once semantics are the weakest guarantees a stream processing engine can provide. This processing guarantee has been termed *gap recovery* in the past [91]. In case of a failure, in-flight records can be lost and never be processed by downstream operators.

To accommodate applications that are intolerant of losing messages, streaming engines support *at-least-once* semantics.

Definition 3.2 (At-least-once). *At-least-once* processing semantics are provided when each ingested message is processed one or more times by each streaming operator.

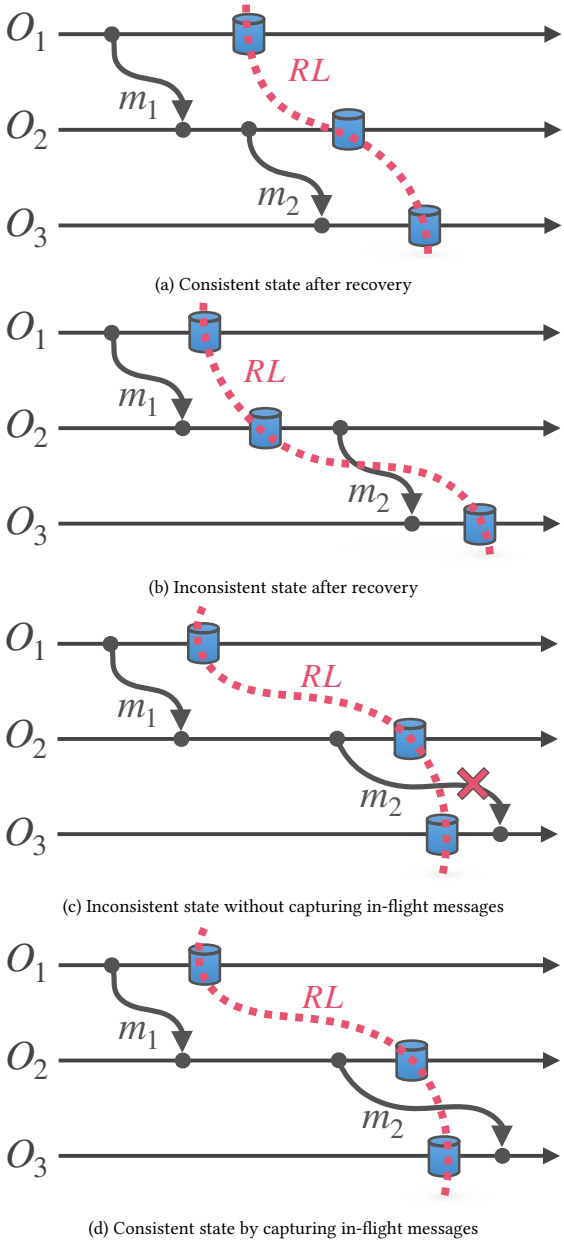
By providing at-least-once semantics, a streaming engine can avoid data loss, but at the same time, it is amenable to accounting for the same message more than once. For sensitive applications, such as bank transaction handling or aggregations, duplicate processing can cause serious anomalies. In such cases, *exactly-once* semantics are necessary.

Definition 3.3 (Exactly-once). *Exactly-once* semantics guarantee that each ingested message is processed exactly once in each operator, i.e., any state changes that occur from processing a message are reflected exactly-once on the checkpointed state.

Exactly-once semantics define strict guarantees, and they can ensure that processing under failures is identical to failure-free processing. Note that there is a distinction [66] between exactly-once *processing* and exactly-once *output* [47]. In exactly-once processing, an external system consuming the output can still observe duplicates. For instance, in case of fault recovery, the streaming system will resume processing after the latest checkpoint

and produce some output that it had already produced (but not yet checkpointed the corresponding state) prior to the failure.

In the rest of the chapter, we only consider *exactly-once* processing guarantees.



(a) Consistent state after recovery

(b) Inconsistent state after recovery

(c) Inconsistent state without capturing in-flight messages

(d) Consistent state by capturing in-flight messages

Figure 3.2: Cases of inconsistent and consistent state after recovery for stateful operators O_1, O_2 and O_3.

3.2.2 Consistency of Global State

Data stream execution is data-driven, where processing is orchestrated by messages being sent and received between tasks, triggering local computation. Without loss of generality, a distributed stream execution consists solely of `send` and `receive` operations corresponding to each message.

Modern distributed stream processing engines refer to the *global state* as the collection of the states of all operators of a streaming pipeline. We refer to an operation (`send` or `receive`) being part of the global state if it occurred before the respective state acquisition. Furthermore, the state of the communication channels can also be included in the global state. These messages are also known as *in-flight messages* or *channel state*. The *consistency* [38] of the global state is of major importance here. In order to define what a consistent state entails, we first define the concept of orphan messages:

Definition 3.4 (Orphan message). Given a global state checkpoint G, an *orphan message* has been received prior to the receiver's local checkpoint S in G, but it was not sent prior to the sender's checkpoint S' in G.

The global state of a streaming pipeline becomes inconsistent in the presence of a dropped or an orphan message [155, 29, 59]. Following the seminal processing model of Chandy-Lamport [38], we define consistent global state as follows:

Definition 3.5 (Consistent global state). The global state G of a streaming pipeline is *consistent* if for each message m :

- **No Orphans:** if `receive(m)` happened before the checkpoint acquisition, the corresponding `send(m)` operation should also happen before the checkpoint.
- **No Dropping:** if `send(m)` happened before the checkpoint acquisition then either `receive(m)` happens before the checkpoint or m is added in the checkpoint as an *in-flight* message.

In principle, consistency is straightforward to maintain and reason about under the normal operation of a streaming system. In the face of failures, however, a streaming system ought to roll back to a previously consistent global state in order to resume its operation and regain consistency. At that point, the recovery mechanism attempts to recover such a global state from the collection of existing operator checkpoints.

Recovery line. A *recovery line* consists of a collection of operator checkpoints that can be used to recover the global state (Figure 3.1). Since not all candidate recovery lines lead to a consistent state, the recovery mechanism must find the most recent recovery line corresponding to a consistent state. Checkpoints that cannot belong to a consistent recovery line are considered *invalid*.

In Figure 3.2, we provide example cases that illustrate when a recovery line and its corresponding global state are consistent. Figure 3.2a showcases a consistent global state since all messages are sent and received before the checkpoints that compose the recovery line. In Figure 3.2b, message m_2 is an orphan message since its side-effects are reflected in the checkpoint of O_3 but not in the checkpoint of the sender operator O_2. Therefore, the global state corresponding to this recovery line is inconsistent, and the recovery line is unsuitable for recovering from a failure.

Table 3.1: Summary of the features of the checkpointing protocols explored in Section 3.3

	Blocking (markers)	In-flight Logging	Deduplication Required	Message Overhead	Independent Checkpoints	Straggler Stalls	Unused Checkpoints	Forced Checkpoints
Coordinated	∘	–	–	–	–	∘	–	–
Uncoordinated	–	∘	∘	–	∘	–	∘	–
Communication-induced	–	∘	∘	∘	∘	–	∘	∘

If in-flight messages (i.e., channel state) are not captured, then a different type of global state inconsistency appears. In Figure 3.2c, operation `send(`m_2`)` occurs before O_2 acquires its checkpoint, whereas, `receive(`m_2`)` occurs after O_3 takes its checkpoint. Using this recovery line without a captured channel state will result in never processing m_2 at operator O_3 and, therefore, dropping messages. In this case, to achieve a consistent global state, capturing the channel state and replaying in-flight messages is necessary (Figure 3.2d). To ensure exactly-once semantics when in-flight messages are replayed, some form of message deduplication must be employed.

3.3 Checkpointing Protocols

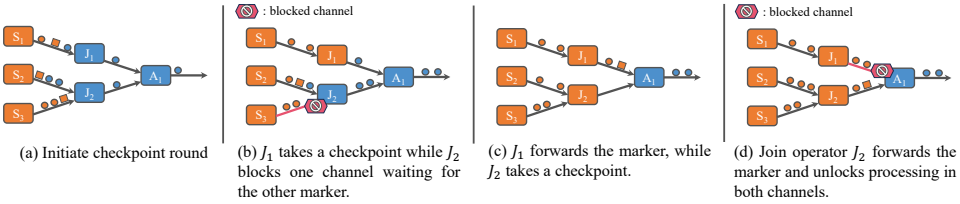
In what follows, we describe the three main checkpointing protocols and discuss their core ideas and some possible drawbacks. In table 3.1, we summarise the necessary mechanisms employed for each protocol to ensure exactly-once processing and the main side effects and features of each protocol.

3.3.1 Coordinated Aligned Checkpointing (COOR)

To the best of our knowledge, virtually every stream processing engine in production that guarantees exactly-once processing, implements a variation of the coordinated checkpointing protocol [30, 73, 35, 46]. Typically, in stream processing engines that implement a coordinated checkpointing protocol, the operators will block processing to allow the alignment of a checkpoint across the system. The checkpoint can be used to create a recovery line in case of a failure. The most adopted version of such a protocol is the Chandy-Lamport marker-based algorithm [38] and its adaptation for acyclic dataflow graphs [30]. In what follows, we describe the core ideas of the protocol, and we illustrate its core functionality with an example.

At its core, the coordinated aligned checkpointing protocol works as follows:

- A checkpoint round initiates at source operators by taking a checkpoint. After taking its checkpoint, each source operator forwards a marker to all its outgoing channels and continues processing.
- When an operator (excluding source operators) receives a marker from an incoming channel, it blocks that channel and buffers the channel's traffic.
- When an operator receives a marker from all its incoming channels, it takes a checkpoint, unblocks processing in all incoming channels, and forwards a marker to all its outgoing channels.
- When the markers reach the end of the pipeline, and the checkpoints are stored in durable storage, the coordinated checkpoint round finishes.



(a) Initiate checkpoint round

(b) J_1 takes a checkpoint while J_2 blocks one channel waiting for the other marker.

(c) J_1 forwards the marker, while J_2 takes a checkpoint.

(d) Join operator J_2 forwards the marker and unlocks processing in both channels.

Figure 3.3: Example execution of the coordinated aligned checkpointing protocol. Messages are represented as circles, and markers are squares. Different colors denote different coordinated rounds.

By blocking processing until all markers are received from the upstream operators, we achieve the alignment of the checkpoints. This alignment guarantees exactly-once processing without the need to capture in-flight messages and the channel state, as it creates a frontier of processed messages through the use of markers.

Figure 3.3 illustrates an example protocol execution. The execution graph presented consists of only the first couple operators of the pipeline. Operators $S_{\{1-3\}}$ are parallel source operators, operators J_1 and J_2 are parallel stateful join operators, and operator A_1 is a stateful aggregation operator. A coordinated checkpoint round is initiated at the source operators by taking a checkpoint. When a parallel source operator finishes with its own checkpoint, it sends a checkpointing marker to all its outgoing channels (figure 3.3(a)) and continues processing. In figure 3.3(b), operator J_1 has received the marker from its sole incoming channel and takes a checkpoint. On the other hand, operator J_2 has received a marker from source operator S_3 and blocks processing in that channel while it waits for the marker from S_2. J_2 takes a checkpoint when it has received all markers, while J_1, after taking the checkpoint, forwards a marker to its downstream operator and unblocks processing in all the incoming channels (figure 3.3(c)). Finally, J_2 also forwards a marker and continues processing after taking a checkpoint (figure 3.3(d)). The markers will then be received by A_1, and the checkpointing process will continue in the same way until it reaches the end of the pipeline.

Strengths. Compared to the in-flight message logging required in uncoordinated approaches (Section 3.3.2), the markers used by the coordinated protocol are lightweight and are not affected by the message size. Additionally, since aligned checkpoints compose a consistent global state, an algorithm that identifies the recovery line is not required.

Drawbacks. One important downside of marker circulation surfaces in cases of stragglers, e.g., due to skewed workloads and/or backpressure. For example, if most of the load falls on a single operator, its downstream operators would have to block the other channels, wait for the straggler to finish processing, and then forward a checkpoint marker. Additionally, in case of shuffling, the protocol needs to transfer as many markers as the parallel instances of the receiving operators (one to each parallel instance). In essence, coordinated checkpoints could take a substantial amount of time in complex topologies due to the markers having to pass through the entire dataflow graph to be completed.

Another drawback of the coordinated protocol is that it does not support cyclic streaming workloads out of the box. Cycles are an integral aspect of iterative computations such as fixpoint calculations, which are common in graph queries [124]. Accounting for cycles

Algorithm 2 Rollback propagation algorithm [172]

Require: all available checkpoints CP ordered by freshness for each operator, a checkpoints graph
Ensure: a consistent recovery line
1: include in *root_set* the latest CP of each operator;
2: mark all CPs in the *root_set* that are strictly reachable from any other CP in the *root_set*;

3: **while** $\exists CP.marked \in root_set$ **do**
4: $\forall CP.marked \in root_set$ replace by the next unmarked CP from the same operator;
5: mark all CPs in the *root_set* that are strictly reachable from any other CP in the *root_set*;
6: **end while**
7: **return** *root_set*

in the coordinated checkpointing protocol entails a) special handling of markers in order to avoid deadlocks owed to the blocking of the cyclic input channel by a marker and b) additional progress tracking mechanisms.

3.3.2 Uncoordinated Checkpointing (UNC)

The *uncoordinated checkpointing* (UNC) [172] protocol allows each operator to decide individually when to take a checkpoint. In contrast to the coordinated approach, there are no markers since there is no need for coordination, and the protocol can only provide at-most-once processing semantics since the checkpoints only contain the operator state. Thus, capturing the channel state between operators is necessary to provide stronger guarantees. To do so, log-based recovery and upstream backup [22, 43] need to be implemented. Pairing uncoordinated checkpointing with a log for keeping track of the channel state allows the replay of messages after recovery, achieving at-least-once semantics. For the protocol to achieve exactly-once semantics, message *deduplication* must be employed when replaying messages from the message log.

Finding Recovery Lines. The freedom of taking checkpoints independently per operator comes with a cost when recovering after a failure. Since the checkpoints are not coordinated, we cannot simply use the most recent operator checkpoints as a recovery line, as it might not correspond to a consistent global state. Therefore, we need to employ a recovery line algorithm to find a suitable recovery line, i.e., one that provides a consistent global state and has the minimum rollback distance. The algorithm for finding such a recovery line is the *rollback propagation algorithm* [172], which requires a checkpoint dependency graph. There are two approaches to creating such a graph, the *rollback dependency graph* [25] and the *checkpoint graph* [172]. Both of these approaches result in the same recovery line, and in this work, we opt for the *checkpoint-graph* [172] since it is more intuitive.

The checkpoint graph has checkpoints as nodes and directed edges between two checkpoints $c_{i,x}$ and $c_{j,y}$ if:

- $i \neq j$, i.e., the checkpoints belong to different operators, and there is at least one orphan message that was sent from operator i after checkpoint $c_{i,x}$ was captured and

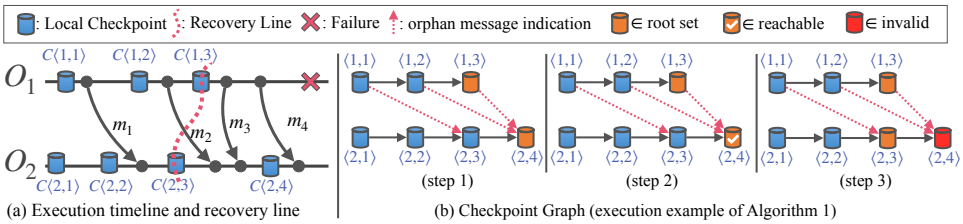


Figure 3.4: Example overview of Rollback propagation algorithm on a given execution timeline

3

was processed from operator j before checkpoint $c_{j,y}$ was taken.

- $i = j$ and $y = x + 1$, i.e., $c_{i,x}$ and $c_{j,y}$ are consecutive checkpoints of the same operator.

In Figure 3.4, we provide an example of a checkpoint graph and showcase step by step how the rollback propagation algorithm uses the checkpoint graph to find a suitable recovery line. To create the checkpoint graph, we include the IDs from channel state logs for the last received and last sent messages alongside the checkpoints. We can identify orphan messages using these IDs and add directed edges in the checkpoint graph (Figure 3.4(a)). The rollback propagation algorithm uses this graph to find the recovery line. First, the algorithm will include the last checkpoints of all operators in a set called the root set (Figure 3.4(b) - step 1). The next step is to identify the nodes in the root set that are strictly reachable from other nodes in the root set and mark them (Figure 3.4(b) - step 2). Then, each marked checkpoint in the root set is replaced by the next most fresh checkpoint for the same operator, and the newly added checkpoints are checked and marked if applicable (Figure 3.4(b) - step 3). When the algorithm reaches a root set that does not include any marked checkpoint, it returns this root set as the desired recovery line.

Strengths. The primary strength of any coordination-free protocol is that it does not block waiting for markers from a coordinator node or its upstream operators, leading to lower latency in the event of a skewed workload. Another benefit yet to be explored by literature is the configurability of such an approach. For instance, the stateless, non-source operators in the uncoordinated approach do not need to participate in the checkpointing pipeline, which is not the case in the coordinated approach because they still would have to propagate the markers. Furthermore, different operators can have different checkpoint intervals, making them adaptive to the current system's needs (e.g., a windowed aggregation operator can checkpoint right after the aggregate is calculated in order to avoid storing the large window's contents).

Drawbacks. To provide exactly-once semantics, message logging is required. However, message logging is costly and can considerably impact the system's performance. Moreover, since checkpoints are not aligned, some captured checkpoints may be rendered invalid when looking for the appropriate recovery line (an invalid checkpoint cannot take part in any recovery line). As seen in Figure 3.5, this problem could be aggravated when dealing with cyclic queries, leading to a phenomenon known in the literature as the *unbounded domino effect* [59], where during recovery, one checkpoint after the other is rendered invalid leading to a considerable rollback distance or even starting from scratch. In Figure 3.5,

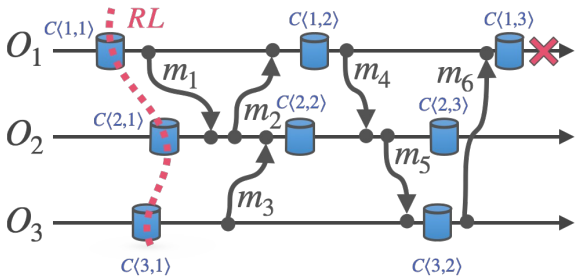


Figure 3.5: Domino effect of invalid checkpoints on a cyclic query.

the first option would be a recovery line consisting of the checkpoints $C_{<1,3>}$, $C_{<2,3>}$, and $C_{<3,2>}$; however, this is invalid due to the orphan message m_6. The next option is the recovery line consisting of $C_{<1,2>}$, $C_{<2,3>}$, and $C_{<3,2>}$ with again m_4 making this invalid. m_5 makes the $C_{<1,2>}$, $C_{<2,2>}$, and $C_{<3,2>}$ invalid. The domino effect continues with the rest of $m_{3,2,1}$ leading to $C_{<1,1>}$, $C_{<2,1>}$, and $C_{<3,1>}$ being the only available recovery line option.

3.3.3 Communication-induced Checkpointing (CIC)

The *communication-induced checkpointing* (CIC) protocol is built on top of UNC and provides a loose coordination of the checkpoints in order to tackle the problem of the *unbounded domino effect*. This loose coordination happens through encapsulating information related to the protocol in the messages containing records across the pipeline. This protocol recognizes two different types of checkpoints: a) *local checkpoints* (equivalent to uncoordinated checkpoints), and b) *forced checkpoints*, which are inserted by the protocol to prevent the domino effect.

Communication-induced protocols are tightly connected to Z-paths and Z-cycles [59] based on the fact that a given checkpoint is invalid if and only if it is part of a Z-cycle. A CIC protocol tries to detect Z-cycles and break them by forcing checkpoints before processing messages that will lead to a cycle. Alvisi et al. [15] have shown that a CIC protocol can handle cyclic communication patterns without the risk of a domino effect, but they may introduce significant overhead.

The most complete and well-documented CIC protocols are BCS [28] and HMNR [83]. Initial tests indicate that the HMNR has better performance than BCS. Therefore, in this work, we adopt HMNR as our CIC protocol. In short, in HMNR each operator keeps a Lamport clock and a vector clock plus three boolean vectors with a length equal to the number of operators participating in the pipeline. Every operator updates his Lamport clock by increasing its value when it takes a new checkpoint. The vector clock *ckpt* stores how many checkpoints have been taken by each operator from the perspective of the current operator. A boolean vector *sent_to* keeps information about messages sent to other operators since the last checkpoint of the current operator. Another boolean vector *taken* stores the existence of Z-paths since the last known checkpoint. The last boolean vector *greater* stores the information whether the operator's clock is greater or not from each other operator's clock. The operator's Lamport clock, the vector clock *ckpt*, the boolean vector *taken*, and the boolean vector *greater* are piggybacked to every message. The protocol

uses all these structures to detect cycles and decide when to force a checkpoint. When an operator receives a message, it checks if there is a message previously sent from it to the sender and the sender's clock is larger than its own or if there is a Z-path detected in the current checkpoint interval of the sender operator. More details on the cycle detection and the forced checkpoints can be found in the original paper [83].

Strengths. The primary strength of the CIC protocol is the forced checkpoints mechanism, leading to a smaller rollback distance and, most importantly, eliminating the domino effect.

Drawbacks. The main drawback of a CIC protocol is the overhead it introduces. For big and complex pipelines, the vector clocks and the boolean vectors can be rather large and greatly impact the size of the messages flowing throughout the system.

3.4 Testbed System

We compared the checkpointing protocols in Styx [135], the backend of Stateflow [136]. For the requirements of our experiments, we developed all necessary protocol mechanisms (e.g., message logging and coordination) and streaming operators (i.e., map, filter, window, join, aggregates).

The Stateflow cluster consists of the typical architecture. A coordinator node is responsible for scheduling/deploying the dataflow graph to workers and running the coordination logic of the checkpointing protocols. Worker nodes execute the dataflow logic and take checkpoints asynchronously based on the checkpointing algorithm. Finally, Stateflow uses Apache Kafka as a replayable fault-tolerant source and Minio as a persistent state store for the operator state checkpoints.

We choose Stateflow for the following reasons: i) unlike other streaming dataflow systems such as ApacheFlink, Stateflow allows for cycles in the dataflow graph; ii) Stateflow provides a sandboxed environment, where we can evaluate the different protocols in isolation, without additional overhead; iii) Other systems (e.g., Apache Flink) base their entire design on coordinated checkpoints – when implementing uncoordinated protocols on Apache Flink, we realized that we needed to virtually rewrite the complete system itself.

3.5 Metrics

Although there is a significant body of work in benchmarking and evaluating stream processing systems and fault tolerance (Section 3.8), no metrics are established to measure the performance of a checkpointing protocol meaningfully. In this work, we argue that the following metrics should be used to evaluate the performance of such a protocol.

End-to-end Latency. A standard metric to evaluate the performance of stream processing systems is the *end-to-end latency*, i.e., the time it takes for a record to result into output in the sink from the moment it is available in the input queue. Although latency is mainly related to the deployed query and the underlying system rather than the checkpointing protocol itself, it allows us to measure the impact of each protocol on normal execution, as the overhead it introduces in terms of latency. We opt to measure the 50th and 99th percentiles.

Sustainable Throughput. Another common metric in stream processing literature is the *maximum sustainable throughput* [95]. The *maximum sustainable throughput* indicates

the maximum throughput that the system can handle for a long period of time without provoking backpressure. Backpressure leads to constantly increasing latencies and an average processing throughput that is lower than the rate of incoming messages. Similarly to end-to-end latency, it allows us to assess the impact of the checkpointing protocol on the overall performance.

Average Checkpointing Time. In this work, we measure the *average checkpointing time*, i.e., the average time it takes for each protocol to take a checkpoint. The fundamental differences between the protocols lie in checkpoint triggering and the additional information that needs to be captured apart from the internal state. Therefore, measuring how these differences affect the time it takes to capture a checkpoint is crucial. Also, as the checkpointing time rises, a significant impact on the processing performance is expected.

Restart & Recovery Time. Restart time consists of all the time the system spends to reload all the needed states and be ready to process data. The recovery time, on the other hand, informs us how long it takes to recover from a failure. The measurement starts when the failure is detected and finishes when the system has managed to return to normal execution. The higher the recovery time, the bigger the impact of a failure. Recovery time also encompasses restart time.

Invalid Checkpoints. Depending on the checkpointing protocol, invalid checkpoints may exist, i.e., checkpoints that cannot be part of a consistent recovery line and, thus, cannot be used for recovering after failure. The existence of invalid checkpoints can be problematic as the state grows since a lot of expensive storage space is occupied by information that will never be used. Moreover, invalid checkpoints can lead to significant rollback distance, which will result in replaying and reprocessing a significant number of messages. Therefore, the number of invalid checkpoints is a good indicator of the performance of a checkpointing protocol. The fewer invalid checkpoints exist, the better a protocol is performing.

Message Overhead. Each protocol introduces messages and requires specific information to be exchanged between workers or sent to the coordinator. Measuring the size of protocol-related information that circulates the system during execution allows us to capture the overhead that the protocol introduces in network usage. A higher percentage of protocol-related information means that a significant portion of our network is used, and additional serialization/deserialization CPU time is spent on information unrelated to processing.

3.6 Streaming Query Workload

To evaluate the checkpointing protocols, we employ four distinct queries from NexMark [163] and our adaptation of the cyclic query introduced in [36].

NexMark Queries. NexMark benchmark [163] simulates an e-commerce application and provides streaming queries with different properties and needs. We selected the following four queries, which allow us to measure the performance and the impact of the checkpointing protocols in different conditions:

- *Query 1* is a stateless map query that transforms the bid values. There is no shuffling.
- *Query 3* implements an incremental stateful join, which joins persons with auctions. It involves a complex topology and shuffling between operators.

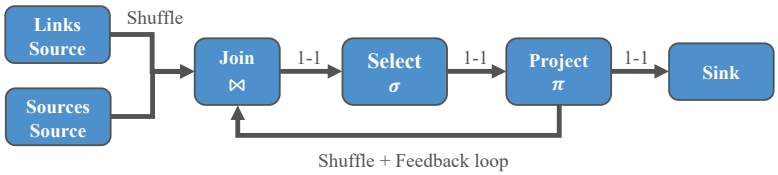


Figure 3.6: Execution graph of the reachability query.

- *Query 8* employs a windowed join between users and auctions. We opt for a processing time tumbling window; however, the type of the time window does not affect the checkpointing protocol's performance. It employs a complex topology, shuffling, and the complexity of the windowing. To meaningfully measure the impact of the protocols on the latency during execution, we implement a running window, i.e., the processing is triggered on record arrival, and the window is cleaned when it expires.

- *Query 12* employs a windowed count over bids. Similarly to query 8, we choose the running version of a processing time tumbling window. The query performs aggregation over time windows and includes minor shuffling.

Fundamental processing operators in modern stream processing engines [73, 31, 13] include maps, joins, windows, and aggregates. The queries we choose represent those fundamental operations and sufficiently cover the operations appearing in the NexMark suite.

Cyclic Query. Most stream processing engines do not support cyclic queries. However, there is existing research on cyclic or recursive queries in stream processing [36, 127, 124]. To further enable research on cyclic streaming queries and to encourage stream processing engines to support such queries, it is essential to evaluate existing checkpointing protocols with cyclic queries. For our evaluation, we adapt the reachability query employed by FFP [36]. Given a static set of nodes, the goal of the query is to identify all reachable nodes from the available source nodes based on the available directed links between the nodes and provide the corresponding paths. The available source nodes and the directed links between the nodes are not known a priori, but they are processed on the fly and are temporal. Figure 3.6 illustrates the execution graph of the query. The query ingests two streams, the directed links between the nodes and the source nodes. Directed links are joined with sources that contain the starting node of the link as a reachable node. In the select operator, we check if the end node of the directed link of a joined pair is contained in the path of the source of the pair, and we discard such pairs. In the project operator, we discard unnecessary information and create a new source with the same source node, the end node of the link as a reachable node, and the path augmented by the pair's link. The new source is provided as output and recursively as input to the join operator. Finally, the join operator can receive direct messages when a specific link or source node is unavailable. In that case, it will remove every link or source affected from its state.

3.7 Experimental Evaluation

3.7.1 Evaluation setup

The experiments are conducted on a local cluster with AMD EPYC 7H12 2.60GHz CPUs and 512GBs of memory. We deploy our benchmarking system using docker and docker-compose. Each worker uses 1 CPU for processing and handles a single parallel instance of each of the operators of the deployed pipeline. We do not use any limits on memory usage. Apache Kafka is used as the source and the sink of our system. Minio is used as a persistent storage for the checkpoints. We extend the NexMark generator from [93, 149] to provide the input in the required format of the system, and we provide a generator that creates source nodes and corresponding links for our cyclic query. We evaluate the three checkpointing protocols using the NexMark queries and our cyclic query. We implement and compare the vanilla versions of the protocols as described in Section 3.3 in order to ensure a fair comparison of their core concepts that is not affected by optimizations tailored to specific system properties.

3.7.2 Results

In what follows, we present the results of our experimental evaluation of the three checkpointing protocols concerning the metrics for benchmarking checkpointing protocols that we previously discussed in Section 3.5. For the NexMark queries, we distinguish two settings: a balanced setting where the distribution of our input follows the uniform distribution and a skewed setting where we leverage NexMark's generator to provide different percentages of hot items.

NexMark Queries. In practice, streaming systems are overprovisioned, ensuring a stable execution that does not cause backpressure in case of input rate fluctuations or transient system issues (e.g., garbage collection). In our experiments, we run all queries at 80% of the maximum sustainable throughput that each protocol achieves for each query and parallelism. We found 80% to be the most stable configuration. Each run lasts for 60 seconds with 30 seconds of warmup. We introduce a failure on the 18th second of a 60-second run.

– *Maximum Sustainable Throughput (MST).* In Figure 3.7, we present the maximum sustainable throughput (MST) each protocol achieved normalized by the MST of the checkpoint-free execution for each query. For Q1, Q8, and Q12, the coordinated approach outperforms the rest and reaches the same MST as the checkpoint-free execution until we reach 70 workers. For 70 and 100 workers, we observe a slight decrease in MST for Q1 and Q12, which results in approximately 90% of the checkpoint-free MST. The impact of the increase in parallelism is more significant for Q8, which employs a join. The uncoordinated protocol follows closely, achieving an MST around 10% lower than the coordinated approach in all cases. On the other hand, the communication-induced protocol fails to keep up and, in higher parallelism, can reach an MST lower than 50% of the checkpoint-free MST. None of the protocols can keep up with the checkpoint-free execution for Q3. However, the coordinated and uncoordinated protocols achieve an MST higher than 70% of the optimal for Q3 in most cases while maintaining an MST of 50% of the optimal for the edge case of 100 workers. On the contrary, the communication-induced protocol fails to achieve an MST higher than 50% for Q3 primarily due to the high message overhead it introduces. In terms

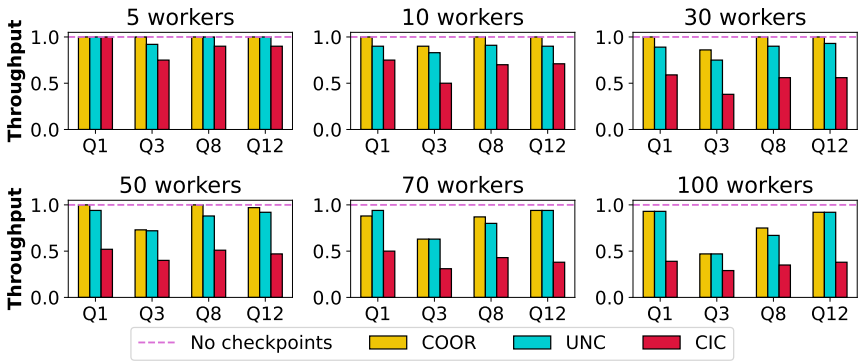


Figure 3.7: Normalized maximum sustainable throughput per query achieved by each protocol for different parallelism.

Table 3.2: Ratio of message overhead with respect to an execution without checkpoints.

	10 workers				50 workers			
Protocol	Q1	Q3	Q8	Q12	Q1	Q3	Q8	Q12
COOR	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
UNC	1.00x	1.00x	1.00x	1.00x	1.00x	1.01x	1.01x	1.00
CIC	2.10x	1.82x	1.74x	1.79x	2.53x	2.58x	2.49x	2.58x

of MST, the coordinated approach outperforms the others, while only the uncoordinated can remain competitive.

– *Message Overhead.* The overhead of the protocol-related information transferred throughout the system can either be in the form of additional protocol messages and/or piggybacked information to process messages. The only protocol-related overhead for the coordinated approach is the messages between workers and the coordinator when starting and concluding a coordinated round, and the markers forwarded from the sources to the pipeline sinks. The uncoordinated protocol requires the operators to send the metadata of every checkpoint they take to the coordinator. Table 3.2 shows that the overhead that the coordinated and the uncoordinated introduce is insignificant in all cases. On the contrary, as explained in Section 3.3, additionally to the information required by the uncoordinated protocol, the communication-induced protocol piggybacks to the process messages all the information required to decide on forcing a checkpoint. The size of this information depends on the number of total instances of the operators employed. As Table 3.2 indicates, even for a parallelism of 10, the overhead can double the size of the messages that are communicated between the workers and the coordinator, while for 50 workers, the message size can reach up to 2.58x the size of messages of a checkpoint-free execution. Increased message size does not only result in the need for higher network bandwidth but also cripples the processing power of our system as it has to serialize and deserialize much larger messages. Therefore, it significantly affects the maximum sustainable throughput we can achieve using the communication-induced protocol, as shown in Figure 3.7.

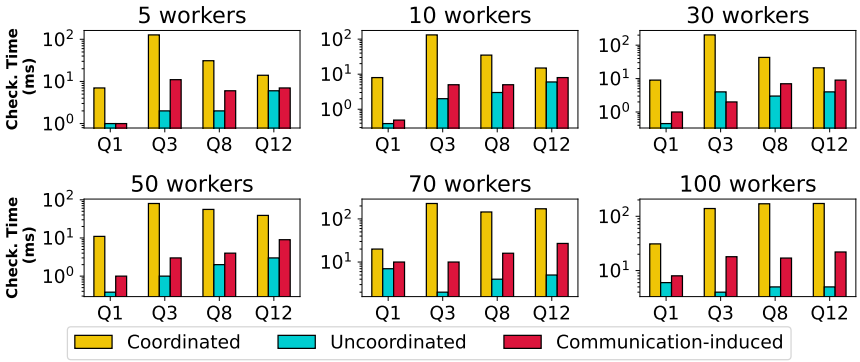


Figure 3.8: Average checkpointing time on different parallelisms.

– *Average Checkpointing Time.* We showcase the average checkpointing time for each protocol for all settings in Figure 3.8. The uncoordinated and communication-induced protocols have an average checkpointing time of a few milliseconds for all settings. The coordinated approach requires a full checkpointing round to be completed to consider its checkpoints as valid. Therefore, in contrast to the other protocols, it incurs an average checkpointing time of up to two magnitudes higher for Q3, Q8, and Q12, which involve shuffling. This is especially the case for Q3, which employs a complex topology and has a high computational complexity, as well as for the higher parallelisms that result in a higher degree of shuffling. The latency overhead caused by the increased checkpointing time in Q3 is also visible in Figure 3.9 for 10 workers.

– *Impact on the 50th and 99th percentile of latency.* In Figure 3.9 and Figure 3.10, we present the 50th and 99th percentiles per second for each protocol and query for different parallelisms. Due to space limitations, we include 10, 30, and 50 workers in our discussion. However, the other settings follow a similar trend. The 50th percentile latency allows us to evaluate the mean performance of the protocols, while the 99th percentile highlights the stragglers and the outliers. For the settings of 10 and 30 workers, the 50th percentile for all protocols for the simpler queries Q1, Q8, and Q12 is similar before the failure occurred and after the system recovered to a stable execution. However, for the 50-worker case, the communication-induced protocol requires piggybacking additional protocol information of significant size at every message. This results in a slight increase observed in the 50th percentile, which is considerably higher in Q8 because it employs a costly join. As for Q3, the coordinated approach suffers from latency spikes every time a checkpoint is taken, which is more evident as the state grows and for the 10-worker case. The 99th percentile follows the same patterns as the 50th percentile for the execution period prior to the failure. Q3 employs an incremental join; the spikes and the increasing instability in latency that we observe are expected and attributed to a combination of the query's nature and checkpointing.

– *Recovery & Restart Time.* Recovery time is the time passed from detecting the failure until the system returns to normal and stable execution. Looking at the 50th percentile (Figure 3.9), all protocols require around 10 seconds to recover to normal execution for

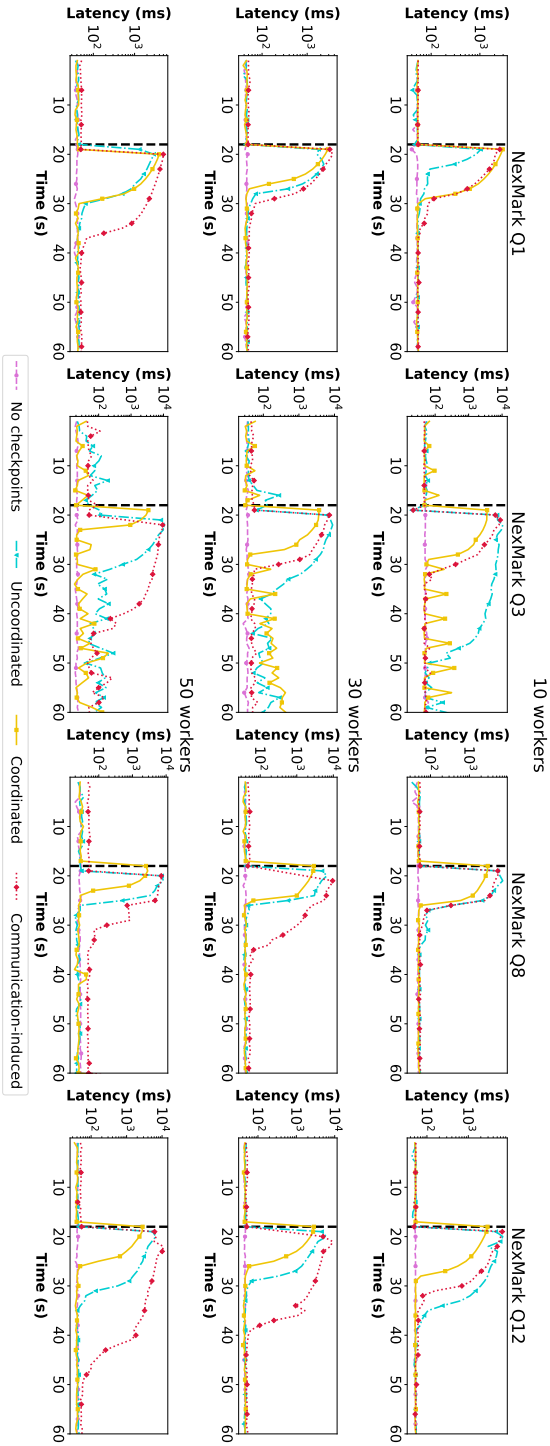


Figure 3.9: 50th percentile latency. The black dashed vertical line indicates the moment of failure.

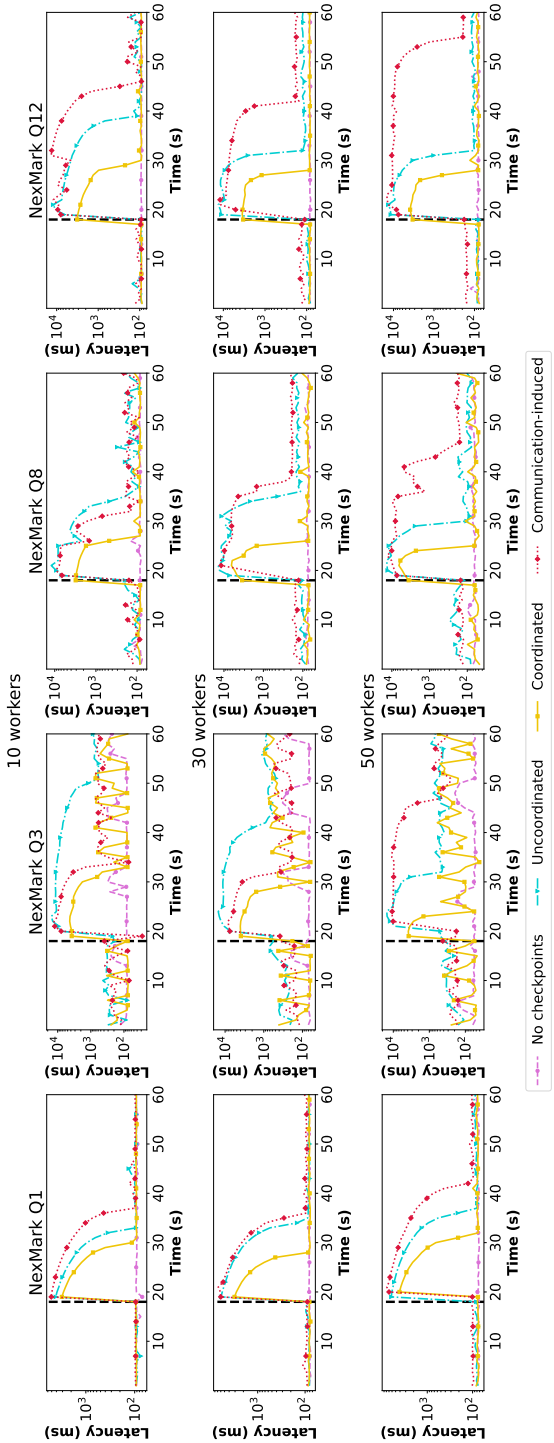


Figure 3.10: 99th percentile latency. The black dashed vertical line indicates the moment of failure.

Figure 3.11: Restart time after failure per query for each protocol on different levels of parallelism.

Q1 for a parallelism of 10, while very small differences are also observed for Q1 for 30 workers. For 50 workers, the communication-induced protocol requires around 10 more seconds to recover due to the significant message overhead it introduces. For Q8 and Q12, the communication-induced protocol performs marginally better than the uncoordinated protocol for 10 workers, but it falls behind when the parallelism increases as it requires around 10 more additional seconds to recover. In Q3, the communication-induced protocol has a smaller recovery time than uncoordinated by up to 20 seconds for 10 and 30 workers, resulting from replaying fewer messages due to forced checkpoints closer to the failure. On the other hand, it requires 10 additional seconds for 50 workers. On average, the coordinated protocol greatly outperforms the other protocols regarding recovery time, mostly because the uncoordinated and communication-induced protocols have to replay many messages.

The restart time (Figure 3.11) is part of the recovery time and reflects the time passed from detecting the failure until the system is ready to restart processing. On average, the coordinated protocol restarts faster than the other two protocols. This is especially evident for a larger number of workers. For example, the restart process for the uncoordinated and communication-induced protocols can take up to 10 times longer than the coordinated for 100 workers. The UNC and CIC protocols need to fetch and prepare the messages to replay and, therefore, take more time to restart. On the other hand, finding the recovery line has an insignificant cost.

– *Invalid checkpoints.* The percentage of invalid checkpoints over the total checkpoints indicates how much the system rolled back. Low percentages show no domino effect and better utilization of the checkpointed state. The coordinated approach does not introduce any invalid checkpoints. Table 3.3 shows that for all the acyclic queries, the uncoordinated and communication-induced protocols introduce very few invalid checkpoints and result in similar total checkpoints. Overall, the uncoordinated and communication-induced protocols result in more checkpoints than the coordinated protocol since every operator independently decides when to take a checkpoint based on its worker's clock.

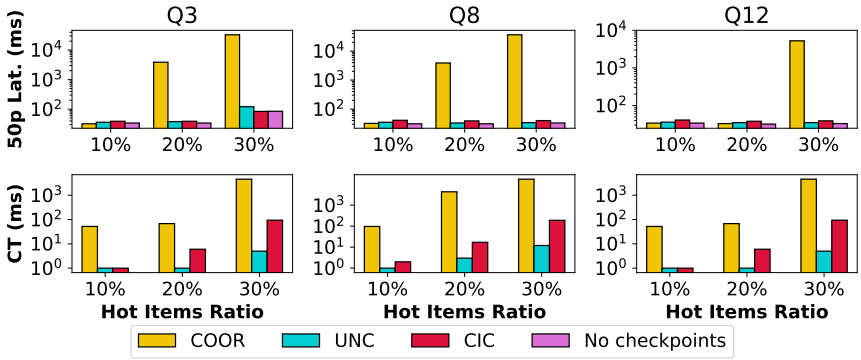
Table 3.3: Total checkpoints and percentage of invalid checkpoints.

	10 workers			50 workers		
	Total(Invalid)			***Total(Invalid)***		
Query	*UNC*	*CIC*	*COOR*	*UNC*	*CIC*	*COOR*
Q1	303(0%)	285(0%)	240(0%)	1437(0%)	1428(0%)	1200(0%)
Q3	455(4%)	471(3%)	400(0%)	2399(3%)	2517(4%)	2000(0%)
Q8	384(2%)	386(3%)	360(0%)	1924(2%)	1920(3%)	1800(0%)
Q12	282(3%)	282(4%)	240(0%)	1446(3%)	1451(3%)	1200(0%)

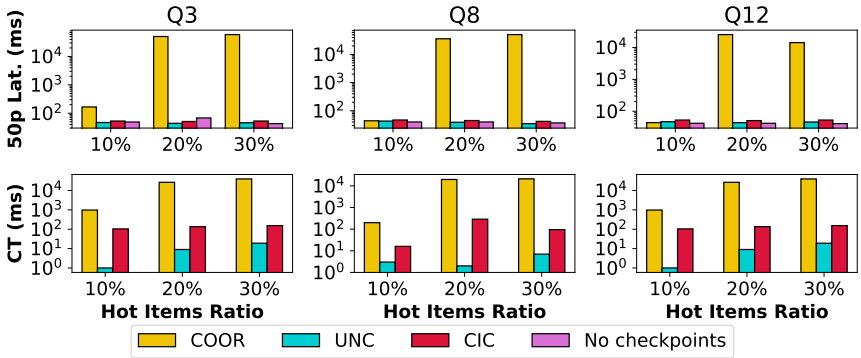
Skewed NexMark. Operating under a skewed workload usually results in workers straggling to process the excessive load they are responsible for. Although operating under such conditions is not preferable, avoiding it is not always feasible. Therefore, it is important to investigate how the different protocols perform under skew. To measure the impact of skew on the protocols' performance, we employ Q3, Q8, and Q12 under different hot item ratios provided by the NexMark generator. Q1 is not affected by skew as it involves non-keyed operations. Therefore, we omit it. We run Q3, Q8, and Q12 on 10 workers at 50% and 80% of the maximum sustainable throughput of the non-skewed execution of every protocol without introducing any failure. Both throughputs result in straggling workers. However, the latter stresses significantly more the system, resulting in fewer checkpoints taken and higher sensitivity to skew. We consider these settings representative of a real-world deployment, where overprovisioning is employed to handle spikes and unexpected skews. We employed three different hot item ratios to increase the skew gradually, from 10% to 30%. The straggling workers heavily affect the 99th percentile of latency, so we focus on the 50th percentile. We also report the average checkpointing time, as it is also heavily affected by the skew and can significantly affect the latency.

Unlike the non-skewed experiments, as illustrated in Figure 3.12, the coordinated protocol performs the worst regarding 50th percentile latency and average checkpointing time in both throughputs. With every increase in the hot items ratio, latency and checkpointing time increase by at least an order of magnitude for the lower throughput, while for the higher throughput, even the lowest skew ratio has a significant impact on Q3. The coordinated protocol is so heavily impacted by skew because not only are the straggling operators slow to take their checkpoints, but they also delay propagating their markers to downstream operators that block processing in other channels to wait for the delayed markers. Meanwhile, both UNC and CIC keep both metrics relatively low. In summary, the uncoordinated and communication-induced protocols can handle skew more effectively in every case.

Similar to the non-skewed experiments, we perform another run using the 50% MST, introducing a failure. Figure 3.13 shows the time needed to restart processing. Unlike the non-skewed experiments, where the coordinated outperformed the other approaches, the differences are mitigated under skew, and all protocols perform similarly. This is an immediate result of the coordination under skew with the stragglers. Invalid checkpoints remain the same under skewed and non-skewed conditions. We do not report recovery time since none of the protocols managed to recover within the time frame for 20% and



(a) 50% of the MST of the non-skewed execution.



(b) 80% of the MST of the non-skewed execution.

Figure 3.12: 50th percentile latency & average checkpointing time under different hot items percentages.

30% skew, while for 10% skew, the performance is similar to the non-skewed experiments.

Cyclic query. For the cyclic query, we only evaluate the *uncoordinated* and *communication-induced* checkpointing protocols. The aligned version of the coordinated protocol cannot handle cyclic queries. That is because at least one operator would be waiting for a marker that originates from itself, thus leading to a deadlock.

We evaluate the protocols with two parallelisms, 5 and 10 workers. We refrain from using higher parallelisms since CIC is greatly affected by complex topologies and higher parallelism, as shown in Figure 3.7. For both deployments, we use the same configuration for our generator. It creates events with the following probabilities: 60% chance of creating a new link, 15% of creating a source node, 20% chance of deleting an existing link, and 5% of deleting an existing source node. The generator also assumes a static set of 1M nodes. We evaluate the two protocols with an input rate of 75% - 80% of their MST for the query. We run the experiments for 60 seconds and introduce a failure at the 48th second.

Regarding latency and maximum sustainable throughput, both protocols perform similarly to a checkpoint-free execution; therefore, we omit these metrics. We present the average checkpointing time, the recovery time, and the number of invalid checkpoints

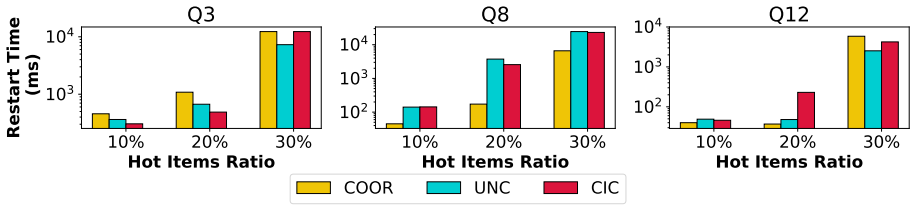


Figure 3.13: Restart time after failure per query in the presence of skew.

Table 3.4: Average checkpointing time (CT), restart time (RT), and invalid checkpoints (IC) for the cyclic query.

	Uncoordinated			Communication-induced		
#Workers	**CT**	**RT**	**IC**	**CT**	**RT**	**IC**
5	0.01 ms	620 ms	1.4%	2.73 ms	347 ms	1.7%
10	1.38 ms	344 ms	1.4%	8.39 ms	399 ms	1.6%

in table 3.4. Regarding average checkpointing time, the uncoordinated protocol is faster than the communication-induced protocol since the communication-induced protocol requires checkpointing additional protocol-related information apart from an operator's state. However, the difference between the two measurements is practically insignificant. The communication-induced protocol required less time to restart after a failure for a parallelism of 5 workers, as it forced checkpoints that led to fewer messages being prepared to be replayed. For 10 parallel workers, the uncoordinated protocol restarts slightly faster than the communication-induced protocol, although the difference is insignificant. Based on the literature and the core characteristics of both protocols, the uncoordinated protocol was expected to introduce many invalid checkpoints and lead to a domino effect. Although this might still hold in some extreme cases, our experiments show that both protocols unexpectedly share very similar percentages of invalid checkpoints for both parallelisms. Neither protocol outperforms the other when employed on top of cyclic queries in any meaningful aspect, and the uncoordinated protocol does not introduce a domino effect.

Summary. In our experiments, we explore three different cases: the NexMark queries with a uniformly distributed workload, the three more complex NexMark queries, i.e., Q3, Q8, and Q12 for a skewed input, and a cyclic query. In the first case, the coordinated approach outperforms the rest regarding latency, recovery time, and maximum sustainable throughput but has a significantly higher checkpointing time. Surprisingly, in contrast to the theoretical analysis, although parallelism and shuffling impact the checkpointing time of the coordinated protocol, they hardly affect the overall performance and only result in mild spikes in latency when a checkpoint is taken. Additionally, the uncoordinated protocol remains competitive in all queries and parallelisms. However, under skewed inputs, the uncoordinated greatly outperforms the coordinated one, which suffers both in terms of latency and checkpointing time. For the cyclic query, surprisingly, the uncoordinated does not showcase an increased number of invalid checkpoints (e.g., a domino effect) and performs slightly better than the communication-induced.

3.8 Related Work

This section presents the related work regarding *benchmarking* for stream processing systems and *experimental evaluation* of fault tolerance in stream processing.

Benchmarking for stream processing. Linear Road [18] is one of the first benchmarks proposed for stream processing that simulates a traffic monitoring application and evaluates the benchmarked solution in terms of latency, throughput, and accuracy. CityBench [14] and RioTBench [147] are real-time analytics benchmarks that employ real-world Internet of Things (IoT) data and extend the evaluation using metrics such as memory and CPU utilization and completeness of query results. SparkBench [106] is tailored to Apache Spark and targets CPU and memory utilization, network and disk I/O, job execution time, and throughput. NEXMark [163] is a widely adopted benchmark, also extended by Apache Beam [4], represents an e-commerce application, and provides streaming queries that cover all the fundamental processing workloads.

Experimental evaluation of fault tolerance. StreamBench [115] employs seven workloads on Spark and Storm and performs an evaluation focusing on throughput and latency. Qian et al. [137] evaluate fault tolerance, including additionally Samza and Kafka. However, their evaluation lacks representative workloads as they only consider a simple workload that consumes input and performs no operations.

3.9 Conclusions

In this chapter, we surveyed the three checkpoint protocol families for fault-tolerance in stream processing and discussed the theoretical advantages and drawbacks of each one of them. We developed an open-source testbed system that allows for isolated comparison of the approaches and performed a thorough experimental evaluation. While our experiments empirically confirmed the reasons behind the universal adoption of the coordinated approach, they also highlighted cases (e.g., skewed input) where the uncoordinated approach shows more robustness and better performance. Based on these results, we urge the research community to research the uncoordinated approach further since even a "vanilla" implementation of it was proven to perform well in uniformly distributed workloads, and it is the only viable solution for skewed workloads.

4

Evaluating Stream Processing Autoscalers

While the concept of large-scale stream processing is very popular nowadays, efficient dynamic allocation of resources is still an open issue in the area. The database research community has yet to evaluate different autoscaling techniques for stream processing engines under a robust benchmarking setting and evaluation framework. As a result, no conclusions can be made about the current solutions and problems that remain unsolved. Therefore, we address this issue with a principled evaluation approach.

This chapter evaluates the state-of-the-art control-based solutions in the autoscaling area with diverse, dynamic workloads, applying specific metrics. We investigate different aspects of the autoscaling problem as performance and convergence. Our experiments reveal that current control-based autoscaling techniques fail to account for generated lag cost by rescaling or underprovisioning and cannot efficiently handle practical scenarios of intensely dynamic workloads. Unexpectedly, we discovered that an autoscaling method not tailored for streaming can outperform others in certain scenarios.

This chapter is based on the following papers:

George Siachamis, Job Kanis, Wybe Koper, Kyriakos Psarakis, Marios Fragkoulis, Arie Van Deursen and Asterios Katsifodimos. 2023. Towards Evaluating Stream Processing Autoscalers. In SMDB (ICDEW). IEEE, 95-99. [149]

George Siachamis, George Christodoulou, Kyriakos Psarakis, Marios Fragkoulis, Arie Van Deursen and Asterios Katsifodimos. 2024. Evaluating Stream Processing Autoscalers. In DEBS. Association for Computing Machinery, 25–36. [148]

4.1 Introduction

A plethora of applications utilize services provided by cloud computing vendors with a variable demand for resources over time. Thus, the vendors have to prepare their platforms for a highly dynamic allocation of resources, depending on the configurations set by users. Furthermore, cloud platforms offer pay-per-use pricing models so that applications only pay for the resources they actually consume. To tackle this multi-conditional problem, the resources should be able to upscale and downscale elastically, adapting to the dynamic demand of the applications.

Most of the widely adopted stream processing engines (SPEs) were originally developed for deployment on clusters of fixed resources. These SPEs provide limited autoscaling capabilities and require substantial operational effort to adapt to changes in needs and workloads. An operations team has to always monitor the performance of the deployed system or application, estimate the required resources, decide whether to scale, and perform a manual rescaling. This process is time-consuming and offers slow reactions to workload changes with serious performance implications.

To provide automated solutions, specialized autoscalers have been developed to equip SPEs with the missing self-managing capabilities. However, it remains unclear how these autoscalers perform in different practical scenarios due to the absence of a proper comparison framework. We argue that without a principled and configurable experimental analysis, it is doubtful that these autoscalers will have the desired impact on modern stream processing engines.

Identifying the difference in resource demand is a critical point of this problem. The shift in demand can be identified by monitoring the underlying infrastructure. Autoscaling methods can vary in terms of problem modeling, heuristics, parameters, provisioning metrics, granularity, and performance [139, 114]. Furthermore, the more fine-grained the rescaling actions can be, based on the operators employed in the pipeline, the better an autoscaler will adjust the resources to the workload patterns. The most prevalent categories of autoscalers include reactive, such as *threshold-based* [81, 87], *reinforcement learning* [112, 54], *queue-based* [67, 111], *control-based* [93, 65] and proactive solutions, like *time series forecasting* [113, 21].

Contributions. In this work, we thoroughly investigate the existing control-based autoscaling solutions for SPEs and provide a concrete set of metrics, queries, and workloads to evaluate them principally. We focus on control-based solutions due to their versatility, their simplicity and the lack of training requirements. In short, the contributions presented in this chapter are the following:

- We stress the importance of extensive experimental evaluation of autoscalers for stream processing.
- We reproduce state-of-the-art autoscalers for stream processing under a common framework.
- We extend the autoscaling solutions operating on the deployment level to rescale on an operator level to ensure that the resource allocation will harmonize with the demand.

- We extend the experimental evaluation of the state-of-the-art control-based autoscaling solutions with heavily dynamic workloads, and we establish important metrics for evaluating autoscaling. We present our experimental results over diverse queries.
- We reach a series of interesting conclusions which, in our opinion, will spark additional research in the area:
  - The design choices of each operator heavily influence their performance and their ability to adhere to different objectives.
  - General-purpose autoscalers can perform better for stateless queries than the evaluated solutions specifically tailored to stream processing.
  - The evaluated autoscalers struggle with complex stateful queries under dynamic workloads.
  - The stop-and-restart state migration process of current SPEs hinders the performance of autoscalers that do not account for the lag generated during the rescaling action.

All the resources are publicly available:
`https://github.com/delftdata/espa-autoscaling.git`
`https://zenodo.org/doi/10.5281/zenodo.11652756`

Outline. In Section 4.2 we present preliminaries and necessary notation. Section 4.3 reviews autoscaling techniques and benchmarks relevant to our work. In Section 4.4, we present in detail autoscaling solutions, which we evaluate in this work. In Section 4.5, we describe the metrics, workloads, and queries used for this evaluation. Section 4.6 includes our experimental evaluation. In Sections 4.6.5 and 4.7, we discuss our key findings and limitations, highlight open challenges, and share the lessons we learned, concluding the chapter.

4.2 Background

In this section, we dive into the autoscaling process and discuss the necessary concepts to discuss the selected autoscalers.

4.2.1 Autoscaling Process

The process of autoscaling resembles the MAPE loop from control theory. As depicted in Figure 4.1, the first step includes *monitoring* of a stream processing job and acquiring all the metrics needed both for the evaluation of its performance and for the decision to perform rescaling actions. Then, the *analysis* step takes place, where we evaluate the job's current state and calculate the job's needs to adhere to the enforced agreements. The analysis outcome is then used from the *planning* step to decide on the proper rescaling actions. The goal is to satisfy the calculated needs while minimizing the resources employed. The last step is *executing* the devised plan. The monitoring API of the SPE or any applicable monitoring tool is usually responsible for retrieving the metrics, while execution usually falls on the SPE and its rescaling mechanism. The analysis and planning steps are handled from the *autoscaler*.

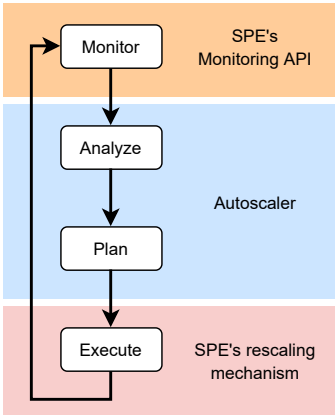


Figure 4.1: MAPE loop for stream processing autoscaling

4.2.2 Common notions

Workers & Operators. In this work, we use Apache Flink as our SPE. We choose Flink among other SPEs since it is the current state-of-the-art and the most widely adopted system in production while providing all the mechanisms expected by the autoscalers. Apache Flink refers to workers as task managers. By default, a task manager runs multiple operators that share its resources. However, we have configured Flink to isolate operators and assign a single operator to each task manager.

Back pressure. Back pressure is a rate control mechanism employed by many SPEs. When an operator cannot handle the input rate, the system uses the back pressure mechanism to regulate the output rate of the upstream operator. The backpressure can be propagated up to the source operator and the input queue.

Lag. Lag is defined as the number of unprocessed records waiting in the input queue or the operator buffers.

Elasticity. Elasticity in cloud computing is the system's ability to dynamically adjust the resource allocation to evolving workloads transparently. The system's cost is optimized by aligning resource allocation with actual demand.

4.3 Related Work

In this section, we discuss the related work on autoscalers specifically designed for stream processing and the available stream processing benchmarks.

Threshold-based. In [81], threshold-based rules are shown to boost performance when applied on individual hosts but not on the entire system. The distributed stream processing engine in [87] supports system scaling at run-time. The proposed autoscaler uses threshold-based rules to test the scaling capabilities of the system.

Reinforcement Learning. Heinze et al. [81] also propose a reinforcement learning approach that can result in high performance while minimizing the initial configuration costs. This problem is further addressed in [82], where an online parameter optimization

technique is proposed, which detects changes in the workload pattern and adapts the scaling policy accordingly. Lombardi et al. [112] propose ELYSIUM, an autoscaler that optimizes resource consumption considering the trade-off between horizontal and vertical scaling. In another work, Lombardi et al. [113] propose PASCAL, a general-purpose autoscaler based on reinforcement learning. A proactive approach forecasts incoming workloads, while a profiling system estimates the optimal provisioning. Doan et al. [54] propose a fuzzy deep reinforcement learning method for autoscaling streaming architectures. Although effective, the parameter tuning of the method is a non-trivial task. Cardellini et al. [33] propose an autoscaler for stream processing in a decentralized environment. It consists of two reinforcement-based learning approaches on a two-layered hierarchical structure for handling each operator in the system individually.

Queue-based. Lohrmann et al. [111] propose a generalized Jackson network, allowing for more precise performance estimations. The scaling decision is determined by comparing various resource allocations. Similarly, Fu et al. [67],[68] propose DRS to capture the impact of provisioned resources using a queuing-theory-based autoscaler. [21] propose an autoscaling method for distributed stream processing in geo-distributed environments. A performance model decides which geo-distributed servers need additional resources to optimize the maximal sustainable throughput of the system.

Control-based. Gedik et al. [72] proposed one of the first autoscalers specifically designed for distributed stream processing engines (SPEs) with stateful operations support. Floratou et al. [65] propose a framework to create self-regulating streaming systems using scaling policies based on the back-pressure status of the system. A self-adaptive processing graph was introduced in [85], which divides the workload of overloaded operators over multiple replicas. Using a control algorithm, the topology can be reactively and proactively scaled, improving the performance and resource efficiency of the system. Kalavri et al. [93] propose DS2, a control-based autoscaler for distributed stream processing. The authors introduce true processing and output rate notions and estimate the optimal parallelism for every operator in a single iteration. Mencagli et al. [121] propose a two-fold autoscaling method based on adaptive scheduling techniques for the short-term spikiness, while a fuzzy logic controller handles the long-term rate variability. Liu et al. [110] develop a profiling model to capture the impact of provisioned resources on the performance and scale the application accordingly. Varga et al. [165] propose two custom metrics combined with Kubernetes' out-of-the-box autoscaler HPA [3] for scaling SPEs.

Benchmarking. NEXMark [163], later extended by Beam [4], is a streaming benchmark that includes a set of analytical queries on streaming data from an online auction platform. Linear Road benchmark [19] simulates a toll system for a fictional urban area. The system monitors traffic and supports operations on live and historical data. Another benchmark on traffic sensor analytics is OSPbench [164]. SmartBench [116] focuses on querying IoT data derived from a smart building monitoring system. The benchmark performs a diverse set of temporal and spatial queries. SparkBench [107], a benchmark focused on Apache Spark, emphasizes on popular Spark applications, including machine learning, graph computation, SQL query and streaming. Analogously, ESPBench [84], use multiple types of data to test workloads of varying complexity (e.g. filtering, machine learning). DSPBench [27] covers multiple streaming scenarios with 15 different benchmark workloads. Yahoo Streaming

Benchmark (YSB) [44] uses an advertisement campaign simulation focusing on relational algebra operations, including filtering, projections and joins. StreamBench [116] generated streams from real-time web log processing and network traffic monitoring seeds. The operational workload varies in complexity and scenarios (e.g. performance, fault-tolerance).

4.4 Control-based Autoscalers

In this section, we delve into the core concepts of the selected autoscalers and how we extended some of them. For our evaluation, we select the state-of-the-art DS2[93] and Dhalion[65]; these solutions are easily deployed and widely accepted by the community. We also consider Horizontal Pod Autoscaler [3], a solution applied to a commercial product. Finally, we employ the metrics suggested by Varga et al. [165] to extend HPA towards a solution more tailored to stream processing.

4.4.1 Dhalion

Dhalion [65] is a framework that provides self-regulating capabilities to underlying stream processing systems that employ a backpressure mechanism to perform rate control. It utilizes user-defined policies to handle performance issues related to different underlying causes, such as load skew, slow instances, and provisioning. In this work, we are only interested in its proposed policy for autoscaling. The policy distinguishes two cases: an overprovisioning and an underprovisioning case.

Overprovisioning. For an operator of a running job to be considered overprovisioned, two conditions must hold: (a) there is no backpressure anywhere in the pipeline, and (b) the input queue of the operator has a length of almost zero. For each operator considered overprovisioned, new parallelism is calculated using a provided *scale down factor*.

Underprovisioning. If there is any backpressure along the pipeline, the job is considered to be in an unhealthy state and underprovisioned. To resolve the issue, the first step is to identify the operator which is the root of the backpressure. Then, a *scale up factor* is calculated for this operator based on the amount of time the job used to process the input normally and the amount of time backpressure occurred over the monitoring window. The current monitoring window is denoted as w_i. More precisely, the *scale up factor* is provided by the following formula:

$$scaleUpFactor = \frac{backpressuredTime_{w_i}}{normalProcessingTime_{w_i}} \tag{4.1}$$

As we consider Kafka as our source, we need to scale up/down Flink's KafkaSource operators. Since there is no backpressure information available for these operators, we decided to use the increase of lag noticed in Kafka as an indicator of backpressure caused by the KafkaSource operators. We denote as *pendingRecordsRate*, the average lag increase per second, and the average number of records consumed per second as *consumedRecordsRate*. Finally, the *scale up factor* is calculated as:

$$scaleUpFactor_{KS} = \frac{pendingRecordsRate_{w_i}}{consumedRecordsRate_{w_i}} \tag{4.2}$$

We gather the needed metrics using the monitoring API of Flink and Prometheus. Since Flink does not report the input queue size of each individual operator, we use the percentage of input buffers used to decide on the lag in the input queues.

4.4.2 DS2

In contrast to Dhalion, which scales each operator independently, DS2 [93] attempts to combine the scaling of all operators in a single step by leveraging the topology of the streaming query. To do so, it introduces the notions of *useful time*, *true processing rate*, and *true output rate*. *Useful Time* is the time spent by an operator in (de)serializing and processing records. *True processing rate* is the number of records an operator processes per unit of useful time, while *true output rate* is the number of records an operator outputs per unit of *useful time*. Based on these notions, DS2 calculates progressively the optimal parallelism of each operator o_i as follows:

$$OP_{o_i} = \frac{\sum \text{true output rate of upstream operators}}{avg(\text{true processing rate}) \text{ of } o_i} \tag{4.3}$$

In this work, to calculate the optimal parallelism for the KafkaSource operators, we use the rate at which records are written to Kafka as the *true output rate* of the upstream operators. In addition, we extend DS2 with a user-provided overprovisioning factor to help DS2 to handle noisy spikes and the lag accumulated due to scaling actions. This is the only tunable parameter of DS2.

4.4.3 HPA

The Horizontal Pod Autoscaler (HPA) [3] is the default autoscaling solution shipped with Kubernetes. HPA scales horizontally a deployment by adding or removing pods in order to match user-provided target values based on an observed metric. The observed metric can be either the standard average CPU/memory utilization or any custom user-defined metric, applied as shown in Equation (4.4).

$$desiredPods = \lceil currentPods \times \frac{currentMetricValue}{targetMetricValue} \rceil \tag{4.4}$$

When scaling down, HPA opts for a conservative approach. It records the scaling recommendations over a stabilization window and picks the highest recommendation as the desired amount of resources. Thus, it ensures a gradual scaledown that is not affected by fluctuations in the metric values.

HPA as a Streaming Topology Autoscaler. Since a given worker in the streaming topology runs on an individual pod, HPA can be used as a basis for building a streaming topology autoscaler that will add or remove workers when required. Although, HPA works over the workers' deployment of Flink and is agnostic of the underlying operators. Our version of HPA monitors the actual operators within a pod instead of the deployment of the workers. We employ the average CPU utilization as a metric. From now on, we will refer to this custom version of HPA as HPA-CPU. HPA-CPU has two tunable parameters: the CPU utilization target value and the length of the stabilization window.

4.4.4 HPA-Varga

Varga et al. [166] extend the HPA autoscaler to use metrics tailored for stream processing; relative lag change and utilization can be used in an ad-hoc fashion for HPA.

Utilization. Utilization provides additional system performance insights. It separates over-provisioning from optimal provisioning by analyzing the percentage of the available resources currently employed for stream processing tasks. To do so, utilization employs the idle-time-per-second metric that most modern stream processing engines provide out of the box. The utilization of the system is calculated using the following formula:

$$Utilization = 1 - avg(\text{idle time per second}) \tag{4.5}$$

Utilization can take values between 0 and 1. A value close to 1 means that the system is using the provided resources to their limits. Otherwise, the resources are underutilized. If the targeted value is close to 1, the autoscaler suggests intensive resource utilization, resulting to fewer scale-up and more aggressive scale-down actions. Although such a strategy might lower resource costs, it might also result in underprovisioning. When lowering the target value, the autoscaler issues scale-up actions more frequently. Such a setting has a higher chance of leading to overprovisioning.

Relative lag change rate. To mitigate the effects of a possible utilization's misconfiguration, Varga et al. [165] pair utilization with another metric. Relative lag change estimates the portion of the workload the system cannot handle. It uses the derivative of the system's lag and the application's input throughput recorded at the input queue. The following formula calculates the relative lag change rate:

$$RelativeLagChangeRate = 1 + \frac{deriv(\text{total lag})}{\text{input throughput}} \tag{4.6}$$

The relative-lag-change rate denotes the rate at which the lag in the input queue is increasing (> 1) or decreasing (< 1). When equal to 1, the lag is not changing, and the current resources are sufficient to handle the workload. Therefore, the relative lag change rate's target value is usually 1.0. When HPA is provided with two monitored metrics, it decides on a scaling actions based on the metrics resulting to the highest parallelism. To allow the autoscaler to scale down in cases of overprovisioning, the authors propose ignoring the relative lag change rate when the lag is below a user-provided threshold. As a result, the HPA will only consider utilization when the lag is below the threshold, allowing for scale-down actions when overprovisioning

As discussed, in Section 4.4.3, the original HPA autoscaler targets deployment-level autoscaling, which is insufficient for a stream processing engine. The extensions suggested by Varga et al. [165] operate similarly. To achieve operator-level autoscaling, we use the same procedure with HPA-CPU, monitoring the operators. While measuring the utilization is straightforward, measuring the relative lag change rate per operator is not trivial. We measure it at the input queue and propagate the result to the operator responsible based on the utilization metric and the backpressure mechanism.

4.5 Evaluation Components

We now focus on establishing a principled evaluation framework for stream processing autoscaling. First, we establish the metrics that can provide feedback on the effectiveness of the autoscaling solutions. Then, we discuss the most interesting NEXMark queries, and finally, we propose a set of dynamic workloads that enable a meaningful evaluation of the autoscalers. We conclude this section with a discussion about the evaluation contributions in contrast to the original papers of the methods we cover.

4.5.1 Performance Evaluation Metrics

Latency. Stream processing usually targets processing data and acquiring results in real-time. Therefore, the most important metric characterizing the performance of an SPE is latency [94]. Since the goal of every autoscaler is to provision just the optimal resources for an SPE to perform efficiently, the SPE's latency can also be used to evaluate the performance of an autoscaler [69, 110]. Typically, the latency is measured as the time it takes for a record to be processed and produce results from the moment it becomes available in the input queue [94]. However, this definition of latency is difficult to measure and depends significantly on the underlying query. Instead, we measure latency as the time a record stays in the input queue until the SPE processes it. We focus on the 50th and the 95th percentile of this latency.

Throughput. Throughput is also one of the primary and most important metrics used to evaluate the performance of an SPE under the current deployment condition and, therefore, the performance of an autoscaler. We define throughput as the number of records the SPE ingests and processes over a second. By comparing the SPE's throughput with the input rate of our source, we can see whether the system can keep up with the input rate and evaluate whether an autoscaler has provided enough resources to the SPE.

Resource efficiency. To estimate the autoscalers' efficiency regarding resource consumption, we consider the number of workers deployed at any time during execution. This is equal to the sum of the operators' parallelism.

Number of scaling actions. We also consider the total number of scaling actions. Depending on the system's topology, scaling the application can induce a significant overhead. This is especially the case with stateful operators and stop and restart migration mechanisms such as the one of Apache Flink, where the state needs to be persistently stored, then migrated offline and reloaded according to the new topology when the system restarts.

Convergence time & steps. Finally, we also investigate the convergence time of the autoscaler, i.e., the total amount of time it takes for the autoscaler to converge to a specific deployment for a new throughput. We also measure the total number of scaling actions required before converging to a new configuration.

4.5.2 Queries

For the evaluation of the autoscalers, we employ queries from the original NEXMark benchmark [163] and the extended version provided by the Apache Beam project [4]. NEXMark simulates an e-commerce application, mainly featuring three types of records: people, auctions, and bids. NEXMark provides a set of different streaming queries with

different properties and complexities. From these queries, we select the following subset that covers the most common types of streaming queries.

Map (*Map*). We first evaluate the autoscalers using an implementation of *Q1* of the original NEXMark benchmark [163]. We will refer to *Q1* as the *Map* query. The *Map* query transforms the values of auctions' bids between different currencies, for example, converting U.S. dollars to Euros. Thus, it performs a map over the stream of data. We choose *Q1* as a representative stateless query with a low-complexity topology and a low computational load.

Filter (*FQ*). *FQ* filters out bids that do not belong to a number of selected auctions. Similarly to *Q1*, *Q2* is also a stateless query with low computational complexity that employs a flatmap as a filter.

Incremental Join (*IJ*). *IJ* focuses on profiling user lifecycles within an online marketplace or an auction platform. It creates insights about user behavior and engagement inside the platforms. The query performs operations involving filtering and grouping. It is a stateful query with a constantly growing state and a non-linear to the input computational cost and, therefore, a complex and heavy workload-wise query. *IJ* has the highest number of operators among the selected NEXMark queries. We apply a time-to-live setting for the state's records, as a continuously expanding state is an unrealistic streaming scenario and will eventually strain our limited resources.

Sliding Window Aggregate (*SA*). *SA* targets items in the auction platform based on their popularity over a certain time period measured by the volume of bids. *SA* operates a sliding window aggregation query, i.e., it computes the total number of bids received for each auction over a sliding time window. The query needs filtering and grouping steps before performing the aggregation step. *SA* is also a computationally expensive query with a large state; however, in contrast to *IJ*, the computational cost is linear to the input rate.

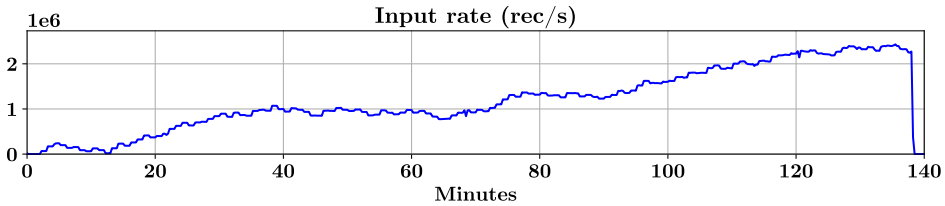
Session Window Aggregate Query (*SWA*). *SWA* focuses on calculating the number of bids a user makes in each active session. To do so, it performs a windowed aggregate (count) over a session window. Therefore, it also comprises a stateful complex computation task, like the other windowed queries.

4.5.3 Workloads

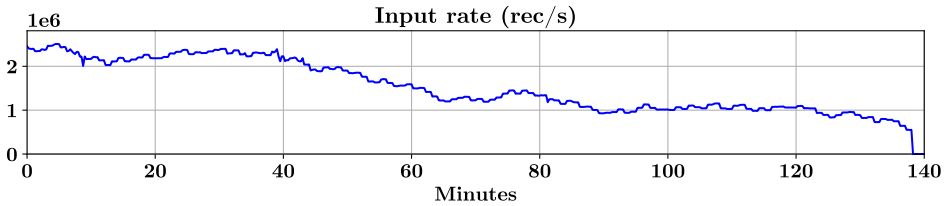
We use a scalable generator that utilizes the NEXMark's entity generators to create dynamic workloads following specified patterns. We employ five different workload patterns.

Increasing. The increasing workload pattern (figure 4.2a) starts from zero input rate and constantly increases over time. The underlying system starts from the minimum parallelism and must scale up following the increase of the input rate. The increasing workload allows for focusing only on scaling up actions and investigating how each autoscaler handles them.

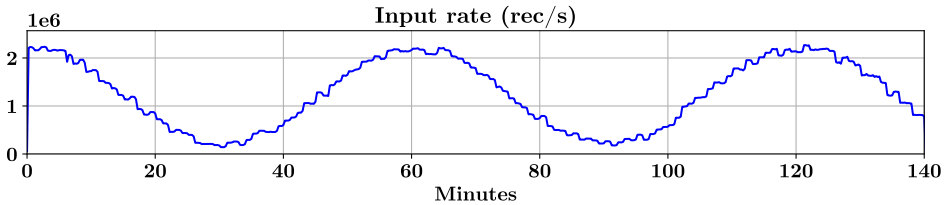
Decreasing. The decreasing workload (figure 4.2b) provides a symmetrically different scenario than the increasing workload. It starts at a maximum input rate and then constantly decreases towards zero input rate. The system starts from an appropriate for the high throughput configuration and scales down following the decrease in the input rate. Contrary



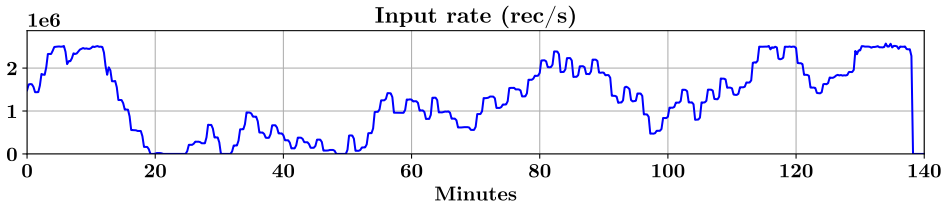
(a) Increasing



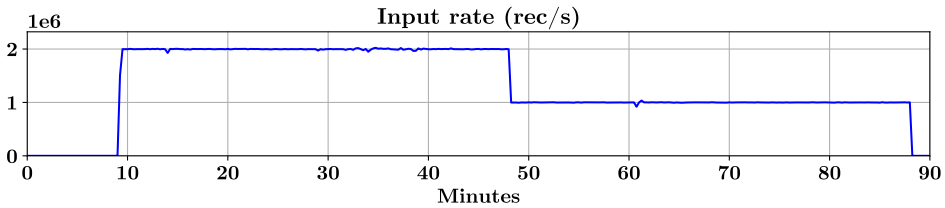
(b) Decreasing



(c) Cosine



(d) Random



(e) Steps

Figure 4.2: Workloads

to the increasing workload, the decreasing workload allows us to focus only on scaling down actions.

Cosine. The cosine workload (figure 4.2c) combines the increasing and decreasing workloads following a cosine pattern. The cosine workload allows for evaluating both the scaling up and scaling down capabilities of an autoscaler, a more complex scaling scenario. It imitates a significant subset of real-world scenarios of dynamic workloads that show some periodical or seasonal behavior.

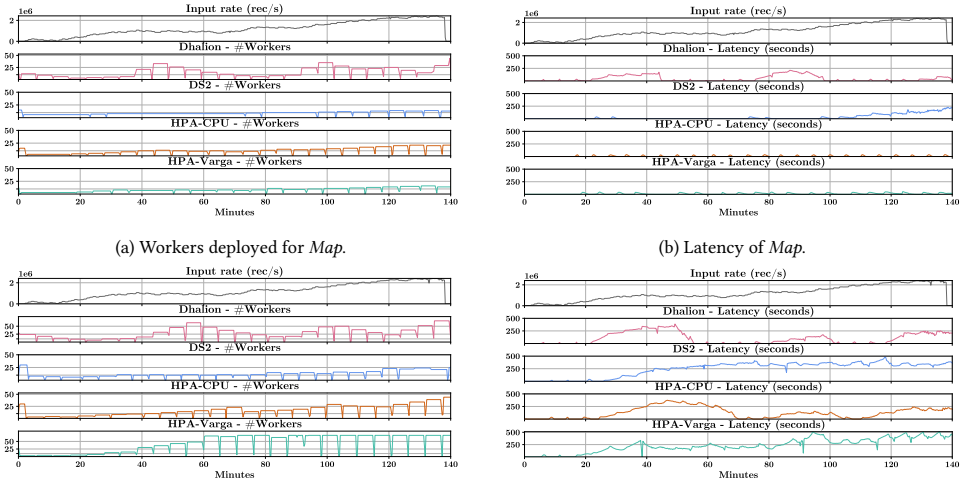
Random. Another workload that mimics a real-world scenario is the random workload (figure 4.2d). The random workload starts at a specific input rate, which is randomly increased or decreased over time. Not following a predefined pattern makes it more difficult for the autoscalers to anticipate changes in the input rate, potentially uncovering unwanted behavior programmed into the autoscalers.

Steps. Finally, the steps workload (figure 4.2e) simulates a workload that consists of fixed pulses of input rates. This workload allows for investigating the performance of autoscalers when the objective is to handle specific changes in the input rate efficiently. It resembles a real-world scenario of changes in the throughput SLAs between a provider and a client. It also enables us to investigate the time autoscalers take to converge to the optimal parallelism configuration after a change in the targeted input throughput.

4.5.4 Discussion

To the best of our knowledge, this work is the first to compare multiple autoscalers under a common framework and establish specific workloads and metrics for this evaluation. The original works presenting the evaluated autoscaling methods are limited, using metrics tailored to a specific goal, and do not include extensive comparisons with competitors under a variety of scenarios. Dhalion's evaluation includes a single wordcount query for throughput performance measurement, a convergence experiment measuring the provisioned resources, and an experiment with two input rate changes while omitting experiments with competitors. Varga HPA focuses on snapshot capturing duration and operator loading during rescaling without including experiments with competitors. DS2 performs the most comprehensive evaluation across three systems, employing six NEXMark queries and a wordcount query. However, it only compares against Dhalion, while the evaluation includes experiments with only two input rate changes and convergence experiments. DS2's primary focus lies in outlining the convergence steps required to reach a requested throughput. Finally, HPA has not been evaluated in a stream processing setting.

In this work, we establish the metrics that are relevant to autoscaling and should always be used for evaluating autoscaling solutions. Additionally, we propose four heavily dynamic workloads that constantly change their input following specific patterns. In this specific evaluation, we focus on latency performance, which none of the evaluated solutions have previously considered. Finally, we stress-test all methods in real-world conditions where data sources continuously produce items during rescaling, leading to potential lag. Notably, none of the evaluated solutions has been previously tested in this setting.



(a) Workers deployed for *Map*.

(b) Latency of *Map*.

(c) Workers deployed for *SWA*.

(d) Latency of the *SWA*.

Figure 4.3: Increasing pattern

4.6 Experimental Evaluation

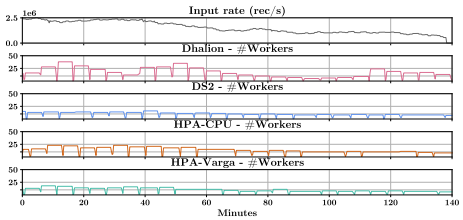
We now present our detailed experimental analysis. First, we evaluate the performance of the autoscalers with queries from the NEXMark benchmark on different workloads. Then, we demonstrate the performance of the autoscaling solutions on additional queries. Next, we compare the convergence ability of each autoscaler. Finally, we discuss the results of our experimental evaluation.

4.6.1 Experimental Setup

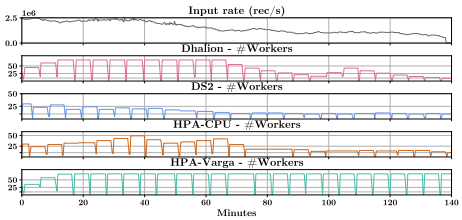
The experiments are conducted on a 3-node Kubernetes cluster with AMD EPYC 7H12 2.60GHz CPUs. On top of this Kubernetes cluster, we have configured an Apache Flink cluster in application mode. The JobManager (Flink's coordinator) instance is provided with 1 CPU and 8GB of memory, while each employed TaskManager (Flink's worker) consists of 1CPU and 4GB of memory. An NFS server is deployed as a persistence layer, Prometheus[1] is used for scraping and gathering all the metrics, and an Apache Kafka[2] deployment is used as a source for the experiments. We cap the available resources to 70 task managers, resulting in a maximum of 70 CPUs and 280GB of memory available for processing.

In our experiments, we set Dhalion's scale-down factor to 0.2, a value suggested in the original work. We use an overprovisioning factor of 0.2 for DS2, which we consider to be sufficient as the intention of DS2 is to avoid any overshooting of resources. We use the default stabilization window of 5 minutes for HPA-CPU, and we choose a target CPU utilization of 70% as the best performing among the values tested. For HPA-Varga, we also consider a CPU utilization target of 70%, the same with HPA-CPU. In addition, we employ

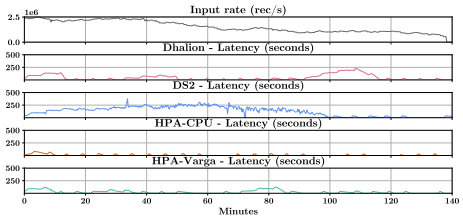
[1] `https://prometheus.io/`
[2] `https://kafka.apache.org/`



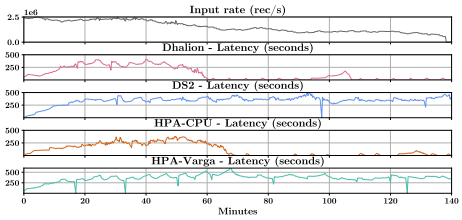
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of *SWA*.

Figure 4.4: Decreasing pattern

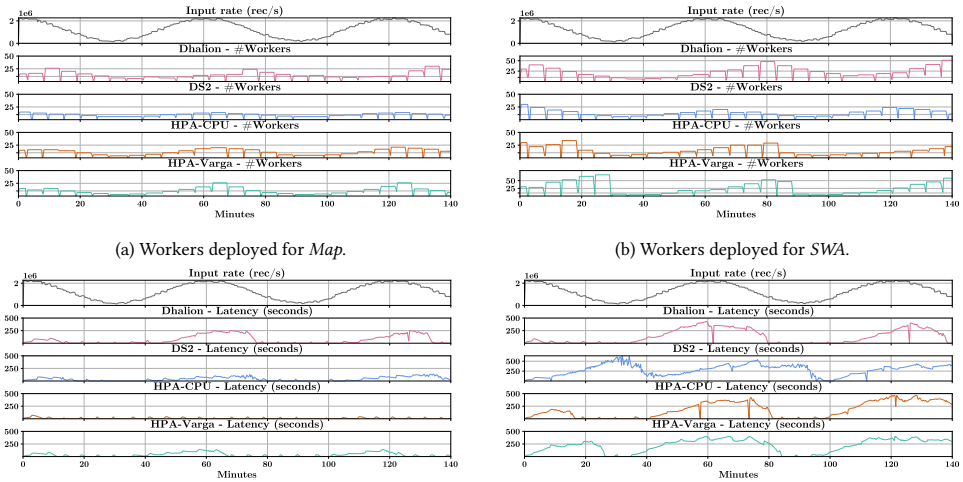
a cooldown window of 5 minutes after every scaling action to allow time for the system to reach a stable state and avoid back-to-back scaling actions due to a slow restart of the system or the lag produced by the scaling action.

4.6.2 Workload Comparison

Our first set of experiments involves four workloads from Section 4.5.3: increasing, decreasing, cosine, and random pattern workloads. The selected workloads represent heavily dynamic workload patterns whose input rate changes constantly. Due to space constraints, we limit our evaluation of the autoscalers across different workloads to two queries, the *Map* and the *Session Window Aggregate*. We choose the *Map* query as a reference query because of its simplicity and lack of state, allowing us to investigate the performance of the autoscalers on a query with low computational complexity. For such a query, performance is mostly influenced by the ability of the system to ingest and circulate the input to its operators rather than the actual computation. Alternatively, the *Session Window Aggregate* query represents a common operation in real-time analytics. Calculating time intervals for session windows makes the *Session Window Aggregate* one of the most computationally complex queries available. In contrast to the *Map* query, the *Session Window Aggregate* query employs a heavy computational task that dominates the impact on the performance of the underlying system and, thus, the employed autoscaler.

Increasing pattern. As discussed in Section 4.5.3, the increasing workload pattern allows us to study the scaling-up behavior of the autoscalers in isolation. Figure 4.3 showcases the performance of the autoscalers on this workload in terms of the number of deployed workers and the resulting latency throughout the execution of the two deployed queries.

– *Performance on Map.* Figure 4.3a present the number of deployed workers per minute during the *Map* query execution. We observe that the number of workers Dhalion recom-



(a) Workers deployed for *Map*.

(b) Workers deployed for *SWA*.

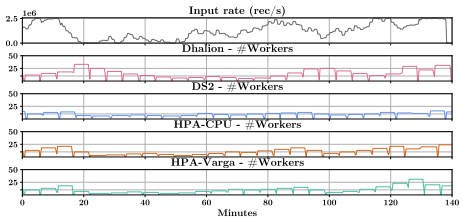
(c) Latency of *Map*.

(d) Latency of *SWA*.

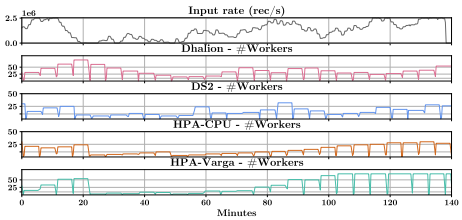
Figure 4.5: Cosine pattern

mends does not follow the input pattern. Although the input rate constantly increases, Dhalion has large intervals of decreasing resources. At first, Dhalion reacts slowly to the increasing rate; the backpressure keeps increasing, translating to increasing latency (Figure 4.3b), which triggers an aggressive scale-up. Evidently, Dhalion fails to suggest the right resources leading to overprovisioning and a large scale-down interval that does not match the expected behavior. DS2 avoids unnecessary rescaling actions while providing the minimum required resources for low latency when the input remains relatively low. When the input increases significantly, every scaling action generates progressively more lag, and DS2 rescales more frequently. This causes a constant increase in latency due to DS2 failing to accommodate the generated lag. The HPA-based solutions assign workers to follow the input pattern smoothly. Both autoscalers keep latency low and perform more frequent rescaling actions than DS2. However, HPA-Varga constantly suggests a lower number of workers without compromising performance. Overall, HPA-based autoscalers match the input pattern and perform better in latency. However, they assign a slightly higher number of resources than DS2.

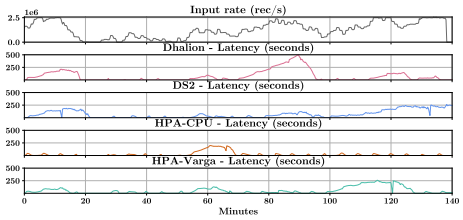
– *Performance on* SWA. We observe in Figure 4.3c that the methods follow a similar trend. Dhalion shows the same behavior as in the *Map* query; it fails to react on time to the input increase, then aggressively overprovisions resources, leading to large scale-down intervals. Contrary to the *Map* query, DS2 performs frequent rescaling actions even while the input remains low. The latency (figure 4.3d) starts to increase early on, and DS2 never manages to recover. However, DS2 assigns the lowest number of workers throughout the experiment. HPA-Varga follows the input pattern but aggressively issues scale-up actions, reaching the maximum available resources early. Despite the many resources, it fails to ensure low latency. HPA-CPU matches the input pattern, steadily providing more resources. However, it still fails to retain low latency.



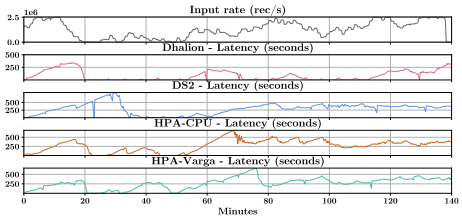
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of *SWA*.

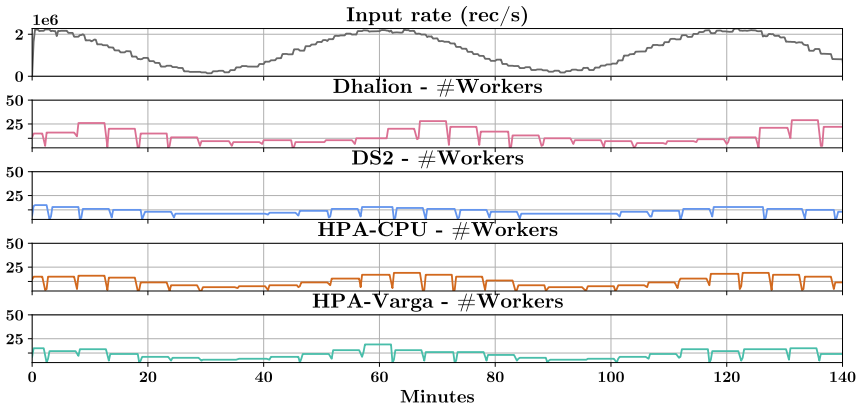
Figure 4.6: Random pattern

Decreasing pattern. In contrast to the increasing pattern, the decreasing workload employs a constantly decreasing input rate. This experiment demonstrates the scale-down performance of the autoscalers in isolation. Figure 4.4 illustrates the behavior of the autoscalers in terms of workers deployed and latency.

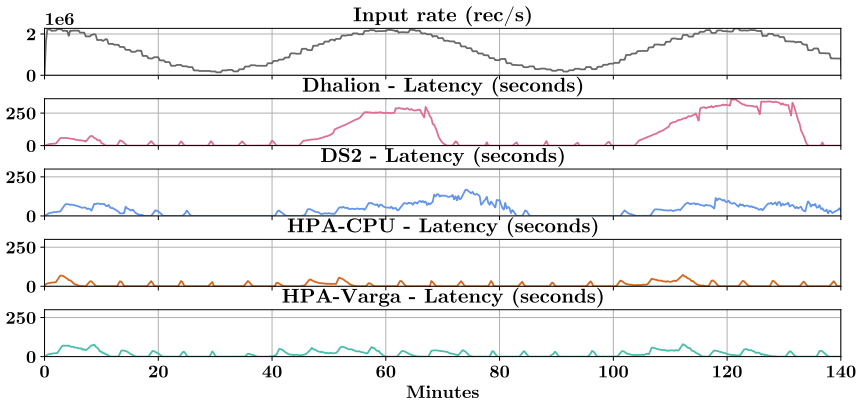
– *Performance on Map.* Similar to the increasing workload, Dhalion does not react to input on time, resulting in fluctuating behavior. DS2 again keeps the resources low but fails to reduce the latency before the input decreases sufficiently. HPA-CPU follows the decreasing pattern and retains low latency throughout the execution while issuing slightly fewer rescaling actions than the rest. HPA-Varga also keeps latency relatively low while recommending fewer resources than HPA-CPU.

– *Performance on* SWA. The *SWA* query presents a challenge to all methods in the decreasing pattern. Dhalion immediately considers the initial parallelism inadequate and quickly scales the system to the maximum available resources. Despite utilizing all the available workers, Dhalion fails to handle the high latency early on. It only manages to reduce the latency after the input rate has been decreased significantly, although it issues an unexpected upscale. HPA-CPU achieves the same performance in terms of latency. However, it assigns significantly fewer resources throughout the execution. DS2 and HPA-Varga cannot retain low latency throughout the entire run. DS2's provisioning adheres to the input pattern, while HPA-Varga quickly maximizes the resources, remaining at the highest parallelism for the entire run.

Cosine pattern. The cosine pattern combines increasing and decreasing input behavior and composes a representative real-world workload that requires varied scaling actions. It is an interesting experiment that allows us to evaluate the autoscalers on an explainable, highly dynamic workload. We illustrate the performance of the autoscalers in Figure 4.5.



(a) Workers deployed



(b) Latency

Figure 4.7: *Filter.*

– *Performance on* Map. Contrary to its behavior on the increasing and decreasing patterns, Dhalion's resource allocation follows the input with a small delay. This delay causes a latency increase during periods of high input rate. Dhalion manages to recover during periods of lower input rate. Similarly, DS2 suffers from a small latency increase only during periods of high throughput. DS2 keeps providing fewer resources throughout the experiment while following the input pattern. This is a consequence of failing to handle the generated lag arising from rescaling. The same latency behavior is observed for HPA-Varga, resulting from a slow scale-up of workers. In contrast, HPA-CPU maintains low latency while adjusting resources on time to keep up with the input.

– *Performance on* SWA. Similar to *Map* query, Dhalion does not react in time to the input changes, resulting in a latency increase. DS2 follows the input accurately. However, it suffers from high latency since it doesn't recover even during low input periods. Both HPA-based autoscalers have high latency for high input periods as they react late to the

input changes. However, HPA-CPU achieves the same performance with fewer deployed workers.

Random pattern. The random workload pattern is the most complex and another representative real-world challenging pattern. It resembles real-world traffic with sudden spikes and irregular input changes, making the scaling actions varying and less obvious.

– *Performance on Map.* Unsurprisingly, Dhalion fails to match resources with the input pattern. It reacts slowly to the random pattern's sudden input changes, leading to large periods of high latency. Despite the randomness of the pattern, DS2 manages to assign resources according to the input pattern. However, in terms of latency, it fails to adapt during prolonged periods of high input rates. HPA-Varga and HPA-CPU adapt to the input pattern for the majority of the time. However, HPA-Varga suffers from sudden increases in input rate, leading to temporary higher latency. HPA-CPU maintains low latency for most of the experiment, except for a high latency period at a sudden increase in input rate.

– *Performance on SWA.* Only DS2 manages to follow the progression of the input pattern for the *SWA* query. However, latency remains high for all periods of medium to high input rates. Both HPA-based autoscalers fail to match the input at any moment, and after a while, they constantly scale up, trying to deal with the lag accumulated in the input queue. Although Dhalion does not keep up with the changes in the input rate, it has the best overall performance in terms of latency and the longest periods of low latency.

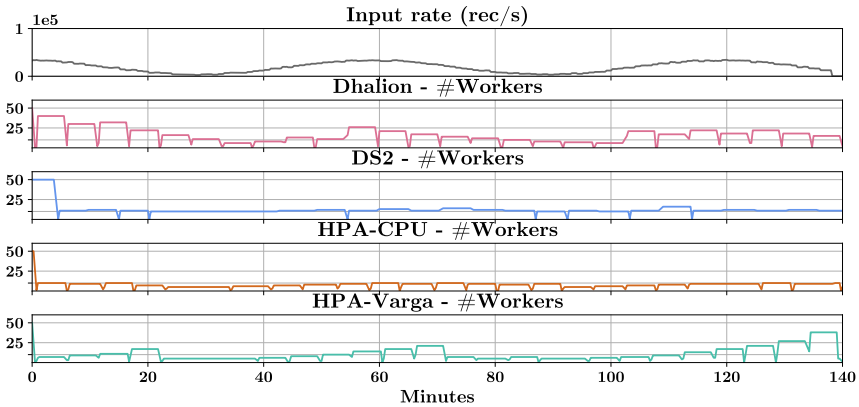
4.6.3 Query Comparison

Although the *Map* and the *Session Window Aggregate* queries are representative processing tasks, we evaluate the autoscalers on additional queries for completeness. Due to space limitations, we only present the performance of additional queries deployed on the cosine workload. Figures 4.7, 4.8 & 4.9 illustrate the performance of the autoscalers on the additional queries.

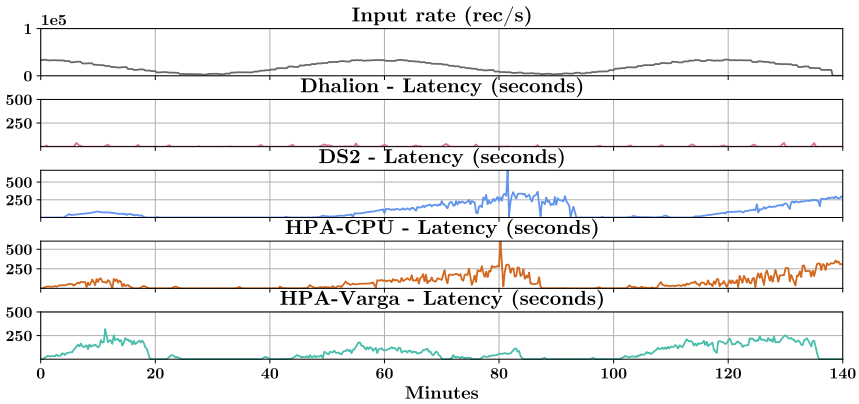
Performance on Filter. The *Filter* query belongs to the same class of stateless low-complexity computation tasks as the *Map* query. As so, we expect similar behavior from the autoscalers. Indeed, DS2 and HPA-CPU show the same behavior as in *Map*. During peak periods of activity, Dhalion follows the input with a slight delay, resulting in high latency; similar to Dhalion's behavior in *Map*. HPA-Varga has a similar resource provisioning as in *Map* but performs slightly better in terms of latency because of better resource allocation.

Performance on Incremental Join. *Incremental Join* is the most complex query employed. This gives us an opportunity to evaluate the autoscalers on a system under stress, even during low input rates. Dhalion is the only autoscaler that matches the input rate and achieves low latency for the whole experiment duration. However, it employs twice as many resources as DS2 or HPA-CPU. DS2 and HPA-CPU minimize the scaling actions and the deployed resources but also see latency increases. HPA-Varga reacts slowly to the input increases, reflecting high latency during periods of high input rate.

Performance on Sliding Windowed Aggregate. The *Sliding Windowed Aggregate* (SA) differentiates from *SWA* in the type of window employed. While a session window produces a constant flow of records throughout the system, a sliding window produces output only



(a) Workers deployed



(b) Latency

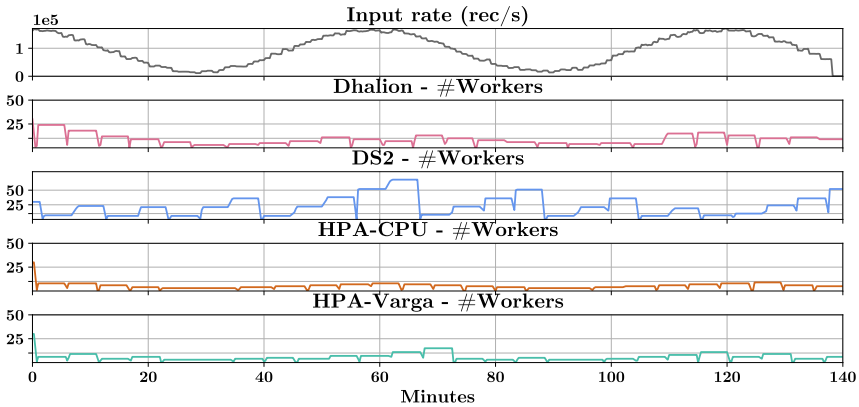
Figure 4.8: *Incremental Join*

when a time interval ends.

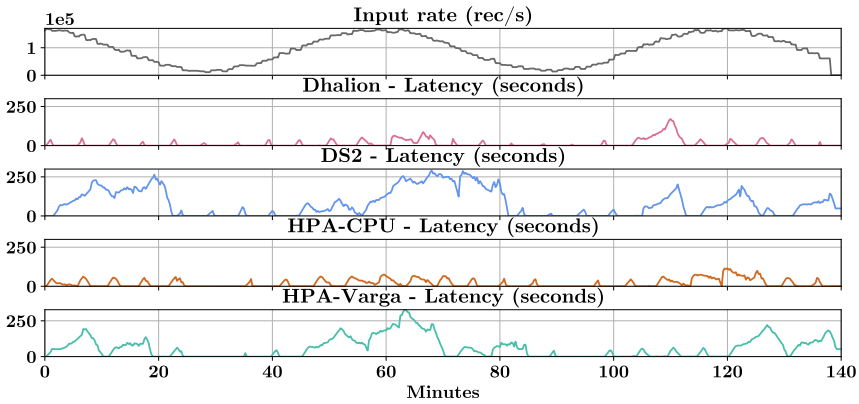
For *SA*, HPA-CPU performs the best as it follows the input pattern and keeps latency low while provisioning a minimal number of workers for the entire run. DS2 cannot align with the input pattern as it perceives dead periods as underprovisioning. Thus, it decides that the system underperforms and falsely raises the resources. Dhalion has competitive performance with delayed scale-up decisions that lead to small latency spikes when the input rate increases. At the same time, on average, it deploys more workers than HPA-CPU. HPA-Varga provisions resources similarly to HPA-CPU but fails to allocate them correctly, impacting the latency.

4.6.4 Convergence comparison

We perform a convergence experiment to evaluate the ability of the autoscalers to converge to an optimal configuration, given a specific input rate. We use the steps workload discussed



(a) Workers deployed

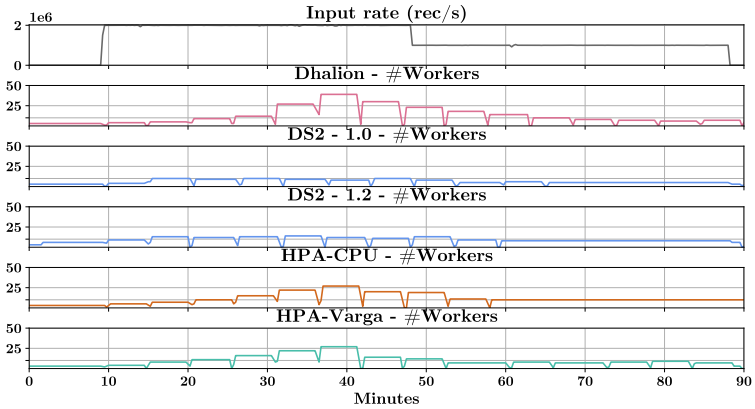


(b) Latency

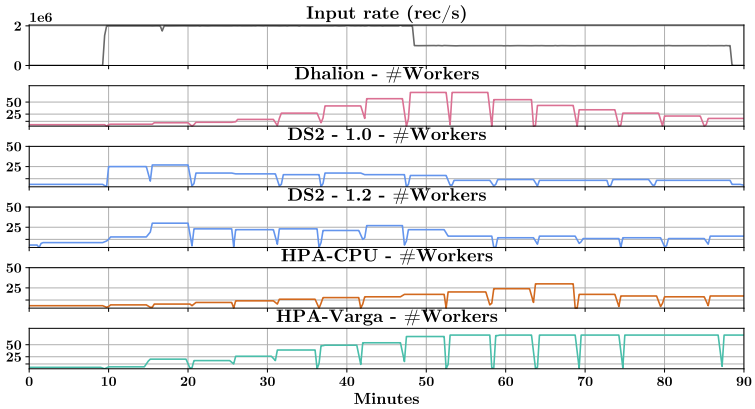
Figure 4.9: *Sliding Windowed Aggregate.*

in Section 4.5.3 and the *Map* and *SWA* queries. The experiment assesses the time and scaling actions required to converge to an optimal configuration.

Figures 4.10a and 4.10b show the deployment of workers over time. Dhalion shows a slow reaction to input changes. As a result, it fails to converge within the provided time frame to any of the two input rates. HPA-Varga reacts slowly to the increased input rate and does not converge regardless of the input rate and the query. Surprisingly HPA-CPU is also slow to react to the high input change for both queries. However, it converges for the medium input rate within two scaling actions for the *Map* query. We deploy DS2 with and without overprovisioning. Both versions react quickly to input changes but only manage to converge for the medium input rate of the *Map* query within two and three scaling actions, respectively. Although DS2, without overprovisioning, temporarily decides on a stable configuration, it continues oscillating after a while.



(a) *Map*



(b) *Session Window Aggregate*

Figure 4.10: Convergence

4.6.5 Summary of findings

In our experiments with different workloads, we observed varying behavior among autoscalers. DS2 consistently follows the input pattern but may encounter occasional high latency. Dhalion struggles to adapt to less complex patterns, reacts slowly to more complex patterns, and allows for high latency. HPA-Varga generally aligns with the input patterns but may react slowly to input rate increase. HPA-CPU outperforms all autoscalers for less complex queries in terms of latency, adjustment, and resource utilization. However, HPA-CPU fails to sustain low latency when facing high-input periods for more demanding queries.

Experiments with additional queries support our earlier observations. HPA-based autoscalers perform well for stateless queries. HPA-CPU matches the input pattern while maintaining low latency. HPA-CPU also performs adequately for the *sliding windowed aggregate* query but fails to provide enough resources in the case of the *Incremental*

Join query. HPA-Varga performs worse on complex queries, both in terms of latency and deployed workers. Dhalion reacts slowly to input changes and allocates resources inefficiently, leading to high latency. DS2 allocates fewer resources and avoids unnecessary scaling actions. However, it struggles to maintain low latency in high throughput periods, especially for complex queries. Finally, our convergence experiments show that none of the evaluated autoscalers can converge within the time limits of our experiments for complex queries. Only HPA-CPU and DS2 converge when scaling down from a higher load to a medium input rate.

The design choices of each autoscaler reflect on its performance. DS2 adjusts to the input rate accurately and fast due to effective metrics and efficient scaling of multiple operators at once, by propagating changes to downstream operators. However, DS2 does not consider the lag generated when it issues scaling actions, leading to high latency. Dhalion relies on backpressure and input buffer usage to decide on scaling actions. However, when backpressure can be detected, the system has already entered an unhealthy state. Additionally, Dhalion only scales a single operator in each scaling action, reacting slowly to changes. Dhalion fails to distribute efficiently resources to the operators resulting to unstable performance and slow convergence. HPA-CPU solely depends on CPU load, which may not accurately reflect the performance impact of complex stateful queries that involve accessing large datasets from memory or disk. Despite our best efforts, we could not overcome that HPA-Varga is designed to work on a deployment level rather than on an operator level. Its utilization metric can be directly measured per operator, while the relative lag can be calculated only indirectly.

As seen in our experiments, the evaluated autoscalers are affected by the lag generated during the rescaling actions, and none can currently consider it when deciding on the optimal configuration. This generated lag is partially a side effect of current systems' inability to migrate their state without a stop-and-restart process. Although the problem of state migration is orthogonal to autoscaling, it plays a crucial role in the performance. Despite prior work introducing proposals for on-the-fly state migration techniques with low overhead, stream processing systems have yet to adopt it.

Previous evaluations. The original evaluation of Dhalion shows a necessity for numerous rescaling actions and considerable time to achieve desired throughput convergence. Our experiments validate this observation, as the time frame are insufficient for Dhalion to reach convergence. In our work, we observe that Dhalion deploys more resources than the other autoscalers. The findings from the DS2 evaluation show lower resource deployments and faster convergence with fewer actions than Dhalion. Unlike the original evaluation, DS2 does not always converge within the time frame set in our experiments. HPA-Varga and HPA-CPU do not conduct an evaluation using the same metrics or offer similar insights.

Limitations. In this work, we propose a principled evaluation framework for evaluating control-based autoscalers. Evaluating additional autoscalers under this framework can provide rich insights, as we showcase with our experiments, and might lead to different conclusions. Furthermore, we evaluate the performance of the autoscalers on top of Apache Flink. Our evaluation framework assumes durable input and output queues and a stream processing engine that allows for per-operator scaling. The evaluated autoscalers are agnostic to the specifics of the underlying rescaling mechanisms of an SPE and only require specific metrics provided by the engine as well as a rate control mechanism in the case of

Dhalion. However, extending the current evaluation to other stream processing engines, such as Storm[3] and Heron [102], can provide valuable insights regarding the autoscalers' applicability and the configurations' performance based on the rescaling mechanisms.

4.7 Conclusion

In this work, we highlighted the lack of significant comparison between existing autoscaling solutions in stream processing. We provided a principled experimental framework to evaluate performance and identify unsolved challenges. We extensively evaluated four control-based autoscalers on dynamic workloads and queries. Surprisingly, a method utilizing CPU usage outperforms state-of-the-art solutions for minimal queries in all workloads. We showcased that none of the evaluated autoscalers can perform well for complex queries over highly dynamic workloads. We discuss the impact of the autoscalers' design choices on their performance, and we argue that the poor performance of the evaluated autoscalers is a result of their inability to account for the lag generated during an autoscaling action or due to slow reactions to the input changes. Finally, we urge stream processing engines to adopt online state migration techniques as it would significantly improve the performance of autoscaling.

[3] https://storm.apache.org

5 Conclusion

In this thesis, we studied the ability of streaming dataflow engines to adapt to unforeseen events related to their processing workload or the underlying infrastructure. In the era of *cloud computing*, where flexible resource allocation and flexible pricing schemes are provided, stream processing engines must be able to automatically adapt to occurring changes in the workload or the underlying infrastructure in order to leverage the provided capabilities and facilitate the prosperity of small and medium-sized businesses. We focused on three main adaptivity problems for streaming dataflow engines: adaptivity to *statistical workload changes* (Chapter 2), *infrastructure failures* (Chapter 3), and *input rate changes* (Chapter 4). We investigated adaptivity to *statistical workload changes* through streaming similarity joins, an important data processing task heavily influenced by statistical changes and imbalanced workloads.

Several solutions have been proposed to enhance the stream processing engines with the desired adaptive capabilities. With respect to *streaming similarity joins*, existing solutions targeted single-node executions, load balancing without reducing the number of unnecessary computations, or specific subproblems, such as similarity joins on sets of words. No efficient solution was proposed that is generalizable to different types of data and similarity metrics. In this thesis, we proposed a novel adaptive distributed streaming similarity joins method that targets the general metrics space (Chapter 2). In terms of adaptivity to *infrastructure failures*, most modern stream processing engines have adopted a coordinated checkpoint-based fault tolerance mechanism. However, there was no empirical or experimental evidence that supports the superiority of the preferred coordinated checkpointing protocol. Therefore, it remained unclear how the preferred coordinated approach performs under executions of different scales or input distributions and if the existing alternatives are better options under certain conditions. This lack of experimental evaluation hindered further research on the important topic of fault tolerance for stream processing. In this thesis, we tackled this issue with Checkmate, a principled evaluation framework for checkpointing protocols for streaming Dataflows (Chapter 3). Finally, regarding adaptivity to *input rate changes*, many autoscaling solutions that target stream processing have been proposed. However, the experimental evaluation of these solutions is shallow and usually does not include detailed comparisons with other autoscalers. Therefore, it was unclear

what the state of the field is, if the existing solutions could cover the needs of stream processing effectively, and how a stream processing autoscaler should be evaluated. In Chapter 4, we proposed a principled evaluation framework for stream processing autoscaling and compared the state-of-the-art control-based autoscalers extensively.

In this chapter, we conclude the thesis and summarize our findings and insights from our quest to answer our main research questions and tackle the discussed research gaps. We then discuss the limitations of our work and present possible future research directions based on our insights.

5.1 Main Findings

In this section, we discuss our findings and our insights when addressing each of the targeted adaptivity problems.

5.1.1 Adaptivity to statistical changes for streaming similarity joins

Statistical changes are very frequent when dealing with high-velocity data streams. Their impact can be detrimental to the performance of certain tasks. In Chapter 2, we investigated the adaptivity of streaming similarity joins, a task that is heavily affected by load imbalance due to statistical changes in the input load. We drove our research based on the following main research question:

> **M-RQ1:** How can we perform streaming similarity joins on multidimensional streams in a distributed fashion, even when distribution changes, achieving low latency?

In order to address **M-RQ1**, we studied the problem of distributed streaming similarity joins in the general metric space. We conducted a thorough investigation of existing related work, including not only streaming solutions but also all relevant solutions from the MapReduce paradigm. We revisited the Inner-Outer partitioning paradigm (section 2.2) and discussed its main concepts. Driven by the research gaps identified in the related work, we proposed S^3J; the first adaptive distributed streaming similarity joins solution in the general metric space that employs an adaptive partitioning scheme influenced by the Inner-Outer partitioning paradigm and a load balancing scheme that leverages the partitioning scheme to provide low-cost online adaptivity to distribution changes. More specifically, the partitioning scheme consists of two operations. First, the input records are divided among the available workers into space partitions following the Inner-Outer paradigm with random partition centroids. Then, within each space partition, records are partitioned into fine-grained partitions, called worksets, that allow for reducing unnecessary computations. Each workset consists of an inner, an outer, and an outlier subset. Similarity computations are performed only between records that belong to the same workset. The load balancing scheme uses the existing worksets to adapt to occurring load imbalances due to distribution changes by redistributing the existing worksets to workers by using a workset balancing algorithm and without reconstructing the costly fine-grained partitions.

We evaluated S^3J against the ClusterJoin-based baseline under queries of varying selectivities and varying operator parallelism [48]. Our experimental evaluation shows that S^3J's adaptive partitioning scheme outperforms the baseline in all scenarios. More specifically, S^3J maintains sub-second latency for low selectivities, even when the parallelism of the task

is low. For high selectivities, S^3J's partitioning scheme results in higher processing throughput than the baseline, even for unsustainable input rates. Our experiments with varying parallelism show that S^3J can scale efficiently by leveraging the additional resources better than the baseline. Regarding record duplication, S^3J outperforms the baseline under both scenarios. Finally, S^3J's adaptive partitioning manages to reduce effectively the unnecessary computations performed by leveraging its worksets. Finally, our load balancing experiment shows that the load balancing scheme can gradually reduce the latency observed and redistribute the load to the workers adequately.

As our results show, S^3J is an efficient adaptive streaming similarity joins solution that can handle distribution changes while maintaining low latency. S^3J's adaptive partitioning can follow the distribution changes efficiently and does not require any prior knowledge of the workload. Its load balancing scheme allows for fast reconfigurations that can gradually reduce the observed latency and balance the load among the workers. We argue that an effective solution to **M-RQ1** must provide, first and foremost, an adaptive partitioning scheme alongside an efficient load balancing mechanism.

5.1.2 Recovering from infrastructure failures using checkpoints

Infrastructure failures are common in distributed systems. Reacting effectively to such a failure is crucial for long running streaming analytics. Most modern streaming dataflow engines employ a checkpoint-based fault tolerance mechanism. However, there was no empirical or experimental evidence on the performance of the different checkpointing protocols. In Chapter 3, we addressed this research gap by focusing on the following main research question:

> **M-RQ2:** How do checkpointing protocols for streaming dataflows perform in different workloads and input data distributions?

We investigated **M-RQ2** by proposing CheckMate. CheckMate is an open-source principled evaluation framework for checkpointing protocols. In CheckMate, we summarized all the necessary preliminaries a researcher or a practitioner needs to explore checkpoint-based fault tolerance. We discussed in detail the three main protocol families and performed a theoretical account of the advantages and drawbacks of each protocol. As a principled evaluation framework, CheckMate establishes the most important and relevant metrics, queries, and distribution scenarios needed to measure effectively the performance of a checkpointing protocol. It provides a common open-source testbed system that allows for an isolated and comparable evaluation of the checkpointing protocols. Finally, CheckMate implements the vanilla versions of each checkpoint protocol and evaluates them under the proposed scenarios, leading to interesting and insightful experimental results.

CheckMate's experimental evaluation sheds much-needed light on the performance of the different checkpointing protocols. First of all, it attests to the strong preference of most modern streaming dataflow engines towards coordinated checkpoints. As our results indicate, coordinated checkpoints are indeed the best-performing choice for uniformly distributed workloads in the absence of straggling nodes. However, the uncoordinated checkpointing protocol is a competitive choice for uniform workloads as it results in slightly worse performance despite the expensive message logging that it employs. In the presence of skew in the input data, CheckMate shows that uncoordinated checkpoints

are more robust and outperform the alternatives. Coordinated checkpoints suffer from high checkpointing time as the percentage of skew increases. Straggling nodes slow down processing due to coordination, resulting in high observed latency. Finally, the uncoordinated and communication-induced approaches are further evaluated on a cyclic graph reachability query. In theory, the uncoordinated checkpointing protocol can lead to a domino effect when employed on cyclic queries. However, no domino effect is observed in our experiments. Therefore, the uncoordinated protocol is the better choice for cyclic queries as it slightly outperforms the communication-induced approach.

In summary, CheckMate answers **M-RQ2** effectively. It provides the theoretical background of checkpoint-based fault tolerance and a principled evaluation framework, accompanied by an open-source testbed system. Its experimental evaluation provides a detailed comparison of the checkpointing protocols that will inspire further research on checkpoint-based fault tolerance.

5.1.3 Adapting to input rate changes using automated solutions

To fully leverage the features offered by cloud providers, stream processing engines rely on autoscaling techniques that leverage the metrics provided by the engine to adapt the deployed resources to the observed input rate. Although multiple solutions have been proposed, it remained unclear how these solutions perform and what open problems still exist. In Chapter 4, we acknowledged this research gap that we summarise in the following main research question:

> **M-RQ3:** How well can existing stream processing autoscalers perform? How can we effectively evaluate them, and what are their inefficiencies?

In Chapter 4, we addressed **M-RQ3** through a detailed experimental evaluation of stream processing autoscalers. We first explored the landscape of autoscaling solutions that target stream processing applications. We then focused on control-based autoscalers due to their simplicity and applicability to many different topologies. We discussed in detail the state-of-the-art control-based autoscalers, DS2 [93] and Dhalion [65], a commercial generic autoscaler from Kubernetes called HPA [3] and an extension of HPA with metrics tailored for stream processing which we call HPA-Varga. Since HPA operates on a deployment level, we adapt HPA and HPA-Varga to achieve per-operator scaling instead. Then, we established the relevant metrics from the literature that are important for evaluating the performance of stream processing autoscalers. We employed different representative queries from NexMark [163, 4], and we proposed five heavily dynamic workload patterns that allow for a thorough evaluation. Finally, we implemented the discussed autoscalers on top of Apache Flink and performed an extensive evaluation based on the proposed framework.

Our experimental evaluation validates some of the results of the original works. DS2 maintains a better performance than Dhalion in most evaluated scenarios, both in terms of latency and deployed resources. However, our evaluation also leads to surprising results. Unexpectedly, none of the autoscalers can adjust the resource efficiently under heavily dynamic workload patterns when a complex query is employed. At the same time, although HPA is a generic autoscaler not tailored for stream processing, it outperforms the state-of-the-art control-based autoscalers when dealing with simple stateless queries. Our results show that the design choices and the employed metrics of each autoscaler play a crucial

role in the performance. Dhalion depends on the backpressure mechanism of the stream processing engine to decide on scaling actions. As a result, Dhalion is slow to react to workload changes since the system is already in an unhealthy state when backpressure is observed. HPA and HPA-Varga decide on scaling actions based on CPU utilization, which fails to capture the complex non-linear queries. DS2 can accurately calculate the required resources for the input rate observed at the time. However, it does not take into account the lag generated due to a scaling action. Generally, the performance of all the evaluated autoscalers is heavily affected by the underlying stop-and-restart state migration mechanism and the lag it generates after rescaling actions.

Our experimental framework provides concrete guidelines and establishes the required resources to evaluate effectively the performance of autoscaling. It answers adequately **M-RQ3**, providing thorough details on the performance of the existing control-based autoscaling solutions.

5.2 Limitations

Despite our best efforts and the thorough design of the research conducted in this thesis, some limitations remain that are worth mentioning.

In Chapter 2, most of our experiments employed synthetic datasets of only two dimensions created by a custom generator following a specified distribution. The proposed solution can handle any kind of data, including data of higher dimensionality. Therefore, our evaluation may be limited with respect to the capabilities of the solution. At the same time, the lack of high-quality, real-world datasets does not allow us to evaluate our solution under real-world conditions. Moreover, we limited our evaluation to a single baseline. Although no other solutions can handle streaming similarity joins in the general metric space, our evaluation could benefit from comparing our solution to available solutions on specific sub-problems. Such an evaluation would reveal the trade-off between a generalizable solution and a solution tailored to a specific subproblem.

In Chapter 3, CheckMate proposes a principled framework for evaluating checkpointing protocols for streaming dataflows. However, it still largely performs its evaluation on synthetic data produced by the NexMark benchmark, which was designed and developed more than a decade ago. Although the results and the insights of CheckMate are reliable and insightful, including representative real-world workloads in the evaluation would provide useful insights into the performance of the protocols on modern, real-world use cases. Additionally, CheckMate implements the vanilla versions of the three main protocol families. It is, therefore, intentionally limited to evaluating the effectiveness of their main concepts. We did not consider any optimization tailored to the specifics of any commercial system.

In Chapter 4, we proposed a principled evaluation framework, and we compared the state-of-the-art control-based stream processing autoscalers. However, we limited our scope to control-based autoscalers, and we omitted the existing literature proposing other types of solutions. Although our results on the performance of the control-based autoscalers are reliable and insightful, they only provide insights for a specific subset of the existing autoscaling landscape. Moreover, for our evaluation, we employed an official Apache Flink distribution that did not provide any online state migration mechanism. Therefore, in order to perform a scaling action, we used the stop-and-restart mechanism, which stops

the execution temporarily, takes a savepoint, and restarts Flink with the new operator parallelism. However, this rescaling mechanism introduces lad that significantly affects the performance of the autoscalers.

5.3 Future Research Directions

In this section we identify open problems and discuss future research directions in the field of stream processing that derive from our insights during the design of our research work and the implementation of the frameworks and the methods included.

5.3.1 Evaluating MapReduce solutions in a streaming environment

As we mention in Chapter 2, there are currently very few works that target similarity joins in the stream processing paradigm. To the best of our knowledge, our proposed solution is the first to address the problem in the general metric space, while the solution proposed in [174] is the only other solution that combines load balancing with the reduction of unnecessary similarity computations in a distributed stream processing environment. However, there is a significant body of work that deals with similarity joins on a MapReduce environment [48, 171, 167, 123]. Although these works are not directly applicable to the stream processing paradigm, they can be of great inspiration and a good starting point moving forward in the topic of streaming similarity joins. These solutions have been extensively evaluated and compared in the MapReduce environment [64], but their concepts have never been applied and evaluated in a streaming environment. Thus, it remains unclear which of these solutions can be adapted for streaming applications, how well they would perform in such a scenario, and what could be the optimization that would enhance their performance in the new setting. Therefore, we argue that a detailed and extensive evaluation of these solutions in the stream processing paradigm, which also discusses the hurdles of adapting these solutions to the new environment, is necessary, and it will certainly inspire more research on the topic of streaming similarity joins.

5.3.2 Learned Partitioning for Streaming Similarity Joins

In Chapter 2, we proposed a solution for streaming similarity joins in the general metric space that leverages an adaptive partitioning scheme to reduce the number of unnecessary similarity computations without losing any potential joins. From our experience, designing and implementing such a generic partitioning scheme is difficult and may be outperformed by alternative approaches tailored to the specific subproblem. In the last decade, machine learning approaches have revolutionized the way of thinking and addressing particular data management tasks [100, 57, 120]. Lately, machine learning approaches have also been proposed for partitioning cloud databases [86], but also for adaptively partitioning stream processing tasks [176]. Dalton [176] uses reinforcement learning to learn the optimal partitioning for a specific data stream and a specific query and quickly adapt this partitioning scheme on the fly. Dalton is oblivious to the specifics of the streaming similarity joins task, and therefore, it does not optimize for reducing the unnecessary similarity computations, and it cannot ensure the completeness of the result. However, it has shown great results and inspires further research. We argue that an interesting future direction involves investigating different types of learning, such as reinforcement or active

learning, to learn an optimal partition that minimizes unnecessary computations and can adapt on the fly to retain a balanced load. Indeed, we believe that a reinforcement learning model could be combined with S^3J's partitioning scheme to provide efficient load balancing by learning the most efficient distribution of the worksets to the workers.

5.3.3 Hybrid Checkpointing for Streaming Dataflows

In Chapter 3, we revisit checkpoint-based fault tolerance, we establish a principled evaluation framework, and we perform a detailed evaluation and comparison of three main checkpointing protocol families. Our experimental evaluation provides rich insights into the performance of each protocol and showcases the scenarios in which each of them prevails. Our experiments show that the coordinated approach outperforms the uncoordinated approach when the load is uniformly distributed across the parallel operators. However, in the presence of stragglers, for example, in the case of a skewed input load, the uncoordinated approach is a better option since it does not block the processing of other operators due to the straggling ones. At the same time, uncoordinated checkpoints can efficiently support cyclic queries that the aligned coordinated checkpoints do not. All in all, our experiments indicate that there is no protocol that can efficiently cover all use cases. Thus, we believe that hybrid checkpoint-based fault tolerance is an interesting research direction. Similar to [76], which proposes adopting different checkpointing intervals for different parts of a pipeline, a hybrid approach may use different checkpointing protocols for different parts of a pipeline based on the needs and the observed workload of each operator.

5.3.4 Tackling Adaptivity Problems with One Stone

Both *autoscaling* and *load balancing*, which are studied in this thesis, are important adaptivity problems that the research community has studied extensively, each one on its own accord. Although these problems are related to different workload properties, systems that are in an unhealthy system often require to tackle both in order to recover. Additionally, it is not always trivial to decide which of the adaptivity mechanisms should be employed to treat the system, and the benefits of each mechanism may affected by the reconfigurations applied by other adaptivity mechanisms. Therefore, it is important to not only study each problem independently but also integrate these problems into a common solution. An approach that was also proposed in [119]. From the autoscaling solutions that we discuss in detail in Chapter 4, only Dhalion can address multiple adaptivity problems, but still, it only considers them independently, one at a time. At the same time, none of the load balancing techniques, also discussed in Chapter 2, consider other adaptivity or reconfiguration problems that are relevant and may jointly affect the performance of the system. As shown in [119], tackling multiple problems with one stone is difficult. However, we argue that looking at adaptivity holistically can have a huge impact on the performance of the proposed tools.

5.3.5 Rethinking Stream Processing Benchmarking

In Chapters 3 & 4, we proposed two principled evaluation frameworks concerning two different problems of adaptivity. Although the insights we gather from applying these frameworks to evaluate state-of-the-art solutions to the problems are rich and valuable,

they can be further improved. Both of these frameworks rely on the most complete, to this date, benchmark for stream processing, the NexMark benchmark [163, 4]. However, it has been almost two decades since NexMark was first proposed. Since then, the stream processing landscape has changed significantly. New engines have been proposed, and new applications have risen. Therefore, NexMark does not fully capture the current state of stream processing. Although it includes the main processing tasks that are part of the modern pipelines, it does not describe the modern use cases and workloads.

Consequently, we argue for the need to revamp benchmarking for stream processing. To build such a modern benchmark, we need to consider all the current needs of a stream processing engine, such as autoscaling, fault tolerance, or load balancing. We believe that the first step towards a new benchmark is investigating the current state of stream processing from the practitioners' point of view. Inspired by previous work in the graph processing field [141], questionnaires, interviews, and a survey of industrial applications can be employed to unveil the current needs, the challenges, and the open problems of stream processing, but more importantly, the applications, the workloads and the policies that are currently relevant in the industry. Based on these insights, a new benchmark can be designed and implemented in order to better represent modern stream processing, allowing experimental evaluation to provide richer and more valuable insights that are also very useful in practice.

Bibliography

References

[1] From Aligned to Unaligned Checkpoints - Part 1: Checkpoints, Alignment, and Backpressure. https://flink.apache.org/2020/10/15/from-aligned-to-unaligned-checkpoints-part-1-checkpoints-alignment-and-backpressure/. Archived at web.archive.org on 2024-06-13.

[2] Improving Speed and Stability of Checkpointing [...]. https://www.alibabacloud.com/blog/599048. Archived at web.archive.org on 2024-06-13.

[3] Kubernetes horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Archived at web.archive.org on 2024-2-20.

[4] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. Archived at web.archive.org on 2024-06-13.

[5] Optimize checkpointing in your Amazon Managed Service for Apache Flink applications. https://aws.amazon.com/blogs/big-data/part-1-optimize-checkpointing-in-your-amazon-managed-service-for-apache-flink-applications-with-buffer-debloating-and-unaligned-checkpoints/. Archived at web.archive.org on 2024-06-13.

[6] Stateful Stream Processing. https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/concepts/stateful-stream-processing/}exactly-once-vs-at-least-once. Archived at web.archive.org on 2024-06-13.

[7] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. volume 5, pages 277–289, 01 2005.

[8] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, page 99–110, New York, NY, USA, 2010. Association for Computing Machinery.

[9] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.

[10] A. Aggarwal. Does your 2024 it budget account for cloud waste? https://www.financialexpress.com/business/digital-transformation-does-your-2024-it-budget-account-for-cloud-waste-3397257/. Financial Express, 2023. Archived at web.archive.org on 2024-06-13.

[11] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, page 258–265, New York, NY, USA, 2003. Association for Computing Machinery.

[12] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[14] M. I. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, editors, *The Semantic Web - ISWC 2015*, pages 374–389, Cham, 2015. Springer International Publishing.

[15] L. Alvisi, E. Elnozahy, S. Rao, S. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 242–249, 1999.

[16] K. Aoyama, K. Saito, H. Sawada, and N. Ueda. Fast approximate similarity search based on degree-reduced neighborhood graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 1055–1063, New York, NY, USA, 2011. Association for Computing Machinery.

[17] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: the stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[18] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 480–491. VLDB Endowment, 2004.

[19] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 480–491. Morgan Kaufmann, 2004.

[20] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, pages 918–929, 2006.

[21] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth. Model-based stream processing auto-scaling in geo-distributed environments. In *30th International Conference on Computer Communications and Networks, ICCCN 2021, Athens, Greece, July 19-22, 2021*, pages 1–10. IEEE, 2021.

[22] M. Balazinska, J. Hwang, and M. Shah. *Fault Tolerance and High Availability in Data Stream Management Systems*, pages 1–8. 01 2017.

[23] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In R. Hull and M. Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 212–223. ACM, 2014.

[24] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed system—an optimistic approach. 1987.

[25] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[26] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-nn queries on data streams. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 156–165, 2007.

[27] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[28] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1984, Silver Spring, Maryland, USA, October 15-17, 1984, Proceedings*, pages 207–215. IEEE Computer Society, 1984.

[29] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, 1998.

[30] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, aug 2017.

[31] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

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

[33] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo. Decentralized self-adaptation for elastic data stream processing. *Future Gener. Comput. Syst.*, 87:171–185, 2018.

[34] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB J.*, 10(2-3):199–223, 2001.

[35] B. Chandramouli, J. Goldstein, M. Barnett, and J. F. Terwilliger. Trill: Engineering a library for diverse analytics. *IEEE Data Eng. Bull.*, 38(4):51–60, 2015.

[36] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *Proc. VLDB Endow.*, 2(1):241–252, aug 2009.

[37] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org, 2003.

[38] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[39] R.-S. Chang, J.-S. Chang, and P.-S. Lin. An ant algorithm for balanced job scheduling in grids. *Future Generation Computer Systems*, 25(1):20–27, 2009.

[40] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 5–5, 2006.

[41] G. Chen, K. Yang, L. Chen, Y. Gao, B. Zheng, and C. Chen. Metric similarity joins using mapreduce. 29(3):656–669, Mar. 2017.

[42] L. Chen, C.-L. Wang, and F. C. Lau. Process reassignment with reduced migration cost in grid load rebalancing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–13, 2008.

[43] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. 01 2003.

[44] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1789–1792. IEEE Computer Society, 2016.

[45] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[46] G. J. da Silva, F. Zheng, D. Debrunner, K. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. Consistent regions: Guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.*, 9(13):1341–1352, 2016.

[47] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 108–115. IEEE, 1996.

[48] A. Das Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *Proc. VLDB Endow.*, 7(12):1059–1070, Aug. 2014.

[49] G. De Francisci Morales and A. Gionis. Streaming similarity self-join. *Proc. VLDB Endow.*, 9(10):792–803, June 2016.

[50] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[51] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[52] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2471–2486, New York, NY, USA, 2020. Association for Computing Machinery.

[53] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *2014 IEEE 30th International Conference on Data Engineering*, pages 340–351, 2014.

[54] D. N. Doan, D. Zaharie, and D. Petcu. Auto-scaling for a streaming architecture with fuzzy deep reinforcement learning. In U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, and S. L. Scott, editors, *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30, 2019, Revised Selected Papers*, volume 11997 of *Lecture Notes in Computer Science*, pages 476–488. Springer, 2019.

[55] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *VLDB*, volume 4, pages 372–383, 2004.

[56] M. Dossinger and S. Michel. Scaling out multi-way stream joins using optimized, iterative probing. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*, pages 449–456. IEEE, 2019.

[57] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang. Distributed representations of tuples for entity resolution. *Proc. VLDB Endow.*, 11(11):1454–1467, 2018.

[58] H. Edelsbrunner. Dynamic rectangle intersection searching. Technical Report 47, Institute for Information Processing, Technical University of Graz, Austria, 1980.

[59] E. Elnozahy, L. Alvisi, Y.-m. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34, 06 2002.

[60] M. ElSeidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. page 16, 2014.

[61] P. S. Eric Lam. Cloud finops: The secret to unlocking the economic potential of public cloud. https://www.forbes.com/sites/googlecloud/2022/04/04/cloud-finops-the-secret-to-unlocking-the-economic-potential-of-public-cloud/?sh=5b392ab222a5, 2022. Forbes, 2022. Archived at web.archive.org on 2024-05-21.

[62] B. Everman, M. Gao, and Z. Zong. Evaluating and reducing cloud waste and cost—a data-driven case study from azure workloads. *Sustainable Computing: Informatics and Systems*, 35:100708, 2022.

[63] R. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. pages 725–736, 06 2013.

[64] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag. Set similarity joins on mapreduce: An experimental survey. *Proc. VLDB Endow.*, 11(10):1110–1122, June 2018.

[65] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 2017.

[66] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. A survey on the evolution of stream processing systems. *VLDB Journal*, 2023.

[67] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 411–420. IEEE Computer Society, 2015.

[68] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

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

[70] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[71] Y. Gao, H. Rong, and J. Z. Huang. Adaptive grid job scheduling with genetic algorithms. *Future Generation Computer Systems*, 21(1):151–161, 2005.

[72] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1447–1463, 2014.

[73] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. *Proc. VLDB Endow.*, 14(12):3110–3121, jul 2021.

[74] S. Ghanbari and M. Othman. A priority based job scheduling algorithm in cloud computing. *Procedia Engineering*, 50(0):778–785, 2012.

[75] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[76] I. Gog, M. Isard, and M. Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 373–387, New York, NY, USA, 2021. Association for Computing Machinery.

[77] T. Greene. The hidden costs of cloud and where to find overspending. https://www.forbes.com/sites/forbestechcouncil/2023/01/19/the-hidden-costs-of-cloud-and-where-to-find-overspending/. Forbes, 2023. Archived at web.archive.org on 2024-06-13.

[78] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 146–155, 2007.

[79] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[80] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, 2012.

[81] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 296–302. IEEE Computer Society, 2014.

[82] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In U. Bellur and R. Kothari, editors, *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, pages 318–321. ACM, 2014.

[83] J. Hélary, A. Mostéfaoui, R. H. B. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Comput.*, 13(1):29–43, 2000.

[84] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner. Espbench: The enterprise stream processing benchmark. In J. Bourcier, Z. M. J. Jiang, C. Bezemer, V. Cortellessa, D. D. Pompeo, and A. L. Varbanescu, editors, *ICPE '21: ACM/SPEC*

International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, pages 201–212. ACM, 2021.

[85] N. Hidalgo, D. Wladdimiro, and E. Rosas. Self-adaptive processing graph with operator fission for elastic stream processing. *J. Syst. Softw.*, 127:205–216, 2017.

[86] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 143–157, New York, NY, USA, 2020. Association for Computing Machinery.

[87] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Elastic stream processing for the internet of things. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 100–107. IEEE Computer Society, 2016.

[88] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *21st International Conference on Data Engineering (ICDE'05)*, pages 619–630. IEEE, 2005.

[89] X. Hu, Y. Tao, and K. Yi. Output-optimal parallel algorithms for similarity joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–90. ACM, 2017.

[90] X. Hu, K. Yi, and Y. Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Trans. Database Syst.*, 44(2):6:1–6:36, 2019.

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

[92] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[93] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, Oct. 2018. USENIX Association.

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

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

[96] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *2012 IEEE 28th International Conference on Data Engineering*, pages 510–521. IEEE, 2012.

[97] A. Klein. Backblaze drive stats for 2022. https://www.backblaze.com/blog/backblaze-drive-stats-for-2022/. Archive at web.archive.org on 2024-05-21.

[98] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, dec 2000.

[99] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. - approximate nn queries on streams with guaranteed error/performance bounds. In *Proceedings 2004 VLDB Conference*, pages 804–815. Morgan Kaufmann, St Louis, 2004.

[100] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[101] H.-P. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. Probabilistic similarity join on uncertain data. In *Database Systems for Advanced Applications: 11th International Conference, DASFAA 2006, Singapore, April 12-15, 2006. Proceedings 11*, pages 295–309. Springer, 2006.

[102] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, page 239–250, New York, NY, USA, 2015.

[103] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 25–36, New York, NY, USA, 2012. Association for Computing Machinery.

[104] A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5(12):2032–2033, aug 2012.

[105] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[106] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery.

[107] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In C. D. Napoli,

V. Salapura, H. Franke, and R. Hou, editors, *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*, pages 53:1–53:8. ACM, 2015.

[108] X. Lian and L. Chen. Similarity join processing on uncertain data streams. *IEEE Transactions on Knowledge and Data Engineering*, 23(11):1718–1734, 2011.

[109] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 811–825, New York, NY, USA, 2015. Association for Computing Machinery.

[110] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.*, 12(4):24:1–24:33, 2018.

[111] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 399–410. IEEE Computer Society, 2015.

[112] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans. Parallel Distributed Syst.*, 29(3):572–585, 2018.

[113] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, and L. Querzoni. PASCAL: an architecture for proactive auto-scaling of distributed services. *Future Gener. Comput. Syst.*, 98:342–361, 2019.

[114] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[115] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, page 69–78, USA, 2014. IEEE Computer Society.

[116] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*, pages 69–78. IEEE Computer Society, 2014.

[117] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 950–961. ACM, 2007.

[118] S. Madden. From databases to big data. *IEEE Internet Computing*, 16(3):4–6, 2012.

[119] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 227–230, 2017.

[120] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery.

[121] G. Mencagli, M. Torquati, and M. Danelutto. Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Gener. Comput. Syst.*, 79:862–877, 2018.

[122] A. Metwally, D. Agrawal, and A. E. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 241–250. ACM, 2007.

[123] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, Apr. 2012.

[124] 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, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[125] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. SplitJoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 493–505, Denver, CO, June 2016. USENIX Association.

[126] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Scalable multiway stream joins in hardware. *IEEE Transactions on Knowledge and Data Engineering*, 32(12):2438–2452, 2020.

[127] A. Pacaci, A. Bonifati, and M. T. Özsu. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1415–1430, New York, NY, USA, 2020. Association for Computing Machinery.

[128] G. Papadakis, G. M. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. Three-dimensional entity resolution with jedai. *Inf. Syst.*, 93:101565, 2020.

[129] G. Papadakis, L. Tsekouras, E. Thanos, N. Pittaras, G. Simonini, D. Skoutas, P. Isaris, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. Jedai^3 : beyond batch, blocking-based entity resolution. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 603–606. OpenProceedings.org, 2020.

[130] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. Andrienko, and Y. Theodoridis. Similarity search in trajectory databases. In *14th International Symposium on Temporal Representation and Reasoning (TIME'07)*, pages 129–140. IEEE, 2007.

[131] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, 2006.

[132] E. Pinheiro, W. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, editors, *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, pages 17–28. USENIX, 2007.

[133] V. Poosala, V. Ganti, and Y. E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999.

[134] I. Popivanov and R. J. Miller. Similarity search over time-series data using wavelets. In *Proceedings 18th international conference on data engineering*, pages 212–221. IEEE, 2002.

[135] K. Psarakis, G. Siachamis, G. Christodoulou, M. Fragkoulis, and A. Katsifodimos. Styx: Transactional stateful functions on streaming dataflows, 2024.

[136] K. Psarakis, W. Zorgdrager, M. Fragkoulis, G. Salvaneschi, and A. Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. In *EDBT*, 2024.

[137] S. Qian, G. Wu, J. Huang, and T. Das. Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT)*, pages 592–598, 2016.

[138] Y. Qiu, S. Papadias, and K. Yi. Streaming hypercube: A massively parallel stream join algorithm. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 642–645. OpenProceedings.org, 2019.

[139] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), 2019.

[140] S. Sagiroglu and D. Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, 2013.

[141] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, dec 2017.

[142] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire. Correlation sketches for approximate join-correlation queries. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1531–1544, New York, NY, USA, 2021. Association for Computing Machinery.

[143] S. Schelter, C. Boden, and V. Markl. Scalable similarity-based neighborhood methods with mapreduce. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 163–170, 2012.

[144] T. Schlegl, S. Schlegl, D. Tomaselli, N. West, and J. Deuse. Adaptive similarity search for the retrieval of rare events from large time series databases. *Advanced Engineering Informatics*, 52:101629, 2022.

[145] A. Shahvarani and H.-A. Jacobsen. Distributed stream knn join. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1597–1609, New York, NY, USA, 2021. Association for Computing Machinery.

[146] M. Sheikhalishahi, R. M. Wallace, L. Grandinetti, J. L. Vazquez-Poletti, and F. Guerriero. A multi-dimensional job scheduling. *Future Generation Computer Systems*, 54:123–131, 2016.

[147] A. Shukla, S. Chaturvedi, and Y. Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017. e4257 cpe.4257.

[148] G. Siachamis, G. Christodoulou, K. Psarakis, M. Fragkoulis, A. van Deursen, and A. Katsifodimos. Evaluating stream processing autoscalers. In *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*, DEBS '24, page 110–122, New York, NY, USA, 2024. Association for Computing Machinery.

[149] G. Siachamis, J. Kanis, W. Koper, K. Psarakis, M. Fragkoulis, A. van Deursen, and A. Katsifodimos. Towards evaluating stream processing autoscalers. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 95–99, Los Alamitos, CA, USA, apr 2023. IEEE Computer Society.

[150] G. Siachamis, K. Psarakis, M. Fragkoulis, O. Papapetrou, A. van Deursen, and A. Katsifodimos. Adaptive distributed streaming similarity joins. In *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*, DEBS '23, page 25–36, New York, NY, USA, 2023. Association for Computing Machinery.

[151] G. Siachamis, K. Psarakis, M. Fragkoulis, A. van Deursen, P. Carbone, and A. Katsifodimos. Checkmate: Evaluating checkpointing protocols for streaming dataflows. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4030–4043, 2024.

[152] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1637–1650, New York, NY, USA, 2021. Association for Computing Machinery.

[153] J. Smith, P. Watson, A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. *Int. J. High Perform. Comput. Appl.*, 17(4):353–367, 2003.

[154] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[155] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, aug 1985.

[156] A. Stupar, S. Michel, and R. Schenkel. Rankreduce–processing k-nearest neighbor queries on top of mapreduce. *Large-Scale Distributed Systems for Information Retrieval*, 15:3, 2010.

[157] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6(14):1930–1941, 2013.

[158] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, sep 2013.

[159] S. Suri, I. F. Ilyas, C. Ré, and T. Rekatsinas. Ember: No-code context enrichment via similarity-based keyless joins. *Proc. VLDB Endow.*, 15(3):699–712, nov 2021.

[160] M. Tang, Y. Yu, W. G. Aref, Q. M. Malluhi, and M. Ouzzani. Efficient processing of hamming-distance-based similarity-search queries over mapreduce. In *EDBT*, pages 361–372, 2015.

[161] Y. Tian, T. Yan, X. Zhao, K. Huang, and X. Zhou. A learned index for exact similarity search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 35(8):7624–7638, 2023.

[162] A. Tsymbal. The problem of concept drift: definitions and related work. 2004.

[163] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark–a benchmark for queries over data streams (draft). Technical report, 2008.

[164] G. van Dongen and D. V. den Poel. Evaluation of stream processing frameworks. *IEEE Trans. Parallel Distributed Syst.*, 31(8):1845–1858, 2020.

[165] B. Varga, M. Balassi, and A. Kiss. Towards autoscaling of apache flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13:1–21, 04 2021.

[166] B. Varga, M. Balassi, and A. Kiss. Towards autoscaling of apache flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13(1):39–59, 2021.

[167] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. SIGMOD '10, page 495–506, New York, NY, USA, 2010. Association for Computing Machinery.

[168] J. Wang, J. Feng, and G. Li. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. *Proc. VLDB Endow.*, 3(1–2):1219–1230, sep 2010.

[169] Q. Wang and T. Palpanas. Deep learning embeddings for data series similarity search. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 1708–1716, 2021.

[170] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: Workload distribution and adaptation by time-slicing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, page 299–310, New York, NY, USA, 2009. Association for Computing Machinery.

[171] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. KDD '13, page 829–837, New York, NY, USA, 2013. Association for Computing Machinery.

[172] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.

[173] J. Wu, Y. Zhang, J. Wang, C. Lin, Y. Fu, and C. Xing. Scalable metric similarity join using mapreduce. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1662–1665, 2019.

[174] J. Yang, W. Zhang, X. Wang, Y. Zhang, and X. Lin. Distributed streaming set similarity join. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 565–576. IEEE, 2020.

[175] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical report, UCB/EECS-2009-55, EECS Department, University of California, 2009.

[176] E. Zapridou, I. Mytilinis, and A. Ailamaki. Dalton: Learned partitioning for distributed data streams. *Proc. VLDB Endow.*, 16(3):491–504, 2022.

[177] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 759–770, 2017.

[178] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 54–71, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

List of Figures

1.1 An example application using the MapReduce paradigm. 2
1.2 Prevailing processing models. 3
1.3 A stream processing pipeline. S1 is the source operator, while O1 is the output operator of the pipeline. M1 and M2 are parallel instances of the map operator. Similarly, A{1-3} are parallel instances of the aggregate operator. 4
1.4 An example of a windowed streaming similarity join between two streams. 6
1.5 Streaming similarity joins employed in an infrastructure monitoring application. Within a large organization, different teams use different monitoring tools to monitor their assets (hardware or software). These different monitoring tools may represent the same assets differently. However, a major incident response team wants to match the produced alerts that contain relevant information in order to enrich the information provided to downstream tasks, such as root cause analysis or incident prediction model training. 7
1.6 Types of failures in distributed stream processing. 8
1.7 Different ways of provisioning resources. 10

2.1 Overview of proposed solution's workflow 25
2.2 (a) Paradigm's similarity computations workflow. (b) Example with a duplicate evaluation of a candidate pair. 27
2.3 Workset Formulation Workflow 30
2.4 99% latency percentile per worker for varying selectivities. Each line represents a single worker (in this case, 5 workers total). Incoming ratio 4000 records per second, Parallelism: 5, Uniform distribution. 36
2.5 99% latency percentile per worker for varying parallelism. Each line represents a single worker (in this case, 5, 10, and 15 workers accordingly). Incoming ratio 8000 records per second, Selectivity: 0.1%, Uniform distribution. 38
2.6 99% latency percentile per worker. Each line represents a single worker (in this case, 5 workers in total). Uniform distribution, Incoming ratio 8000 records per sec, Selectivity: 0.1%, Parallelism: 5. 39

3.1 Examples of valid recovery lines when in-flight messages are included in the global state. 44
3.2 Cases of inconsistent and consistent state after recovery for stateful operators O_1, O_2 and O_3. 45

3.3 Example execution of the coordinated aligned checkpointing protocol. Messages are represented as circles, and markers are squares. Different colors denote different coordinated rounds. 48
3.4 Example overview of Rollback propagation algorithm on a given execution timeline 50
3.5 Domino effect of invalid checkpoints on a cyclic query. 51
3.6 Execution graph of the reachability query. 54
3.7 Normalized maximum sustainable throughput per query achieved by each protocol for different parallelism. 56
3.8 Average checkpointing time on different parallelisms. 57
3.9 50th percentile latency. The black dashed vertical line indicates the moment of failure. 58
3.10 99th percentile latency. The black dashed vertical line indicates the moment of failure. 59
3.11 Restart time after failure per query for each protocol on different levels of parallelism. 60
3.12 50th percentile latency & average checkpointing time under different hot items percentages. 62
3.13 Restart time after failure per query in the presence of skew. 63

4.1 MAPE loop for stream processing autoscaling 68
4.2 Workloads 75
4.3 Increasing pattern 77
4.4 Decreasing pattern 78
4.5 Cosine pattern 79
4.6 Random pattern 80
4.7 *Filter*. 81
4.8 *Incremental Join* 83
4.9 *Sliding Windowed Aggregate*. 84
4.10 Convergence 85

List of Tables

2.1 Related work comparison. 19
2.2 Effects of selectivity on duplication ratio and comparisons reduction. . . . 37
2.3 Effects of parallelism to duplication ratio and comparisons reduction. . . . 38

3.1 Summary of the features of the checkpointing protocols explored in Section 3.3 47
3.2 Ratio of message overhead with respect to an execution without checkpoints. 56
3.3 Total checkpoints and percentage of invalid checkpoints. 61
3.4 Average checkpointing time (CT), restart time (RT), and invalid checkpoints (IC) for the cyclic query. 63

Curriculum Vitæ

Georgios Siachamis

1994/05/28	Born in Athens, Greece

Professional Experience

2020-2024	Academic Consultant, ING, Netherlands
2019	Research Intern, Université de Cergy-Paris, France
2015-2016	Software Engineer, FOCUS ON DIGITAL LTD., Greece

Education

2019-2024	Doctor of Philosophy (PhD), Computer Science Delft University of Technology, Netherlands
20014-2019	Diploma (M.Eng), Electrical and Computer Engineering National Technical University of Athens, Greece

List of Publications

1. **George Siachamis**, George Christodoulou, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Asterios Katsifodimos. Evaluating Stream Processing Autoscalers, in ACM Conference on Distributed and Event-Based Systems (DEBS), 2024.

2. **George Siachamis**, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, Asterios Katsifodimos. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows, in IEEE International Conference on Data Engineering (ICDE), 2024.

3. Kyriakos Psarakis, **George Siachamis**, George Christodoulou, Marios Fragkoulis, Asterios Katsifodimos. Styx: Transactional Stateful Functions on Streaming Dataflows, in arXiv, 2024.

4. **George Siachamis**, Kyriakos Psarakis, Marios Fragkoulis, Odysseas Papapetrou, Arie van Deursen, Asterios Katsifodimos. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows, in ACM Conference on Distributed and Event-Based Systems (DEBS), 2023.

5. **George Siachamis**, Job Kanis, Wybe Koper, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Asterios Katsifodimos. Towards Evaluating Stream Processing Autoscalers, in International Workshop on Self-managing Database Systems (SMDB), co-located with ICDE, 2023.

6. Christos Koutras, Kyriakos Psarakis, **George Siachamis**, Andra Ionescu, Marios Fragkoulis, Angela Bonifati and Asterios Katsifodimos. Valentine in Action: Matching Tabular Data at Scale, in Very Large Data Bases (VLDB), 2021.

7. Christos Koutras, **George Siachamis**, Andra Ionescu, Kyriakos Psarakis, Marios Fragkoulis, Jery Brons, Christoph Lofi, Angela Bonifati and Asterios Katsifodimos. Valentine: Evaluating Matching Techniques for Dataset Discovery, in International Conference on Data Engineering (ICDE), 2021.

8. **George Siachamis**. Integrating Massive Data Streams, in Very Large Data Bases (VLDB), PhD Workshop, 2021.

Included in this thesis.

SIKS Dissertation Series

Since 1998, all dissertations written by PhD students who have conducted their research under auspices of a senior research fellow of the SIKS research school are published in the SIKS Dissertation Series.

2016
01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VUA), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VUA), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VUA), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UvA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VUA), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VUA), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VUA), Refining Statistical Data on the Web
19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UvA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VUA), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UvA), Query Auto Completion in Information Retrieval

24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VUA), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (TiU), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis

50 Yan Wang (TiU), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UvA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VUA), Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TU/e), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (TiU), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UvA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VUA), Logics for causal inference under uncertainty
23 David Graus (UvA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VUA), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VUA), Architecture Practices for Complex Contexts

29 Adel Alhuraibi (TiU), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (TiU), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VUA), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
02 Felix Mannhardt (TU/e), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process
06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations

10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TU/e), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
14 Bart Joosten (TiU), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OU), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (TiU), Emotion Recognition from Mandarin Speech
30 Wouter Beek (VUA), The "K" in "semantic web" stands for "knowledge": scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
06 Chris Dijkshoorn (VUA), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes

09 Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VUA), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TU/e), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VUA),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VUA), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VUA), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OU), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VUA), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VUA), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
33 Anil Yaman (TU/e), Evolution of Biologically Inspired Learning in Artificial Neural Networks
34 Negar Ahmadi (TU/e), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES
35 Lisa Facey-Shaw (OU), Gamification with digital badges in learning programming
36 Kevin Ackermans (OU), Designing Video-Enhanced Rubrics to Master Complex Skills

37 Jian Fang (TUD), Database Acceleration on FPGAs
38 Akos Kadar (OU), Learning visually grounded and multilingual representations

---

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
04 Maarten van Gompel (RUN), Context as Linguistic Bridges
05 Yulong Pei (TU/e), On local and global structure mining
06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
07 Wim van der Vegt (OU), Towards a software architecture for reusable game components
08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search
09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
10 Alifah Syamsiyah (TU/e), In-database Preprocessing for Process Mining
11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data AugmentationMethods for Long-Tail Entity Recognition Models
12 Ward van Breda (VUA), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
15 Konstantinos Georgiadis (OU), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
17 Daniele Di Mitri (OU), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VUA), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VUA), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TU/e), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization

27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst
31 Gongjin Lan (VUA), Learning better – From Baby to Better
32 Jason Rhuggenaath (TU/e), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TU/e), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (UM), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

---

2021 01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
02 Rijk Mercuur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UvA), Modeling Users Interacting with Smart Devices
04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
19 Roberto Verdecchia (VUA), Architectural Technical Debt: Identification and Management

20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
22 Sihang Qiu (TUD), Conversational Crowdsourcing
23 Hugo Manuel Proença (UL), Robust rules for prediction and description
24 Kaijie Zhu (TU/e), On Efficient Temporal Subgraph Query Processing
25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
26 Benno Kruit (CWI/VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs

---

2022 01 Judith van Stegeren (UT), Flavor text generation for role-playing video games
02 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
03 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
04 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
05 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
06 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
07 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
08 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
09 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
14 Michiel Overeem (UU), Evolution of Low-Code Platforms
15 Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
16 Pieter Gijsbers (TU/e), Systems for AutoML Research
17 Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification

18 Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
19 Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
20 Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media
21 Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
22 Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
23 Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
24 Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
25 Anna L.D. Latour (UL), Optimal decision-making under constraints and uncertainty
26 Anne Dirkson (UL), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
27 Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
28 Onuralp Ulusoy (UU), Privacy in Collaborative Systems
29 Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
30 Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
31 Konstantinos Traganos (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
32 Cezara Pastrav (UU), Social simulation for socio-ecological systems
33 Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management
34 Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
35 Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction

---

2023 01 Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
02 Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
03 Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
04 Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval
05 Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
06 António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
07 Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning

08 Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
09 Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
10 Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing
11 Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
12 Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
13 Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
14 Selma Čaušević (TUD), Energy resilience through self-organization
15 Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models
16 Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters
17 Meike Nauta (UT), Explainable AI and Interpretable Computer Vision – From Oversight to Insight
18 Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
19 George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
20 Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
21 Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
22 Alireza Shojaifar (UU), Volitional Cybersecurity
23 Theo Theunissen (UU), Documentation in Continuous Software Development
24 Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
25 Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
26 Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
27 Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
28 Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts
29 Tim Draws (TUD), Understanding Viewpoint Biases in Web Search Results

---

2024 01 Daphne Miedema (TU/e), On Learning SQL: Disentangling concepts in data systems education
02 Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
03 Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
04 Mike Huisman (UL), Understanding Deep Meta-Learning
05 Yiyong Gou (UM), Aerial Robotic Operations: Multi-environment Cooperative Inspection & Construction Crack Autonomous Repair

06 Azqa Nadeem (TUD), Understanding Adversary Behavior via XAI: Leveraging Sequence Clustering to Extract Threat Intelligence
07 Parisa Shayan (TiU), Modeling User Behavior in Learning Management Systems
08 Xin Zhou (UvA), From Empowering to Motivating: Enhancing Policy Enforcement through Process Design and Incentive Implementation
09 Giso Dal (UT), Probabilistic Inference Using Partitioned Bayesian Networks
10 Cristina-Iulia Bucur (VUA), Linkflows: Towards Genuine Semantic Publishing in Science
11 withdrawn
12 Peide Zhu (TUD), Towards Robust Automatic Question Generation For Learning
13 Enrico Liscio (TUD), Context-Specific Value Inference via Hybrid Intelligence
14 Larissa Capobianco Shimomura (TU/e), On Graph Generating Dependencies and their Applications in Data Profiling
15 Ting Liu (VUA), A Gut Feeling: Biomedical Knowledge Graphs for Interrelating the Gut Microbiome and Mental Health
16 Arthur Barbosa Câmara (TUD), Designing Search-as-Learning Systems
17 Razieh Alidoosti (VUA), Ethics-aware Software Architecture Design
18 Laurens Stoop (UU), Data Driven Understanding of Energy-Meteorological Variability and its Impact on Energy System Operations
19 Azadeh Mozafari Mehr (TU/e), Multi-perspective Conformance Checking: Identifying and Understanding Patterns of Anomalous Behavior
20 Ritsart Anne Plantenga (UL), Omgang met Regels
21 Federica Vinella (UU), Crowdsourcing User-Centered Teams
22 Zeynep Ozturk Yurt (TU/e), Beyond Routine: Extending BPM for Knowledge-Intensive Processes with Controllable Dynamic Contexts
23 Jie Luo (VUA), Lamarck's Revenge: Inheritance of Learned Traits Improves Robot Evolution
24 Nirmal Roy (TUD), Exploring the effects of interactive interfaces on user search behaviour
25 Alisa Rieger (TUD), Striving for Responsible Opinion Formation in Web Search on Debated Topics
26 Tim Gubner (CWI), Adaptively Generating Heterogeneous Execution Strategies using the VOILA Framework
27 Lincen Yang (UL), Information-theoretic Partition-based Models for Interpretable Machine Learning
28 Leon Helwerda (UL), Grip on Software: Understanding development progress of Scrum sprints and backlogs
29 David Wilson Romero Guzman (VUA), The Good, the Efficient and the Inductive Biases: Exploring Efficiency in Deep Learning Through the Use of Inductive Biases
30 Vijanti Ramautar (UU), Model-Driven Sustainability Accounting
31 Ziyu Li (TUD), On the Utility of Metadata to Optimize Machine Learning Workflows
32 Vinicius Stein Dani (UU), The Alpha and Omega of Process Mining

33 Siddharth Mehrotra (TUD), Designing for Appropriate Trust in Human-AI interaction
34 Robert Deckers (VUA), From Smallest Software Particle to System Specification - MuDForM: Multi-Domain Formalization Method
35 Sicui Zhang (TU/e), Methods of Detecting Clinical Deviations with Process Mining: a fuzzy set approach
36 Thomas Mulder (TU/e), Optimization of Recursive Queries on Graphs
37 James Graham Nevin (UvA), The Ramifications of Data Handling for Computational Models
38 Christos Koutras (TUD), Tabular Schema Matching for Modern Settings
39 Paola Lara Machado (TU/e), The Nexus between Business Models and Operating Models: From Conceptual Understanding to Actionable Guidance
40 Montijn van de Ven (TU/e), Guiding the Definition of Key Performance Indicators for Business Models