A PROCRUSTEAN APPROACH TO STREAM PROCESSING

by

Nikolaos Romanos Katsipoulakis

Bachelor of Science, National Kapodistrian University of Athens,

2011

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2018

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Nikolaos Romanos Katsipoulakis

It was defended on

December 18th 2018

and approved by

Dr. Alexandros Labrinidis, Department of Computer Science, University of Pittsburgh

Dr. Panos K. Chrysanthis, Department of Computer Science, University of Pittsburgh

Dr. John Lange, Department of Computer Science, University of Pittsburgh

Dr. Andrew Pavlo, Department of Computer Science, Carnegie Mellon University

Dissertation Advisors: Dr. Alexandros Labrinidis, Department of Computer Science, University of Pittsburgh,

Dr. Panos K. Chrysanthis, Department of Computer Science, University of Pittsburgh

A PROCRUSTEAN APPROACH TO STREAM PROCESSING

Nikolaos Romanos Katsipoulakis, PhD

University of Pittsburgh, 2018

The increasing demand for real-time data processing and the constantly growing data volume have contributed to the rapid evolution of Stream Processing Engines (SPEs), which are designed to continuously process data as it arrives. Low operational cost and timely delivery of results are both objectives of paramount importance for SPEs. Given the volatile and uncharted nature of data streams, achieving the aforementioned goals under fixed resources is a challenge. This calls for adaptable SPEs, which can react to fluctuations in processing demands.

In the past, three techniques have been developed for improving an SPE's ability to adapt. Those techniques are classified based on applications' requirements on *exact* or *approximate* results: *stream partitioning*, and *re-partitioning* target *exact*, and *load shedding* targets *approximate* processing. *Stream partitioning* strives to balance load among processors, and previous techniques neglected hidden costs of distributed execution. *Load Shedding* lowers the accuracy of results by dropping part of the input, and previous techniques did not cope with evolving streams. *Stream re-partitioning* is used to reconfigure execution while processing takes place, and previous techniques did not fully utilize window semantics.

In this dissertation, we put stream processing in a *procrustean bed*, in terms of the manner and the degree that processing takes place. To this end, we present new approaches, for window-based aggregate operators, which are applicable to both *exact* and *approximate* stream processing in modern SPEs. Our stream partitioning, re-partitioning, and load shedding solutions offer improvements in performance and accuracy on real-world data by exploiting the semantics of both data and operations. In addition, we present SPEAr, the

design of an SPE that accelerates processing by delivering *approximate* results with accuracy guarantees and avoiding unnecessary load. Finally, we contribute a hybrid technique, ShedPart, which can further improve load balance and performance of an SPE.

TABLE OF CONTENTS

PREFACE xvii

1.0 INTRODUCTION 1

1.1 Challenges 2

1.1.1 Partitioning 3

1.1.2 Load Shedding 4

1.1.3 Re-partitioning (Scale-out) 5

1.2 Our Approach 5

1.2.1 Exact Stream Processing 6

1.2.2 Approximate Stream Processing 7

1.2.3 Combination of methods 7

1.3 Research Contributions 7

1.4 Road Map 8

2.0 BACKGROUND 9

2.1 System Model 9

2.2 Query Model 11

2.3 Execution Plan Generation 14

2.3.1 Logical Plan 14

2.3.2 Physical Execution Plan 15

2.4 Stateful Execution in a SPE 17

2.4.1 Tuple Arrival 18

2.4.2 Watermark Arrival 19

2.5 Result Approximation in Stream Processing 20

2.5.1 Error Metrics . . . 21
2.5.1.1 Algebraic and Distributive: . . . 22
2.5.1.2 Holistic: . . . 23
2.6 Experimental Infrastructure . . . 23
2.6.1 Real-world datasets . . . 23
2.6.2 Storm Runtime . . . 24
3.0 STREAM PARTITIONING . . . 27
3.1 A new formulation for Parallel Stateful Operations . . . 27
3.2 Proposed partitioning cost model . . . 29
3.3 The pitfall of ignoring aggregation costs . . . 31
3.3.1 Partition Algorithm Taxonomy . . . 32
3.3.2 Mapping existing Partition Algorithms to each Category . . . 32
3.3.2.1 Shuffle . . . 32
3.3.2.2 Hash (or Field) . . . 33
3.3.2.3 Partial-Key . . . 34
3.4 Minimizing imbalance with low aggregation cost . . . 34
3.4.1 Incorporating Cardinality in Partitioning . . . 35
3.4.1.1 Incorporating Cardinality in measuring Imbalance . . . 35
3.4.1.2 Incorporating Cardinality in Measuring Aggregation Cost . . . 36
3.4.2 Cardinality Estimation data structures . . . 37
3.4.2.1 Naive . . . 37
3.4.2.2 Hyperloglog . . . 37
3.5 Proposed Cardinality-aware Partitioning algorithms . . . 38
3.5.1 Cardinality Imbalance Minimization (CM) . . . 40
3.5.2 Group Affinity and Imbalance Minimization (AM & cAM) . . . 40
3.5.3 Hybrid Imbalance Minimization (LM) . . . 43
3.6 Experimental Setup . . . 44
3.6.1 Stream Partitioning Algorithms . . . 45
3.6.2 Data sets and Workloads . . . 45
3.7 Experimental Results . . . 47

3.7.1 Performance . . . 47
3.7.1.1 TPCH Query 1 (Figure 12) . . . 48
3.7.1.2 TPCH Query 3 (Figure. 13 - 14) . . . 48
3.7.1.3 DEBS Query 1 (Figure. 15a - 15c) . . . 49
3.7.1.4 DEBS Query 2 (Figure. 16a - 16c) . . . 50
3.7.2 Scalability . . . 51
3.7.2.1 GCM Query 1 (Figs. 17 & 18) . . . 52
3.7.2.2 GCM Query 2 (Fig. 19) . . . 53
3.7.3 Partition algorithm cost . . . 53
3.7.3.1 Partition latency (Figure. 20a - 20c) . . . 54
3.7.3.2 Partition Memory (Table 4) . . . 54
3.8 Discussion . . . 55
3.9 Related Work . . . 58
3.10 Summary . . . 59
4.0 STREAM RE-PARTITIONING . . . 60
4.1 Synefo: Elasticity in Storm . . . 60
4.1.1 Architecture Overview . . . 61
4.1.1.1 Synefo Server . . . 64
4.1.1.2 Synefo Spout . . . 65
4.1.1.3 Synefo Bolt . . . 66
4.1.2 *Synefo* Summary . . . 68
4.2 Unimico . . . 68
4.2.1 Migration timestamp . . . 70
4.2.2 Stoping and resuming continuous queries . . . 73
4.2.2.1 Stopping the query at the originating node . . . 73
4.2.2.2 Starting the query at target node . . . 74
4.2.3 Experimental Evaluation . . . 75
4.2.3.1 Experimental Setup . . . 75
4.2.3.2 Simple CQ migration (Figures 28, 29, 30, & 31) . . . 76
4.2.3.3 Complex CQ migration (Figure 32) . . . 77

4.2.4 UniMiCo Related Work . . . 78
4.2.5 UniMiCo Summary . . . 79
5.0 APPROXIMATE STREAM PROCESSING (WITHOUT GUARANTEES) . . . 82
5.1 Continuous Queries with Service Level Objectives . . . 82
5.2 The need for Load Shedding . . . 83
5.2.1 Shortcomings of existing techniques . . . 83
5.2.2 Concept Drift Evidence . . . 84
5.3 Concept Drift Definition . . . 85
5.4 Load Shedding . . . 87
5.4.1 Focal point of this work: what to shed . . . 88
5.5 Existing Shedding Algorithms . . . 89
5.5.1 Normal Execution (Baseline) . . . 89
5.5.2 Uniform Shedding (State of the Art) . . . 89
5.6 Concept-Driven Load Shedding Algorithm . . . 90
5.6.1 Designing CoD to eliminate overhead . . . 91
5.6.2 Implementation for different query types . . . 93
5.6.2.1 Scalar User-Defined Aggregations . . . 94
5.6.2.2 User-Defined Aggregations . . . 94
5.6.2.3 Two-way Equality Joins . . . 94
5.7 Experimental Evaluation . . . 95
5.7.1 Experimental Setup . . . 95
5.7.2 Datasets . . . 96
5.7.3 Experimental Results . . . 97
5.7.3.1 Scalability (Figure 37) . . . 98
5.7.3.2 Performance (Figure 38) . . . 98
5.7.3.3 Accuracy (Figure 39) . . . 99
5.7.3.4 Data Volume Reduction (Figure 40) . . . 100
5.8 Related Work . . . 101
5.9 Summary . . . 102

6.0 APPROXIMATE STREAM PROCESSING (WITH GUARANTEES) 104
6.1 The need to accelerate processing . . . 105
6.2 Short Primer on Approximate Query Processing (AQP) . . . 108
6.3 Approximate Stream Processing . . . 110
6.3.1 Continuous Query Model Design Options . . . 111
6.3.1.1 Unbound Budget–Unbound Accuracy: . . . 111
6.3.1.2 Bound Budget–Unbound Accuracy: . . . 111
6.3.1.3 Unbound Budget–Bound Accuracy: . . . 112
6.3.1.4 Bound Budget–Bound Accuracy: . . . 112
6.4 SPEAr System Overview . . . 113
6.4.1 Architecture . . . 113
6.4.2 Supported Queries . . . 114
6.4.3 Workflow . . . 116
6.4.3.1 Tuple Arrival . . . 116
6.4.3.2 Watermark Arrival . . . 117
6.5 Sampling . . . 117
6.5.1 Background: Sampling for AQP . . . 118
6.5.1.1 Uniform Sampling . . . 118
6.5.1.2 Stratified Sampling . . . 118
6.5.2 Online sampling creation for SPEAr . . . 119
6.5.2.1 Uniform Sampling for Scalar Aggregations . . . 119
6.5.2.2 Stratified Sampling for Grouped Aggregations . . . 120
6.5.2.3 Quantile Sampling . . . 121
6.6 Accuracy Check . . . 121
6.6.1 Scalar Confidence Check . . . 122
6.6.2 Grouped Confidence Check . . . 123
6.6.3 Quantile Confidence Check . . . 124
6.7 Experimental Evaluation . . . 124
6.7.1 SPEAr Implementation Details . . . 124
6.7.2 Experimental Setup Details . . . 125

6.7.2.1 Datasets . . . 126
6.8 Experimental Results . . . 127
6.8.1 Scalability (Figures 45-46) . . . 127
6.8.2 Performance (Figure 47 and Table 9) . . . 130
6.8.3 Window Size Sensitivity Analysis (Figure 48) . . . 132
6.8.4 SPEAr compared to Incremental Processing (Figure 49) . . . 133
6.8.5 Error (Figure 50) . . . 136
6.8.6 Data Volume Reduction (Figure 51) . . . 136
6.9 Related Work . . . 137
6.9.1 Incremental Processing . . . 138
6.9.2 Load Shedding . . . 138
6.9.3 Sketches . . . 139
6.9.4 Elastic SPEs . . . 139
6.10 Summary . . . 140
7.0 SHED-PARTITION: THE POWER OF TWO TECHNIQUES IN ONE 141
7.1 The need for tighter integration of adaptation techniques . . . 141
7.2 Problem Statement . . . 144
7.3 Partition Model . . . 145
7.3.1 Sample Size Estimation . . . 146
7.4 ShedPart . . . 146
7.4.1 ShedPart Architecture . . . 147
7.4.2 ShedPart Algorithm . . . 148
7.4.2.1 Tuple Arrival: . . . 148
7.4.2.2 Watermark Arrival: . . . 150
7.4.3 ShedPart Optimizations . . . 152
7.4.3.1 Storage Optimization . . . 152
7.4.3.2 Incremental Variance Monitoring . . . 153
7.5 ShedPart Experimental Evaluation . . . 153
7.5.1 Experimental Setup . . . 154
7.5.2 Experimental Results . . . 154

7.5.2.1 Sensitivity Analysis 155
7.5.2.2 Impact on Imbalance 156
7.5.2.3 Overall Performance 159
7.6 Summary 162
8.0 EPILOGUE 163
8.1 Summary of Contributions 163
8.2 Future Research Directions 166
Bibliography 168

LIST OF TABLES

1 Model Symbol Overview 10
2 Stream partitioning algorithms 44
3 Summary of data characteristics. 46
4 Memory requirements in MBs 57
5 Examples of concept per query type 86
6 Dataset Characteristics 95
7 Properties of design space 105
8 Datasets and Queries Used 124
9 Processing time (msec) of SPEAr and Count-Min sketch. 131

LIST OF FIGURES

1 The actions taken by an SPE administrator. 2

2 Example Query 12

3 Logical Plan for CQ of Figure 2. 14

4 Example query plan for a CQ with a join. 15

5 Physical Execution Plan for CQ of Figure 2. 16

6 Tuple arrival: *Single* and *Multiple* Buffer(s) design operations. 18

7 Watermark arrival: *Single* and *Multiple* Buffer(s) design operations. 19

8 Query with an accuracy specification. 20

9 The windowed grouped average of the sample query (Figure 2) as a three stage process. 28

10 Expected performance of stream partitioning algorithms. 31

11 Existing stream partitioning algorithms lack a unified model that limits *imbalance* while keeping *aggregation* cost low. 33

12 TPCH Query 1 performance. 47

13 TPCH Query 3 performance. 49

14 TPCH Query 3 relative imbalance to FLD. 50

15 DEBS Query 1 performance. 51

16 DEBS Query 2 performance. 52

17 Latency Improvement (%) over PK for GCM Query 1. 53

18 Aggregation runtime (%) for GCM Query 1. 54

19 Latency Improvement (%) over PK for GCM Query 2. 55

20 Per window Partition latency for DEBS Query 1 (i.e., processing cost of partitioning algorithm). . . . 56
21 Synefo's system stack. . . . 61
22 *Synefo* physical and active topology for the query defined with the code of Listing 4.1. . . . 64
23 Scale-out operation in *Synefo*. . . . 65
24 Scale-in operation in *Synefo*. . . . 66
25 UniMiCo's migration strategy 69
26 Calculating migration timestamp with two consecutive windows 70
27 Queries used in our experimental evaluation. . . . 76
28 Results of Query 1 around the migration point (i.e., 10 second mark). . . . 77
29 The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay. . . . 78
30 Results of Query 2 around the migration point (i.e., 10 second mark). . . . 79
31 The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay (similar behavior with Query 1) 80
32 Results and response time of the complex query Q3 around the migration point of 10 second mark. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay 81
33 Summary of mean response times for all Queries when no migration takes place and when UniMiCo is used to migrate state. . . . 81
34 Real-world datasets feature varying *concept drift* characteristics, in terms of both inter-window concept distance and overlap. . . . 84
35 CoD tuple arrival: the tuple is optionally added to the windows it belongs to, and also stored in the buffer space. . . . 92
36 CoD watermark arrival: the concept is extracted from the sample and then tuples are shed based on it. . . . 93

37 Scalability of normal execution compared to Uniform load shedding and CoD, with the shedding ratio set to 98%. 96
38 Overall average and 95 percentile window latency with four worker nodes. . . 97
39 Average and 95 percentile relative error. 99
40 CoD achieves much better error with a fraction of the data required by Uniform. 101
41 Workflow between a traditional and an AQP data warehouse. 109
42 Sample Query. 110
43 Sample Query with accurary. 110
44 Example Scalar query for SPEAr. 115
45 Processing time of Storm and SPEAr on the DEC dataset. 128
46 Average window worker memory of Storm and SPEAr on the DEC dataset. . 129
47 Average and 95-percentile processing time. 130
48 Processing Time on GCM with different window sizes. 132
49 Processing time (nsec) on DEC between Storm, Storm with incremental processing, and SPEAr (without the incremental optimization). 134
50 Relative error per window on DEC. The red (flat) line indicates the user-defined acceptable error (i.e., 10% at 95% confidence). The blue line indicates the error achieved by SPEAr. 135
51 Percentage of total tuples processed by SPEAr. 137
52 The internal architecture of a worker thread in modern SPEs, with the ShedPart module. 147
53 Sample rate estimated by Equation 7.1 for 99% confidence. 156
54 Imbalance. 157
55 Maximum worker load. 158
56 Window runtime analysis for ShedPart on DEBS. 159
57 Max parallel step runtime on GCM. 160
58 Breakdown of time spent on each window, in terms of ShedPart and processing (GCM). 161
59 Design Space of techniques for adaptability techniques. 164

PREFACE

First and foremost, I want to express my appreciation to my academic advisors: Alexandros and Panos. Their guidance and support were crucial for the completion of this dissertation. Alexandros and Panos taught me patience, positive thinking, and "*focusing on the big picture*". Collaborating with them matured me both on a personal and a professional level.

Next, I would like to express my gratitude to the committee members: Jack R. Lange, and Andy Pavlo. Jack has been a source of insightful comments, which led to improving the quality of my thesis. Andy has been more than just a committee member: Not only would he treat me as a member of his research group, but he would also provide guidance and constructive comments to many of my thesis' projects.

Moreover, I would like to thank my dear friends, which I met through the PhD program: Anatoli Shein, Briand Djoko, Cory Thoma, Daniel Petrov, Rakan Alseghayer, and Salim Malakouti. I feel fortunate to have them around during stressful times. Furthermore, I would like to express my gratitude to all other graduate students of the Computer Science department, which made the uphill climb of the Doctorate degree less steep. Some honorable mentions: Adriano, Alireza, Angen, Brian, Debashis, Ekaterina, Kenrick, Longhao, Mohammad, Qiao, Vineet, Xiaoyu, Xiaoyu, Xiaozhong, Zhipeng.

Furthermore, I want to thank my friends, which I met during my time in Pittsburgh. I feel blessed that I met you: Angela J., Annia M., Antonis T., Antonis P., Christine S., Dimitris M., Elina K., Freddie O., Jak L., Katia L., Klea P., Molly L., Nikos L., Hank B., Panos B., Petty T., Sadaf T., Shadi S., Stephanie N., Steve K., Vangelis P.. Also, I want to thank my teammates from the University of Pittsburgh's Water Polo Club Team. Special appreciation and gratitude goes to Zheru L., who supported me during the hard times of this dissertation.

I dedicate this work to my parents, Nopi and Mpampis, and my brother Manolis. They have always been a pillar of support in my life. This Doctorate would not be possible without you. Thank you.

1.0 INTRODUCTION

Modern data-intensive applications are prevalent in a plethora of social and economic sectors, which require real-time processing of voluminous and velocious data streams. Those carry rapidly fluctuating data in content, and (sometimes) structure. Examples of such applications include high-frequency trading [120], social network analysis [40], targeted advertising [14], click-stream analysis [54], urban analytics [19], real-time data visualization [104, 26, 130], to name a few. The goal of those applications is to capture trends by identifying patterns in the data streams, and use the extracted knowledge for online, time-critical, decision support.

Traditional Database Management Systems (DBMSs) have been optimized either for Online Transaction Processing (OLTP) workloads, which aim at persisting ad-hoc concurrent transactions that read and write shared data, or for Online Analytical Processing (OLAP) workloads, which aim at answering complex queries fast [62, 117]. Both models are not fit for the aforementioned applications as they follow a "pull-based" processing model, which requires queries to pull data from persistent storage [8, 37]. OLTP and OLAP systems follow a *Human-Active*-DBMS-*Passive* approach (HADP) since all operations in the DBMS are triggered by user actions [33]. Stream processing has been deemed as a more appropriate processing model, since it follows a "push-based" approach for processing (i.e., DBMS-*Active-Human-Passive* model–DAHP). As a model, stream processing involves a static set of continuously-running queries, which are often called *Continuous Queries* (CQs), applied on evolving data streams of unbounded size (i.e., dynamically arriving and cannot be fully predicted). Those streams can carry updates of any form, based on the general *turnstile* model of execution [57].

Stream Processing Engines (SPEs) have been developed to cover the needs for this type

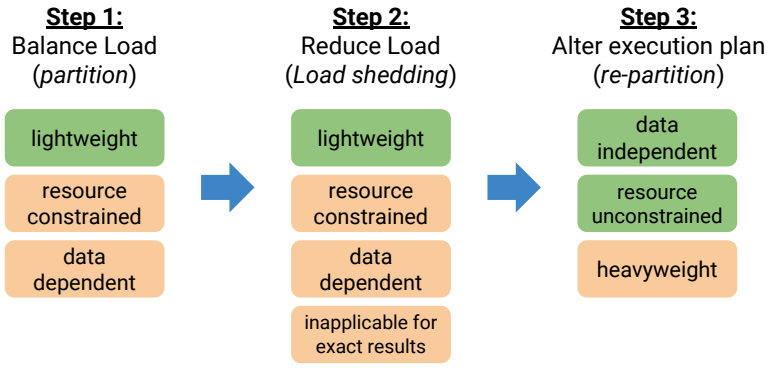


Figure 1: The actions taken by an SPE administrator.

of processing [25, 37, 8, 1, 7, 17, 138, 14, 97, 15, 36, 30, 131, 83]. Their design differs from DBMSs in that SPEs are optimized for continuous and timely execution of registered CQs, which are known prior to the beginning of execution. To cope with the input load and produce results fast, SPEs have their processing load constrained in size (i.e., RAM), and any type of secondary storage is excluded from the main processing pipeline. As a result, the notion of "windowing" is adopted by SPEs, which dictates that continuous execution produces a stateful operation's result on a window (i.e., snapshot) of the data [22, 18].

1.1 CHALLENGES

Even though the set of CQs is known, a major challenge for SPEs is their ability to provide the expected level of service during the whole lifetime of a CQ. This can be difficult because the exact resources needed for a given CQ at any point in time of its execution are not known at query submission time, which is when resources are provisioned. A CQ's resource needs are estimated based on historical information, without precise forecasting of future data characteristics (i.e., input rate, volume etc.). As a result, SPE administrators end up over-provisioning resources, which leads to high operational costs. Matters get exacerbated when

resources are not ample to cover the processing needs of a CQ. Under such circumstances, either data can get lost or results can be delayed. Ultimately, static resource allocation is precarious and can result in severe costs emanating from either loss of data or increased operational expenses.

In the past, three techniques have been developed to address the challenge of resource provisioning in SPEs. Those are *stream partitioning*, *load shedding*, and *stream re-partitioning* (or *scaling-out/-in*). The first aims at distributing load among processors as evenly as possible, the second selectively drops load during congestion periods, and the third involves reconfiguration of a CQ's execution plan in an online fashion[1]. These techniques can be classified based on their usage for *exact* and *approximate* processing scenarios: the former entails the production of an exact result, whereas the latter entails the production of an estimation of a result. Stream *partitioning* and *re-partitioning* are targeted for *exact*, whereas *load shedding* is aimed for *approximate* processing. Each one of these techniques comes with its individual merits and drawbacks. Hence, each one is better suited for different applications and for particular congestion circumstances. Figure 1 illustrates the sequence of steps followed by an SPE administrator, in terms of applying each technique (based on our empirical evidence), along with each technique's pros and cons. *Stream partitioning* and load shedding can be adapted without disrupting execution (hence "lightweight" in terms of impact in performance), but both are constrained by the available resources and their effectiveness depends on the underlying data. Therefore, both of them are applied first when the execution pipeline gets congested. Load shedding has the additional limitation that it might not be applicable to some use-cases that require *exact* results. The final step is *stream re-partitioning* that is able to overcome any type of congestion, but it requires temporarily suspending execution (hence "heavyweight" in terms of impact in performance).

1.1.1 Partitioning

Stream *partitioning* was introduced for SPEs that support distributed execution [37, 111]. With the appearance of multiple parallel processes (and multi-core CPUs) the need to bal-

[1]We use the term *stream re-partitioning* to describe changes in the execution plan for both static and dynamic resource allocation strategies.

ance load had arisen. In the beginning, stream *partitioning* algorithms distributed tuples in a load-agnostic fashion, which was borrowed from parallel DBMSs. Subsequently, more sophisticated techniques involved monitoring incoming streams and making informed decisions for balancing load [58, 98, 99, 108].

Adaptive stream *partitioning* is a lightweight process, since it does not dictate costly re-organization of a CQ's execution plan. Also, it can alleviate performance degradation emanating from skewed input [75]. However, stream *partitioning* solutions are limited by the underlying resources and the shape of input data. Existing stream *partitioning* solutions fail to incorporate costs imposed by distributed execution in their decision process, which harms their ability to balance load. As shown in Figure 1, stream *partitioning* is the first step for adapting an SPE's performance. Stream *partitioning* is capable of improving the performance on skewed input, but it is resource constrained, and its efficacy depends on the underlying data.

1.1.2 Load Shedding

Load shedding was proposed to reduce the volume of data being processed, and is applied when input temporarily increases [124, 45, 24, 119, 125, 123, 96]. In practice, an SPE monitors the processing time, and if a users' Service Level Objectives (SLOs) are violated, it drops input tuples based on a calculated shed rate. By dropping tuples while the *delay target* is met, the accuracy of the result decreases and the main challenge for load shedding is to strike a balance between performance and accuracy.

This technique is best fitted for short-term input spikes and for CQs that can sustain accuracy drop in the results (i.e., *approximate* processing). As a result, load shedding is inapplicable to workloads that require *exact* processing, and is to no avail when data are not fit for approximation. To make matters worse, existing load shedding techniques make impractical assumptions about stream processing (e.g., stationary data, or known tuples' utilities), which constitute load shedding futile for current use-cases. Figure 1 indicates that load shedding is applied after the efforts for addressing load with stream *partitioning* have been tried. Load shedding is a lightweight process, since it does not require suspending a

CQ's execution. However, it suffers from similar shortcomings with *partitioning*. On top of that, load shedding might be deemed inapplicable to use-cases where *exact* processing is required.

1.1.3 Re-partitioning (Scale-out)

With the evolution of Cloud Computing and the ability to provision additional resources on-demand, *stream re-partitioning* has been employed as a better alternative to load shedding for handling long-lasting input spikes [111, 141, 110, 34, 63]. The main idea of *re-partitioning* is to (i) add additional processing units, without severely disrupting on-going execution, and (ii) release them when they are no longer needed. Furthermore, *re-partitioning* has been used for re-organizing a CQ's execution plan on static resource allocations, to alleviate the performance impacts of executing multiple processes on the same machine.

Stream *re-partitioning* involves state migration, stream re-direction, and maintaining data consistency, which make it useful for long-lasting increases in input load. In addition, existing techniques require suspending execution temporarily [111, 34]. Thus, stream *re-partitioning* is a costly process, and existing techniques are not designed to leverage stream processing semantics effectively. Consequently, an SPE's performance deteriorates momentarily during re-configuration. Figure 1 depicts that *re-partitioning* is the ultimate step of action in order to improve an SPE's performance. Despite *re-partitioning*'s ability to improve performance, it remains a heavyweight operation.

1.2 OUR APPROACH

Each technique cannot individually cover the adaptability needs of a SPE by itself, as they all fall short under specific conditions. For instance, stream *partitioning* is able to only handle workloads that can be accommodated by existing infrastructure; load shedding is applicable for short-term spikes on workloads that allow *approximate* processing; stream *re-partitioning* involves heavy re-organizations of executing components. Total adaptability

requires the competence to identify situations under which each technique can be applied. On top of that, combining two (or more) techniques is imperative to compensate for the weaknesses associated with each one and can further improve the performance of an SPE. Therefore, we state that stream processing needs to be approached in a *procrustean* fashion[2], in terms of the manner and the volume that data are processed.

In this dissertation, we study existing SPEs and modern stream workloads imposed on them, with a focus on *stateful* operations[3]. Our investigation reveals that existing solutions fall short, and do not fit well with current operational circumstances. To this end, we develop adaptive algorithms and protocols that can be applied in both *exact* and *approximate* stream processing situations. Our contributions leverage the design of modern SPEs to improve performance and maintain high results' accuracy at a low cost.

1.2.1 Exact Stream Processing

In the context of *exact* processing, we developed a novel holistic model for stream *partitioning*, which incorporates the associated costs of distributed execution in the decision process. Based on this model, we invented a family of stream *partitioning* algorithms, which materialize it in a heuristic fashion. Our experimental evaluation shows that our algorithms lead to increased throughput and reduced processing time in real workloads. Turning to stream *re-partitioning*, we invented a novel CQ migration protocol that does not cause any temporal performance degradation, and can migrate CQs with complex and/or nested window semantics. Previous work on migration protocols did not leverage the fact that in stream state appears in the form of windows, which expire as execution progresses. Our protocol is taking advantage of this peculiarity of stream processing, and is able to migrate a CQ without having to migrate any state.

[2] *Procrustes* was a notorious bandit in Ancient Greece, who would adapt his sinister plans based on his victims' dimensions [128].

[3] Stateless operations can be trivially handled, since they do not involve any state. Therefore, we do not focus on them in this dissertation

1.2.2 Approximate Stream Processing

Previous load shedding techniques made impractical assumptions that would either assume that prior knowledge for prioritizing data for shedding exists, or that historical information of input streams is stationary and can be used during shedding to maintain high accuracy. We identified the importance of a window's concept and introduce an algorithm that treats concept-drift as an inherent characteristic of data streams. This algorithm exploits modern SPEs execution model, and improves results' accuracy at no additional cost. Furthermore, we brought *Approximate Query Processing* (AQP) in SPEs, by providing approximate results with bounded error. The motivation for this work is to accelerate processing time without compromising the quality of results. In detail, we developed SPEAr, which a prototype SPE able to detect opportunities for accelerating processing with a bounded degradation in accuracy.

1.2.3 Combination of methods

Our research on all three adaptation techniques revealed that no single technique can solve all possible overload situations. Adaptive stream *partitioning* is unable to handle extremely skewed workloads, without having to migrate state (i.e., costly operation). Similarly, load shedding can not operate if either the application or the data do not allow for *approximate* stream processing. Finally, stream *re-partitioning* comes with a high cost and can cause missing *delay targets* (i.e., violation of SLOs). To this end, we explored the combination of load shedding and stream *partitioning*, which further improves load allocation and reduces processing time.

1.3 RESEARCH CONTRIBUTIONS

1. A family of stream *partitioning* algorithms that aim to balance load among multiple workers. We present a novel model for estimating the cost of a stateful operation, by including the aggregation overhead. This work has been published as [75] and we present

it in Chapter 3.

2. The design of *Synefo*, an SPE built on top of Apache Storm with the ability to alter the resources of an active CQ. This work has been published as [78]. In addition, we contribute the design of UniMiCo, which is a state migration protocol with zero state transfer overhead. This work has been published as [107]. Both *Synefo* and UniMiCo are presented in Chapter 4.
3. The design and implementation of a novel load shedding method, which prioritizes tuples for shedding based on a window's concept. This work has been published as [76] and we present it in Chapter 5.
4. The design of SPEAr, which is a prototype SPE built on top of Apache Storm. Given a user's accuracy specification (in the form of error and confidence), SPEAr is able to identify opportunities for accelerating processing, by producing an approximate result of bounded error. This work is presented in Chapter 6.
5. The design of ShedPart, which is a complementary module for stream *partitioning*, that trades accuracy for load balance. ShedPart selectively drops parts of the input in order to achieve an even allocation of work, while the result appears with a bounded error. This work is presented in Chapter 7.

1.4 ROAD MAP

To set the stage for this dissertation, in Chapter 2 we present the background and the prerequisite information for our work. In Chapter 3 we present our contributions in stream *partitioning* for *exact processing*. Next, in Chapter 4 we present our work on stream *re-partitioning*, followed by Chapter 5 with our work on load shedding, which aims *approximate* processing. In Chapter 6 we present our work on *approximate* processing with bounded accuracy, which consists of the design of our prototype SPE SPEAr. Then, in Chapter 7 we present our work on the combination of stream *partitioning* and load shedding, which targets *approximate* processing with bounded accuracy. Finally, we conclude this dissertation in Chapter 8 with our concluding remarks and future research directions.

2.0 BACKGROUND

In this chapter, we establish the theoretical foundations, important details, and background information of this dissertation. First, we present the SPE system model (Section 2.1), the query along with the data model (Section 2.2), and the execution pipeline of SPEs (Section 2.3). Then, we provide background information on internal mechanisms of modern SPEs for stateful window processing (Section 2.4), which is our focus on this dissertation since it poses a plethora of challenges for SPEs. Next, we offer background information and prerequisites for approximate processing (Section 2.5). Finally, we conclude this chapter with general information on the real workloads we employ in our experiments (Section 2.6.1), and background information on Apache Storm, which we use as the "fabric" for building many of our prototypes (Section 2.6.2). Table 1 summarizes the symbols that are introduced in this chapter.

2.1 SYSTEM MODEL

Despite the fact that the need for CQs and differential execution first appeared in the 1990s [126, 87, 88], the first system that covered stream processing partially appeared much later, in the early 2000s [41]. At the time of writing of this dissertation, SPEs' design has gone through three generations each one introducing pivotal changes in design.

In the beginning, SPEs appeared as single-node systems with serial execution runtimes. The most popular prototypes were STREAM [25], Aurora [8], NiagaraCQ [41], PSoup [38], and AQSIOS [1]. Those featured a monolithic design, akin to a DBMS, and carried a query parser, a memory manager, and Input/Output modules for network/storage. First genera-

Table 1: Model Symbol Overview

$\mathcal{V}$	number of workers
S_i	input streams $1 \leq i \leq N$
$\mathcal{X}_i = (\tau, k, p)$	schema of S_i
$e_{\mathcal{X}_i}$	tuple of S_i
$\mathcal{W}_{r,s} : S_i \rightarrow \{S_i^1, \ldots, S_i^w\}$	window definition function for S_i with range r and slide s
$P : S_i^w \rightarrow \{L_{S_i^w}^v, \text{for } 1 \leq v \leq \mathcal{V}\}$	*partition* function for window S_i^w
$\epsilon : R_w, \hat{R}_w \rightarrow [0, 100]$	error function between R_w and approximation $\hat{R}_w$
α	confidence level

tion SPEs featured a serial execution engine, which required the existence of a centralized scheduler [23], and multiple CQs shared the same CPU [20]. Some SPEs carried a workload manager and supported execution of operations with different priorities [106]. A unique characteristic of the first generation of SPEs was operator sharing, which allowed multiple queries to share a common operation on the same stream(s) [112]. SPEs featured an optimization process, whose goal was to create groups of CQs with matching semantics [41].

The second generation of SPEs focused on distributing processing to (mainly) overcome the limitations of a single-node environment (i.e., main memory, processing speeds). A CQ would be broken down to a series of operators, which would be spread out in a cluster of servers, in order to scale processing. Borealis [7] was the distributed version of Aurora and allowed a user to scale-out queries by spreading their operators to multiple machines. In a similar fashion, TelegraphCQ [37] allowed parallel and distributed execution of continuous queries. IBM's System S [17] and StreamCloud [63] expanded the processing and memory capacity of a SPE and were able to scale processing further, by adding more operators in specific stages of the execution plan [100, 101].

The current generation of SPEs features distributed execution with a less monolithic design, in order to achieve higher scales. In essence, SPEs of this generation resemble systems

like MapReduce [47], with a focus on low-latency processing and data that can fit in main-memory. The SPE offers a small set of utilities, like the transfer layer and the execution layer, and leaves the memory management and the processing logic to the user. Storm [131] (along with its evolution Heron [83]) offers the transfer layer and the processing fabric, and the application logic is responsible for the rest. Spark Streaming [138] simulates stream processing through micro batches and offers strict delivery semantics. A number of other systems have became popular, such as Samza [4], Kafka Streams [6], Trill [36]. Two other systems, Flink [30] and Apex [3], focus on providing a unified environment for both stream and batch processing. Those SPEs are designed to operate on (public) clusters and on commodity hardware (i.e., scale-out architectures). Recently, SPEs that operate as a Service have appeared from Cloud Service Vendors [2, 5, 15].

In this work, we focus on the current third generation of SPEs due to their ability to scale. Their modular design allows for scalability and fits well with Cloud Infrastructures. However, all of our work can be applied on any type of distributed SPE.

2.2 QUERY MODEL

A CQ, represented as $\mathcal{Q}$, is submitted to an SPE in either declarative (i.e., SQL-like syntax) or imperative/functional form, consists of data source definitions and operators, which operate serially to produce the expected result.

A CQ's data sources can be input streams, or static relations [18]. $\mathcal{Q}$'s input streams are represented by S_i, where $1 \leq i \leq N$ (N is the number of input streams). Each S_i is a sequence of tuples $e_{\mathcal{X}_i}$ (e stands for event), with a predefined schema $\mathcal{X}_i$. Often, the schema presents normalized fields, but in some use-cases input tuples can hold nested fields (i.e., records from JSON files [85]). From this point on, we will continue our model's presentation based on a single input stream S_i. However, without any loss of generality our model can accommodate multiple streams as well.

$\mathcal{Q}$ represents a series of transformations, which are represented by operators. Operators are characterized based on their need to maintain state in order to apply their transformation.

```
Select route, AVG(fare)
From Rides
    [Range 15 Minutes,
    Slide 5 Minutes]
With Time time
Group By route
Order By AVG(fare)
Limit 10;
```

a: CQL

```
query = rides
      .time(x -> x.time)
      .windowSize(15, MINUTES)
      .windowSlide(5, MINUTES)
      .mean(x -> x.fare)
      .group(x -> x.route)
      .order()
      .limit(10);
```

b: Functional

Figure 2: Example Query

Stateless operators carry logic that does not require state and act based on a $e_{\mathcal{X}_i}$'s attribute values. For instance, *stateless* operators are filter (relational select σ), and map operations (e.g., relational rename ρ, project π), which operate based on the information of a single input tuple. *Stateful* operators require state to apply their transformation, and act based on a subset of $e_{\mathcal{X}_i}$ from S_i. Examples of *stateful* operators are aggregate operations (e.g., `sum`, `mean`), relational joins (i.e., $\bowtie$), cartesian products ($\times$), set operations (i.e., relational union, intersection, and difference), or *user-defined* aggregate operations (UDAs). In the past, *stateful* operations have appeared as iterative processes [97], which require to maintain state among different invocations. In this dissertation, our focus is going to be on *stateful* operators as they produce a plethora of challenges for existing SPEs. Due to the unbounded size of an input stream, a *stateful* operator's state is constrained to a subset of past tuples, which is called a *window*.

Often, $\mathcal{Q}$ will have one or more window definitions. The notion of a window definition $\mathcal{W}_{r,s}$ is to support *stateful* operators, which are applied to a group of tuples when $\mathcal{W}_{r,s}$'s conditions have been met (i.e., a *stateful* operator's process is triggered). A logical window definition can be represented as a function $\mathcal{W}_{r,s} : S_i \rightarrow \{S_i^1, \ldots, S_i^w\}$, where $w \rightarrow \infty$. Each S_i^w represents the tuples of S_i that belong to window w according to $\mathcal{W}$ (i.e., S_i^w is the window w). In $\mathcal{W}_{r,s}$, r is the *range* of the window, which indicates the size of a window, either in

terms of tuple count, or time duration. Early SPEs, like STREAM [25] and Aurora [8], introduced two types of window trigger conditions: count- (or tuple-) and time-based. The former produces a result when the desired number of tuples are available in the input buffer. In this case, $\mathcal{W}$'s r denotes a number of tuples, and when r tuples have accumulated, the *stateful* operation associated with $\mathcal{W}$ is evaluated. The latter, produces a result when a user-defined time condition has been met. In this case, $\mathcal{W}$'s r denotes a time unit, which indicates that if a window of r duration has been established, the *stateful* operation associated with $\mathcal{W}$ is evaluated. For time-based windows, the notion of time to establish a window can be based either by the SPE (called "processing time" [14]), or by a specific attribute value of $e_{\mathcal{X}_i}$ (called "event time" [14]).

The window definition $\mathcal{W}_{r,s}$ features a *slide* s, which indicates the progression step of a window. When the slide $s = r$, two consecutive windows do not overlap. Windows with this characteristic are called *tumbling* windows, and each tuple of S_i is assigned to a single window, When the slide $s < r$, two consecutive windows overlap with each other, and each tuple of S_i contributes to $\lceil \frac{r}{s} \rceil$ consecutive windows. This type of windows are called *sliding*. Recently, a third type of windows named session-based have appeared [32], which carry more complex window completion requirements that can not be expressed just by time or number of tuples.

Figure 2 presents an example query in both declarative (Figure 2a) and functional notation (Figure 2b). The example query calculates the arithmetic mean of fares per *route* on a sliding window of $r = 15$ minutes, with a slide of $s = 5$ five minutes (i.e., $\mathcal{W}_{15,5}$). This CQ has access to a single source stream named "*rides*" (i.e., $N = 1$ and $S_1 =$ rides). The query demonstrated in Figure 2a is in CQL–like [18] notation [1], and the query of Figure 2b is phrased using a functional notation (similar to one used in Spark Streaming, Flink, and Heron). Each tuple of the *rides* stream has the following schema: $\mathcal{X}_{\text{rides}} =$(*time*, *route*, *fare*). For each window, the query produces the ten routes with the highest arithmetic mean of fares. The main difference between CQL and the query syntax of Figure 2a is the additional specification of the time field. Most of the CQs that we will present in this dissertation feature *event time* processing. In the example query of Figure 2, the field *time* is used to

[1] CQL did not support explicit usage of event-time processing as appears in Figure 2a

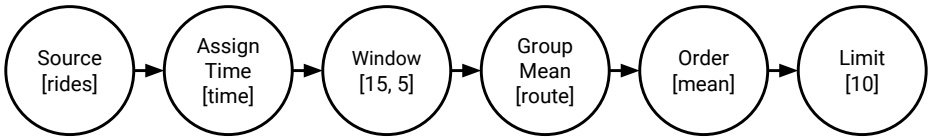


Figure 3: Logical Plan for CQ of Figure 2.

assign tuples to windows.

2.3 EXECUTION PLAN GENERATION

A CQ is submitted to an SPE in the forms that are presented in Figure 2. In the event that a CQ is phrased using a CQL-like notation (e.g., Figure 2a), an SPE's toolset is able to transform it to an execution topology (e.g., Flink, Heron, and Spark Streaming support this functionality). Next, an SPE will perform the following steps: (a) logical plan creation, and (b) physical plan formation.

2.3.1 Logical Plan

The logical plan of a CQ is a *Directed Acyclic Graph* (DAG), similar to an evaluation tree of a DBMS. The main difference lies in that a CQ's logical plan does not need to have a single output. Figure 3 presents a possible evaluation plan created by an SPE for the CQ presented in Figure 2. The logical plan presented in Figure 3 is a single branch tree because all nodes receive a single input. Each vertice (node) of the logical plan denotes an operator of a CQ, and each edge a data transfer. In case there are operators with more than one inputs, such as relational joins, then the logical plan looks like a tree (e.g., the query plan depicted in Figure 4). The left-most node of the execution plan represents a data source (e.g., an input stream). The right-most node represents the last operator that will be performed and it is often followed by a consumer node, which can either be an external

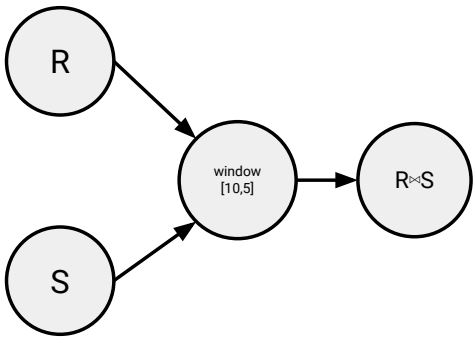


Figure 4: Example query plan for a CQ with a join.

storage system, or an ingress operator by another processing pipeline. Intermediate nodes represent operators imposed on flowing tuples and can be either *stateless* or *stateful*. As we have mentioned previously, our focal point is limited on issues related to *stateful* processing. In the plan depicted in Figure 3 the "Assign Time" operator is a *stateless* operator, since it acts as a map operation where each input tuple is augmented with a time field. However, the "Window" and the "Group Mean" operators are *stateful*, since they require the window semantics to be triggered in order to process tuples.

2.3.2 Physical Execution Plan

As mentioned in Section 2.1, the target system of this dissertation is a distributed SPE, which scales processing by utilizing multiple CPUs on different servers (or multiple cores on a single server with multiple CPUs). To this end, the SPE will turn the logical plan (i.e., Figure 3) to a physical execution plan. This process entails instantiating execution workers for each operator (i.e., threads, processes etc.), creating the physical communication channels among different stages of execution (i.e., I/O buffers, network sockets, etc.), initiating monitoring and reliability mechanisms, and commencing the synchronization components. Depending on the available resources for $\mathcal{Q}$, the SPE will produce a different physical execution plan. For instance, the parallelism of a *stateful* operator relies on the number ($\mathcal{V}$) of workers assigned to it. Each worker can be a machine, a process, a thread, or an actor, depending on the

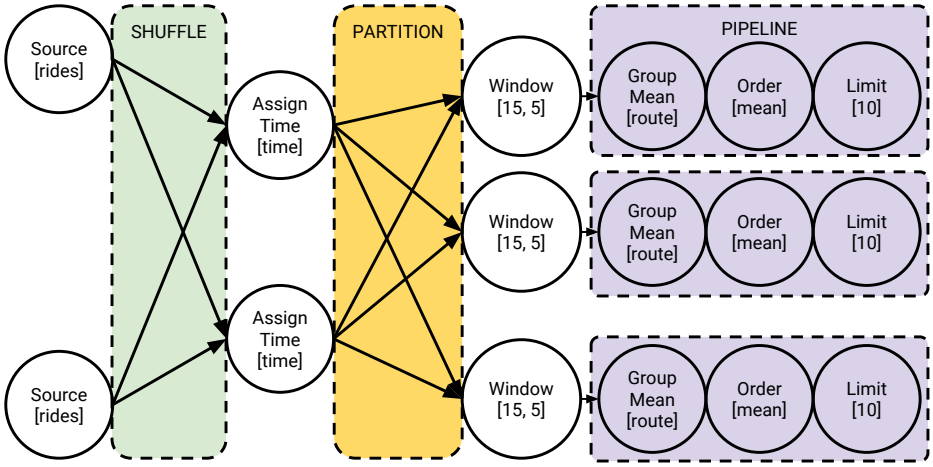


Figure 5: Physical Execution Plan for CQ of Figure 2.

parallelization granularity allowed by an SPE.

Figure 5 shows a possible execution plan when 10 workers are available. To this end, the SPE assigns two workers for ingress operations (i.e., source), two for timestamp assignment, three for windowing, and three for the rest of the operators. In most SPEs (e.g., Storm, Heron, Spark, and Flink), the user decides on the number of workers assigned to each operation. In Figure 5 three different distribution policies are depicted, which are offered by most SPEs. The *Shuffle* operation is performed when input tuples are sent to downstream operators in a balanced fashion (i.e., round-robin). The *Partition* operation is when the data need to be either collocated or distributed in a particular fashion. Finally, *Pipeline* (or *operator fusion*) is performed when consecutive operators can be performed on the same thread, to avoid unnecessary network hops and serialization.

Partition can be modeled as a function that takes a subsequence S_i^w and produces another sequence of equal length that indicates the worker to which each $e_{\mathcal{X}_i}^w$ is going to be sent ($e_{\mathcal{X}_i}^w$ indicates a tuple $e_{\mathcal{X}_i}$ belonging to window S_i^w). In other words, *Partition* is a function $P : S_i^w \rightarrow \{L_{S_i^w}^v, \text{for } 1 \leq v \leq \mathcal{V}\}$. Each $L_{S_i^w}^v$ denotes a sub-sequence of S_i^w and is the part sent to worker v for processing. $\mathcal{V}$ represents an SPE's parallelism degree for a particular operator and is materialized by $\mathcal{V}$ workers, which are responsible for processing the (partial)

result of a window w. In the example query of Figure 2, a partition step is required before the "Window" operations, if tuples of the same *routes* are (preferably) collocated. In this case, the *partition* function is hashing tuples based on the value of the *route* attribute.

An $e_{\mathcal{X}_i}$ can be represented as a triplet $(\tau_{\mathcal{X}_i}, k_{\mathcal{X}_i}, p_{\mathcal{X}_i})$, where $\tau_{\mathcal{X}_i}$ is the attribute responsible for ordering tuples in S_i and is used to assign each $e_{\mathcal{X}_i}$ to a logical window (either time- or count-based), $k_{\mathcal{X}_i} \subset \{\mathcal{X}_i - \tau_{\mathcal{X}_i}\}$ are the attributes, which identify a tuple and are utilized in the partition process, and $p_{\mathcal{X}_i} \subset \{\mathcal{X}_i - (\tau_{\mathcal{X}_i} + k_{\mathcal{X}_i})\}$ are the remaining attributes, which comprise $e_{\mathcal{X}_i}$'s payload. Often, those appear in predicates, projection lists, or are used by aggregate functions. In the example query of Figure 2, $\tau_{\mathcal{X}_{rides}} =$ (time), $k_{\mathcal{X}_{rides}} =$ (route), and $p_{\mathcal{X}_{rides}} =$ (fare). After the physical execution plan is established by an SPE, the execution plan is materialized by the SPE's scheduler, and processing initiates.

2.4 STATEFUL EXECUTION IN A SPE

Stateful operators are defined over a window, and SPEs are designed with internal mechanisms to identify when a window (i.e., a subset of tuples) is ready for processing. Those internal mechanisms consist of an *eviction* and a *trigger* module. The former is responsible for identifying tuples that are fully processed and can be safely discarded; the latter is responsible for identifying the conditions indicating that a window is complete and ready to be processed.

In detail, we present the actions taken for *stateful* processing of time-based windows, which are similar for both count and session based windows. A *stateful* operator needs to know when a window is complete, and it is time to commence processing. To this end, *stateful* processing in current SPEs makes use of *Watermark* tuples. These indicate when a window is complete and it is safe to stage its tuples for processing [14, 15, 30, 36]. Watermarks are tuples having only a timestamp (τ_W), and indicate that all tuples with $\tau \leq \tau_W$ have been observed by an SPE. Concretely, a watermark acts as a "contract" that a *stateful* operator will have access to all tuples up to a particular point in time, and that no late tuples will be

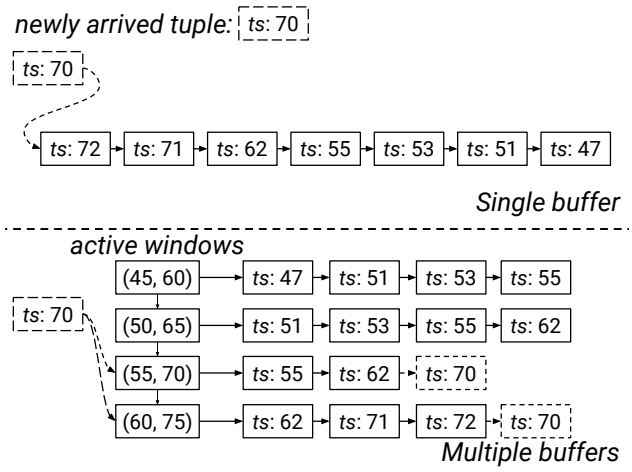


Figure 6: Tuple arrival: *Single* and *Multiple* Buffer(s) design operations.

encountered [2]. In addition, the watermark mechanism is used as a checkpointing mechanism to guarantee exactly-once processing semantics [31]. Watermarks are produced by data source operators periodically.

A *stateful* operator's behavior can be described in two phases: (a) tuple arrival, and (b) watermark arrival. At tuple arrival, a *stateful* operator needs to store the tuple until the time arrives when a window is complete. At watermark arrival, a *stateful* operator needs to prepare the window for processing, and apply the operator's logic on the tuples that correspond to the window.

2.4.1 Tuple Arrival

When a tuple arrives at a stateful operator, it is stored in the operator's internal buffer(s). Figure 6 illustrates the two widely used designs among existing SPEs: (a) *Single Buffer* dictates that all tuples are stored in a single buffer based on their order of appearance. This design is adopted by Storm and Heron and its advantage is that each tuple is stored only once. (b) *Multiple Buffer* design dictates that a copy of the input tuple is stored to a buffer for each of the windows that it participates in. Flink uses the *Multiple Buffers* design and

[2]Out-of-order processing is an orthogonal topic to this dissertation.

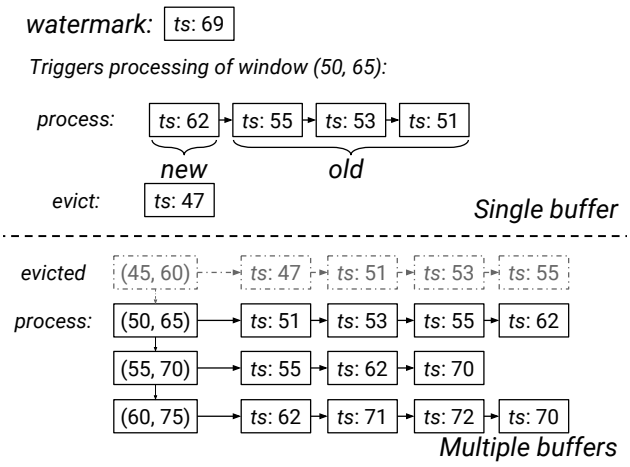


Figure 7: Watermark arrival: *Single* and *Multiple* Buffer(s) design operations.

its merit is that when the time comes, the window is ready to be handed for processing.

In the example of Figure 6, a tuple with timestamp 70 arrives and participates in windows: $(55, 70)$, $(60, 75)$, $(65, 80)$, and $(70, 85)$. With the *Single Buffer* design, the tuple is appended to the buffer and the operator waits until the next one arrives. In contrast, with the *Multiple Buffers* design, the tuple's timestamp is extracted, and the windows that it belongs to are determined. Given a tuple's timestamp τ and a window specification $\mathcal{W}_{r,s}$, the starting timestamp of the latest window that τ participates in (τ^l_{start}) is calculated by:

$$\tau^l_{start} = \tau - (\tau + s) \mod s \tag{2.1}$$

In Equation 2.1, s is the window slide taken from $\mathcal{W}_{r,s}$. Then, the rest of the windows are found iteratively. A copy of the tuple is stored in each window's buffer.

2.4.2 Watermark Arrival

Every time a *stateful* operator receives a watermark, it triggers the processing of all affected windows. At the arrival of the watermark, the window operator has to prepare all non-processed windows ending on (or before) τ_W. The operator extracts the tuples for each window, and stages them for processing. With the *Single Buffer* design, the operator scans

```
query = rides
      .time(x -> x.time)
      .windowSize(15, MINUTES)
      .windowSlide(5, MINUTES)
      .mean(x -> x.fare)
      .group(x -> x.route)
      .error(10) // percentage
      .confidence(95); // percentage
```

Figure 8: Query with an accuracy specification.

the buffer and gathers all the tuples belonging to a particular window. At the same time, it evicts expired tuples (those that they carry a timestamp earlier than the start of the current window). Optionally, a *stateful* operator might perform an additional scan on the window to separate tuples processed for the first time. This scan is done by SPEs that offer incremental processing (e.g., Storm, Heron) [20, 84, 32]. Turning to the *Multiple Buffers* design, the operator picks the appropriate window, and passes it to the processing logic. Figure 7 presents the window preparation process when the watermark with timestamp 69 arrives. With the *Single Buffer* design, the operator scans its buffer, collects the tuples of window $(50, 65)$, and marks tuple 47 for eviction. Finally, the operator sends tuples for processing. Conversely, with the *Multiple Buffers* design, the operator simply picks the buffer for the corresponding window and sends the tuples for processing.

2.5 RESULT APPROXIMATION IN STREAM PROCESSING

Some real-time applications do not require an *exact* result and can sustain a drop in accuracy by approximating an *exact* answer. Often, such applications have strict response time requirements to offer an interactive experience [89, 44]. Examples of such applications can be found in real-time visualizations [139, 104], online decision making [109], etc. In these circumstances, accuracy of the result can be sacrificed for performance, as long as there is an implicit level of accuracy to bound the error.

This accuracy can be controlled by the user when they submit a $\mathcal{Q}$ with an accuracy specification. Often, the specification consists of an accuracy measure ϵ and a confidence level α (i.e., the probability that the error will be less or equal to ϵ). For instance, the CQ of Figure 2 can carry an approximation specification like the one illustrated in Figure 8. In this dissertation, we will use "bounded error" and "accuracy requirement" (or "accuracy guarantee") to define the specification that quantifies the amount by which a result differs from an actual value.

The query of Figure 8 features two additional clauses compared to the CQ of Figure 2b. This query model is similar to the one used in *Approximate Query Processing* (AQP) systems such as Aqua [12], Strat [39], BlinkDB [13], Quickr [73], Idea [55], and VerdictDB [105]. It comes with an error and confidence specification. Specifically, ϵ is defined with the `.error()` command, and the probability that $\epsilon \leq 10\%$ with the `.confidence()` (i.e., $\Pr(\epsilon \leq .1) \approx .95$) [42, 69]. In this particular example, the error can be a relative or an absolute error. However, depending on $\mathcal{Q}$, the error specification varies [45, 119]. In general, for *stateful* operators, we will represent the actual window result as R_w, and the approximate window result as $\hat{R}_w$.

2.5.1 Error Metrics

An approximate window result $\hat{R}_w$ entails that only a subset of the data is going to be used for its estimation. Thus, to measure the difference between R_w and $\hat{R}_w$, the user provides a metric $\epsilon : R_w, \hat{R}_w \rightarrow e$, which is a function to calculate the accuracy (or the error in an equivalent fashion). Depending on the *stateful* operation, different functions are used for calculating ϵ. In this section, we present some of the functions that we used in this work to calculate ϵ.

Our focus is mainly on *stateful* operators that carry aggregate operations (e.g., `mean`, `sum`), which are widely used in SPEs. Those transform a window of input tuples to either a scalar value or a vector of values (in case there is a `group by` argument). First, an aggregate operation extracts a number of attributes from $e_{\mathcal{X}_i}$ and feeds them to a transformation function to produce R_w. In case there is a group statement in a CQ, a result is produced for each distinct group.

Aggregate operations are categorized based on their ability to be distributed [65, 135]. *Distributive* aggregate operations include functions whose computation can be "distributed" or partially calculated and combined. For example, `sum` and `count` are two paradigms of *distributive* aggregate examples. *Algebraic* aggregate operations are functions which can be calculated by combining distributive aggregation results. For example, the average (i.e., `mean`) is an *algebraic* aggregate operation since it entails the calculation of both `sum` and `count`. Finally, *holistic* aggregate operations are functions whose computation requires processing all data at once. A widely used holistic aggregate operation is the `median`, since it requires all tuples of a window in sorted order. Below, we present the most widely used error metrics for the different types of aggregate operations.

2.5.1.1 Algebraic and Distributive: Often, for aggregate operations and UDAs the most popular ϵ metric is the relative error (Equation 2.2):

$$\epsilon_w = \frac{|R_w - \hat{R}_w|}{\max(R_w, \hat{R}_w)} \tag{2.2}$$

and rarely the absolute error (Equation 2.3):

$$\epsilon_w = |R_w - \hat{R}_w| \tag{2.3}$$

Sometimes, an aggregate operation of a UDA requires a result for each group (distinct key) in S_i^w (i.e., it carries a `group by` clause). Often, the attributes that form a group for are the partitioning key k (see Section 2.3.2). The number of distinct groups (or keys) in S_i^w will be presented as $\|S_i^w\|$ throughout this dissertation. The error between R_w and $\hat{R}_w$ is calculated by combining all groups errors. First, the error for each group $g \in \|S_i^w\|$ is calculated (i.e., e_w^g). Then, the error among all groups is aggregated using one of the following functions [11]:

$$\epsilon_\infty^w = \max_{1 \le g \le \|S_i^w\|} \epsilon_w^g \tag{2.4}$$

$$\epsilon_{L1}^w = \frac{1}{\|S_i^w\|} \sum_{g=1}^{\|S_i^w\|} \epsilon_w^g \tag{2.5}$$

$$\epsilon_{L2}^w = \sqrt{\frac{1}{\|S_i^w\|} \sum_{g=1}^{\|S_i^w\|} (\epsilon_w^g)^2} \tag{2.6}$$

Equation 2.4 assigns ϵ the maximum error among groups. Equation 2.5 returns the mean error among ϵ_w^g, and Equation 2.6 returns the square root of the mean of squares.

2.5.1.2 Holistic: As far as holistic aggregate operations and UDAs are concerned, different error metrics have been previously used. Quantile statistics are very popular and use a different metric for measuring error. Specifically, a ϕ-quantile, for $\phi \in [0, 1]$, is defined to be the element in position $\lceil \phi |S_i^w| \rceil$ in the sorted sequence of S_i^w. For $\phi = 0.5$ the quantile is equivalent to the median. An element is said to be an ϵ-approximate ϕ-quantile if its rank is between $\lceil (\phi - \epsilon)|S_i^w| \rceil$ and $\lceil (\phi + \epsilon)|S_i^w| \rceil$ [91].

2.6 EXPERIMENTAL INFRASTRUCTURE

In this dissertation, in order to evaluate our developed algorithms, methods, and protocols, we conducted experiments with synthetic and real-world datasets. Below, we present general details of the real workloads that we employ in our evaluations. Those details can be used by the reader as a reference for additional information on the workloads. In addition to home grown simulations, we employed Apache Storm, which is a widely used open-source SPE, to test our contributions in most of our experiments.

2.6.1 Real-world datasets

Below, we present general details of the real-world datasets and workloads that we use in our experimental evaluation. We offer additional details for each dataset in the experimental evaluation in which each one is used.

TPC-H (TPCH): Despite the fact that TPCH is a benchmark targeted for measuring the performance of OLAP DBMSs, it has been extensively used for throughput-oriented streaming scenarios [50, 102, 36, 30, 40]. This benchmark comes with a data generation tool (we used v2.17) and a list of 22 queries designed to measure the performance of a DBMS. TPCH is ideal for measuring throughput of complex CQs on very large windows. We

generated TPCH data using the `dbgen` tool that comes with it using a *scale factor* of 10 (for a total of 10GB of raw data).

ACM DEBS 2015 Grand Challenge (DEBS): This workload features rides' data accumulated from a New York Taxi Company. DEBS totals 32GB in raw size, and comes with two sliding window queries [71]: the Top-10 most frequent routes query (Query 1), and (ii) the Top-10 most profitable areas of NY (Query 2). DEBS is ideal for measuring window processing time of an SPE, and its two queries offer group numbers that can potentially range from 62.5 thousand to 8.1 million. The average window size is $\approx$ 10 thousand tuples.

Google Cluster Monitoring (GCM): This data set contains execution traces from one of Google's cluster management software systems [61]. This dataset is about 40GB in size, but we have used only data from the Task Events table. In the past, this dataset has been used for sliding window CQs [82].

DEC Network Monitoring Dataset (DEC): This dataset is the smallest real dataset in raw size that we use (175MB). Previously, it has been used in stream processing scenarios [24] and consists of network packets monitored over an hour in the DEC Corporation. We used only a single CQ for this dataset, which calculates the average network packet size on a sliding window of 45 seconds, with a slide of 15 seconds. DEC is an ideal dataset for measuring window processing time on scalar aggregate operations.

2.6.2 Storm Runtime

In this section, we present a brief overview of Storm's runtime [131], which is an open-source SPE widely used in both academia and enterprise. Our focus is on the core components of Storm's engine, required by the reader to better comprehend the technical details of our work. In order to demonstrate CQ formation for Storm (up to version 1.2) we illustrate the user code (in Java) in Listing 2.1, which illustrates a subset of the transformations presented in the CQ of Figure 2 (i.e., without the `order` and the `limit` statements).

Storm uses a *Spout* to define a data stream source, and a *Bolt* to define a transformation operation. In the code of Listing 2.1, a `RideSpout` object is instantiated to be the entry point of *ride* tuples in the topology (Line 3) and operates as a data source. The `RideSpout` object

Listing 2.1: Part of the query of Figure 2 for Storm.

```
1  long size = TimeUnit.MINUTES.toMillis(15);
2  long slide = TimeUnit.MINUTES.toMillis(5);
3  RideSpout spout = new RideSpout();
4  AverageFare avg = new AverageFare();
5  TopologyBuilder builder = new TopologyBuilder();
6  builder.setSpout("rides", spout, 1);
7  builder.setBolt("avg-fare", avg
8      .withTimestampExtractor(x -> x.getLong(0))
9      .withWindow(Duration.of(size), Duration.of(slide)),
10     3)
11     .fieldGrouping("rides", new Fields("route"));
12 return builder.createTopology();
```

is registered to the topology by passing its reference to the *TopologyBuilder* object, along with a unique identifier "rides", and a parallelism hint number (Line 6). The parallelism hint dictates the number of worker threads that will be instantiated in the execution topology (i.e., physical execution plan) for the Spout (i.e., copies of the object) (Section 2.3.2). In the example code, the Spout's parallelism is set to one, which will create a single instance of the `RideSpout` object. Next, an `AverageFare` Bolt object is instantiated, which is the object that will produce the mean fare per distinct *route* for each window (Line 4). The `AverageFare` object is registered to the topology (Line 7), its windowing properties are set (Lines 8 - 9), and its number of instances are set to three (Line 10). At this point, the topology has not a communication channel between the Spout instance, and the three downstream Bolt instances. In Storm, this is defined by setting the "*grouping*" of tuples to downstream operators, and in the sample code this is done at Line 11. In this example, each tuple will be distributed (or grouped) based on the *route* value.

Storm supports a *single master-multiple workers* distributed runtime, that executes on a

shared-nothing set of servers. A `Nimbus` process (i.e., the *master*) is responsible for scheduling, managing, and overlooking the execution of active CQs (i.e., topologies) in a distributed network of `Supervisor` processes (i.e., *workers*). More recent versions of Storm support multiple `Nimbus` instances for fault-tolerance and high availability. An Apache ZooKeeper cluster aids in coordinating the distributed network of `Supervisors` [53]. Storm's `Nimbus`, network of `Supervisors`, and the Apache ZooKeeper cluster is considered the required "fabric" in this dissertation. Each of the `Supervisor` instances is periodically checked for liveness. Often, a single `Supervisor` instance executes per server. A `Supervisor` is a single Java process and consists of local I/O modules (input and output buffers), memory management (i.e., JVM's garbage collector), and has a predetermined number of execution slots. It is common practice to configure the number of execution slots (i.e., threads) in a `Supervisor` to be equal to the number of available cores of a server's CPU.

When a CQ is submitted to Storm for execution, `Nimbus` schedules the operations of the physical plan (see Section 2.3) to available `Supervisor` slots, which are called `Executioner` threads. In addition, `Nimbus` informs `Supervisor` processes about the data dissemination policy between `Executioner` threads. This is the type of grouping used to push data from one layer of the CQ (topology) to the next. In the example query depicted in Listing 2.1, the grouping is defined between the single `RideSpout` thread and the three instances of `AverageFare` bolts.

A *grouping* can be any of the following: (i) Shuffle, (ii) Field/Hash, (iii) *Direct*, and (iv) *Broadcast*[3]. We provide additional details on the *grouping* (i.e., partitioning) types in Chapter 3. *Direct grouping* is used for controlled uni-casting of tuples to specific threads, and *Broadcast grouping* is used for multi-casting tuples to all downstream threads. When a `Supervisor` receives a scheduling command from the `Nimbus`, it allocates as many `Executioner` threads as the number of tasks in the schedule command, and it establishes the IO modules (e.g., input and output buffers). By the time all the Spouts and Bolts, with their corresponding parallelism, and communication channels are established, `Nimbus` commences the execution of the topology.

[3]Newer versions of Storm include additional groupings, but we omit them because they are irrelevant to our study.

3.0 STREAM PARTITIONING

Stream Partitioning is the first technique that we address as part of *exact* processing in this dissertation. In this Chapter, we provide essential details for *stateful* processing and the way it is materialized for distributed execution in SPEs. As mentioned in Section 2.3.2, an operator is executed in parallel by multiple workers and in Section 3.1 we present a generic formulation for this process. Often, a *stateful* operator consists of the distribution (partition) of S_i^w and the materialization of processing.

We dive into the details of our work on load-aware stream partition algorithms. In detail, we present our formulation for *stateful* operators (Section 3.1). Next, we describe our novel cost model (Section 3.2), followed by our proposed stream partition algorithm taxonomy (Section 3.3). In Section 3.4, we present the required framework for our algorithms, which are discussed in Section 3.5. Finally, our experimental evaluation is presented in Sections 3.6 and 3.7, followed by a discussion of our findings (Section 3.8).

3.1 A NEW FORMULATION FOR PARALLEL STATEFUL OPERATIONS

By the time a *stateful* operation is scheduled to execute in parallel, it gets transformed into a three-stage process for each window S_i^w. Its input consists of $S_1^w, \ldots, S_N^w$ and the three stages are in order: (i) *partition*, (ii) *partial evaluation*, and (iii) *aggregation*. Figure 9 depicts the windowed *group-by* average on the stream *rides* of the example query (Figure 2) as the three stage process.

Partition can be modeled as a function that takes a subsequence S_i^w and produces another sequence of equal length that indicates for each tuple the worker to which it will be sent.

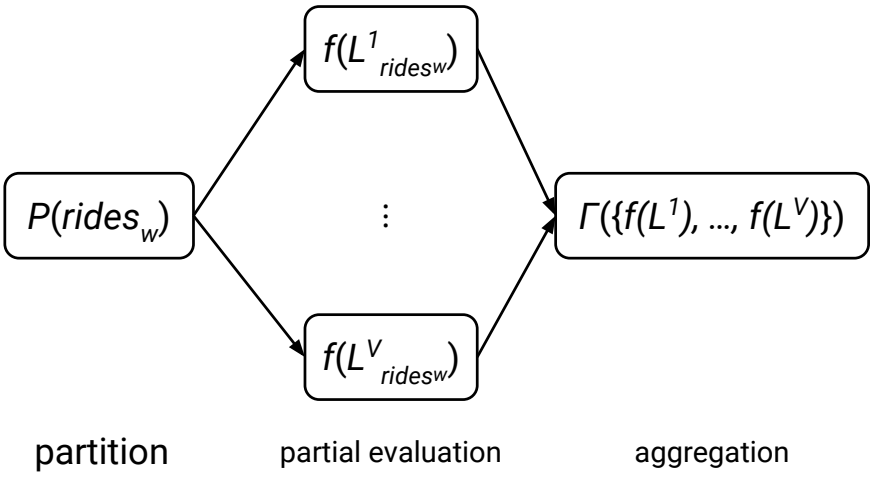


Figure 9: The windowed grouped average of the sample query (Figure 2) as a three stage process.

In other words, *partition* is a function $P : S_i^w \rightarrow \{v_1, \ldots, v_{|S_i^w|}\}$, where $1 \leq v_l \leq \mathcal{V}$ [1]. The resulting sequence consists of elements v_l, where $1 \leq l \leq |S_i^w|$, each one mapping $e_{\mathcal{X}_i}$ indexed by l to a number in $[1, \mathcal{V}]$. $\mathcal{V}$ represents the SPE's parallelism degree for a particular stateful operation and is materialized by $\mathcal{V}$ workers, which are responsible for processing the partial result in window w (a worker can either be a thread or a process). $L_{S_i^w}^v = \{e \in e_{\mathcal{X}_i^w} | P_w(S_i^w)[e] = v\}$ denotes the sequence of tuples from S_i^w that will be sent to worker v, by the *partition* process. *Partial evaluation* is executed by $\mathcal{V}$ workers in parallel. Each worker receives its corresponding $L_{S_i^w}^v$ sequence and applies the user-defined transformation f. f produces a set of key-value pairs: $f : L_{S_i^w}^v \rightarrow \{(k_1, v_1), \ldots, (k_m, v_m)\}$ of arbitrary size m. m is naturally bounded by the cardinality of S_i^w, which is defined as the number of distinct values $k_{\mathcal{X}_i}$ in S_i^w. For the rest of this work, the cardinality of a stream/sequence x will be represented by $\|x\|$, which is used to refer to the number of distinct keys (groups) held by a worker. Finally, *aggregation* combines all the key value pairs

[1] $|x|$ represents the length of a sequence x

$f(L^{v}_{S^{w}_{i}})$ produced by each worker v, $\forall v \in [1, \mathcal{V}]$, into a final result, using an aggregation function $\Gamma(\{f(L^{1}_{S^{w}_{i}}), \ldots, f(L^{\mathcal{V}}_{S^{w}_{i}})\})$.

Going back to the query shown in Figure 9, *partition* would be a function $P(\{rides_w\})$ that partitions each windowed stream based on *route*. *Partial evaluation* would be the partial count and sum of the *group-by* operator and the result of each worker would produce a sequence of *key value* pairs, in which the *keys* would consist of distinct *route* values and *values* would be the sum and the count of fares for each corresponding *route*. Finally, the *aggregation* stage would combine partial results by adding partial counts and sums for every matching *route* key. In essence, if two workers are used with #1 producing $\{(x, 12, 2), (y, 123, 3)\}$ and #2 producing $\{(x, 43, 1), (y, 1, 1), (z, 4, 2)\}$, then *aggregation* ($\Gamma$) produces: $(x, 18.333)$, $(y, 31)$, and $(z, 2)$, similar to the processing model of [113].

3.2 PROPOSED PARTITIONING COST MODEL

Partition aims to: (i) divide S^{w}_{i} as evenly as possible among $\mathcal{V}$ workers, while (ii) the aggregation load (Γ) remains low. This way, execution can benefit from employing multiple workers: *the more* $\max_{\mathcal{V}}(L^{\mathcal{V}}_{S^{w}_{i}})$ *gets reduced, the faster the partial evaluation step is going to progress.* In this dissertation, we adopt the assumption that there exists a monotonic relation between the number of tuples and load increase (similar to previous work on stream partitioning [98]). The previous entails that when a tuple is assigned to a worker, the latter's load will either increase or stay the same.

[98] introduced tuple *imbalance* as a metric for quantifying a *partition algorithm's* efficiency in terms of balancing load among workers. However, [98] expressed *imbalance* on the entire stream (i.e., counting from the beginning of time), which we believe is limited, given the dynamic nature and characteristics of data streams. In this work, we extend *imbalance* to cover the window aspect of a streaming query:

$$I(P(S^{w}_{i})) = |\max_{j}(L^{j}_{S^{w}_{i}}) - \underset{j}{avg}(L^{j}_{S^{w}_{i}})|,\ j = 1, \ldots, \mathcal{V} \tag{3.1}$$

Equation 3.1 defines tuple *imbalance* as the difference of the maximum $\max_j(L^j_{S^w_i})$ minus the average $\underset{j}{avg}(L^j_{S^w_i})$, as partitioned by a partition algorithm P. The less the *imbalance* achieved, the less the maximum runtime of each worker will be.

We propose a new model for measuring the effectiveness of a *partition algorithm* by incorporating *aggregation* cost, which has been ignored in the past. As we discussed in Section 3.1, the *aggregation* stage will have to ingest all $f(L^{\mathcal{V}}_{S^w_i})$ and combine every pair of (k, v) tuples with a matching key k. Hence, the number of operations for processing partial results is proportional to the sum of the sizes of all partial aggregations $|f(L^{\mathcal{V}}_{S^w_i})|$:

$$\Gamma(S^w_i) = O(\sum_{o=1}^{\mathcal{V}} |f(L^o_{S^w_i})|) \tag{3.2}$$

Equation 3.2 captures both the processing and the memory cost of the *aggregation*, since partial results need to be stored until they are processed. In fact, the larger $\Gamma(S^w_i)$ is, the more memory is required to accommodate partial results. Therefore, we model stream partitioning as the following minimization problem:

$$\begin{aligned} &\text{minimize} && I(P(S^w_i)) \\ &\text{while} && \Gamma(S^w_i) \leq \max_{1 \leq j \leq \mathcal{V}}(L^j_{S^w_i}) \end{aligned} \tag{3.3}$$

The reason $\Gamma(S^w_i)$ should be less or equal than the maximum $L^{\mathcal{V}}_{S^w_i}$ is so that execution benefits from parallelizing the workload and not having *aggregation* become more than the maximum partial processing. Finally, in *scale-out* architectures, workers' load might diverge due to external factors (i.e., communication, multi-tenancy etc.). Our model (Equation 3.1) focuses on identifying load generated by the stateful operation and act accordingly to balance it. To the extent of our knowledge, broader load monitoring in a SPE involves architectural interventions, such as monitoring modules and feedback loops [66, 141, 50, 67, 78, 108]. If a SPE features the aforementioned components to detect load divergence caused by external factors, our cost model (Equation 3.3) can be extended to incorporate that information in its decision process as well [2].

[2]By changing Equation 3.1 to multiply $L^j_{S^w_i}$ with *load-divergence* coefficients, produced periodically by monitoring components.

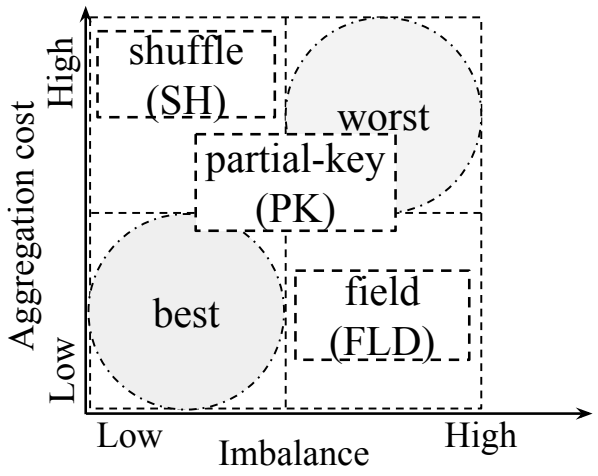


Figure 10: Expected performance of stream partitioning algorithms.

3.3 THE PITFALL OF IGNORING AGGREGATION COSTS

To better understand inherent trade-offs among existing partition algorithms, we present Figure 10, which illustrates the two dimensions with which each algorithm is measured. The horizontal axis represents the *ability* of an algorithm to balance the load among workers (i.e., keep the *imbalance* low), and the vertical axis represents an algorithm's *ability* to maintain the *aggregation* cost low, based on our model (Equation 3.3). In Figure 10 we have placed previously proposed partition algorithms based on their expected behavior in terms of *imbalance* and *aggregation* cost.

As indicated by Equation 3.3, partitioning becomes a trade-off between tuple *imbalance* and *aggregation* cost: *the more tuples are spread, the more aggregation time increases*. Consider S_i to be an input stream with schema $\mathcal{X}_i = (t, a, b)$, where $\tau_{\mathcal{X}_i} = \{t\}$, $k_{\mathcal{X}_i} = \{a\}$ and $p_{\mathcal{X}_i} = \{b\}$. In a *stateful* operation, a partitioning algorithm has to make a choice of where all tuples with a particular key will be sent. Partitioning algorithms can be categorized based on how many worker options are presented for a given $k_{\mathcal{X}_i}$. We review the different alternatives next.

3.3.1 Partition Algorithm Taxonomy

A *1-choice partition algorithm* offers no mechanisms to balance the skewness of the input data. As a result, the workers that happen to be assigned the part of the data that appear the most (i.e., most frequent) will always have more work compared to others. That leads to higher *imbalance* (Equation 3.1). In addition, when a single option for each $k_{\mathcal{X}_i}$ is presented, *aggregation* cost (Equation 3.2) is minimal, because each worker will produce a subset of the full result.

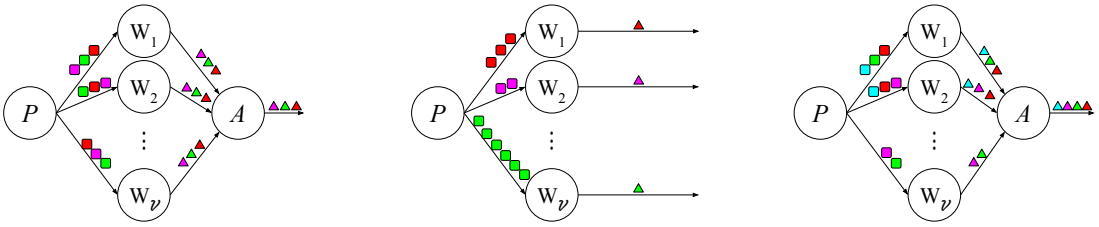
On the other hand, an *M-choice partition algorithm* ($M \leq \mathcal{V}$) considers M candidate workers for each $k_{\mathcal{X}_i}$. Thus, load for $k_{\mathcal{X}_i}$ is divided into M equal parts and handled by M workers. As a result, *imbalance* (Equation 3.1) is reduced, and the SPE takes better advantage of parallelism. Unfortunately, partial results produced by the M workers handling a particular $k_{\mathcal{X}_i}$ have to be gathered and combined. That entails an inflated *aggregation* cost, which is expected to increase by a factor of M. For example, in a single window S_i^w, if tuples with $k_{\mathcal{X}_i} = a_x$ are assigned to 4 workers, then the *aggregation* stage will process 4 partial results (i.e., one from each worker). As a result, the *aggregation* process affects the time it takes to produce a window result R_w.

3.3.2 Mapping existing Partition Algorithms to each Category

At this point, we are going to categorize existing partition algorithms in the two aforementioned categories (i.e., single and multiple choice partition algorithms). Even though there exists a plethora of complex partition algorithms in existing bibliography, we will focus on two of the most popular [3] and the state of the art.

3.3.2.1 Shuffle (**SH**) blindly sends tuples to workers, without making any attempt to balance load and collocate keys (Figure 11a). Therefore, SH is categorized as a *M-choice partition algorithm* because an *aggregation* stage is required to produce the final result. SH manages to minimize tuple *imbalance* (Equation 3.1) since each worker receives the same number of tuples in a given window S_i^w: if $\mathcal{V}$ workers exist, each one will receive

[3] Popularity is considered by the fact that those are implemented in most SPEs.



a: Shuffle (SH). b: Hash/Field (FLD). c: Partial-Key (PK).

Figure 11: Existing stream partitioning algorithms lack a unified model that limits *imbalance* while keeping *aggregation* cost low.

$\frac{|S_i^w|}{\mathcal{V}}$ tuples. Turning to *aggregation* cost $\Gamma(S_i^w)$ (Equation 3.2), when SH is used it becomes computationally expensive, because tuples are partitioned without an attempt to collocate keys. Therefore, in a worst case scenario, each worker will produce a partial result ($f(L_{S_i^w}^{\mathcal{V}})$) with all the keys that exist in S_i^w (illustrated in Figure 11a). In that case, $\Gamma(S_i^w)$ will become equal to M times $\|S_i^w\|$. As far as our cost model is concerned (Equation 3.3), *SH minimizes* imbalance*, but does not act to limit the* aggregation *cost.*

3.3.2.2 Hash (or Field) (**FLD**) follows a different approach than SH, by collocating tuples with the same $k_{\mathcal{X}_i}$ on the same worker (Figure 11b). FLD feeds $k_{\mathcal{X}_i}$ to a hash function and selects a worker based on the result. It guarantees that keys from the same group will be collocated, resulting in minimal *aggregation* cost (Equation 3.2). Hence, FLD is characterized as a *1-choice partition algorithm.* Nevertheless, FLD fails to balance the load effectively when input is skewed and some keys appear more often than others. (i.e., there is tuple *imbalance* - Equation 3.1). Matters can get exacerbated if initial expectations (or assumptions) on input load do not hold true overtime. Under such circumstances, "struggling" workers with excess load will hinder the progress of a query and even compromise the correctness of the result. In conclusion, FLD imposes minimal *aggregation* cost but does not act on limiting tuple *imbalance* (based on the cost model - Equation 3.3).

3.3.2.3 Partial-Key (**PK**), is the current state of the art algorithm [98]. It adopted the idea of *key splitting* [21] to alleviate the load of processing keys that are part of the skew. PK was first to incorporate load in terms of the number of tuples assigned to each worker (i.e., $L^{\mathcal{V}}_{S^w_i}$). *Key splitting* is materialized by using a pair of independent hash functions (i.e., $\mathcal{H}_1, \mathcal{H}_2$) and feed $k_{\mathcal{X}_i}$ to both. Also, PK maintains an array of size $\mathcal{V}$ with the total tuple count sent to each worker. Every time a tuple arrives, its $k_{\mathcal{X}_i}$ is fed to $\mathcal{H}_1$ and $\mathcal{H}_2$ to identify two candidate workers. The partition algorithm will forward the tuple to the candidate that has received the least number of tuples up to that point. PK was extended to more than two candidates [99], when two are not sufficient to handle skew. Even though PK succeeded in improving *imbalance* (Equation 3.1) compared to FLD, it did so by adding an essential *aggregation* step (Equation 3.2). Therefore, PK is expected to incur *aggregation cost* proportional to the number of candidates. Turning to our cost model (Equation 3.3), PK can potentially violate the *aggregation* cost constraint, when $\Gamma(S^w_i)$ exceeds the maximum workload experienced by each worker.

Summary: Our goal is to propose partitioning algorithms that belong to the *best* part of the chart (Figure 10), where there is both low imbalance and low aggregation overhead. As a guide, we will use our cost model (Equation 3.3). We present our proposed approach next.

3.4 MINIMIZING IMBALANCE WITH LOW AGGREGATION COST

Designing a partitioning algorithm that achieves low *aggregation* cost entails keeping track of the number of keys produced by each worker on every window S^w_i (i.e., $f(L^j_{S^w_i})$, for $1 \leq j \leq \mathcal{V}$). Equation 3.2 indicates that if the sum of $f(L^j_{S^w_i})$ is reduced, then the *aggregation* cost gets reduced as well. However, the boundaries of the *aggregation* cost need to be identified first.

Proposition 1. *For a given stream S^w_i, a* stateful *operation f, and $\mathcal{V}$ number of workers, $\Gamma(S^w_i)$ is bounded by: $\|S^w_i\| \leq \Gamma(S^w_i) \leq \mathcal{V}\|S^w_i\|$.*

Proof. $\Gamma(S^w_i)$ will always be greater or equal to $\|S^w_i\|$ and that happens when the partitioning

algorithm sends each key to a single worker only. In this case, $L^i_{S^w_i} \cap L^j_{S^w_i} = \emptyset$, $\forall\, 1 \leq i \neq j \leq \mathcal{V}$. Hence, $\|L^1_{S^w_i}\| + \ldots + \|L^{\mathcal{V}}_{S^w_i}\| = \|S^w_i\|$. Similarly, if the partition algorithm sends at least one tuple for each key to every worker (i.e., $k \in \|L^j_{S^w_i}\|$, $\forall k \in S^w_i$ and $1 \leq j \leq \mathcal{V}$), then $\|L^1_{S^w_i}\| + \ldots + \|L^{\mathcal{V}}_{S^w_i}\| = \underbrace{\|S^w_i\| + \ldots + \|S^w_i\|}_{\mathcal{V}} = \mathcal{V}\|S^w_i\|$ □

A mechanism to control $\Gamma(S^w_i)$'s value has to be established. Equation 3.2 can be expanded to the sum of its operands as:

$$\Gamma(S^w_i) = |f(L^1_{S^w_i})| + \ldots + |f(L^{\mathcal{V}}_{S^w_i})| \tag{3.4}$$

Hence, in order to monitor *aggregation* cost, the partition algorithm has to keep track of the number of distinct keys sent to each worker, for each S^w_i.

3.4.1 Incorporating Cardinality in Partitioning

Assuming a mechanism for keeping track of workers' cardinalities has been established, the cost model (Equation 3.3) can be extended to incorporate the knowledge of the number of distinct keys sent to each worker. As indicated by Equation 3.3, the information about workers' cardinalities can be used in two places: (a) *imbalance* (Equation 3.1), and (b) *aggregation* cost (Equation 3.2).

3.4.1.1 Incorporating Cardinality in measuring Imbalance The load of each worker has been modeled in terms of number of tuples (Equation 3.1). In the same manner, a worker's load can be expressed in terms of cardinality using the following formula:

$$CL^j_{S^w_i} = \|L^j_{S^w_i}\|, 1 \leq j \leq \mathcal{V} \tag{3.5}$$

Equation 3.5 depicts the load of a worker in terms of the number of distinct keys sent to it. Therefore, cardinality *imbalance* can be expressed as the difference between the maximum and the mean cardinality of all workers for a given window S^w_i, as a result of a partitioning algorithm P:

$$CI(P(S^w_i)) = \max_j(CL^j_{S^w_i}) - \underset{j}{avg}(CL^j_{S^w_i}), 1 \leq j \leq \mathcal{V} \tag{3.6}$$

At this point, *imbalance* is determined by tuple count and cardinality. However, different *stateful* operations are affected by each metric differently. Hence, there might be a need for a more diverse load estimation formula, which combines tuple count and cardinality. In order to avoid one metric dominating the other, the initial values should be scaled accordingly:

$$L^{j}_{S^w_i}{}' = \frac{L^{j}_{S^w_i} - \min_{1\leq k\leq \mathcal{V}}(L^{k}_{S^w_i})}{\max_{1\leq k\leq \mathcal{V}}(L^{k}_{S^w_i}) - \min_{1\leq k\leq \mathcal{V}}(L^{k}_{S^w_i})} \tag{3.7}$$

$$CL^{j}_{S^w_i}{}' = \frac{CL^{j}_{S^w_i} - \min_{1\leq k\leq \mathcal{V}}(CL^{k}_{S^w_i})}{\max_{1\leq k\leq \mathcal{V}}(CL^{k}_{S^w_i}) - \min_{1\leq k\leq \mathcal{V}}(CL^{k}_{S^w_i})} \tag{3.8}$$

$$H^{j}_{S^w_i} = pL^{j}_{S^w_i}{}' + (1-p)CL^{j}_{S^w_i}{}', \text{ where } 1 \leq j \leq \mathcal{V} \tag{3.9}$$

Equation 3.9 combines the normalized loads both in terms of tuples (Equation 3.7) and distinct keys (Equation 3.8) in a unified score. That score is adjustable based on a user's (or query optimizer's) parameter p, which controls the bias for each score accordingly:

- *the smaller the p, the less the load in terms of tuples affects Equation 3.9*
- *whereas the higher the p, the less the load in terms of distinct keys affects Equation 3.9*

Finally, *imbalance* can be expanded to a hybrid form that incorporates load in terms of both tuple count and cardinality as follows:

$$HI(P(S^w_i)) = \max_j(H^{j}_{S^w_i}) - \underset{j}{avg}(H^{j}_{S^w_i}),\ 1 \leq j \leq \mathcal{V} \tag{3.10}$$

3.4.1.2 Incorporating Cardinality in Measuring Aggregation Cost

Aggregation cost is determined by $\Gamma(S^w_i)$ (Equation 3.2) and reducing it can be achieved by reducing the count of distinct keys sent to each worker. Its minimum value can be $\|S^w_i\|$ when each key is sent to only a single worker. This behavior resembles FLD and it might result in *imbalance* on workers. To avoid this, we employ *key splitting* for keys that have not been sent to a worker before in a particular window S^w_i. By sending each newly encountered key to the worker with either the least keys or the least number of tuples up to that point, the *aggregation* cost remains low. Also, *imbalance* is expected to be lower compared to the one achieved from FLD.

3.4.2 Cardinality Estimation data structures

The partition algorithm needs to keep track of each worker's cardinality. Hence, it has to maintain an array of $\mathcal{V}$ cardinality estimation structures (C), which will offer two methods: (i) *update*($k_{\mathcal{X}_i}$): for updating the count of distinct keys; and (ii) *estimate()*: for returning the count of distinct keys.

3.4.2.1 Naive The naive approach for estimating a worker's cardinality involves keeping track of the exact number of distinct keys. Therefore, a partition algorithm responsible for $\mathcal{V}$ downstream workers, requires $\mathcal{V}$ unordered set structures. This way, the *update* and the *estimate* methods will offer constant execution time (O(1)).

One caveat of using an unordered set structure for each worker is the memory overhead. Depending on the algorithm used, a key can end up in multiple workers (e.g., SH). This way, the memory required for maintaining the number of keys on each worker can become $O(\mathcal{V}\|S_i^w\|)$, since all unordered sets can end up having each key. The memory cost of a naive cardinality estimation structure is related to the cardinality of S_i^w and the choice of the partitioning policy: If $\|S_i^w\|$ remains low and the partition algorithm does not send the same keys to multiple workers, the memory requirements for C will remain low. However, if $\|S_i^w\|$ is high and the partition algorithm tends to send tuples with the same key to multiple workers, then memory load can hinder the partition process.

3.4.2.2 Hyperloglog HyperLogLog (HLL) introduced by Flajolet et al. [52] is an approximation data structure for estimating the number of distinct elements in databases with a bounded error. It requires $\mathrm{O}(\log_2 \log_2 N)$ memory for a relation expected to have N distinct elements. Every time a new tuple arrives, its $k_{\mathcal{X}_i}$ is extracted and fed through a hash function. HLL extracts the m most significant bits of the hash result, and uses them to identify which register (out of 2^m) to update. Each register is $\log_2 \log_2 N$ bits long, and its value is updated depending on the left-most zero of the m most significant bits of the hashed value. HLL has been shown to present an accuracy of $\frac{1.04}{\sqrt{m}}$. Recent work from Heule et al. [70] presented a number of improvements that need to take place so that cardinalities

Algorithm 1 Partition.

```
1: procedure PARTITION(e_{X_i}, t_w, C, L)
2:     k ← GetKey(e_{X_i})
3:     τ ← GetTime(e_{X_i})
4:     if τ ≥ t_w then
5:         reset(C)
6:         reset(L)
7:     c_1 ← H_1(k)
8:     c_2 ← H_2(k)
9:     return decide(C, L, k, c_1, c_2)
```

in the orders of billions can be estimated efficiently. For cardinality estimation, a partition algorithm is required to use $\mathcal{V}$ HLL's to measure the number of distinct keys sent to each of the $\mathcal{V}$ workers.

3.5 PROPOSED CARDINALITY-AWARE PARTITIONING ALGORITHMS

PK [98, 99] motivated the merits of *key splitting* [21] for reducing *imbalance* among workers; in a nutshell, the dynamicity and variability of input data streams pretty much guarantee the need for it. All the variations of our proposed algorithms leverage *key splitting*. This entails the use of multiple hash functions for identifying candidate workers. For simplicity, our algorithms are presented with only two candidates, but they can be easily extended to accommodate more.

The basis of our algorithms is presented in Algorithm 1 and is invoked by the SPE's *partitioning algorithm* when a new tuple arrives. The *partitioning algorithm* maintains two arrays of size $\mathcal{V}$: one with cardinality estimation structures (C), and one with tuple counters (L). L is identical to the one used by PK and gets updated the same way in all our proposed algorithms. $e_{\mathcal{X}_i}$'s key is extracted and fed to the two hash functions: $\mathcal{H}_1$ and $\mathcal{H}_2$. If the partitioning algorithm uses more than two candidates (i.e., $M > 2$), then an equal number

Algorithm 2 Cardinality Imbalance Minimization (CM)

```
1:  procedure CM(C, L, k, c_1, c_2)
2:      l_1 ← C[c_1].estimate()
3:      l_2 ← C[c_2].estimate()
4:      if l_1 ≤ l_2 then
5:          C[c_1].update(k)
6:          L[c_1] += 1
7:          return c_1
8:      else
9:          C[c_2].update(k)
10:         L[c_2] += 1
11:         return c_2
```

of hash functions are used in the decision process. The resulting choices (c_1 and c_2) along with the arrays C and L are passed to *decide()*.

During query execution, an SPE might have multiple instances of partition algorithms running on different machines (especially in a *scale-out* setting, where thousands of threads are involved in a query). The advantage of using hash functions is that no exchange of information is required among different instances of *partitioning algorithms*. On top of that, C and L have their counts monotonically increasing on each window. Therefore, if each of the *partitioning algorithms* tries to reduce *imbalance* and/or *aggregation* cost, then (through the additive property) the overall *imbalance* and/or *aggregation* cost are reduced. Finally, C and L need to be reset when a window expires (Algorithm 1 line 4). This guarantees that decisions consider the temporal nature of stream processing. Algorithm 1 receives t_w as an argument, which is the expiration timestamp of the current window. In the following sections we go over our variations for *decide()*:

1. Cardinality *imbalance* Minimization (CM)
2. Group Affinity with *imbalance* Minimization (AM & cAM)
3. (iii) Hybrid *imbalance* Minimization (LM)

3.5.1 Cardinality Imbalance Minimization (CM)

The first partitioning algorithm aims at limiting cardinality *imbalance* (Equation 3.6) and the decision is made based on the cardinality estimate retrieved by the C array structure (Equation 3.5). The newly arrived tuple $e_{\mathcal{X}_i}$ is sent to the candidate worker that has the least cardinality. Algorithm 2 illustrates the cardinality *imbalance* minimization algorithm (CM) and works as a counterpart to PK. CM can utilize either the Naive (Section 3.4.2.1) or the HLL (Section 3.4.2.2) as its cardinality estimation structure.

This algorithm is expected to be used in operations in which processing cost is dominated by the amount of distinct keys. This way, *imbalance* in terms of cardinality will be minimal. However, *imbalance* in terms of tuple counts will be increased, since CM is tuple-count agnostic and does not make any efforts to limit the *aggregation* cost.

3.5.2 Group Affinity and Imbalance Minimization (AM & cAM)

Group Affinity algorithms try to impose no additional *aggregation* cost, while balancing load with the use of *key splitting*. The name affinity comes from keeping track of whether a key has been encountered before in S_i^w, and if it did, then it is forwarded to the worker that received it previously.

The first variation of affinity-based algorithms, is AM and focuses on cardinality *imbalance* (Algorithm 3). AM tries to minimize *aggregation* cost by not splitting keys among workers. First, it checks if one of the candidate workers has encountered key k previously. If one of them did, then the tuple is forwarded to that worker; otherwise, it is sent to the worker with the least cardinality up to that point.

A different variation of AM, named cAM (Algorithm 4) behaves similarly, but it forwards the tuple to the worker with the least tuple count up to that point. This way, both *aggregation* cost and *imbalance* are considered during partitioning. Despite the fact that AM and cAM resemble FLD, they are expected to perform better because of the multiple number of choices that are presented to them.

Both AM and cAM require cardinality estimation structures to keep track of each key's location. As discussed earlier, HLL can be used to limit the memory used by the cardinality

Algorithm 3 Group affinity combined with cardinality *imbalance* minimization (AM)

```
1: procedure AM(C, L, k, c_1, c_2)
2:     if C[c_1].contains(k) then
3:         L[c_1] += 1
4:         return c_1
5:     else if C[c_2].contains(k) then
6:         L[c_2] += 1
7:         return c_2
8:     else
9:         l_1 ← C[c_1].estimate()
10:        l_2 ← C[c_2].estimate()
11:        if l_1 ≤ l_2 then
12:            C[c_1].update(k)
13:            L[c_1] += 1
14:            return c_1
15:        else
16:            C[c_2].update(k)
17:            L[c_2] += 1
18:            return c_2
```

Algorithm 4 Group affinity with *imbalance* minimization (cAM)

```
1: procedure AM(C, L, k, c_1, c_2)
2:     if C[c_1].contains(k) then
3:         L[c_1] += 1
4:         return c_1
5:     else if C[c_2].contains(k) then
6:         L[c_2] += 1
7:         return c_2
8:     else
9:         l_1 ← L[c_1].estimate()
10:        l_2 ← L[c_2].estimate()
11:        if l_1 ≤ l_2 then
12:            C[c_1].update(k)
13:            L[c_1] += 1
14:            return c_1
15:        else
16:            C[c_2].update(k)
17:            L[c_2] += 1
18:            return c_2
```

estimation structures. However, HLL uses an irreversible operation to update its internal buckets when a new element is encountered. That makes it unable to check whether a key has been previously sent to a worker. We address this shortcoming by introducing an optimistic mechanism for checking if a key has been seen before: upon the arrival of a key that hashes to workers i and j, their cardinality estimates, c_i and c_j, are retrieved. A trial update of those is performed and the new cardinality estimates are retrieved: c'_i and c'_j. If either workers' cardinality difference $\Delta(|c_x - c'_x|)$ is 0, then AM and cAM optimistically assume that the key has already been sent to that worker (they forward the tuple to it). Otherwise, AM's and cAM's behavior is similar to CM's or PK's. HLL is expected to make wrong decisions at the benefit of a constant memory cost.

3.5.3 Hybrid Imbalance Minimization (LM)

Algorithm 5 Hyblid *imbalance* minimization (LM)

1: **procedure** CM(C, L, k, c_1, c_2)
2: $l_1 \leftarrow pL^{c_1}_{S^w_i} + (1-p)CL^{c_1}_{S^w_i}$
3: $l_2 \leftarrow pL^{c_2}_{S^w_i} + (1-p)CL^{c_2}_{S^w_i}$
4: **if** $l_1 \leq l_2$ **then**
5: $C[c_1].update(k)$
6: $L[c_1] += 1$
7: **return** c_1
8: **else**
9: $C[c_2].update(k)$
10: $L[c_2] += 1$
11: **return** c_2

For *stateful* operations equally affected by tuple count and cardinality, we propose the hybrid load *imbalance* minimization algorithm (LM). It combines a worker's tuple count with cardinality and calculates hybrid load as indicated in Equation 3.9. A tuple is forwarded to the worker with the least load and LM's main goal is to minimize hybrid load imbalance (Equation 3.10). LM is depicted on Algorithm 5.

Table 2: Stream partitioning algorithms

Symbol	Algorithm	Choices	Cardinality Estimation Structure used
SH-w	shuffle	w	*None*
FLD-1	field	1	*None*
PK-2	partial-key [98]	2	*None*
PK-5	partial-key [99]	5	*None*
CM-2	Alg. 2	2	Naive, Sec. 3.4.2.1
AM-2	Alg. 3	2	Naive, Sec. 3.4.2.1
AM-5	Alg. 3	5	Naive, Sec. 3.4.2.1
cAM-2	Alg. 4	2	Naive, Sec. 3.4.2.1
cAM-5	Alg. 4	5	Naive, Sec. 3.4.2.1
LM-2	Alg. 5	2	Naive, Sec. 3.4.2.1
CM-2-H	Alg. 2	2	HLL, Sec. 3.4.2.2
AM-2-H	Alg. 3	2	HLL, Sec. 3.4.2.2
LM-2-H	Alg. 5	2	HLL, Sec. 3.4.2.2

3.6 EXPERIMENTAL SETUP

This section details our setup in terms of the infrastructure, algorithms, datasets, and workloads used in our experimental evaluation.

Our experiments were conducted on an AWS c4.8xlarge instance, running Ubuntu v14.04. For all experiments, we used our own multi-threaded stream partitioning library, developed in C++11 and compiled with GCC v4.8.2. Our performance analysis involved varying numbers of workers (8 up to 32), and partitions (from 8 to 256). The reason we did not experiment with more workers was because we did not want to pollute results with context-switching overheads. All reported runtimes are the averages of seven runs, after removing minimum

and maximum reported times, to compensate for anomalies related to running concurrent processes in a real system.

3.6.1 Stream Partitioning Algorithms

For our experimental evaluation, we used algorithms *shuffle* (SH), *field* (FLD), and *partial key* (PK) [98] (with 2 and 5 candidate workers), as baselines and compared them with different variations of our proposed algorithms:

- *Cardinality Imbalance Minimization* (CM)
- *Group Affinity with Cardinality Imbalance Minimization* (AM)
- *Group Affinity with Imbalance Minimization* (cAM)
- *Hybrid Imbalance Minimization* (LM)

For all variations of LM we set p to 0.5 to achieve unbiased load estimation.

As a reference implementation for SH, FLD and PK we used the ones found in Apache Storm. In addition, we used the open source implementation of Murmur-Hash v3. All our proposed algorithms appear in two versions: one with naive and one with HLL as the cardinality estimation structure. For the former, we used C++ STL's implementation of unordered set, and for HLL, we implemented our version with $4096(= 2^{12})$ registers and a register size of 5 bits. The choice of the number and size of registers was made to accommodate up to 10^7 distinct keys of 32 bits, with an accuracy lower than 2%, as instructed in [52]. Table 2 explains the algorithm acronyms we use in our graphs, in which w indicates the number of candidate workers.

3.6.2 Data sets and Workloads

Section 2.6.1 presents general information about the datasets we use in our experimental evaluation. In this section we present specific details for this chapter's experiments. Table 3 summarizes the characteristics of each data set or benchmark used in our experiments. Below, we go over each data set and the queries we apply on it.

TPC-H (TPCH): Out of 22 TPCH queries, 16 of them feature a *grouping* statement: half of the queries maintain a constant number of groups, whereas the other half features

Table 3: Summary of data characteristics.

Dataset	Size	Groups	Window	Metric
TPC-H	10GB	4 up to ∼100k	N/A	throughput
DEBS	32GB	62.5K up to 8.1M	sliding	latency
GCM	16GB	4 to ∼670K	sliding	latency

a scaling number that increases when the *scale factor* grows. Due to the fact that our work addresses *stateful* operations, we focused on *grouping* TPCH queries and picked Query 1 (as a constant *grouping* query) and Query 3 (as a scaling *grouping* query). Those two differ significantly in the number of resulting groups, and this enabled us to document the performance of different partition algorithms, when the *aggregation* cost varies in terms of size. As indicated in Table 3, Query 1 presents four and Query 3 presents up to 110,000 resulting groups. We used TPCH to measure window throughput using different partitioning algorithms.

ACM DEBS 2015 Grand Challenge (DEBS): We conduct experiments using both of DEBS's queries, and we measure partitioning algorithms' impact in performance. In detail, for DEBS we measure window latency, which is the time it takes to process a window's tuples.

Google Cluster Monitoring (GCM): For GCM we use two sliding window queries, which, like DEBS, are aimed for measuring window latency. GCM Query 1 was taken from [82], and scans the `task_event` table to calculate every 60 seconds (with a slide of 1 second) the total CPU cores requested by each scheduling class. In addition, we introduced GCM Query 2 that calculates every 45 minutes (with a slide of 1 second) the average CPU cores requested by each job ID. There are more than 600 thousand job IDs in the whole data set.

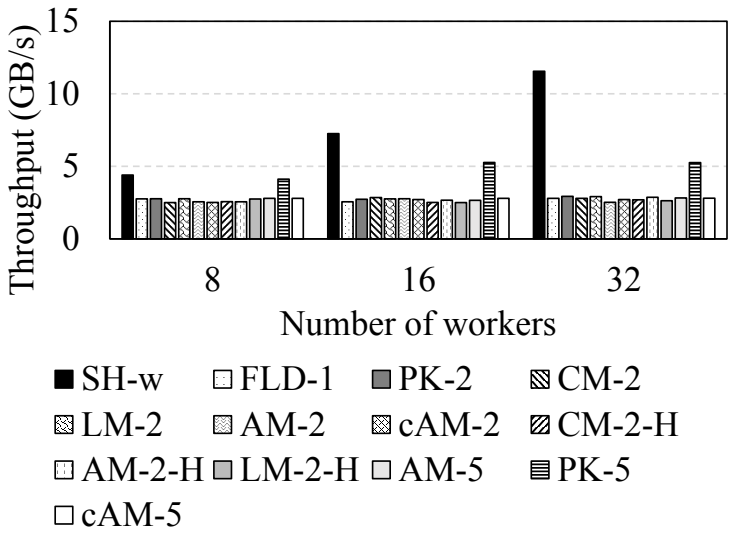


Figure 12: TPCH Query 1 performance.

Our experiments evaluate the impact of a partition algorithm on performance (Section 3.7.1), in terms of throughput (using the TPCH data set) and window latency (using the DEBS data set). Moreover, we evaluate the scalability (Section 3.7.2) of our algorithms compared to the state of the art (using the GCM data set). For all experiments, data were loaded in main memory before execution, to better emulate a real SPE and to minimize any side-effects from hard disk latencies. The time to load data and write output to storage was not included in the reported times. Finally, for the experiments of Section 3.7.1 and 3.7.2, the time it takes to partition tuples is not included, because it is analyzed in Section 3.7.3 in terms of both processing and memory costs.

3.7.1 Performance

In this set of experiments, we used the TPCH and DEBS benchmarks to evaluate performance.

3.7.1.1 TPCH Query 1 (Figure 12) Figure 12 shows the results of emulating the performance of all partitioning algorithms for TPCH query 1. For this experiment, SH-w performs the best. This behavior is expected since there are only 4 groups for Query 1. Therefore, the *aggregation* cost is negligible and performance is affected only by tuple *imbalance*. SH-w is expected to offer optimal tuple *imbalance* (≤ 1), which is reflected on the results shown in Figure 12. Those agree with our model (Equation 3.3), which identifies that SH-w offers minimal *imbalance* with constant *aggregation* runtime of $O(4\mathcal{V})$, where $\mathcal{V}$ is the number of workers. In addition, PK-5 offers the next best throughput, since it reduces tuple *imbalance*, compared to all other algorithms with 2 and 5 alternative choices per group. CM and LM do not scale well with two choices, since they are affected by cardinality *imbalance*. LM is expected to perform similarly to PK, if p takes a value of 1 (as indicated in Equation 3.9). Turning to *1-choice partition algorithms* (i.e., FLD-1, AM and cAM), they present constant performance and do not scale when the number of workers increases. This happens because each group, is presented to a single candidate worker.

Take-away: If the number of groups is constant and much smaller than the size of the *aggregation*, SH performs the best.

3.7.1.2 TPCH Query 3 (Figure. 13 - 14) The query plan of TPCH Query 3 consists of a parallel hash join for the *customer* and *orders* tables, followed by a broadcast join with the *lineitem* table. Then, a parallel computation of the *group by* follows, and execution concludes with a final aggregation step to materialize the result.

Figure 13 illustrates the performance of *1-choice partition algorithms* (i.e., FLD-1, AM, and cAM), SH-w, PK-2, and PK-5. *M-choice partition algorithms* performed worse, from 2.5x up to an order of magnitude worse (LM and CM offered similar performance to PK-2). As shown on Table 3, TPCH Query 3 involves 110 thousand groups (before applying the *limit* statement), and *aggregation* can take up to 60% of the total execution time for *M-choice partitioning algorithms*. As a result, *M-choice partitioning algorithms* (i.e., SH, PK, CM, and LM) experience a substantial performance overhead on the final *aggregation* step.

Figure 14 illustrates the tuple *imbalance* relative to FLD-1 achieved by different variations of AM and cAM. Even though, AM-2-H achieves better tuple *imbalance* compared to FLD-1,

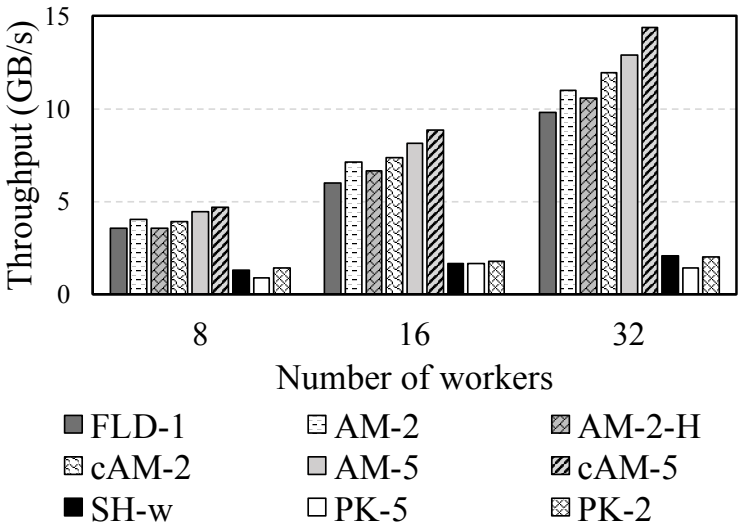


Figure 13: TPCH Query 3 performance.

it fails to perform in the same level as AM-2. Tuple *imbalance* results justify the throughput shown on Figure 13, in which (apart from AM-2-H) all variations of our proposed algorithms perform significantly better than FLD-1. By adopting *key splitting*, throughput increases with the use of multiple candidate workers. AM-5 and cAM-5 offer improved throughput up to 47% compared to FLD-1.

Take-away: For throughput-oriented queries, with a large number of groups, cAM and AM perform the best. They achieve up to an order of magnitude better throughput compared to PK, and outperform FLD by up to 47%.

3.7.1.3 DEBS Query 1 (Figure. 15a - 15c) Turning to DEBS, both queries involve window semantics and the performance is measured in terms of window latency. Figure 15 shows the mean and 99-percentile window latency achieved by each partitioning algorithm. It is clear that in all worker settings, FLD-1, AM-2, cAM-2, AM-2-H, AM-5, and cAM-5 perform the best. This emanates from a lack of *aggregation* overhead, which makes those algorithms

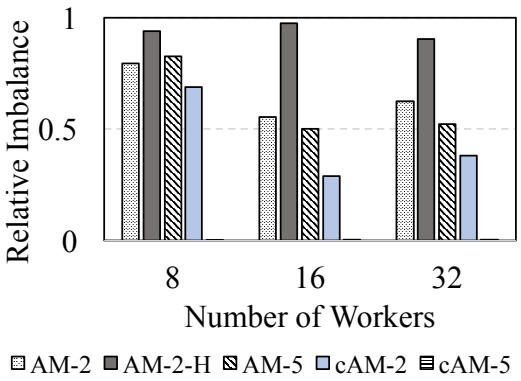
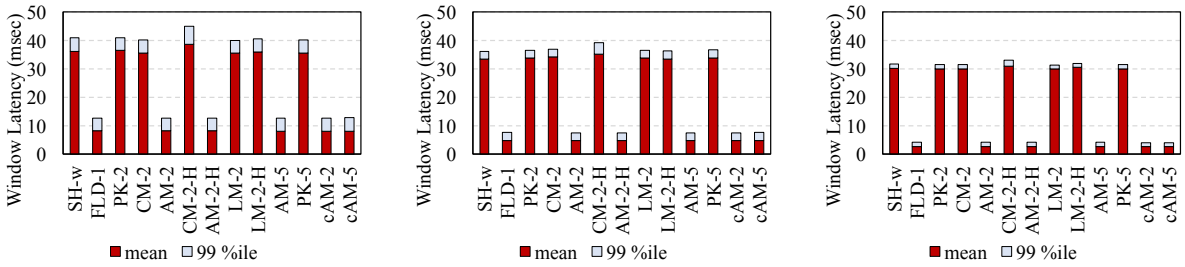


Figure 14: TPCH Query 3 relative imbalance to FLD.

scalable when the number of workers increases. In fact, the *aggregation* cost amounts to more than 70%, 84%, and 88% of total runtime for SH-w, PK-2, PK-5, CM-2, and LM-2. Finally, AM-2-H achieves identical performance with AM-2, which leads us to believe that our optimistic mechanism for cardinality estimation maintains a low error.

Take-away: For latency-oriented *stateful* queries, AM, and cAM perform from 4.5x up to 11.6x better compared to PK.

3.7.1.4 DEBS Query 2 (Figure. 16a - 16c) The execution plan starts by partitioning incoming tuples based on the medallion of each ride and each worker has to create two local indices: one for accumulating fares for each pickup cell, and one for keeping track of the latest drop-off cell. Then, an *aggregation* step follows, which gathers each pickup cell's fares and determines the latest cell for each medallion. The two resulting streams are partitioned based on pickup and drop-off cell IDs. Next, a gather step is executed, in which the median fare and the number of vacant taxis are processed to calculate the profit of each cell. Finally, partial results are merged and ordered to produce the Top-10 most profitable cells. This query represents a problematic case for our model (Equation 3.3), because partitioning does not necessarily affect the workload imposed on each worker (i.e., the partitioning key is the



a: 8 workers. b: 16 workers. c: 32 workers.

Figure 15: DEBS Query 1 performance.

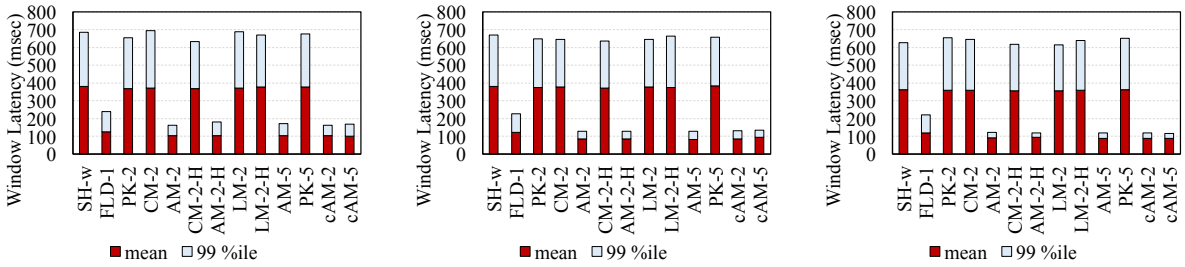
medallion but each worker's state is affected by the number of distinct cells).

Figure 16 depicts window latency achieved by each algorithm, and it is apparent that FLD-1, AM-2, AM-5, cAM-2, cAM-5, and AM-2-H offer the best performance. AM and cAM in all their variations outperform FLD in both mean (from 1.2x up to 1.5x) and 99-percentile (from 1.3x to 1.9x) latency. This is justified by AM's and cAM's ability to partition data more evenly compared to FLD. *M-choice partition algorithms* underperform because they do not act on limiting the *aggregation* overhead. In comparison with PK (in both PK-2 and PK-5), AM and cAM perform up to 5.7x faster. In order to examine AM's and cAM's scalability, we also ran DEBS Query 2 with 64 and 128 workers. They performed up to 6.2x better than PK and up to 2.3x better than FLD.

Take-away: For latency-oriented queries, with multiple *stateful* operations, AM and cAM have window latency between 1.2x and 1.9x lower than FLD, and up to 5.7x lower than PK.

3.7.2 Scalability

In this set of experiments, we used the GCM dataset to measure the scalability of our *1-choice partitioners* (i.e., AM and cAM) compared to SH and PK. The reason for picking GCM for scalability experiments is because it presents a conventional monitoring scenario,



a: 8 workers. b: 16 workers. c: 32 workers.

Figure 16: DEBS Query 2 performance.

in which the groups are not significantly more than the number of tuples in a window (like in TPCH and DEBS), and the queries consist of a single *stateful* operation. This way, *M-choice partitioners* would not be impeded by the *aggregation cost*.

3.7.2.1 GCM Query 1 (Figs. 17 & 18) GCM Query 1 features up to 4 groups. Its main difference compared to TPCH's Query 1 because the number of tuples in every window is comparable to the number of groups (the average window size is 42 groups). For this query, we measured SH's, AM's and cAM's scalability compared to PK-2, which is the current state of the art and is expected to be scalable due to the small number of groups (as in Section 3.7.1.1).

Figure 17 presents the percentage improvement in window latency of SH, AM, and cAM compared to PK-2. SH-w has its latency improvement decrease, because *aggregation cost* increases when more workers are employed. In contrast, AM and cAM have their latency decrease when the number of workers increases and they exhibit lower latency than PK-2. As Figure 18 indicates, AM's and cAM's scalability result from their constant *aggregation* cost while the partial evaluation latency decreases. The former is not the case with SH-w and PK-2, which have the *aggregation cost* percentage increase with the number of workers. **Take-away:** AM and cAM are scalable, maintain a constant *aggregation cost*, and outperform

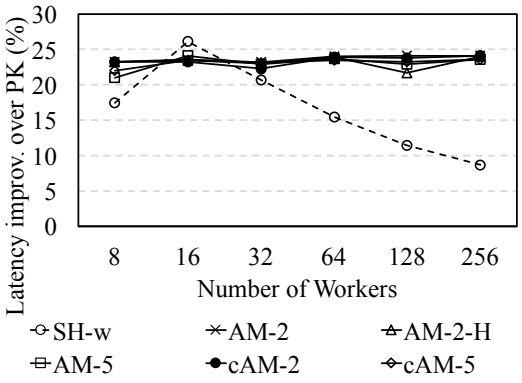


Figure 17: Latency Improvement (%) over PK for GCM Query 1.

PK-2 by up to 1.3x.

3.7.2.2 GCM Query 2 (Fig. 19) Figure 19 illustrates the percentage improvement in window latency of SH-w, AM, and cAM over PK-2. Even though this query contains a large number of groups (Table 3), its average window size is only 181 tuples and group repetition is scarce. Therefore, *M-choice partitioners* will not have their performance deteriorate due to an overwhelming *aggregation cost* (the case in TPCH Query 3-Section 3.7.1.2). However, SH-w is not scalable because when additional workers are employed its *aggregation cost* becomes higher. Turning to PK-2, it manages to be scalable, but it underperforms compared to AM and cAM in all worker settings.

Take-away: AM and cAM are scalable and present more than 1.4x better latency compared to PK.

3.7.3 Partition algorithm cost

In this set of experiments, we measure overhead imposed by each algorithm in terms of processing and memory cost. To that end, we picked DEBS Q1, because it features the longest group identifier (15 bytes), and the number of groups averages to millions per window.

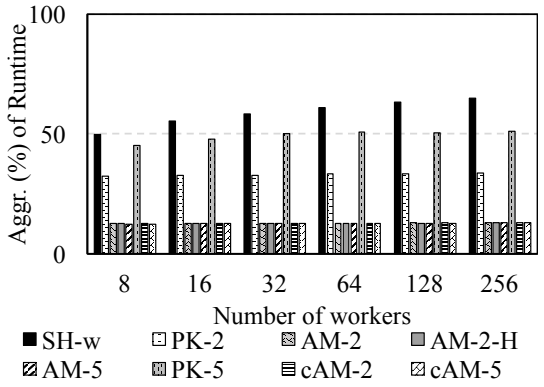


Figure 18: Aggregation runtime (%) for GCM Query 1.

3.7.3.1 Partition latency (Figure. 20a - 20c) To measure processing time, we marshaled DEBS data to each algorithm and measured partition latency on each window. As described in Section 3.4.2, the cardinality estimation structure size relies on the number of workers. Therefore, we measured partition latency for 8, 16, and 32 workers. Figure 20 illustrates the total time spent on each window with each partitioning algorithm. We included 90 and 99-percentile window latency. Most of the algorithms present constant values for 8, 16, and 32 workers. However, noticeable difference can be seen with 32 workers (Figure 20c) for the 99-percentile window latency of CM-2, CM-2-H, LM-2, and LM-2-H. The increase is a result of additional processing required for those algorithms.

Take-away: Using our proposed algorithms does not incur any noticeable overhead in latency.

3.7.3.2 Partition Memory (Table 4) Our proposed algorithms make use of cardinality estimation structures. Hence, we ran a micro-benchmark, in which we produced each possible key for DEBS Query 1 and replicated it to both available candidate workers. This experiment aims at examining an extreme scenario, in which all of the 8.1M groups appear in a single window. We measured memory consumed in MBs (Table 4). The naive cardi-

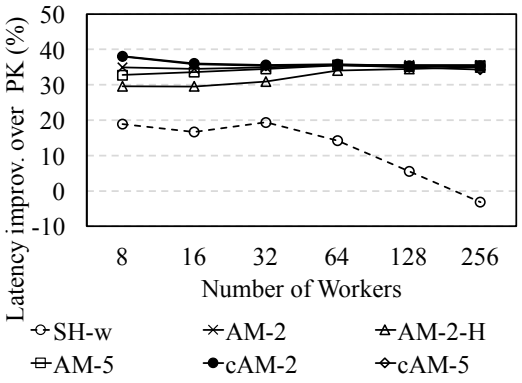


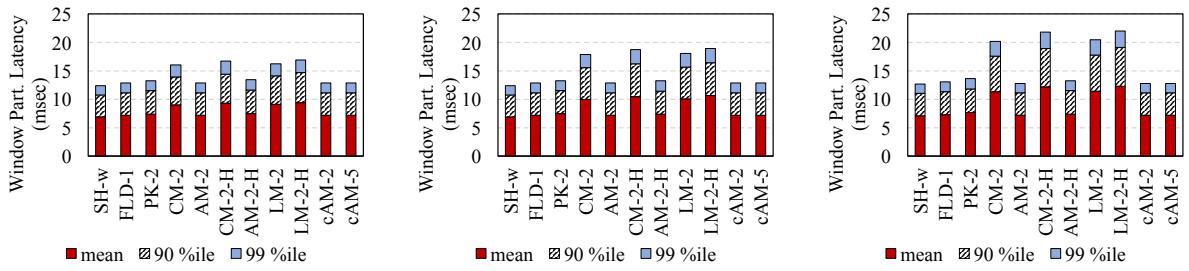
Figure 19: Latency Improvement (%) over PK for GCM Query 2.

nality estimation structure size quickly increases with the number of keys. Since each key is sent to both of the two candidates, the naive cardinality estimation structure's size increases further. Conversely, when HLL is used, memory consumption increases when the number of workers increases and its size does not get affected by neither the number of keys, nor the number of candidates. However, if the expected cardinality of the input stream is more than 10 million, then each HLL structure needs to double its number of buckets.

Take-away: Memory requirements of the cardinality estimation structure can be significantly limited with the use of HLL.

3.8 DISCUSSION

An SPE's performance can be affected by both *imbalance* and *aggregation* cost. According to our experimental results, the state of the art solution (i.e., PK) fails to perform well, when a large number of groups appears, and *1-choice partitioners* like FLD can make use of *key splitting* [21] to achieve better performance.



a: 8 workers b: 16 workers c: 32 workers

Figure 20: Per window Partition latency for DEBS Query 1 (i.e., processing cost of partitioning algorithm).

Maintaining low *imbalance* does not necessarily lead to limiting *aggregation* cost. Even if an "improved" and diverse load metric is used (i.e., CM with Equation 3.6 and LM with Equation 3.10), performance will degrade when the number of groups increases. In fact, *M-choice partitioners* underperform when a large number of keys appears, because they focus solely on minimizing *imbalance*. After conducting a sensitivity analysis on *M-choice partitioners* and their inaction for limiting *aggregation cost*, we found that they achieve minimal *aggregation cost* (i.e., $\Gamma(S_i^w) = O(\|S_i^w\|)$) only under specific circumstances. One such scenario is when tuples appear in an order that prohibits key sharing among workers. For example, S_i^w contains tuples with keys x and y, which map to 2 available workers w_1 and w_2. If all tuples with key x appear in even positions, and all tuples with key y appear in odd positions, then $\Gamma(S_i^w) = O(2)$. Another scenario in which LM achieves minimal *aggregation cost* is when keys are uniformly distributed and each one appears at most once. None of the data sets we used in our experiments demonstrated any of these scenarios and we could not find real-world data sets that behaved as such.

Addressing the important question of when to use each partition algorithm. As indicated by our experiments, SH is the best option for *stateful* operations, in which the expected number of groups is constant and far less than the window size divided by

Table 4: Memory requirements in MBs

DEBS Query 1			
no. of workers	8	16	32
CM-2	243	243	243
AM-2	122	122	122
LM-2	243	243	243
CM-2-H	0.02	0.04	0.08
AM-2-H	0.02	0.04	0.08
LM-2-H	0.02	0.04	0.08

the number of workers. Our results on TPCH Query 1 (Figure 12) agree with the previous statement, since SH performed the best and was the only scalable algorithm. Conversely, when a *stateful* operation involves a large number of groups, that constitute a significant percentage of total tuples on each window, AM and cAM have to be used. They offer minimal *aggregation* cost (like FLD), while leveraging the merits of *key splitting* to decrease *imbalance*. We codify our results into the following:

Stream partitioner selection Rule: *If the number of groups in a stateful operation is expected to be constant and significantly less than the average number of tuples assigned to a worker on each window, then SH must be used. Otherwise, AM and cAM will offer the best trade-off between imbalance and aggregation cost.*

In Chapter 7 we address the problem of *one partition algorithm for every scenario*, by presenting a novel partition technique that combines load shedding with stream *partitioning*.

3.9 RELATED WORK

Load Balancing: Seminal work on stream re-partitioning is presented in Flux [111], in which the Flux operator is presented for monitoring load on each operator. Compared to our work, Flux employs FLD to distribute tuples and is orthogonal to ours because we do not consider solutions that employ state migration and re-partitioning in the event of *imbalance*. Furthermore, Flux's model does not consider the *aggregation* cost of *stateful* operations. Closer to our work is PK, presented by Nasir et al. [98], which combines *key splitting* when tuple *imbalance* appears. Our proposed algorithms rely on a more complete model that considers both tuple *imbalance* and *aggregation* cost. Thus, as shown in Section 3.7.1, our proposed AM and cAM algorithms achieve better performance. Recent work on partitioning by Rivetti et al. [108] has presented a solution for online adjustment of SH. Their goal is to decrease processing latency by making better decisions of tuple counts. Our work differs from theirs in the sense that we aim to improve stream partitioning in terms of both *imbalance* and *aggregation* cost. The model presented in [108] does not consider *aggregation* cost, and it only focuses on SH, which is expected to underperform in *stateful* operations that involve multiple groups. In addition, THEMIS [72] presented a federated way of controlling load shedding to preserve balanced execution in a SPE. Our work is orthogonal to THEMIS because our goal is to improve tuple *imbalance* by altering the partitioning algorithm and not by shedding tuples.

Elasticity - Query Migration: Previous work that involves query (or state) migration [111, 137, 141, 63, 34] is orthogonal to this work. Our focus is on the performance of the partition algorithm and its performance in the event that migration is not an option. Nevertheless, our partition algorithms can be integrated as part of an SPE that allows for state migration, but is beyond the focus of our work. Work by Gedik [58] performs an extended study on partition algorithms for SPEs by improving state migration and adaptability by measuring skewness. That work is similar to ours in terms of monitoring input stream cardinalities, but, our model differs because it incorporates the *aggregation* cost in the decision process. Finally, a lot of work focuses on online elasticity of an SPE and adaptive behavior based on the input. Recent works [50, 86] aim to combine efficient stream parti-

tioning and load migration for distributed stream joins. Additional work has been done on scaling-out [136, 67, 68, 78], where SPEs are presented with the ability to detect struggling operators, decide on migration policies, and perform online reconfiguration of the execution plan. Online reconfiguration and state migration are outside the scope of this work.

3.10 SUMMARY

In summary, we presented a novel model for stream *partitioning* which considers tuple *imbalance* and *aggregation* cost for *stateful* operations. Inspired by it, we introduced novel partition algorithms, which adopt our model by keeping track of the cardinalities of each worker. Our experiments indicated that when the number of groups is large, our algorithms are significantly faster compared to both the state of the art (PK) and previously proposed solutions. We conclude that selecting a partitioning algorithm for a *stateful* operation has to be made after considering the expected number of distinct groups, and when that number is large, our proposed algorithms offer the best performance.

4.0 STREAM RE-PARTITIONING

In this dissertation, we use *stream re-partitioning* to describe all the operations that need to take place when an SPE detects that its allocated resources are not enough to process input in a timely manner. In this scenario, an SPE needs to (i) provision additional resources, (ii) update dataflow traffic, (iii) re-organize state of the participating workers, and (iv) resume normal operation. The aforementioned steps need to be completed without imposing significant performance degradation by pausing execution.

In this Chapter, we present our contributions to *stream re-partitioning*, which are two-fold. First, the design of an SPE enhancement, built to work with a widely used SPE, Apache Storm. This enhancement offers the ability to reconfigure the execution plan of an active CQ. This prototype delivers all the required functionality (i.e., the steps presented in the previous paragraph) and does not require changes in an existing Apache Storm setup. Second, a state re-configuration protocol for efficiently bringing up-to-date workers' states, during a stream *re-partition* operation (i.e., step (iii)). The SPE enhancement is named *Synefo*, and we present its details in Section 4.1; the state migration protocol, named UniMiCo, is presented in Section 4.2.

4.1 SYNEFO: ELASTICITY IN STORM

At the time of this work, a complete mechanism for stream *re-partitioning* of an operational CQ was not offered by any of the most popular open-source SPEs (i.e., Storm, Spark Streaming, Samza, Flink). To this end, we developed *Synefo*, which is a Storm [131] component[1]

[1] *Synefo*'s Storm components could be loaded as a `jar` library for a CQ.

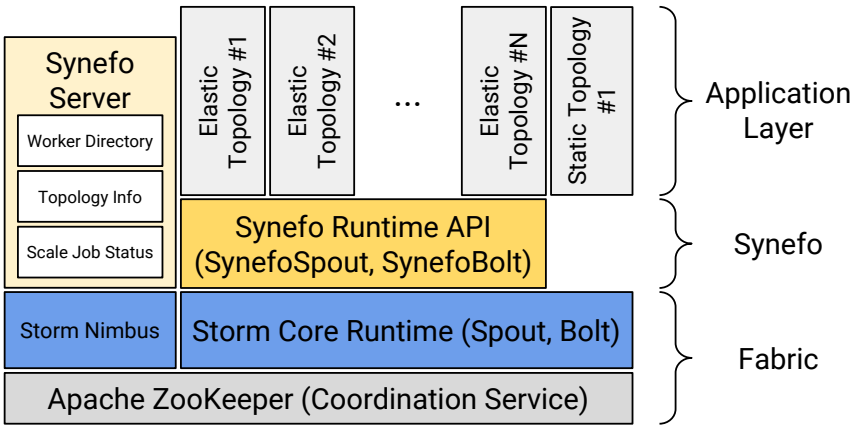


Figure 21: Synefo's system stack.

able to handle stream *re-partition* actions in a non-disruptive manner. In this dissertation, we will also call the stream *re-partition* action a scale-out/scale-in task of an SPE. *Synefo*'s goal is to offer scale-out/scale-in capabilities without having to make major changes to Storm's execution engine.

4.1.1 Architecture Overview

Synefo was developed as part of CE-Storm, which is a system that provides Confidential and Elastic processing on top of Storm (hence the initials CE). CE-Storm's full stack is depicted in Figure 21. At the bottom, Storm's [131] engine and an Apache ZooKeeper cluster provide the essential fabric that CE-Storm relies on. We preferred Storm's runtime over other SPEs (e.g., Spark Streaming [138]) because of its per-tuple processing model. On top of Storm, *Synefo* operates as an add-on library, and offers the elasticity properties for stateful processing, by managing tuple flow among the available `Executioner` threads. On top of *Synefo*, *CryptStream* [129] is the application layer, and is responsible for ensuring confidentiality and access control of streaming data, according to user policies. Both *Synefo* and *CryptStream* are designed independently and each one can operate by itself.

Synefo offers an elastic execution environment, which is able change the number of active processing components in an online fashion. This functionality requires the four essential operations:

1. Provision (or decommission) resources.
2. Update tuple routing information.
3. Update state of participating `Executioner` threads.
4. Resume normal operation.

These operations are serviced by *Synefo* by leveraging the underlying *fabric*: Storm runtime, and Apache ZooKeeper (Figure 21). *Synefo* uses the ZooKeeper cluster for (i) thread synchronization, and (ii) state checkpointing. In addition, *Synefo* utilizes Storm runtime's communication channels (i.e., groupings) for direct messaging among workers to control tuple routing and communication among threads. Also, to provision additional resources, *Synefo* makes use of "stand-by" `Executioner` threads, which are provisioned during CQ submission, but are not processing any data until needed. If needed, "stand-by" `Executioner` threads can be included in the processing pipeline. This approach is similar to the technique presented in [95] for scaling-out a VoltDB database. Listing 4.1 presents CQ registration for *Synefo* of the query depicted in Listing 2.1.

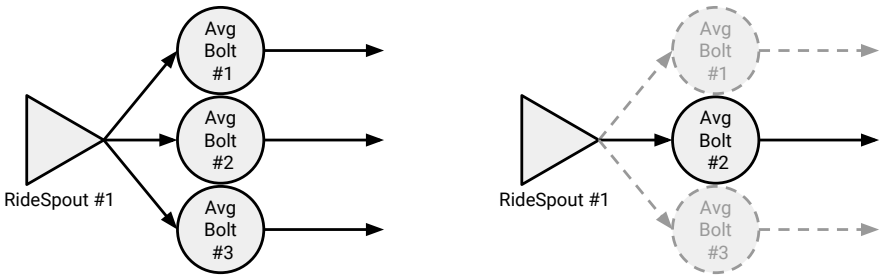
The Java code shown in Listing 4.1 illustrates CQ registration for *Synefo*, which is similar to Storm. The main difference is that with *Synefo*, from the three `AverageFare` instances, only one is going to be receiving tuples until input traffic gets high enough to require more instances to be enabled. As a result the other two instances will remain dormant until needed. In detail, in the code depicted in Listing 4.1 the Spout and the Bolt instances are wrapped by a `SynefoSpout` (Line 4), and a `SynefoBolt` (Line 5). Another difference is that each `SynefoBolt` instance gets two direct streams from the upstream SynefoSpout: one for data (Line 9) and one for control-flow messages (Line 10).

`Executioner` threads in *Synefo* are divided into *active* and *inactive*. An *active* thread is participating in processing, since *Synefo* is allowing tuples to be sent to it. On the other hand, an *inactive* thread is provisioned as part of a CQ's topology, but does not participate in processing, since *Synefo* is not allowing tuples to be sent to it. In essence, *Synefo*'s role

Listing 4.1: Part of the query of Figure 2 for *Synefo*.

```
1  SynefoSpout rideSpout = new SynefoSpout(new RideSpout());
2  SynefoBolt avgBolt = new SynefoBolt(new AverageFare());
3  TopologyBuilder builder = new TopologyBuilder();
4  builder.setSpout("rides", rideSpout, 1);
5  builder.setBolt("avg-fare", avg
6      .withTimestampExtractor(x -> x.getLong(0))
7      .withWindow(Duration.of(size), Duration.of(slide)),
8      3)
9      .directGrouping("rides", DEFAULT)
10     .directGrouping("rides", CONTROL);
11 return builder.createTopology();
```

reduces to managing the activity status of available threads and controlling the tuple flow in a CQ's topology. Every time a CQ is submitted to *Synefo*, the CQ's topology is broken into two parts: the physical topology, which consists of both the *active* and *inactive* threads; and the active topology, which consists of only the *active* threads (i.e., the ones participating in processing). In essence, every scale-out (or scale-in) command adds (or removes) threads to (from) the active topology. *Synefo* controls the participation of `Executioner` threads in processing by leveraging Storm's `DirectGrouping`. This way, *Synefo* is able to control the flow of tuples among threads (essentially, inactive threads do not receive any tuples). Figure 22 illustrates the physical and active topology for the CQ of Listing 4.1, right after topology submission to the available cluster. The physical topology (Figure 22a) carries a single Spout instance and three Bolt instances. In addition, all the communication channels between them are established. Conversely, the active topology at the beginning of execution (Figure 22b) illustrates the active components, which include a single Spout and a single Bolt instance. As a result, only one communication channel is operational and tuples pass through it.

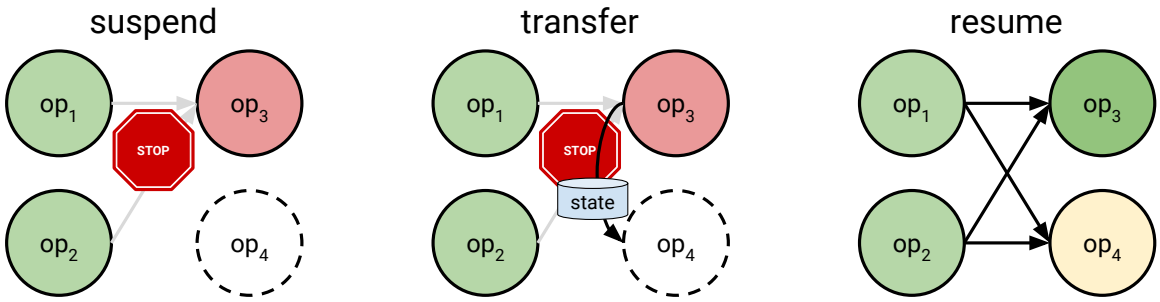


a: Physical Topology. b: Active Topology.

Figure 22: *Synefo* physical and active topology for the query defined with the code of Listing 4.1.

Synefo consists of a runtime API, which includes the *Synefo* Bolt and Spout interfaces, and a *Synefo* Server, which is a coordinating process for orchestrating the scale actions for an operational elastic topology (Figure 21). The Server maintains a directory of the active workers (i.e., threads), topology information, and the completion status of scale actions. It communicates with threads using the underlying Apache ZooKeeper framework. Our design choice to have *active* and *inactive* operational components emanates from Storm's design limitations and our initial goal to have *Synefo* work with existing Storm clusters. As can be seen in Figure 21, a user has to option to not make use of *Synefo*'s elastic capabilities, and schedule a topology with static resources, which makes use of Storm's default fabric (i.e., Storm Nimbus, Storm Runtime, and Apache ZooKeeper). Next, we present in detail *Synefo*'s main components: (i) *Synefo* Server, (ii) *Synefo* Spout, and (iii) *Synefo* Bolt.

4.1.1.1 Synefo Server The *Synefo* Server monitors resource utilization levels in an operational *Synefo* topology, and maintains an active role during the initiation of a CQ. In detail, it awaits for every thread in a topology to be provisioned by Storm's fabric (both *active* and *inactive*) and register to the Server. *Synefo* Server maintains a directory with all the `SynefoSpout` and the `SynefoBolt` instances. By the time all threads have connected and



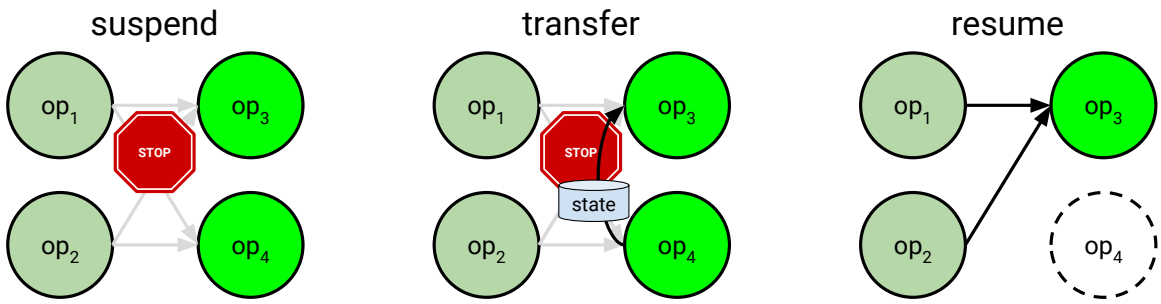
a: Suspend execution. b: Split state. c: Resume execution.

Figure 23: Scale-out operation in *Synefo*.

registered to the Server, topology information is gathered and execution initiates.

During execution, the *Synefo* Server periodically gathers performance information of each thread. When the need for a re-configuration in the *Active* topology arises, the *Synefo* Server drives the operation by sending out the appropriate commands to participating threads. Given a thread that is is going to have its status changed, *Synefo* Server identifies its upstream threads (i.e., threads that send tuples to it), and its sibling threads (i.e., threads that are on the same level in the topology). Then, it identifies the candidate thread(s) which will exchange state with the aforementioned thread, and then the Server notifies all threads accordingly. After this point, the *Synefo* Server does not take any further actions, and updates its directory when the scale action concludes.

4.1.1.2 Synefo Spout The runtime component responsible for disseminating data into the topology is a *Synefo* Spout. It has been designed as a sub-class of Storm's `BaseRichSpout`, which has been extended to: (i) initiate operation by registering to the *Synefo* Server, and (ii) perform scale-out/scale-in operations. In addition, during execution the *Synefo* Spout reports its telemetry information to the *Synefo* Server periodically, and is able to execute scale commands ordered by the *Synefo* Server. A downstream thread of a *Synefo* Spout is assumed to always be a *Synefo* Bolt, because a Spout represents a stream of incoming



a: Suspend execution. b: Merge state. c: Resume execution.

Figure 24: Scale-in operation in *Synefo*.

data in Storm. Every time a *Synefo* Spout receives a scale command, it updates its routing information accordingly. For instance, if a *Synefo* Spout is directed to scale-out by adding a Bolt in its active downstream threads list, the following steps take place:

1. **Suspend:** Stop dissemination of input tuples in the topology.
2. **Notify:** Inform all downstream bolts about the scale-out operation by providing the newly-activated threads id and location information.
3. **Update Routing Information:** Update routing table information, to include the newly added bolt in the active downstream threads list.
4. **Resume:** Await for a confirmation of participating threads that state has been shipped completely.

By the time a scale operation concludes, a *Synefo* Spout reports about the completion back to *Synefo* Server.

4.1.1.3 Synefo Bolt The *Synefo* Bolt is the last component of *Synefo*'s runtime and is similar to a *Synefo* Spout. Their main difference is that a Bolt is able to handle incoming command tuples for scale operations and exchange state with other *Synefo* Bolts.

Bolts are expected to maintain operator state, which is required by the *stateful* transfor-

mation dictated in a CQ. When a *Synefo* Bolt receives a scale command, its state will need to be shared. In a scale-out command state is split, whereas in a scale-in command state is merged. For instance, if a Bolt becomes *inactive*, its state needs to be shipped to the Bolt(s) that will take over its workload and those will have to merge the transferred state to their current state; whereas if a Bolt becomes *active*, it has to receive state from another Bolt, which has its state split. *Synefo* Bolts transfer state transparently from the operation which is currently executed in a component. It is the operator's implementation responsibility to utilize the received state accordingly.

Figure 23 illustrate the steps performed for a scale-out operation in *Synefo*. The circles represent *active* threads (i.e., receiving and processing tuples) and their color indicates their load: *the more red, the more loaded; the more green, the less loaded.* Threads op_1 and op_2 are sending their output tuples to op_3, which has exceeded its operational capacity (Figure 23a). At this point, *Synefo* Server sends a scale-out command indicating that op_3 should be scaled-out and op_4 will be activated. Threads op_1 and op_2 are notified that op_4 is going to be included to their active downstream threads list. At this point, transfer of tuples from op_1 and op_2 suspends (Figure 23a), and op_3 transfers part of its state to op_4 (Figure 23b). In essence, op_3's state is split in two parts, and one of them is sent to op_4 (i.e., op_4 will assume the responsibility for a subset of op_3's workload). After the state migration concludes, op_1 and op_2 are notified accordingly and they resume transfer of new tuples to both op_3 and op_4 (Figure 23c). At this point, the processing load is split between threads op_3 and op_4. During the scale process, incoming tuples to op_1 and op_2 need to be buffered and maintained until execution in their downstream layer (i.e., op_3 and op_4) resumes.

In the event that op_3 and op_4 are under-utilized, *Synefo* will issue a scale-in command (Figure 24). *Synefo* commences a scale-in operation that de-activates thread op_4, in order to avoid wasting operational resources. In a similar fashion to the scale-out operation, the first step is to suspend execution by stopping the transfer of tuples from threads op_1 and op_2 (Figure 24a). Then, the state of op_4 is shipped to op_3 and merged (Figure 24b). By the time the state is transferred completely, threads op_1 and op_2 are directed to exclude op_4 from their active threads list, and they resume tuple production by forwarding those only to op_3 (Figure 24c).

4.1.2 *Synefo* Summary

Synefo offers complete stream *re-partitioning* functionality to Storm, without requiring any changes in the former's runtime. Our prototype supports provision of additional resources by introducing the separation between *physical* and *active* topologies. In addition, it controls the state of a thread by utilizing Storm's direct-grouping mechanism, and utilizes to disseminate control-flow information among threads. Furthermore, it uses the existing ZooKeeper cluster as a state back-end and communication among threads.

Synefo supports scale operations by momentarily suspending execution. This approach to state migration is often called checkpoint-based [66, 34, 86, 136] or suspend-resume [49, 46, 79]. As has been previously documented in [34, 63], protocols for state migration that require temporary suspension of execution cause performance degradation (both in terms of processing latency [34] and throughput [63]). In addition, the effect in performance is directly affected by the size of the transferred state: *the more the state, the higher the performance degradation* [34]. Our initial experimentation with *Synefo* verified previous findings in terms of the high overhead of stream *re-partitioning* when the transferred state increases. As a result, we were motivated to investigate novel techniques for state migration, that do not require prolonged pauses in execution and are capable of accommodating complex window semantics.

4.2 UNIMICO

As discussed earlier, state migration is a crucial (and heavy) operation during a scale action for the stream *re-partition* process: it can affect the completion time of the scale operation and might lead to a prolonged delay in processing latency. The former occurs because the state migration time is analogous to the size of the state being transferred. In essence, the more the state, the more it takes to complete its transfer.

The key goal of the **Uni**nterruptible **Mi**gration of **Co**ntinuous Queries (UniMiCo) protocol is to avoid transferring state during the migration of the state associated with a CQ con-

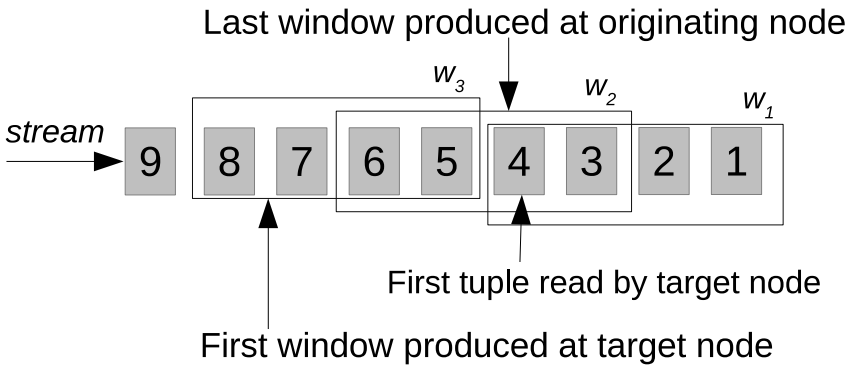


Figure 25: UniMiCo's migration strategy

taining stateful operators. To achieve this, UniMiCo migrates a CQ at the window boundary, meaning that the sender worker/thread (we will refer to it as the *originating node*) continues processing until it completes the last in-progress window, while the receiver worker/thread (we will refer to it as the *target node*) starts processing from the first tuple of the next window[2]. Given that two consecutive sliding windows overlap, the tuples belonging to the overlap of the two windows are processed by both the originating and the target nodes. This way, the state of the operator is reconstructed at the *target node* so there is no need to migrate it.

We illustrate this strategy in Figure 25. In this example, the sliding window of a stateful operator (e.g., aggregate) has a size of four seconds and a slide of two seconds, with input rate 1 tuple/second. The number in each stream tuple is its timestamp, which is assumed to monotonically increase over time (i.e. *in-order* processing of tuples). By the time the migration process starts, the most recent window produced is w_1, whose start timestamp is 1. In addition, the first tuple received by the target node after it connects to the stream has a timestamp of 4. UniMiCo determines that (1) the originating node will continue processing until w_2 expires, which happens to be the last window with start timestamp less than 4, and (2) the corresponding CQ at the *target node* will start processing tuples with timestamp

[2]For UniMiCo, we will use the term "node" instead of worker/thread because of our goal to expedite state transfer among workers that are not part of the same server node in a cluster.

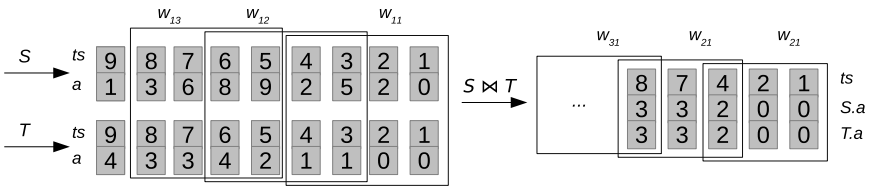


Figure 26: Calculating migration timestamp with two consecutive windows

greater or equal to 5 (w_3).

4.2.1 Migration timestamp

The migration timestamp marks a CQ hand-off from the *originating* to the *target node*. That timestamp is used to synchronize the stop of the last window at the *originating node* and the start of the next window at the *target node*.

Definition 1. *The* migration timestamp *is the start timestamp of the last window to be processed at the* originating node.

In the example of Figure 25, the start timestamp of w_2, which is 3, is the migration timestamp.

Calculating the migration timestamp The exact calculation of the migration timestamp depends on the implementation details of the window operation. Below, we present the way to calculate the migration timestamp on both time-based and tuple-based cases. In all the equations below, s denotes the slide of the window.

Time-based, single-input window: Assuming a time-based window of length l and slide s, let ts_{start} denote the timestamp of the first input tuple the stream source at *target node* was able to read after connecting to the stream. Furthermore, ts_{last_w} is the timestamp of the most recent window processed. The migration timestamp, denoted ts_{mi} is calculated as

follows (note that now s is in number of tuples):

$$ts_{mi} = \begin{cases} ts_{last\_w} & \text{if } ts_{start} \leq ts_{last\_w} \\ ts_{start} - \delta \text{ otherwise} \end{cases}$$
$$\text{where } \delta = \begin{cases} s & \text{if } (ts_{start} - ts_{last\_w})\%s = 0 \\ (ts_{start} - ts_{last\_w})\%s \text{ otherwise} \end{cases} \quad (4.1)$$

Tuple-based, single-input window: For tuple-based windows, the calculation is the same in the case when $ts_{start} \leq ts_{last\_w}$. When $ts_{start} > ts_{last\_w}$, UniMiCo needs to wait until a tuple t comes to the window operator, whose timestamp is equal to or greater than ts_{start}. This way, UniMiCo is aware of the number of tuples with timestamps between ts_{last_w} and ts_{start} (let that number be N). The migration timestamp can be calculated by the following equation:

$$ts_{mi} = timestamp(\delta^{th} \text{ tuple preceding } t)$$
$$\text{where } \delta = \begin{cases} s \text{ if } (N+1)\%s = 0 \\ (N+1)\%s \text{ otherwise} \end{cases} \quad (4.2)$$

Multiple-input window: The most common example of window-based operator with multiple inputs is a binary join. For time-based windows, Equation 4.1 can be used, with $ts_{start} = max(ts_{start_i})$, where ts_{start_i} is the timestamp of the first input tuple the stream source i at *target node* was able to read. For tuple-based window, the number of tuples N_i coming between ts_{start_i} and ts_{last_w} is calculated separately for each input i. Afterwards, Equation 4.2 is applied with $N = max(N_i)$.

Multiple window operators: A CQ can have multiple window-based operators with different window specifications (i.e., range and slide), such as a query with an aggregation on top of a join. For these cases, we introduce the concept of the *controlling window operator*.

Definition 2. *The* controlling window operator *is the closest window operator to the output of the CQ. The* controlling window operator *handles the calculation of the migration timestamp, as well as controlling the start and stop of the migrated query at the target and* originating node*s.*

For simplicity, we assume that the timestamp of an output tuple of a window-based operator is the *earliest* timestamp of input tuples involved in the calculation of that output tuple (we discuss later how this assumption can be relaxed). When the aforementioned condition holds, we know that all the original input tuples, contributing to the result produced by the farthest window of start timestamp ts, have timestamps greater/less than or equal to ts. Therefore, only the farthest window operator (i.e., the *controlling window operator*) in the CQ needs to be involved, and the calculation is the same as in the case of single window. Note that the previous assumption is not required for the *controlling window operator*.

Figure 26 shows an example of a CQ consisting of two window-based operators: a binary join, whose window has length of four seconds and slide two seconds, followed by an aggregation, whose window has length of three tuples and size of two tuples. For each tuple its timestamp is shown on the upper and its join key on the bottom part. For the controlling window, the most recent window being produced is w_{21}, whose start timestamp is 1 (i.e., $ts_{last\_w} = 1$). In addition, assume that out of the two first tuples read from S and T by the target node, the latest timestamp ts_{start} equals 5. In this case, the migration timestamp is calculated as if there is only the *controlling window operator* (i.e, the aggregation) with two inputs S and T. Because the *controlling window operator* is tuple-based, UniMiCo has to wait until tuple t of timestamp 7 arrives to know that there are 3 tuples whose timestamps are between 1 and 5, i.e., $N = 3$. Applying the calculation from Equation 4.2 for the case of tuple-based window, UniMiCo decides that the migration timestamp is that of the tuple preceding t, which is 4. In other words, the last window produced at target is w_{21}.

When the previous condition on output tuples' timestamps of preceding window operators does not hold, ts_{start} is measured as the timestamp of the first tuple arriving at the *controlling window operator* on the *target node*. Recall that when this condition holds, ts_{start} is the timestamp of the first tuple coming to the source operator, i.e., it can be captured earlier. With the new ts_{start}, all of the above calculations of the migration timestamp are still applicable. Note that in this case if ts_{start} is smaller than ts_{last_w}, there will be some wasted processing at the target to process tuples from source up to the controlling window between ts_{start} and ts_{last_w}. Because migration happens when the target is lightly loaded, it is expected that processing at the *target node* will be at least as fast as that at the *originating*

Algorithm 6 UniMiCo protocol at *originating node*

1: **procedure** UNIMICO_AT_ORIGIN(TARGET, $\mathcal{Q}$)
2: $send$(TARGET, MIGRATE, $\mathcal{Q}$)
3: $receive$(TARGET, ts_{start})
4: $ts_{mi} = calculate\_migration\_timestamp()$
5: $send$(TARGET, ts_{mi})
6: $finish\_process(\mathcal{Q}, ts_{mi})$

node, hence the wasted processing, if any, would be small.

4.2.2 Stoping and resuming continuous queries

An important operation for stream *re-partition* is the suspend and resume of the nodes participating in the *re-partition* process. As was the case in *Synefo*, when the state has successfully been transferred, one worker thread will have to either start execution (scale-out) or stop (scale-in). The same is required for UniMiCo.

4.2.2.1 Stopping the query at the originating node Once the migration timestamp is determined, stopping the query at the *originating node* is relatively straightforward: all operators in the CQ continue to process normally until they receive the signal from the *controlling window operator* to deactivate themselves, unless they are shared with other CQs. This happens when the *controlling window operator* has consumed its last window, i.e., the window started with the migration timestamp. Shared operators are never deactivated and continue to process normally after migration. Upon stopping, an operator cleans up all its queues. Algorithm 6 presents an overview of the procedure that takes place in the *originating node*.

When the *controlling window operator* is associated with a join, a minor adjustment is needed in order to avoid duplicate outputs between the *originating* and *target node*s. Normally, when there is a match between a tuple t of one input and t' of the other, the join tuple tt' is produced only once, even if both t and t' fall in the overlap of two (or more)

Algorithm 7 UniMiCo protocol at *target node*

1: **procedure** UNIMICO_AT_TARGET(ORIGIN, $\mathcal{Q}$)
2: $receive(\text{ORIGIN}, \text{MIGRATE}, \mathcal{Q})$
3: **for all** $s \in \mathcal{Q}.streams()$ **do**
4: $connect(s)$
5: $ts^s_{start} = read(s)$
6: $send(\text{ORIGIN}, ts_{start})$
7: $receive(\text{ORIGIN}, ts_{mi})$
8: $resume(\mathcal{Q}, ts_{mi})$

consecutive windows. If we start migrating from one of the windows, the join tuple tt' will be produced once at the *originating node*, and again at the *target node*. In the latter case, the production of a duplicate tuple is avoided by suppressing the production of the join result at the *originating node*. Note that when two matching tuples have their timestamps in the window overlap, the previous adjustment is needed only if the join is the last window-based operator in the query. In the event that a join is followed by another window operator, the duplicated intermediate output tt' is needed, as it is an input for the subsequent window at the *target node*.

4.2.2.2 Starting the query at target node

The operators of a migrated CQ can be activated at the *target node*, as soon as the migration is initialized. However, full activation is attained by controlling the flow of tuples based on the migration timestamp. That process is different for time- and tuple-based windows, as we describe below. Algorithm 7 illustrates the steps taken at the *target node* when a CQ migration is initiated.

Time-based *controlling window operator*: If the CQ has a time-based *controlling window operator*, the stream source operator(s) calculate(s) the activation timestamp as migration timestamp increased with the slide of the window. Then, the stream source operator discards any input tuples, which carry timestamps less than the activation timestamp. In addition, it starts producing tuples with timestamp equal to or greater than the activation timestamp. With tuples being outputted from the stream source(s), the query is fully

activated.

Tuple-based *controlling window operator*: In this case, the stream source operator(s) start(s) producing results from tuples with timestamps greater than the migration timestamp. But, the *controlling window operator* will discard all first $(s - \delta)$ tuples, where s is the slide of the window and δ is calculated from Equation 4.2 by the *originating node*.

For both window types, if the output timestamp of the preceding stateful operator is not the window's start timestamp, the *controlling window operator* has the single authority that decides when to output tuples. Thus, the source operator cannot do any early filtering.

4.2.3 Experimental Evaluation

While UniMiCo enhances WRP's functionality, at the same time it inherits from WRP both its performance advantages and its limitations. For example, both protocols are equally effective in migrating operators with small range windows and less so with large ones [63]. In light of this, our experimental evaluation focused on UniMiCo's correctness as well as to show that UniMiCo migrates CQs correctly with single and multiple stateful operators without significantly impacting in their response time for small range windows where UniMiCo is applicable.

4.2.3.1 Experimental Setup We implemented and evaluated UniMiCo in a distributed setup of AQSIOS. As discussed in [1], the window operator in AQSIOS is a separate operator, which receives stream tuples as input, and injects *minus* tuples to the stream to mark the boundary of a window. Windows can have either time-based or tuple-based length, but the window slide is always one tuple. Therefore, window-based operators, such as join or aggregation, will rely on those minus tuples to perform their window-based processing. With the separation of the window operator, each input to a join operator can have a window of different length and type. In the scope of this dissertation, we assume that join inputs have windows of the same length and the same type, however, our design can be extended to heterogeneous window environments.

We ran two types of experiments, one with simple CQs consisting of a single window

```
Select *
From S
    [Range 10 seconds],
     T
     [Range 10 seconds]
Where S.1 = T.1;
```

a: Query 1

```
Select SUM(m)
From S [Rows 5];
```

b: Query 2

```
Select SUM(S.m)
From ISTREAM
    (Select *
     From S
         [Range 10 Seconds],
          T
         [Range 10 Seconds]
     Where S.1 = T.1)
      [Rows 5];
```

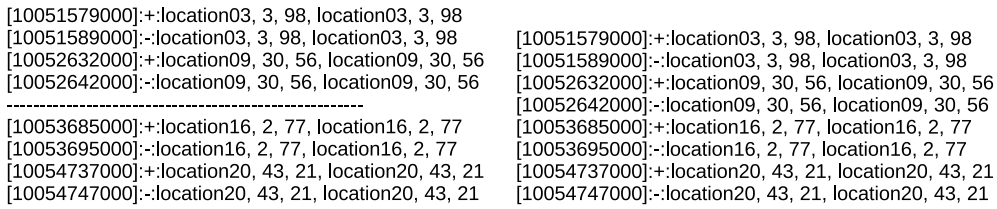
c: Query 3

Figure 27: Queries used in our experimental evaluation.

operator and another with a complex CQ consisting of two window operators. We ran each CQ twice, under the same settings, and changed only whether a migration took place. Then, we compared CQs' outputs and response times around the migration point. All settings were the same between the two runs.

4.2.3.2 Simple CQ migration (Figures 28, 29, 30, & 31) We used UniMiCo to migrate a CQ with a join operator (Q1), and another one with an aggregate operator (Q2). These two queries are presented in CQL [18] are depicted in Figures 27a and 27b. `S` and `T` are input streams. Q1 is associated with time-based windows with size 10 seconds (i.e., [Range 10 seconds]) whereas Q2 is associated with a tuple-based window of size 5 (i.e., [ROWS 5]).

Figures 28 and 30 show the results of Queries 1 and 2 around the migration point, respectively. Figure 28a illustrates is the result under migration, in which the rows above the dash line are the last output tuples at the *originating node*, and those below the dash line are the first output tuples at the *target node*. Figure 28b shows the result without migration, which is exactly the same as the concatenation of the two parts of the top plot. Similar observations can be made in Figure 30 for Query 2. As one can see, the correctness of the output is maintained by using UniMiCo, and its protocol succeeds in performing the hand-off without losing any data.



a: With Migration. b: Without Migration.

Figure 28: Results of Query 1 around the migration point (i.e., 10 second mark).

The plots of Figures 29 and 31 show the response time of queries 1 and 2 two seconds before and after the migration point of about the 10^{th} second. As can be seen in both figures, there are no noticeable hiccups in the response time of the queries throughout the migration. For Query 1, the average and standard deviation of the response time in this period without migration is 3.751 and 3.99 milliseconds, respectively, while under migration they are 3.750 and 3.97 milliseconds. For Query 2, the corresponding numbers are 3.155 and 3.923 milliseconds without migration, and 3.101 and 3.836 milliseconds with migration. In both cases, the difference is negligible.

4.2.3.3 Complex CQ migration (Figure 32) In this experiment we migrated a more complex query Q3, consisting of a join and an aggregate operator, each using a different window definition as shown in Figure 27c. In this case, the last window, which is the tuple-based window of size 5 (i.e., [ROWS 5]) associated with the aggregation, plays the role of the *controlling window*. Figure 32 shows the output tuples and the response time of the query Q3 around the migration point, compared with the run when there is no migration. Similar to the cases of the simple queries, the query output is preserved and the cost of migration is not noticeable. The average and standard deviation of the response time without migration are 6.568ms and 6.133ms respectively, while those with migration are 6.658ms and 6.217ms.

Take-away: With UniMiCo the migration of a CQ has effectively no impact in processing

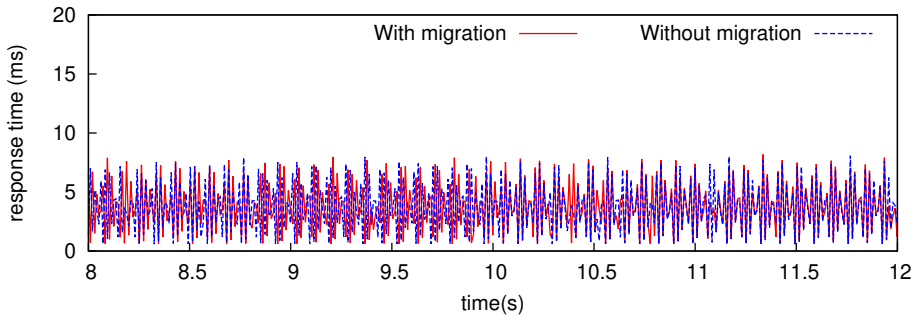


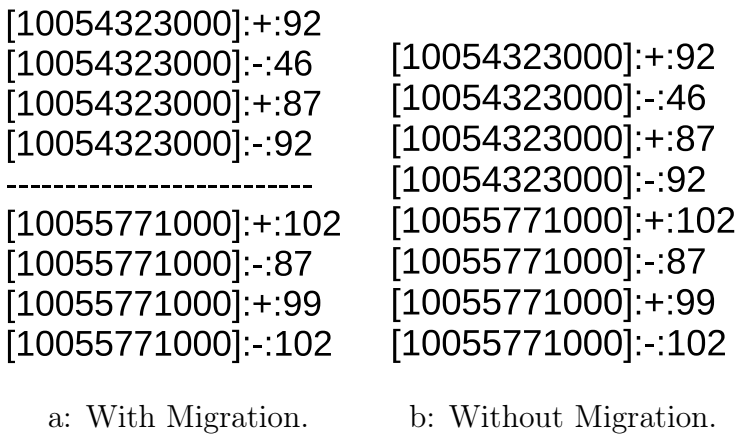
Figure 29: The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay.

times. Figure 33 presents the average response times for all the queries we used in our experimental evaluation.

4.2.4 UniMiCo Related Work

Workload Migration has been previously proposed in many different contexts as a way for alleviating the load of single machines, and moving it to different sites. Examples of migration can be seen in Virtual Machines VMs, SPEs, and distributed storage systems. Due to the lack of space, we are going to focus of previous migration approaches in SPEs.

Migration of queries and operators in SPEs before and has been coined as the default mechanism for load balancing. Flux [111] was one of the early attempts to introduce a monitoring/detection opetator in a query network, and also provided a state migration protocol, to move CQs on different machines. In additon, Fernandez et al. [34] has presented a solution in which, backup (VMs) are used in a distributed network of VMs running a SPE are used to periodically store state. In the event of load imbalance, CQs are migrated by receiving the state from those, and they resume execution by the time the full state has been transferred, along with incremental changes. Streamcloud [63] is a full-fledged SPE prototype that supports two types of migration: no-state and state migration. The latter is similar to the technique used previously, where state needs to be transferred from the



a: With Migration. b: Without Migration.

Figure 30: Results of Query 2 around the migration point (i.e., 10 second mark).

source to the destination. The former, is similar to UniMiCo in the foundation that no state is transferred. However, UniMiCo presents a more general model that supports both tuple-based and time-based windows, does not limit the migrated subquery to only one stateful operator, and does not need the use of an upstream module (i.e, the load balancer in [63]) to synchronize the start and stop of the migration. Finally, the work of Lin et al. [86] discusses distributed theta joins in SPEs and they briefly discuss a migration mechanism between operators. Again, their approach follows the state migration paradigm and significantly differs from the no-state transfer approach of UniMiCo.

4.2.5 UniMiCo Summary

In this Chapter, we presented UniMiCo, a general migration protocol for CQs, used in distributed SPEs that support elasticity. Compared to previous work, UniMiCo is more general because it is applicable to CQs with different window semantics, and can be used for migrating a plethora of stateful operations, without any constraints. Migration is performed without the need to transfer state or stop processing input tuples during CQ hand-off. Finally, our preliminary experimental evaluation demonstrates UniMiCo's feasibility, by implementing it

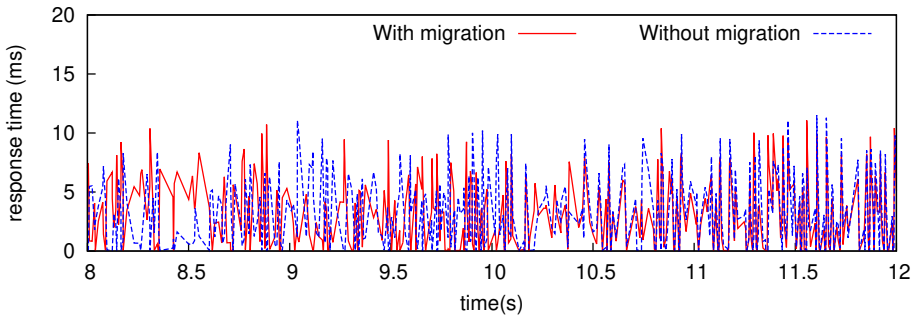


Figure 31: The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay (similar behavior with Query 1)

in a full-fledged prototype SPE (AQSIOS). On top of that, our experiments show that the migration of a CQ does not incur any noticeable disturbances in performance, while maintaining the correctness of output results. In addition, we presented the design of *Synefo*, which is a prototype system operating on top of Storm, and supports stream *re-partitioning*.

[10022574000]:+:109	
[10022574000]:-:104	[10022574000]:+:109
[10022574000]:+:104	[10022574000]:-:104
[10022574000]:-:109	[10022574000]:+:104
----------	[10022574000]:-:109
[10028529000]:+:107	[10028529000]:+:107
[10028529000]:-:104	[10028529000]:-:104
[10028529000]:+:70	[10028529000]:+:70
[10028529000]:-:107	[10028529000]:-:107

a: With Migration. b: Without Migration.

Figure 32: Results and response time of the complex query Q3 around the migration point of 10 second mark. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration does not add any noticeable delay

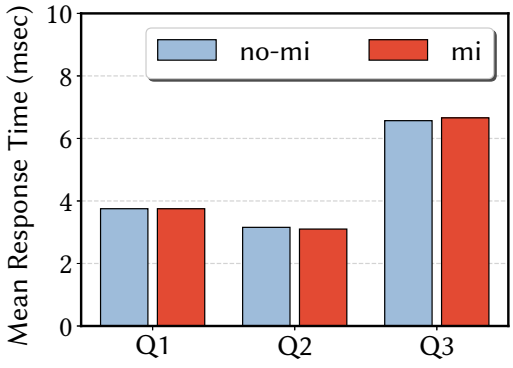


Figure 33: Summary of mean response times for all Queries when no migration takes place and when UniMiCo is used to migrate state.

5.0 APPROXIMATE STREAM PROCESSING (WITHOUT GUARANTEES)

In Chapters 3 and 4 we presented our work on stream *partitioning* and *re-partitioning*. Both of those techniques are targeted for maintaining the performance of an SPE at acceptable levels (as specified by the user) and produce *exact* results. In this Chapter, we present our work on load shedding, which is a technique aimed for situations when an *approximate* result is acceptable by the user. Hence, load shedding is applicable when the need of *exact* processing is not paramount and performance-oriented SLOs take priority.

In Sections 5.1 and 5.2 we discuss the need for load shedding, along with the limitations of existing techniques. Next, in Section 5.3 we propose our model for *concept* and we formalize the model for load shedding in Section 5.4. Then, in Section 5.5 we illustrate existing load shedding algorithms, and in Section 5.6 we present the details of our algorithm. In Section 5.7 we discuss our experimental evaluation, followed by related work (Section 5.8) and this Chapter's summary (Section 5.9).

5.1 CONTINUOUS QUERIES WITH SERVICE LEVEL OBJECTIVES

As mentioned earlier, SPEs [41, 25, 8, 37, 17, 7, 63, 16, 138, 97, 131, 83, 36, 136, 30, 82, 31, 2, 5, 15] have been developed to accommodate real-time processing for continuous queries (CQs). Then, an SPE's responsibility is to produce results that match the CQ until they are requested to stop. A typical use-case for SPEs is when users require real-time insights for online decision making. For instance, an analyst in a ride-sharing company might be interested in the latest trending routes in a city. Trends will help the analyst adjust her

decisions in order to decrease operational costs and increase revenue. Thus, she could submit the CQ shown in Figure 2.

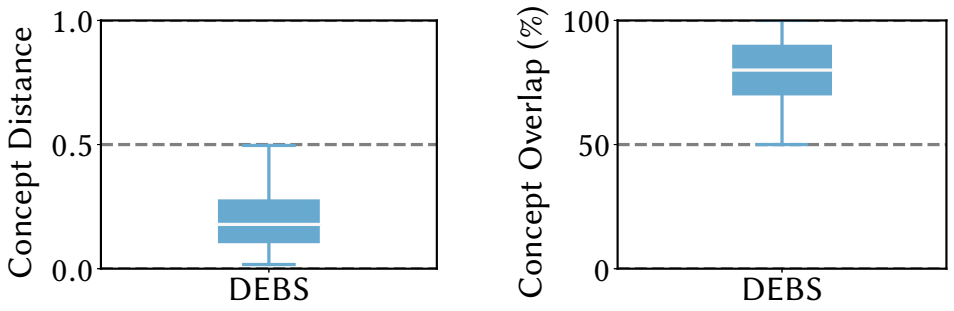
Considering the volatile nature of the input, the workload imposed to the SPE might become exorbitant and lead to violation of a CQ's Service Level Objectives (SLOs). Those SLO's might come in the form of latency requirements (i.e., deadlines), or in the accuracy of the result. In our example, a temporal event might increase the *Volume* and the *Velocity* of the rides stream, and in turn delay the production of results. To address this problem, a lot of work has been done to enhance SPE's adaptability. The most popular solution, involves elastically adding more resources to accommodate processing demands [110, 63, 34, 78, 107]. This solution has been shown to achieve the expected results in the long run, but causes significant performance overhead when used for short-lived input spikes.

5.2 THE NEED FOR LOAD SHEDDING

A more suitable technique for short-term bursts is load shedding, which dictates dropping input tuples to produce results in a timely fashion at the expense of the results' accuracy [124, 24, 59, 125, 132, 123, 119]. This accuracy drop is in agreement with data analysts' needs, which often focus on the qualitative relationships among groups rather on the exact numerical result [104]. This entails that in the example query of Figure 2 the analyst is mostly interested in the order of the top routes, rather than the average fare per route. The main challenge of load shedding is to downsample input without deteriorating the accuracy of a result.

5.2.1 Shortcomings of existing techniques

Existing load shedding techniques require either (a) user input regarding individual tuples' utilities [124] or (b) historical data accumulated over time [24, 45, 119]. The first requirement is impractical for CQs that require extracting patterns in real time. For instance, in the example query of Figure 2 expecting the user to provide a tuple's utility based on the route is equivalent to requesting the top routes of each window. The second requirement is



a: Distance b: Overlap

Figure 34: Real-world datasets feature varying *concept drift* characteristics, in terms of both inter-window concept distance and overlap.

impractical due to the fact that real-world streams evolve over time and relying on historical data accumulated over time will minimize or even ignore concept drift, which we explain in the next section.

5.2.2 Concept Drift Evidence

A data stream's evolution over time is called *concept drift* and is a frequent phenomenon in the real world. As evidence, we produced the results of the example query presented in Figure 2 on a real-world dataset. This dataset consists of taxi data from a New York City Taxi Company, which along with the CQ of Figure 2 were part of the ACM DEBS 2015 Challenge [71]. We ran this query on the whole dataset and produced the result for each window. Figure 34 illustrates the mean set difference (i.e., "concept distance") and mean set intersection (i.e., "concept overlap") among consecutive window results. It is apparent that the top routes (i.e., "concept") change among consecutive windows by up to 50%. This indicates that load shedding techniques, which rely on historical information to prioritize shedding, are destined to deteriorate. As a result, uniform load shedding is the

only applicable technique in current SPEs, which is also stated in the work of Fu et al. [54].

5.3 CONCEPT DRIFT DEFINITION

In the past, various definitions have been given to the *concept* of a stream. The most prevalent defines *concept* either as the underlying mechanism generating stream data, or the concept that is learned by applying $\mathcal{Q}$ to an input stream [134]. Naturally, *concept drift* describes the phenomenon when data are produced by non-stationary distributions [80] and evolve over time. Previous work on identifying *concept drift* in continuous streams, perceives the stream S_i as a continuous signal of information, without leveraging the windowing properties of $\mathcal{Q}$ [134, 56]. This constitutes the *concept drift* identification process difficult, because it requires the segmentation of S_i to "drift periods", which is a computationally heavy process.

We follow a more practical approach to *concept drift*, which leverages the windowing semantics of $\mathcal{Q}$, whose goal is to transform the input stream to the desired output. Therefore, an input stream S_i can be broken down to its window segments S_i^w and *concept drift* can be monitored on the window boundary. Thus, given a $\mathcal{Q}$ and its window definition $\mathcal{W}_\mathcal{Q}$, we define the *concept* of a window w as C_w (i.e., leverage the window boundary). Concretely, each window S_i^w will have its own concept C_w. In the example query shown in Figure. 2, its concept drifts on average by 30% on every window as shown in Figure 34. *Concept drift* can be quantified using two dimensions. First, *concept overlap* is the fraction of common elements among C_w and C_{w+1}. Second, *concept distance* is the similarity of C_w and C_{w+1} in terms of higher-level characteristics other than *concept overlap*.

Our query-driven definition of *concept* indicates that among different stateful operations, the notion of *concept* is different. Table 5 provides an overview of three of the most popular stateful operations in SPEs[1], our definition of *concept* C_w, and the utility of the former in load shedding.

Starting from a mean-like scalar aggregation (e.g., mean, count, variance etc.), C_w is

[1]Those operations are the stateful operations offered by Storm, Heron, Flink, and Spark Streaming.

Table 5: Examples of concept per query type

Operation	**Concept**	**Utility for Load Shed**
Aggregation	(μ, σ^2)	Sampling Rate (b)
Group Aggr.	Group Frequencies	Group Sampling Rate (b_g)
Equi-Join	Group Frequencies	Group Sampling Rate per stream

determined by the measures of central tendency of the distribution followed by the participating attributes in the computation. Babcock et al. [24] has shown that when $\mathcal{Q}$ is a scalar aggregation, measuring the mean value (μ) and the variance (σ^2) of the distribution can provide useful insights for approximating its value: identify the proper sampling rate b for window S_i^w.

Turning to grouped aggregations, like the one appearing in the example query of Figure 2, C_w is the frequency of appearance of each group. This provides information about the distribution of groups in S_i^w and can be used to create a stratified sample for S_i^w when tuples need to be shed. This technique has been previously used in *Approximate Query Processing* for Data Warehouses [13, 105] so that an approximate value appears of each group. In order to build a representative stratified sample, the *concept* determines the sampling rate b_g for every group g in a window S_i^w.

When a *stateful* operation is an equality join, the group frequency on each of the input streams can indicate the size of the result and the number of tuples per matched key [12]. Similar to a grouped aggregation, the *concept* is the frequency of each group appearing on each stream. This can be used during load shedding to identify the sampling rate of each group on each stream.

5.4 LOAD SHEDDING

In this section, we will present load shedding and formalize it as a minimization problem. An SPE enhanced with a load shedding mechanism comes with a load detection component. When the former detects that load exceeds the available processing capacity, it commences shedding tuples [8, 124, 24, 106]. An end-to-end load shedding solution has to decide (a) **when**, (b) **where**, (c) **how much**, and (d) **what** to shed [124].

The decision for the **when** question locates the point(s) in time most beneficial to commence load shedding, or can be a manual command given by the user. More often than not, this decision is independent with the rest of the operations and has to do with the monetary budget or available resources. The decision for the **where** question finds the position in the execution DAG of $\mathcal{Q}$ to place shed operations. In previous work, load shedding took place in the source operators only [54]. This approach benefits from the fact that results produced are easier to justify, especially when they are enhanced with information from other sources. However, their accuracy is difficult to justify when sliding windows are used. For instance, in the query of Figure 2 dropping a tuple might cause different windows to be produced. In turn, this will lead to missed window results. On the other hand, if load shedding takes place in a stateful operator of $\mathcal{Q}$, then results are produced based on the window boundaries of the CQ [125]. In our work, we follow this approach because we target applications that require production of results on every window. Finally, the decision for the **how much** question relates to the available processing capacity, the runtime complexity of $\mathcal{Q}$, and the input rate of source streams. In this work, we consider that to be an expert's decision, and assume that the shedding rate is given to the SPE by the user. Currently, we are investigating dynamic methods for determining the shed rate in an online fashion.

Previously-proposed load shedding techniques aimed at maintaining results accuracy and focused on **what** to shed. For example, *semantic* load shedding prioritizes tuples for shedding [124]. Similarly, the system presented in [106] chooses a different shedding rate for classes of CQs with varying SLOs. When tuples are shed, the accuracy of the result is affected and there are various metrics for measuring error. For a given $\mathcal{Q}$ and a S_i^w, we will represent the result after processing all tuples as R_w. With load shedding in place, which we

Algorithm 8 Uniform Shedding

1: **procedure** UNIFORMSHED(S_i^w, b)
2: $T'_w \leftarrow Sample(S_i^w, b)$
3: **return** T'_w

will represent as $\mathcal{S}$, a subset of tuples is processed. In that case, an approximate answer for S_i^w is produced, which we will illustrate it as $\hat{R}_w$. Depending on the stateful operation of $\mathcal{Q}$, there are various metrics of quantifying $\hat{R}_w$'s error with respect to $\mathcal{Q}$. In our formulation, we will represent the error metric as $\epsilon_{\mathcal{Q}} : R_w, \hat{R}_w \rightarrow [0, 1]$. In essence, the lower the error, the more effective $\mathcal{S}$ is.

5.4.1 Focal point of this work: what to shed

In this work, we focus on providing an answer to the question of **what** to shed. Our goal is to improve the accuracy of load shedding by prioritizing shedding of tuples based on their contents. In order to improve accuracy, an SPE needs to be aware of each window's *concept* C_w. Concretely, this is equivalent to knowing the utility QoS graph required by *Semantic* load shedding [124]. This way, load shedding would be prioritized based on tuples' importance according to C_w. We codify the problem of **what to shed** as the following minimization problem:

$$\begin{aligned} \underset{C_w}{\text{minimize}} \quad & \epsilon_{\mathcal{Q}}(R_w, \hat{R}_w) \\ \text{where} \quad & R_w = \mathcal{Q}(S_1^w, \ldots, S_l^w), \\ & \hat{R}_w = \mathcal{Q}(\mathcal{S}(S_1^w, \ldots, S_l^w, C_w)) \end{aligned} \tag{5.1}$$

In Equation 5.1, $\epsilon_{\mathcal{Q}}$ is the error metric for the result of $\mathcal{Q}$, R_w represents the result for S_i^w without shedding any tuples, and $\hat{R}_w$ represents the result for S_i^w when tuples are shed. $\mathcal{S}$ is the load shedding operation and C_w is the window's *concept*. In essence, Equation 5.1 states that **"what to shed"** is the search for C_w on every window, which can be a hard problem due to the uncharted future inputs and the existence of *concept drift* (Figure 34).

5.5 EXISTING SHEDDING ALGORITHMS

Current SPEs offer strict delivery semantics, which usually lead to applying the processing logic into ordered windows [138, 83, 36, 15, 30]. As mentioned in Section 2.2, tuples are assigned to windows according to $\mathcal{W}_{r,s}$, and the processing logic of $\mathcal{Q}$ is applied to a complete window. For simplicity, we present load shedding algorithms on complete windows (similar to the approach of [125]) instead of single tuples at a time (in Section 2.4 we presented the internal operations of forming a window in an SPE).

5.5.1 Normal Execution (Baseline)

In order to approach the minimization problem of Equation 5.1, we establish normal execution as our *Baseline*. This would represent normal execution on S_i^w without shedding any tuples (i.e., $\mathcal{S} : S_i^w \rightarrow S_i^w$ in Equation 5.1). The *Baseline* features zero shedding runtime overhead since the whole window is processed, and maximum accuracy (i.e., $\epsilon_{\mathcal{Q}} = 0$). With *Baseline* there is no need to estimate C_w and its drawback is that the data volume sent for processing will be the whole window.

5.5.2 Uniform Shedding (State of the Art)

As discussed in Sections 5.1 and 5.4, when C_w is unavailable and the input streams(s) experience *concept drift*, only *Uniform* load shedding can be applied. Recent work presented in [54] explains that *Uniform* load shedding is in fact used in industrial setups. Therefore, in this work we consider *Uniform* load shedding as the state of the art.

Uniform load shedding materializes by extracting a uniform sample from S_i^w and sending it for processing. Algorithm 8 presents an outline of *Uniform* load shedding which receives two arguments: S_i^w and a shed parameter b. The latter can either represent a shed bias (i.e., probability) or a percentage of S_i^w that will be processed. In Algorithm 8, a call to method *Sample* (line 2) creates a uniform sample (e.g., binomial, reservoir etc.).

Uniform load shedding makes no effort in estimating C_w. As a result, the error $\epsilon_{\mathcal{Q}}$ is sensitive to the sampling rate b. *Uniform*'s runtime cost is $O(c|S_i^w|)$, where c is the

Algorithm 9 Concept-driven load shedding

```
1: procedure CoDShed(S_i^w, b, Q)
2:     T_w ← Sample(S_i^w, b)
3:     C_w ← Q(T_w)
4:     T'_w ← ∅
5:     for all t ∈ S_i^w do
6:         if ¬Shed(t, C_w) then
7:             T'_w ← T'_w + t
8:     return T'_w
```

cost of the random process. Finally, the data volume sent for processing is proportional to the bias parameter b: *the higher the bias, the higher the processing cost*. *Uniform* load shedding presents an error-agnostic approach and focuses solely on reducing the processing load imposed to the SPE. As a result, the choice of the bias parameter is extremely important because it directly impacts both the performance and the result accuracy.

5.6 CONCEPT-DRIVEN LOAD SHEDDING ALGORITHM

In order to improve accuracy, an SPE needs to be able to estimate C_w on every window w. Our proposed load shedding technique is named **Co**ncept-**D**riven (CoD) and estimates C_w by applying $\mathcal{Q}$ on a uniform sample of S_i^w. Concretely, CoD decouples sampling from shedding to get insights for improving accuracy. CoD is motivated by work done on approximate query processing, such as Aqua [9] and BlinkDB [13]. Those run queries on samples extracted from the dataset and provide approximate answers. In contrast, CoD uses a sample to estimate C_w, which in turn is utilized for shedding tuples.

Algorithm 9 outlines CoD, whose input consists of a window S_i^w, a load shedding percentage b, and a *stateful* operation represented by $\mathcal{Q}$. b controls the size of uniform sample T_w, to which $\mathcal{Q}$ will be executed for retrieving C_w. Algorithm 9's output consists of $T'_w \subset S_i^w$, which is forwarded for processing to next operation in a CQ's DAG. In Algorithm 9, $\mathcal{Q}$ is required

so that an estimation of the window's *concept* is extracted (see Section 5.3): Depending on the query type, Table 5 presents the information extracted to improve result's accuracy. For example, in the query of Figure 2, C_w will be the frequency of each group *route* appearing in S_i^w. Then, depending on b, each *route* would be assigned a portion of b (i.e., b_r). In essence, $C_w = \{b_1, \ldots, b_r\}$ is the sampling rate for each *route*. By the time C_w is established, the tuples of S_i^w are scanned once more, and each tuple t is passed to a *Shed* method along with C_w (line 6). *Shed* is responsible for deciding whether to shed a tuple, by examining whether tuple t's extracted attributes are part of C_w, i.e., the concept for window w. If yes, the tuple is included in T_w; the tuple is shed otherwise.

CoD requires two passes over S_i^w and has a runtime complexity of $O(c|S_i^w|+b|S_i^w|+c|S_i^w|)$. The first operand denotes the cost of creating the sample, the second operand is the cost of executing $\mathcal{Q}$ on a sample of size $b|S_i^w|$, and the third operand is the cost of applying load shedding to S_i^w. In essence, CoD's runtime cost is two times the runtime cost of *Uniform* (since the dominant factor is the scan of the window and not applying $\mathcal{Q}$ on the sample). If implemented naively, this can be a very high cost. In the next section we present an efficient design that turns CoD's runtime cost equal to that of *Uniform*'s.

5.6.1 Designing CoD to eliminate overhead

The design of modern SPEs provides an opportunity to avoid incurring additional overhead when CoD is applied. In this section, we present the details of event time window processing in modern SPEs, CoD's implementation, and CoD's implementation details for stateful queries.

In both designs, tuple storage is decoupled from window processing. With watermark-based execution, processing is postponed until the watermark has been received (see Section 2.4). This way, CoD can be designed to avoid additional scans and maintain a runtime cost similar to that of *Uniform*.

In order for CoD to avoid the additional scan of a window, uniform sampling is performed incrementally. When a tuple arrives, the operator determines the windows that the tuple belongs to. Also, for each window, CoD maintains a uniform sample of size b. The incoming

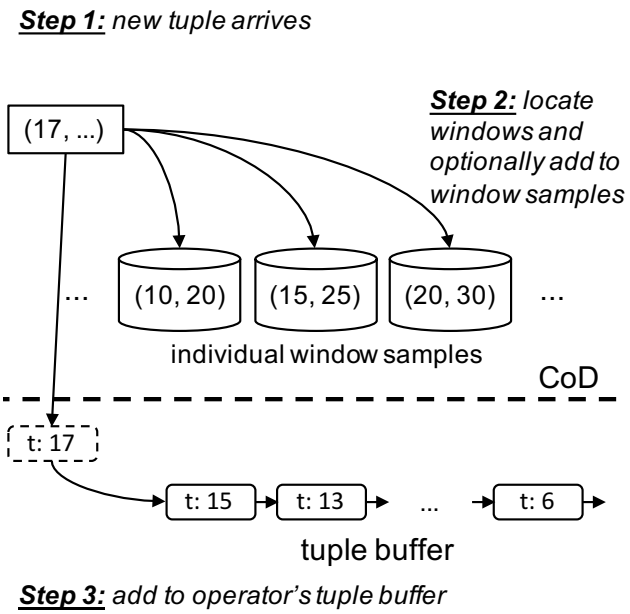


Figure 35: CoD tuple arrival: the tuple is optionally added to the windows it belongs to, and also stored in the buffer space.

tuple is added on each window's sample based on a random process (i.e., coin flip). Figure 35 illustrates the steps that are followed by CoD when a *new tuple arrives* which are as follows:

Step 1: the tuple's timestamp is extracted and it is used to identify the windows that the tuple participates.

Step 2: for each of those windows, a random process (e.g., a coin-flip) determines whether the tuple will be included in the sample. In our implementation, each sample is a reservoir sample, in order to maintain samples of size b for each window.

Step 3: the tuple is stored to the operator's storage space.

At a *watermark arrival*, CoD uses the uniform samples to extract window's *concept* C_w, as follows:

Step 1: Extract watermark's timestamp and identify window to be processed.

Step 2: CoD extracts C_w from the corresponding sample.

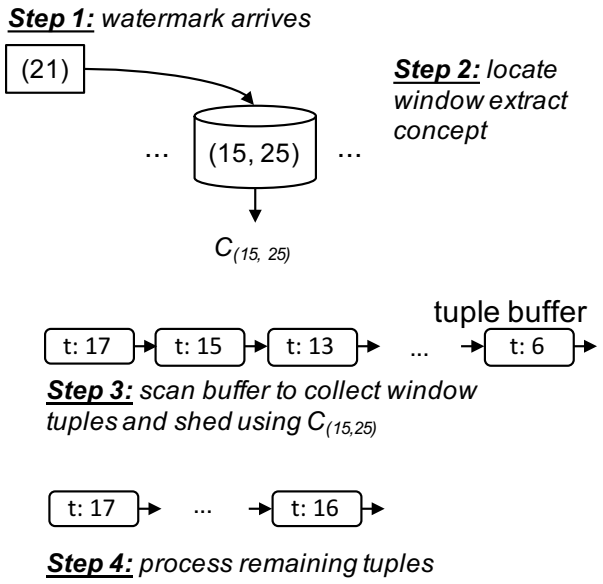


Figure 36: CoD watermark arrival: the concept is extracted from the sample and then tuples are shed based on it.

Step 3: in a single scan of the tuple buffer CoD locates the tuples for the window, uses C_w to prioritize tuples for shedding, and evicts expired ones.

Step 4: surviving tuples are sent for processing and the sample for the window is discarded.

As we presented in Sec. 5.5.2, *Uniform*'s runtime cost requires one scan over S_i^w. This action can be performed when the window is scanned to extract the window's tuples. CoD's design results in a similar runtime cost, with a time complexity of $O(c|S_i^w|)$ (since the dominant factor is the scan of the window). This cost is equivalent to *Uniform* load shedding runtime cost.

5.6.2 Implementation for different query types

As has been illustrated in Table 5, different stateful operations have a different *concept*. In this section we provide details of CoD's sampling process for different query types.

5.6.2.1 Scalar User-Defined Aggregations When $\mathcal{Q}$ is a scalar User-Defined Aggregation (UDA) then CoD will maintain a uniform sample for each window. For instance, if $\mathcal{Q}$ is the arithmetic mean of tuples' payloads (p), then during tuple arrival, based on a random process a tuple's p will be included in the sample. When a watermark arrives, CoD will extract the mean value μ and the variance σ^2 from the window sample (see Table 5). With this information, it can decide the sampling rate required to provide an answer based on a user's accuracy requirements [24].

5.6.2.2 User-Defined Aggregations When $\mathcal{Q}$ is a grouped UDA, then during tuple arrival CoD maintains the frequency of appearance of each group. Given a shed bias b and a window's size, CoD can create a stratified sample with a proportional representation of each group. Concretely, for each group g, CoD has to determine its sampling rate b_g. As indicated in [9, 13, 105], this technique will ensure that every group g will appear in the result. When a watermark arrives, CoD extracts each tuple's k attribute(s), and based on the tuple's group, it maintains a sample of size $b_g|S_i^w|$. In comparison with *Uniform*, this stratified sampling approach is expected to lead to more accurate results and zero missing groups.

5.6.2.3 Two-way Equality Joins If $\mathcal{Q}$ is an equality join between two streams, then CoD needs to maintain a histogram of groups for each of the two streams [12]. This way, CoD will be able to know which groups have a higher likelihood of forming a result. When a watermark arrives, CoD identifies the intersection of the two histograms: for each group g that appears in both histograms, a sample rate $b_g^{\{1,2\}}$ is determined based on the overall shed bias b, for each stream. Similar to grouped UDAs, when the window is scanned, a stratified sample for each stream and each group on each stream is maintained and pushed for processing.

Table 6: Dataset Characteristics

Dataset	**Size**	**Queries**	**Mean Window Size (tuples)**
DEBS	32 GB	Group Aggr.	$\approx$ 10K
GCM	16 GB	Group Aggr.	320K
DEC	175 MB	Aggr.	46K

5.7 EXPERIMENTAL EVALUATION

In our experimental evaluation, our goal is to measure performance, error, and the ability to decrease data volume in real world conditions. We implemented *Uniform* and CoD algorithms on Apache Storm v1.2, by extending the `WindowManager` class. For *Uniform*, our `ShedWindowManager` class requires the shed percentage parameter b. When a watermark arrives, a binomial process ($p = b$) determines whether the tuple is shed or not. Even if a tuple is shed, it is maintained in order to acknowledge its acceptance to the upstream operator. For CoD, our `ConceptShedWindowManager` receives the shed percentage parameter b, which controls the size of the sample for each window. Every time a tuple arrives, CoD augments the corresponding window samples by using a similar binomial process as in *Uniform*. At watermark arrival, the concept is extracted from the corresponding sample, and tuples are shed accordingly.

5.7.1 Experimental Setup

We conducted our experiments on a cluster consisting of 5 AWS r4.xlarge nodes. Each node ran on Ubuntu Linux 16.04 and OpenJDK v1.8. Each node had access to 4 virtual CPUs of an Intel Xeon E5-2686 v4 and 32GBs of RAM. One node was set up as the master. It had a long-running single-instance ZooKeeper server (v3.4.10) and an Apache Storm Nimbus process. The rest of the nodes ran a single Storm Supervisor process and were configured

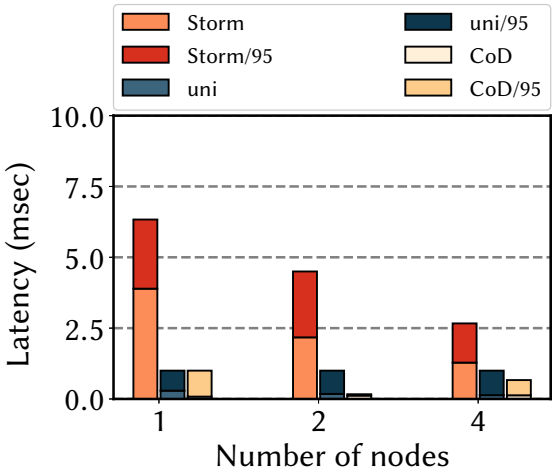


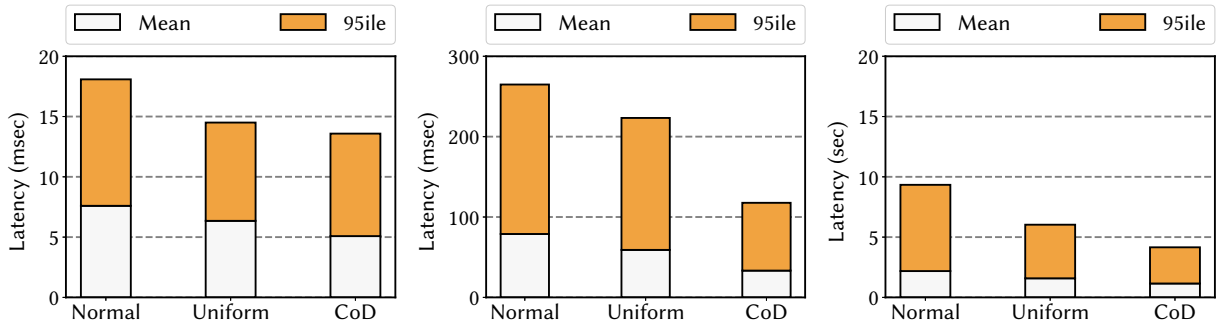
Figure 37: Scalability of normal execution compared to Uniform load shedding and CoD, with the shedding ratio set to 98%.

with up to 3 execution threads (one was reserved for Storm's acknowledgment service). In all our experiments, we enabled Storm's acknowledgment mechanism to guarantee processing of all tuples. All our experiments were repeated five times and the numbers reported are the averages of all runs.

5.7.2 Datasets

Section 2.6.1 presents general information for the real datasets in our study. In this section, we present specific information for the datasets we use in this chapter's experimental evaluation. In detail, we use three real-world datasets, with varying sizes, data characteristics, and window queries. Two of the datasets feature grouped aggregations and one scalar aggregation. Table 6 presents for each dataset its total size, the aggregation type, and the average window size in tuples. We provide additional details for each dataset below:

ACM DEBS Challenge Dataset (DEBS): For this experimental evaluation, we use DEBS's first query (see Section 2.6.1), and implemented it using a single source operator, a varying number of window aggregation operators, and a single collector bolt that sorts



a: DEC b: GCM c: DEBS

Figure 38: Overall average and 95 percentile window latency with four worker nodes.

routes based on the aggregation result.

Google Cluster Monitoring Dataset (GCM): From this dataset, we use part of the *task-events* table and run a sliding window grouped aggregation, taken from [82]. This query calculates the average CPU time requested by each scheduling class, on a 60 minute window with a 30 minute slide. Compared to DEBS, this dataset features fewer groups and smaller window sizes.

DEC Network Monitoring Dataset (DEC): For this dataset, we use the query documented in Section 2.6.1. This CQ calculates the average packet size on a sliding window of 45 seconds with a slide of 15 seconds.

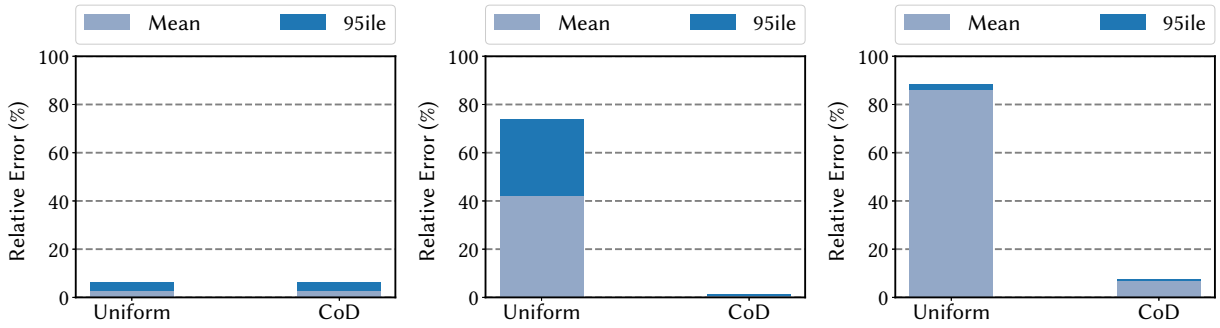
5.7.3 Experimental Results

Below, we present our experimental results on scalability (Section 5.7.3.1), performance (Section 5.7.3.2), accuracy (Section 5.7.3.3), and data volume reduction (Section 5.7.3.4). To this end, we compared normal execution to the current state of the art in load shedding (i.e., *Uniform*) and to CoD. For measuring error, we used the relative error metric for scalar aggregate operations (Equation 2.2), and the mean relative error for grouped aggregate operations (Equation 2.5) defined in Section 2.5.

5.7.3.1 Scalability (Figure 37) First, we examined the scalability of each approach by measuring the average and 95-percentile window processing latency on the DEC dataset. The reason we picked DEC for this experiment is because it is a processing-heavy aggregation, without any need for memory. Therefore, this dataset poses a favorable use-case for normal execution, since the workload is not memory-heavy. Also, we set the shedding rate (b) to 98% for both *Uniform* and CoD, since our analysis has shown that it leaves enough data to produce results with less than 10% error (see Section 5.7.3.3). In addition, we used the whole dataset and we doubled the number of server nodes (i.e., worker nodes) that participate in the process. Figure 37 presents the average and 95-percentile latency for normal (labeled as "Storm"), *Uniform* load shedding (labeled as "uni"), and CoD load shedding (labeled as "CoD").

As can be seen, Storm has the highest average and 95-percentile latency. The reason for this is due to the fact that processing takes place in the whole window and all tuples are scanned. *Uniform*'s performance is faster due to the fact that a fraction of the window is processed. The same is the case with CoD. Both settings with shedding achieve significantly better average processing latency compared to normal and more than four times better 95-percentile latency. Another important observation is that with load shedding, better processing latency is achieved with a fraction of the resources: CoD and *Uniform* running with one node achieve better latency compared to normal execution running on four nodes. **Take-away:** With CoD, processing takes place on a fraction of the window, and one requires much less resources to achieve better processing latency compared to Storm without any load shedding.

5.7.3.2 Performance (Figure 38) Next, we measured overall mean and 95-percentile latency for all three datasets. Compared to the scalability experiment (Section 5.7.3.2) we measure end-to-end latency on the stateful operator, from the watermark arrival until the time the window result is pushed to the next operator. For this experiment, we set the number of worker nodes to four (i.e., utilize all our resources), and the shed-bias to 97% (i.e., less than three percent of the window gets processed). For all datasets, we used a single source operator, and a sink operator.



a: DEC b: GCM c: DEBS

Figure 39: Average and 95 percentile relative error.

Figure 38a presents the overall latency for the DEC dataset. As can be seen, the overall latency is lower for *Uniform* and CoD. For DEC, most of the time is spent on window preparation (i.e., window scan). Therefore, the performance gap between normal and the load shedding methods is small. However, this is not the case in GCM and DEBS. Figure 38b illustrates that CoD and *Uniform* present better performance compared to normal execution. The reason is that GCM requires a grouped aggregation, which can introduce additional load to each worker. Figure 38c presents a similar picture. CoD manages to produce a result faster than normal Storm, and up to 2.25 times faster for the 95-percentile case. In our prototype implementation we measured that *Uniform* was slower compared to CoD due to the frequency of the binomial process (i.e., on every tuple of the window). This is not the case with CoD, since tuples of infrequent groups are simply added to the window.

Take-away: CoD improves overall latency and it can produce a result up to 2.25 times faster compared to normal Storm.

5.7.3.3 Accuracy (Figure 39) For the two load shedding techniques, we measured the average error and the 95-percentile error among all windows on all datasets. For the error metric we used the relative error for the aggregation result of DEC, and for DEBS and GCM

we used the average error among all groups for a given window. In addition, for each dataset we picked a different shed bias (b) which was 99%, 98%, and 98% for DEC, GCM, and DEBS. In the case of CoD, b works as the size of the sample created to identify C_w.

Figure 39a shows that both *Uniform* and CoD produce comparable results for DEC. This happens because DEC features a scalar aggregation query. Therefore, CoD creates a uniform sample, which is equivalent to *Uniform* and the resulting accuracy is the same for both techniques. Figure 39b illustrates the error for GCM, and the difference between the two becomes significant. In this case, *Uniform* sheds infrequent groups from the result, which in turn make the average error more than 40%. This is not the case with CoD, which employs stratified sampling for each group, by measuring their frequency. Then, CoD determines a shed bias for each group, and when it scans the window it creates a sample for each group. As a result, no groups are left behind from the final result and the error remains below 10%. A similar behavior can be seen with DEBS, whose results are shown in Figure 39c. Due to the fact that DEBS features more groups, *Uniform*'s error is more than 80%. This is not the case with CoD which manages to maintain both average and 95-percentile error close to 10%.

Take-away: CoD is able to identify the *concept* of each window and maintain accuracy more than 90%. Compared to *Uniform*, CoD can achieve more than an order of magnitude better accuracy.

5.7.3.4 Data Volume Reduction (Figure 40)

In the last experiment we wanted to analyze the drop of value in error while the data volume processed increases. To this end, we utilized all the workers in our infrastructure and executed the DEBS workload with varying b (shed bias) values: 98%, 90%, 75%, and 50%. On each run, we measured the average relative error achieved by each load shedding technique: *Uniform* and CoD.

Figure 40 illustrates the average relative error when a different percentage of the window is processed. As far as *Uniform* is concerned, when the data percentage processed increases, the average relative error drops (almost) linearly. After examining the quality of the results, *Uniform* exhibits big values for errors for two reasons. First, groups that do not appear often in the window are completely dropped, which leads to 100% error for them. Second,

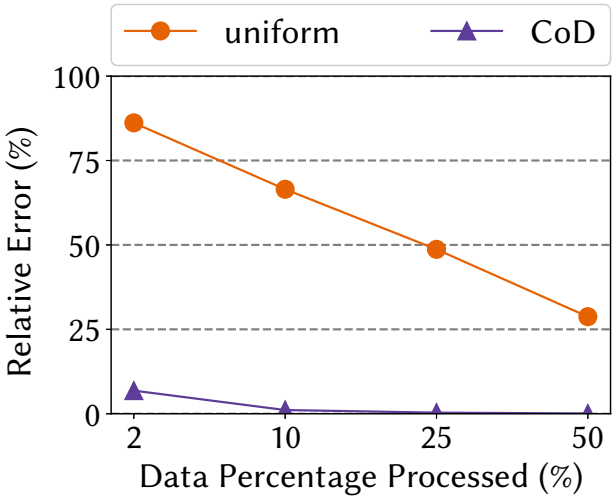


Figure 40: CoD achieves much better error with a fraction of the data required by Uniform.

groups that appear frequently have some of their values dropped, which leads to high error values for those as well. On the other hand, CoD identifies the right amount of sampling needed per group by employing stratified sampling. Therefore, each group is represented proportionally in the final result, and the average relative error remains significantly low. In fact, for a data percentage value 25 times smaller, CoD achieves much higher accuracy compared to *Uniform*.

Take-away: CoD makes it feasible to process significantly less data while achieving more than an order of magnitude less relative error compared to *Uniform*.

5.8 RELATED WORK

Throughout this paper we presented previous work done in load shedding [124, 45, 74, 119, 125, 24]. In addition, *Uniform* load shedding has appeared in a distributed version [123]. A control-based approach to load shedding is presented in [132] and the work of Gedik et al. [59] focuses on load shedding for join operations. Unfortunately, none of those feature a general mechanism for estimating C_w, and in turn they are not robust against *concept drift*. The

work presented in [45] presents two heuristics for estimating the *concept* for equality joins. However, those algorithms are targeted for joins and rely on frequency-based estimation, which might not efficiently estimate the *concept* on all CQ types (i.e., grouped median). In addition, those heuristics do not present a systematic approach for detecting *concept drift*. Furthermore, prior work focused on optimization of continuous queries under resource constraints [74, 119, 59]. [74] presented an optimization framework based on the resource constraints of window joins on unbounded streams. Moreover, work on load shedding has looked at other dimensions beyond correctness. The work of Kalyvianaki et al. [72] presents an algorithm that focuses on fair load shedding among federated SPEs. The work of Pham et al. [106] presents load shedding in agreement with QoS of CQs with different priority classes.

Finally, a lot of work has been done on approximation data structures for stream processing [52, 70, 28, 92, 116]. However, those approaches aim at particular classes of CQs and cannot be used for multiple types of operators. Finally, sampling-based approximation processing has been previously proposed for relational databases [13, 9]. In the case of data streams, data arrive continuously and the data's characteristics are unavailable for the SPE to analyze and pick an optimal sampling strategy.

5.9 SUMMARY

In this chapter we presented our work on load shedding for current SPEs. Our investigation on load shedding has revealed two shortcomings of previously proposed techniques. First that, previously proposed techniques are brittle against the inherent *concept drift* of data streams; second, they rely on knowing tuples' importance at query submission. Those shortcomings constitute *Uniform* load shedding the only applicable solution for modern SPEs. To improve the accuracy of the current state of the art (i.e., *Uniform*), we proposed CoD, a novel technique for load shedding, which relies on estimating a window's *concept* before shedding tuples thus making it more generally applicable. Even though CoD requires two scans of a window, we described a design that incurs zero overhead for modern SPEs. We implemented CoD for Apache Storm and examined its performance, accuracy, and data vol-

ume reduction capabilities compared to normal execution and to *Uniform* load shedding. Our experiments with real workloads show that CoD manages to (a) reduce processing cost by up to 2.25x compared to normal execution, (b) achieve more than an order of magnitude better accuracy compared to the state of the art in load shedding, and (c) reduce the data volume significantly.

6.0 APPROXIMATE STREAM PROCESSING (WITH GUARANTEES)

In previous chapters, we presented our solutions for stream *partitioning* (Chapter 3), stream *re-partitioning* (Chapter 4), and load shedding (Chapter 5). All those adaptation techniques are aimed for situations when an SPE's performance is compromised due to increased workload. However, stateful processing by itself poses significant overhead(s), both in terms of storage and processing. Those materialize by the need to support elaborate window operations, such as handling out-of-order arrival of tuples, cost-effective data management (i.e., garbage collection), the ability to recover from failures etc. Driven by the complexity associated with the aforementioned operations, we identify the need for accelerating processing by approximating results, whenever possible. This becomes feasible by detecting opportunities for approximating results given an accuracy specification provided with a CQ. To this end, we developed a prototype SPE, which is able to deliver this functionality for a plethorea of stateful aggregate operations.

Section 6.1 motivates this line of work, followed by Section 6.2, which provides background information on *approximate* query processing. Next, Section 6.4 discusses the system model of our prototype, followed by a detailed explanation of our prototype's sampling techniques presented in Section 6.5. Section 6.6 outlines the details of the acceleration opportunity detection mechanism, followed by Section 6.7, which presents the experimental evaluation. Finally, Section 6.9 provides information on related work, and Section 6.10 concludes with a summary of our findings.

Table 7: Properties of design space

	Load Shed	Sketch	Incremental	Elastic	SPEAr
Accuracy Guarantees (R1)		✓	N/A	N/A	✓
Window preserving (R2)	✓	✓	✓	✓	✓
General Aggregates (R3)	✓			✓	✓
React to changes in data (R4)			N/A	N/A	✓
Process fraction of tuples (R5)	✓				✓

6.1 THE NEED TO ACCELERATE PROCESSING

The conservative estimation of a CQ's resource needs drives users to proactively provision more resources than truly needed, to accommodate occassionally high processing demands. This behavior leads to high operational costs, and gives rise to the need to accelerate stateful processing in an opportunistic manner; *when possible, an SPE should examine if it can process only part of a window*. This feature is justified by past reports that windows increase with a rate of hundreds of thousands to millions of events per second [131, 40]. This need is similar to the one that motivated the development of *Approximate Query Processing* (AQP) in DBMSs, which produce a result by processing only a fraction of the data [69, 9, 39, 13, 73, 55, 105] but that result deviates from the exact answer by a bounded amount. AQP systems calculate a result from a sample, which has been created by monitoring previous query patterns and data characteristics. AQP has been applicable in Data Warehouses because the correct size and the contents of a sample can be identified offline based on a user's accuracy and resource requirements. Unfortunately, an SPE is not aware of data characteristics at query submission and AQP becomes very challenging in a streaming context.

In order to make it possible for an SPE to opportunistically accelerate processing, an authoritative mechanism is required to uphold the following standards

R1 Each result has to abide to a predetermined level of accuracy, which can be described

as Quality-of-Service. This will guarantee that results are acceptable by the end-user (i.e., application). For the rest of this dissertation, we will refer to requirement **R1** as *accuracy requirements*, *accuracy guarantees*, or *error bounds*. For example, in the CQ of Figure 8 a result cannot deviate more than 10% from the actual result in 95% of the windows.

R2 Results have to be produced for each window of the input stream(s). This requirement materializes only for stream processing and guarantees that window results are preserved, which is crucial for *event-time* processing (windows are determined by the input timestamps and overlooking a timestamp will lead to different windows). Therefore, the timestamps need to be preserved in order to have a reproducible result.

R3 Acceleration needs to be feasible for a plethora of different aggregate operations (or *stateful* operations in this manner).

R4 The system needs to be able to react to input data and effectively detect windows that qualify for accelerated processing; at the same time, the system needs to be able to identify windows that do not allow for safe acceleration (i.e., the accuracy guarantees are likely to be undermined).

R5 Acceleration needs to materialize as processing a fraction of the window, because its size is expected to be the dominating factor in terms of runtime cost. Also, by processing only a fraction of a large window, an SPE's performance will not deteriorate when data are accessed from a secondary storage device.

Unfortunately, none of the existing techniques for improving the performance of an SPE is able to deliver all the requirements above. As we presented in Chapter 5, load shedding enables an SPE to drop tuples when the input load exceeds its processing capacity [124, 24, 45, 96, 106]. Despite the fact that load shedding can be effective in terms of reducing load, it fails to deliver accuracy guarantees (i.e., fails requirement (**R1**)). In addition, load shedding can be applied in a manner to preserve windows at a minimal processing cost [125, 76], and is applicable to many types of aggregate operations (i.e., satisfies (**R2**) and (**R3**)). Furthermore, apart from CoD, previously proposed load shedding techniques, fail to react to changes in the content of data and rely on user intervention (i.e., fail (**R4**)). For example, *semantic shedding* [124] is not applicable to queries that aim at exploring patterns in streams,

because it assumes prior knowledge of the full domain. Finally, load shedding operates by processing a fraction of data (i.e., satisfies (**R5**)).

Sketches and *approximation algorithms* [29, 51, 91, 92, 48, 116, 93, 43, 52, 70, 28, 115, 130, 140] decrease the memory footprint of a *stateful* operation with specific accuracy guarantees. *Sketches* are able to deliver accuracy guarantees for pre-determined data (i.e., satisfy (**R1**)), but they need fine tuning to adapt to the uncharted nature of incoming streams and react accordingly (i.e., fail (**R4**)). They can be used only in specific problems (i.e., fail (**R3**)), and have to process all tuples of a window (the bigger the window, the longer the processing time) (i.e., fail (**R5**)). Variations of time-evolving sketches [115] have not been shown to offer tight accuracy on sliding windows.

Incremental processing (or resource sharing) has been introduced in SPEs as a technique to avoid processing tuples more than once [87, 20, 84, 122, 102, 32, 113, 135, 121, 114]. It can be applied when multiple CQs apply the same aggregation on the same stream(s) with varying window semantics[1]. Incremental processing solutions provide exact answers, hence they do not need to offer accuracy guarantees and they preserve windows (i.e., satisfy (**R1**) and (**R2**)). Even though it has been shown that incremental processing can limit the update cost of a window's result to a constant amount, it can be applied only in associative aggregate operations (i.e., fail (**R3**)). For instance, holistic aggregations are not supported and quantile processing, which happens to be one of the most popular aggregations [91, 92, 116], cannot be processed in an incremental fashion. Finally, incremental processing solutions do not need to react to changes in data, because they produce exact results, which requires to process the whole window (i.e., fail (**R5**)).

As discussed in Chapters 1 and 4, ***elastic*** SPEs are able to alter resources without disrupting execution [77]. However, stream *re-partitioning* aims at processing all tuples, is a time-consuming operation, and requires additional resources (i.e., fail (**R5**)). Therefore, elastic SPEs are slow to react to changes in input data and maintain operational costs low (i.e., fail (**R4**)). Table 7 provides a summary of the shortcomings of current approaches.

Inspired by the performance benefits of AQP and the differential execution of *Incremental Processing* in SPEs, we designed **SPEAr** (SPE Accelerator) that detects opportunities

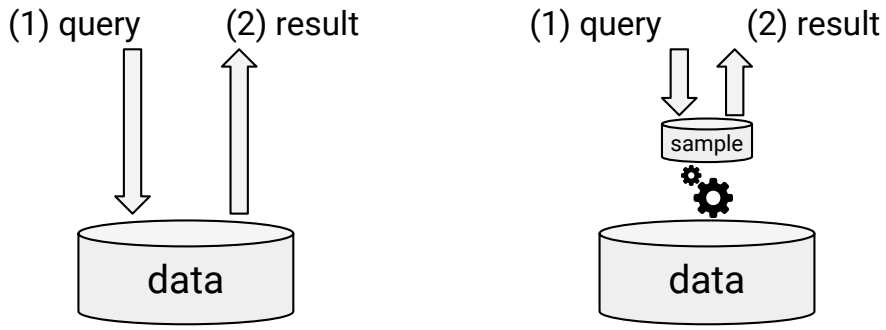
[1] As long as a CQ carries a sliding window definition.

for accelerating execution. In its current version, SPEAr produces approximate results with accuracy guarantees for sliding window aggregate operations, by incurring minimal overhead and avoiding unnecessary actions during processing. SPEAr employs conservative confidence interval estimation on approximate results produced from incremental samples for each window: if the worst-case accuracy estimation is within the user's accuracy requirements, then SPEAr accelerates processing by producing an approximate result; otherwise, it falls back to normal execution by processing the whole window.

To the extent of our knowledge, SPEAr is the first SPE, which can detect acceleration opportunities automatically, does not require offline sample creation, makes no assumptions on the periodicity of data, and delivers approximate results with accuracy guarantees. Our experimental results prove that SPEAr is capable of improving performance, use only a fraction of the resources needed for *exact* processing, while delivering a result within the accuracy specification.

6.2 SHORT PRIMER ON APPROXIMATE QUERY PROCESSING (AQP)

In this section, we present important information for AQP, as it has been applied in the past. AQP has appeared as a method for accelerating processing in OLAP DBMSs (i.e., *Data Warehouses*) [69, 11, 13, 73, 55, 105]. AQP's goal is to provide approximate results with bounded accuracy, by processing only a fraction of the data. This fraction is often termed as the processing *budget*, in terms of either secondary storage access (i.e., I/O operations) or response time (i.e., milliseconds). In the past, proposed AQP systems have appeared with differences in their query model, sampling operations, and prior information reuse. Figure 41 captures the main difference between an AQP system compared to a traditional OLAP DBMS: An OLAP DBMS receives a query and forms the result by accessing all the data in secondary storage (Figure 41a). On the other hand, an AQP DBMS produces a result by applying the query on a sample created from the data (Figure 41b). As a result, an AQP DBMS takes less time to produce a result, but the produced result is an approximate one. AQP functionality is offered by a number of proprietary DBMSs, such as Oracle 12c [103] and SQL Server [94].



a: Traditional workflow b: AQP workflow

Figure 41: Workflow between a traditional and an AQP data warehouse.

There are variations among previously proposed AQP systems, with regards to the query model offered. The major difference materializes on whether the approximation error is defined by the user, or appears as an additional column in the result. In otherwords, this variation comes from a user's requirement on whether she wants to control the accuracy of the result, or she wants to receive an estimation of the error that comes with an approximate result (without imposing a requirement for it). Online Aggregation [69] and Aqua [11] present the error as part of the result. For example, if a user has access to a relation named $Rides(time, route, fare)$ and they are interested in the average fare, they would submit the SQL query depicted in Figure 42. In this case, the AQP DBMS will produce a result with the column for the average fare $\bar{f}$ value, and a confidence interval $\pm c$, where c indicates that the true result is within $(\bar{f} - c, \bar{f} + c)$. Other systems like BlinkDB [13] and VerdictDB [105] require an accuracy specification as part of the query, as illustrated in Figure 43. The result of those systems omits a confidence interval column, and features only the approximate answer (i.e., $\bar{f}$).

Another variation appears on AQP DBMSs' sampling process. Online Aggregation [69], and Quickr [73] perform online sampling and leverage metadata for formulating the query

```
SELECT AVG(fare)
FROM Rides;
```

Figure 42: Sample Query.

```
SELECT AVG(fare)
FROM Rides
ERROR 1%
CONFIDENCE 95%;
```

Figure 43: Sample Query with accurary.

evaluation plan. On the other hand, Aqua, BlinkDB, and VerdictDB require offline sample creation before processing begins. In addition, some AQP DBMSs make use of prior results or query patterns to improve performance and accuracy [13, 55, 73]. Due to the existence of data (and metadata) maintained in a data warehouse, an AQP DBMS is able to deliver approximate results fast with accuracy guarantees.

6.3 APPROXIMATE STREAM PROCESSING

Unfortunately, the availability of data (or metadata) *prior to processing*, which is what traditional AQP DBMS techniques usually require, is not possible for stream processing. This is part of the stream processing model, which dictates that CQs are static and data flow through the system. Only the CQ is known prior to window completion and data are processed as soon as they arrive. Therefore, online sampling is the only option for AQP in a stream processing model.

As discussed in Section 6.2, AQP DBMSs allow the user to either submit their accuracy requirements with the query, or present an approximate result with its confidence interval.

Furthermore, all AQP DBMSs make use of the notion of a *budget* in one form or another. For systems like BlinkDB, VerdictDB, and Aqua, the user controls the *budget* (either in I/O operations or runtime estimation), which is used to determine the samples' sizes. For online sampling-based AQP systems, the budget increases as long as the query is allowed to execute for a longer time. For instance, in the Online Aggregation system [69, 64], a user submits a query without an accuracy specification, and while the query executes, it presents an approximate answer with its confidence interval, which improve the longer a query is allowed to execute.

6.3.1 Continuous Query Model Design Options

In order to define an AQP model for stream processing, one needs to establish the appropriate continuous query model and its parameters. Our investigation has led us to the following parameters: (a) *accuracy*, and (b) *budget*. The first parameter comes naturally with AQP and directly controls the quality of the produced result. The second parameter is the *budget*, which controls both the potential performance gains and the accuracy. As a result, the design space for the query model of an AQP system for stream processing is defined by the four resulting combinations of allowing the user to bound (or not bound) each of the two aforementioned parameters. The query model can vary based on whether the *budget*, and the *accuracy* are bound with the query and we discuss each combination below.

6.3.1.1 Unbound Budget–Unbound Accuracy: This approach does not allow the user to establish a level of service in terms either of accuracy or the processing speed. Thus, we chose not to investigate this combination further.

6.3.1.2 Bound Budget–Unbound Accuracy: In this approach, the SPE receives a *budget* specification by the user. This specification can be similar to the definition of budget of BlinkDB/Aqua and can translate to sample size or process time. On every window, the SPE creates a sample equal to the budget and produces an approximate window result, which is annotated with an accuracy estimation. This query model allows for a straightforward

tracking of the validity of its results, and can be used in real-time applications which allow the user to access previous data and re-process a window in case the accuracy estimation is very wide. Unfortunately, the number of real-time applications which allow "time travel" are fairly limited, especially when the storage pipeline is separated from the SPE [35]. For instance, if the window result produced comes with wide confidence intervals, then it might not be of any use to the end user. In this scenario, the only option for a user will be to pull old data from secondary storage for processing. This might take too long and exceed the allowed decision-making time window. Ultimately, requirement **R2** of Table 7 is undermined for the windows with wide confidence intervals. For example, if the query of Figure 2 is used, and the average fare $\bar{f}$ of different rides overlaps, then the final ranking would be of no use to the end user.

6.3.1.3 Unbound Budget–Bound Accuracy: On every window, the SPE processes tuples until the estimated accuracy is equal to (or lower than) the user-defined accuracy requirement. In this case, the query model is similar to the one from Online Aggregation and the main difference is that the user does not have to explicitly stop the window processing. The SPE can incrementally track the accuracy estimation, and when it reaches the acceptable level, it stops processing and produces the approximate result up to that point. Even though this model is dynamic enough to use exactly as many resources as it requires, it is not able to efficiently process data for a variety of aggregate operations (i.e., violates requirement **R3** of Table 7). For example, if a CQ requires the estimation of a quantile, then every time a tuple is included in the "about-to-be-processed" dataset, the whole part of the window needs to be sorted from scratch. This entails a polylogarithmic amortized cost for a complete sorting, which is equivalent to processing the whole window.

6.3.1.4 Bound Budget–Bound Accuracy: On every window, the SPE produces an approximate result based on a bounded *budget* and *accuracy* requirement. In particular, the SPE processes a number of tuples equal to the *budget* and estimates the accuracy of the approximate result; if the accuracy is lower than the user-defined accuracy requirement, then the SPE uses the approximate result; otherwise, it processes the whole window. This

query model is more versatile because:

- It delivers approximate results within the accuracy requirements (requirement **R1** of Table 7).
- It produces a useful approximate result on every window (requirement **R2** of Table 7).
- It supports a plethora of aggregate operations (requirement **R3** of Table 7).
- It is able to react in case the budget is not ample (requirement **R4** of Table 7).

A possible downside of this approach is its lack of control in its ability to process just a fraction of tuples on every window (requirement **R5** of Table 7). However, this is a pathology of stream processing, which materializes when window data are highly variable and an acceptable approximation is not feasible.

Due to the versatility of the **Bound Budget–Bound Accuracy** model, we selected it as the most appropriate model for SPEAr. In detail, the user needs to define the accuracy requirement (like the query depicted in Figure 8) along with a budget specification. Then, SPEAr is able to detect opportunities for accelerating the processing on every window. We present an overview of the SPEAr system in the next Section.

6.4 SPEAR SYSTEM OVERVIEW

In this section we present the architecture and design details of SPEAr. In Section 6.4.1 we briefly introduce the distributed architecture of SPEAr and its main components. In Section 6.4.2 we list the CQ operations supported by SPEAr, and in Section 6.4.3 we provide an overview of SPEAr's operation.

6.4.1 Architecture

SPEAr is built on top of Apache Storm v1.2 and it has been designed to extend Storm's `storm-client` API. As explained in Section 2.6.2, Storm is a distributed SPE that follows a tuple-at-a-time processing model and has been widely used by both academia and enterprise. Heron [83] is the evolution of Storm, and their main differences lie in the core infrastructure

(described in more detail in [83]). A Storm cluster consists of a single `Nimbus` instance, which is the master coordinator for active topologies. A number of `Supervisor` instances act as the worker processes, and exchange data with each other during processing. The Nimbus process is responsible for monitoring and managing active topologies and is not part of the processing pipeline. Supervisors are responsible for materializing the execution of a CQ.

SPEAr follows the same execution model as Storm, in terms of the communication and physical execution layer. In essence, SPEAr is extending Storm's API, to allow for approximate execution of stateful aggregations. Storm v1.2 supports windows through the use of the `BaseWindowedBolt` base class. The user needs to define the window logic by implementing the former's `execute()` method. Also, Storm supports both tumbling and sliding windows, whose semantics are defined during topology creation. In addition, Storm supports out-of-order (or late) tuples and supports storing state in a secondary storage devices for recovery.

SPEAr supports the aforementioned functionality by defining its own `ApproxWindowedBolt` and `ApproxScalarWindowedBolt` classes. The first is used for *stateful* aggregate operations that group tuples based on a subset of their attribute values; the latter is for scalar *stateful* aggregate operations. At query submission, a user needs to extend those classes, and create their topology using SPEAr's custom made `TopologyBuilder` class.

6.4.2 Supported Queries

As we mentioned earlier, we designed SPEAr to offer the *Bound Budget–Bound Accuracy* query model. A user provides an accuracy requirement, in the form of an upper-bound error ϵ, and a confidence level α at CQ submission: parameter ϵ represents the percentage relative error in the approximate result; α indicates the probability that the approximate result will deviate less than or equal to ϵ from the exact result (see Section 2.5). In addition, the user needs to define a *sampling budget* b, which indicates the maximum size of a sample per window. Essentially, b represents the additional memory that a user is willing to sacrifice for building a sample and it is similar to VerdictDB's I/O budget [105]. In the current version of SPEAr, b is statically assigned by the user. However, future versions of SPEAr will be able

```
q = rides
    .time(x -> x.time)
    .windowSize(15, TimeUnit::MINUTES)
    .windowSlide(5, TimeUnit::MINUTES)
    .mean(x -> x.fare)
    .error(10%)
    .confidence(95%)
    .budget(1000)
    .parallelism(4);
```

Figure 44: Example Scalar query for SPEAr.

to accommodate dynamic methods for online budget estimation.

Currently, SPEAr supports *mean-like* stateful aggregations, including common aggregate functions (e.g., count, sum, avg, quantile, variance, stddev). In addition, those functions are supported both in scalar or grouped format. SPEAr does not support approximate extreme statistics (i.e., *min* and *max*), which can be estimated by techniques presented in [118]. To the extent of our knowledge, statistics of this type are the most popular in real-time data analysis [104]. Approximation takes place on every `Executioner` thread (Section 2.6.2) that performs a *stateful* aggregate operation, and each thread performs the acceleration decision by itself. For instance, let us consider the following query expressed in a functional notation in Figure 44. This CQ indicates that for the *ride* stream, the mean fare will be processed for windows of $r = 15$ minutes, and slide $s = 5$ minutes. In addition, this *stateful* aggregate operation will be executed in parallel by four workers. Each worker will have a budget of 1000 tuples per window, and the accuracy requirement is a relative error of 10% with a confidence of 95%. During execution, each worker will allocate memory equal to 1000 tuples for each window, and calculate the average fare value from it. If the confidence intervals of the approximate value are estimated to be within $\pm 10\%$ of the exact answer, then a worker will produce the approximate result. Otherwise, a worker will process the whole window.

6.4.3 Workflow

In Section 2.4 we presented the details of stateful execution in modern SPEs. SPEAr relies on this workflow model to achieve performance improvement and maintain the estimation error within a user's accuracy requirements. As a result, SPEAr performs different operations at tuple and at watermark arrival.

6.4.3.1 Tuple Arrival At tuple arrival, SPEAr utilizes its budget allowance to maintain information incrementally, which will be used at watermark arrival for decision making. Depending on the *stateful* aggregate operation, the budget is utilized in a different manner. When a tuple t arrives, its timestamp (or other window-determining attribute is extracted) and the windows it contributes to are established using Equation 2.1. For each window that the tuple is part of, a binomial process determines whether this tuple will be included in that window's budget space. This process is similar to the one used by CoD (Section 5) and by incremental processing solutions, which augment the partial result of a window fragment [20, 84, 122, 113, 121, 114].

For scalar aggregations, like the one demonstrated in the CQ of Figure 44, the budget is used to maintain a uniform sample of the attributes that are required for the aggregate operation. In particular, a uniform sample of *fare* values for each window will be used. A sample of size equal to the budget b is allocated for each window separately.

For grouped aggregate operations (e.g., the query of Figure 2), a *stratified* sample needs to be built for each distinct group appearing in a window. This is required so that each group is proportionally represented in the output [12, 11, 10] (i.e., there needs to be an output tuple for each distinct group). Therefore, the budget b is used for keeping track of the frequency of each distinct group in a window, since this meta-information is not known before the window is complete (i.e., its watermark is not received). This operation is a fundamental difference between SPEAr and AQP DBMSs, which rely on either offline sample creation or metadata information. The latter, have the fundamental information prior to query arrival and this allows them to built samples incrementally. In contrast, SPEAr (or any SPE for that matter) does not have this information and has to built the samples after a window is complete (i.e.,

at watermark arrival).

6.4.3.2 Watermark Arrival At watermark arrival, SPEAr needs to produce an accuracy estimate for the approximate result and decide whether it is within the user's confidence intervals. If it is, then the approximate result is produced as the window result; otherwise, SPEAr processes the whole window and produces the exact result.

For scalar aggregate operations, the result can be incrementally processed when tuples arrive. Especially for associative and (non-) invertible operations, SPEAr's budget can accommodate both the result and the current aggregate operation result, using techniques used in incremental processing [20, 84, 113, 114]. As a result, SPEAr requires a constant processing time to retrieve an approximate result at watermark arrival. At the same time, the confidence interval is estimated and a decision is made on whether to use the approximate result. As far as holistic aggregate functions are concerned, those can be processed incrementally, using Manku et al's one pass algorithm [91].

For grouped aggregations, when the watermark arrives, the stratified sample sizes need to be determined. This is done by deciding the sampling rate for each group, based on groups' frequencies and the available budget. Then, the window is scanned once to create the stratified sample and at the same time produce the approximate result for each group. This scan does not impose any additional overhead since it is already done by Storm to identify expired tuples (Section 2.4) (i.e., garbage collection).

6.5 SAMPLING

In this section, we present background information on sampling techniques that have been used in AQP. Furthermore, we present information about the sampling techniques that are employed by SPEAr.

6.5.1 Background: Sampling for AQP

In the past, AQP systems have relied on sampling-based approximations [69, 9, 12, 11, 10, 39, 13, 55, 105]. In essence, if there exists an underlying relation T, then a simple random sample $T_s \subseteq T$ is a subset of tuples extracted from the underlying relation. Depending on a user's query $\mathcal{Q}$, which is applied on the sample T_s to produce an approximate result, different types of sampling have been proposed.

6.5.1.1 Uniform Sampling Uniform sampling has been mainly used for scalar aggregate operations. In this case, the challenge is to come up with a set of size n, and have a simple random sample (or i.i.d. sample). The previous requirement is important to avoid low quality results due to temporal locality of data in the underlying storage.

In essence, if T consists of tuples $\{t_1, \ldots, t_N\}$, the uniform sampling procedure is an assignment to each t_i of a probability p_i, which indicates the probability that t_i will appear in T_s. If the goal is to have $|T_s| = n$ (i.e., the size of the sample to be n), we need that $\sum_{i=1}^{N} p_i = n$. In uniform sampling, every tuple t_i has the same probability of appearing in T_s (i.e., $p_i = \frac{n}{N}, \forall i$). A challenge arises if $\mathcal{Q}$ comes with a predicate P on some of the columns of T, due to the fact that this creates an implicit bias on the uniformity of the result. However, in this work we consider approximation at the operator level of an SPE. As a result, any filter operations appearing in a CQ do not affect the uniformity of the sampling process. We refer the reader to [12] for a more detailed analysis on the matter.

6.5.1.2 Stratified Sampling Stratified sampling has been used when $\mathcal{Q}$ addresses a portion of T's tuples with a specific characteristic. In stateful aggregate operations, this happens when there is a group-by clause in the operator. In this case, the sample T_s needs to maintain a proportional representation of every group. In particular, if the tuples of T can be grouped in $\mathcal{G}$ groups, each group g appears n_g times. In essence, $N = \sum_{i=1}^{g} n_i$. Ideally, for a sample T_s of size n, every group should appear $n\frac{n_g}{N}$ times in T_s.

Essentially, the algorithm for creating a stratified sample requires two passes over T. First, for each group g we need to get its count n_g. Then, based on the sample size n, create

a uniform sample for each group which is going to be of size $min(\lfloor n\frac{n_g}{N} \rfloor, n_g)$ [12, 13]. This way, each group will be proportionally represented in the final result.

6.5.2 Online sampling creation for SPEAr

SPEAr uses different sampling methods depending on the *stateful* aggregate operation. *Uniform* and *stratified* sampling are used for scalar and grouped aggregations, respectively [42]. To avoid the sampling overhead, SPEAr conducts sampling upon tuple arrival by identifying the windows in which a tuple participates (akin to the *Multiple Buffers* design presented in Section 2.4)[2]. The tuple is offered to the sample of each window. When the window preparation phase is triggered, the sample is ready for estimating the stateful operation's result and perform the accuracy check.

6.5.2.1 Uniform Sampling for Scalar Aggregations For scalar aggregations, SPEAr maintains a uniform sample of size b for each window. The main challenge is to maintain the size of each sample $\leq b$, while creating a simple random sample. The more straightforward way to maintain an simple random sample would be to follow a binomial process for each input tuple. However, this would potentially lead to samples with sizes greater than b. To this end, SPEAr employs *Reservoir Sampling* which guarantees the creation of a simple random sample of bounded size [81].

At the same time, SPEAr performs incremental processing of the aggregate operation. Specifically, every time a tuple t appears, the samples of the windows it participates in are retrieved. For each of the samples, a reservoir process determines if the tuple will be added to the sample. If the tuple is added, then the incremental value of the aggregate is augmented. In case a tuple is removed from a reservoir sample, the aggregate value needs to be augmented. For instance, for the query depicted in Figure 44 if a tuple with timestamp `00:07:00` arrives, the windows that it participates in are extracted: $w_1 = (11:55:00, 00:10:00), w_2 = (00:00:00, 00:15:00), w_3 = (00:05:00, 00:20:00)$ (by applying Equation 2.1). Then, if any of the samples of w_1, w_2, or w_3 are not of size b, the tuple

[2] SPEs spend a significat amount of time in network I/O and for putting the window operator's thread in and out of sleep.

t's fare value is added to the corresponding sample. Otherwise, a value might have to be removed (according to the reservoir sampling process), and be substituted by t's fare value. If the substitution takes place, the corresponding sum value for each sample is augmented by removing the previous value and adding the new fare value (a counter for the total size of window tuples is maintained as well).

6.5.2.2 Stratified Sampling for Grouped Aggregations For grouped aggregations, previous work on AQP suggests the usage of *stratified* sampling [42]. The problem of missing groups from the result is eliminated, which appears when skewed datasets are uniformly sampled and small groups have a high probability to not appear in the sample [11, 13, 73]. The *Basic Congress* sampling method [11] guarantees that for a sample of size b, each group will have a number of tuples proportional to its frequency of appearance in the dataset. As mentioned earlier, the frequency of appearance of each group is known in OLAP DBMSs, which is not the case in stream processing. To overcome this problem, SPEAr builds congressional samples *incrementally*.

At tuple arrival, SPEAr extracts the group fields of a tuple and identifies the windows that this tuple participates in. In lieu of a sample, a histogram is maintained for each window that holds the frequencies of each group in each window. When the watermark for a window is received, and the window preparation phase is triggered, the window's histogram is scanned to establish each group's sample size based on its frequency n_g and b. Then, the window's tuples are scanned, the appropriate samples are created, the aggregation for each group is processed, and the result is pushed for the accuracy check. This scan does not present an added cost for SPEAr because it is required for identifying expired tuples (as part of Storm's guaranteed delivery mechanism and garbage collection process).

Even though the congressional sample process is an elaborate process, it requires a single scan on the window's tuples. This scan is faster or on par with the number of scans required by normal processing in SPEs (Section 2.4). Additionally, SPEAr uses a number of optimizations for limiting the sampling cost. For instance, when the number of groups is known or defined by the user, then SPEAr equally divides b among the different groups. This way, sampling is piggybacked on tuple arrival and a window scan is unnecessary.

6.5.2.3 Quantile Sampling

Quantiles present an exceptional case for SPEAr since they are approximated by utilizing the approximation algorithm of Manku et al [91]. This algorithm manages to approximate the ϕ-quantile, with $\phi \in [0, 1]$ with a well defined error ϵ. Given a set of N numbers and a parameter ϕ, their algorithm is able to find the ϵ-approximate ϕ quantile. A ϕ quantile is the element that resides in position $\lceil \phi N \rceil$ in the sorted sequence of the N numbers (see Section 2.5.1.2). An ϵ-approximate ϕ quantile is when the ϕ quantile is between the positions $\lceil (\phi - \epsilon)N \rceil$ and $\lceil (\phi + \epsilon)N \rceil$ of the arrays elements in sorted order.

Manku et al's algorithm works by having two parameters: B and k. The product Bk indicates the total amount of memory used, where k indicates a number of buffers of size B. This algorithm builds a compact representation of the sorted array of elements, and when all N elements have been scanned, it is able to provide any ϕ quantile statistic. The most popular statistic is the median, which is equivalent to $\phi = 0.5$. In their work, the authors present a sampling based algorithm for providing an ϵ-approximate ϕ quantile tied with a confidence parameter δ.

SPEAr utilizes this algorithm to approximate quantile queries under constrained memory using a single pass over the data. For every window, memory of size b is allocated for the estimation of the window's quantile. In essence, SPEAr constraints the algorithm to have $Bk \leq b$. At watermark arrival, if b is enough to provide the ϵ-approximate quantile after the window is complete, then SPEAr presents the approximate value as the window's result. Otherwise, it has to calculate the actual quantile by scanning the whole window. In the case of grouped quantiles, SPEAr maintains each group frequency for the tuples of a window. At watermark arrival, SPEAr will calculate the total needs for the quantile of every group, and if the total is greater than b, it will go ahead and process the whole window. Otherwise, it will accelerate by using the appropriate amount for each group.

6.6 ACCURACY CHECK

In this section, we present the details for estimating the accuracy of an estimated result. As mentioned earlier, SPEAr performs a confidence check when the watermark for a window

arrives, and when the approximate answer has been established. Prior to that, SPEAr features some additional mechanisms to fast-track the check process, and avoid any penalties (in the form of unneccessary execution).

6.6.1 Scalar Confidence Check

Accuracy or Error estimation has been extensively studied for AQP in the past. A relation $T = \{x_1, \ldots, x_N\}$ of size N is sampled to create a sample $T_s = \{X_1, \ldots, X_n\}$ of size $n < N$. Given an aggregate operation $g()$, $\hat{g}(T_s)$ applied is an estimator of $g(T)$. The goal is to measure the quality (i.e., expected error) of the estimate $\hat{g}(X_1, \ldots, X_n)$.

SPEAr approximates the confidence interval for a scalar aggregation by assuming that the estimates are normally distributed about the corresponding population values. For example, given a budget $b = n$, where n is the sample size accumulated using reservoir sampling, and the mean estimate is $\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} v_i$, where v_i are the values that contributed to the estimation of the average. The actual answer for the window's mean value is $\mu = \frac{1}{N}\sum_{i=1}^{N} v_i$. If n is large enough (usually $n \geq 30$ for symmetric distributions), then one can come up with the confidence intervals, using the sample's variance s. In detail, the confidence intervals are calculated as in [42], as follows:

$$\bar{Y}_{low} = \bar{y} - \frac{ts}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}, \quad \bar{Y}_{high} = \bar{y} + \frac{ts}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}$$

For estimating the sum (or total) of the window's values, then the confidence intervals are the following:

$$\hat{Y}_{low} = N\bar{y} - \frac{tNs}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}, \quad \hat{Y}_{high} = N\bar{y} + \frac{tNs}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}$$

In the above Equations, $\bar{y}$ is the sample mean, t is the value of the normal deviate corresponding to the desired confidence probability (e.g., 1.96 for 95% and 2.58 for 99%), n is the sample size, and s is the sample standard deviation.

As a result, SPEAr calculates the confidence intervals for $g()$ and estimates it as a relative distance from the estimated value. If the distance is less than or equal to the error specified by the user, then SPEAr returns this value as output. Otherwise, it calculates the aggregation for the whole window.

Finally, we have to mention that for SPEAr's mechanism to succeed, the budget (or the sample size n) can not be arbitrarily small. This is due to the fact that we employ normal approximation for the confidence intervals (as a direct implication from the *Central Limit Theorem*). Therefore, in the event of a very small sample on a skewed distribution, the confidence interval will not be precise.

6.6.2 Grouped Confidence Check

For grouped aggregations, SPEAr calculates the relative distance from the actual answer for every group and aggregates them using one of the Equations 2.4, 2.5, and 2.6. Each Equation serves different purposes and is applicable to different types of applications. If the resulting error is less than or equal to the user error, then the approximate result is used. Otherwise, SPEAr processes the full result.

Grouped Aggregate Operations present some additional challenges, which can either lead to overcoming the budget b, or pay additional processing costs. The former, can take place when the number of groups exceeds the available budget b. This is common in windows which are very uniform and each group appears sparsily. The latter case happens when the values of each group have a high variance, which can lead to a wide confidence interval. In this case, SPEAr would waste time in processing the approximate confidence interval and then find out that it needs to process the whole window.

To protect performance from those two cases, SPEAr is enhanced with two "early-fail" mechanisms, which are used to quickly detect windows that are not able to be accelerated:

1. **Number of distinct groups is greater than the budget:** In this case, SPEAr will immediately proceed to processing the whole window. This, optimization is also used to alleviate the memory usage when a window is not concluded. If SPEAr detects that the frequency of groups is higher than the budget, then it will release the memory and mark the window as not fit for acceleration.
2. **Variance of group values:** SPEAr monitors the estimated variance of groups in a window. If the average variance leads to a wide confidence interval, then SPEAr detects that it should not spend time to calculate an approximate value for the window.

Table 8: Datasets and Queries Used

	Total Tuples	Win. Size	Win. Slide	Avg. Win. Size
DEBS	56M	30 min.	15 min.	≈10K
GCM	24M	60 min.	30 min.	320K
DEC	4M	45 sec.	15 sec	47K

6.6.3 Quantile Confidence Check

Confidence Check for quantile operations is performed using the expected memory requirements to satisfy an ϵ-approximate ϕ-quantile, by comparing the allocated budget with the expected budget documented in [91]. If the allocated budget b is lower than the one required for the approximate quantile algorithm to provide an answer within the user's accuracy, then SPEAr processes the whole window.

6.7 EXPERIMENTAL EVALUATION

In this section we present the details of our experimental evaluation on SPEAr. Our goal is to investigate SPEAr's ability to improve performance, reduce use of resources, detect opportunities for acceleration, and decrease the amount of data processed. In Section 6.7.1 we provide SPEAr's implementation details, experimental infrastructure, and the datasets used in the experiments. Next, in Section 6.8 we present our experimental results.

6.7.1 SPEAr Implementation Details

SPEAr is built as an extension to Apache Storm's (v1.2.1) `storm-client` module. A user enables approximate processing by submitting their topology with SPEAr's topology builder class named `ShedTopologyBuilder`, which requires the definition of an error, a confidence

level, a memory budget, and the aggregate operation. As shown in the example CQ depicted in Figure 44, a user has to provide a method for extracting the timestamp field that assign tuples to windows, the sliding window specification (i.e., window size and slide), and the aggregate operation in the form of a lambda expression (i.e., anonymous function). Also, they provide the error ϵ, the confidence (α), the budget (b), and the number of workers that SPEAr will use for the aggregate operation.

6.7.2 Experimental Setup Details

Our experiments were conducted on Amazon EC2, with a cluster consisting of 9 r4.xlarge nodes. Each node ran Ubuntu Linux 16.04, OpenJDK Java v1.8, and python v2.7 and has access to 4 virtual CPUs of an Intel Xeon E5-2686 v4 and 32GBs of RAM. One node was set up as the master, having a single-instance Zookeeper v3.4.10 server and a Nimbus process. Each of the remaining nodes ran a single Storm supervisor process, which is configured to accommodate up to 4 executor threads.

In all experiments, we enabled Storm's acknowledgment mechanism to guarantee processing of all tuples. We need to state that the acknowledgment mechanism hinders SPEAr's acceleration mechanism and significantly reduces its performance benefit over Storm. Also, we enabled the Storm's back-pressure mechanism to guarantee in-order delivery of tuples and avoid polluting measured times. We note that all experiments, whose results are presented in this work, did not have any late (or out-of-order) tuples.

As far as the CQ execution plan is concerned, we set a single source operator (i.e., Storm's Spout) that reads data sequentially from a memory-mapped file, and pushes them to the downstream workers. In turn, each worker executes SPEAr's detection mechanism and the aggregate operation whenever a window is complete. A worker forwards its windows' results to a single worker that collects partial results. All of the results that we present are the arithmetic mean of seven runs, without the maximum and the minimum reported values.

In order to measure processing times with high precision, we used Storm's metrics API, which provides periodic reporting of runtime telemetry for each worker. In addition, for the experiments that we compare SPEAr to sketching techniques, we make use of CountMin

sketch [43], as it is considered the state of the art sketch technique for frequency counting. In this experimental evaluation, we used the popular CountMin sketch implementation of stream-lib v2.9.5 [3].

6.7.2.1 Datasets In our experimental evaluation we used three real-world streaming datasets, which are documented in Section 2.6.1. For each one of the datasets, we conducted an aggregate operation on a predefined time-based sliding window, on event time. Table 8 summarizes specific information for the datasets. Two of the datasets feature group aggregate operations (DEBS and GCM) and one features two scalar aggregate operations (DEC). For all queries we set the relative error to 10% and the confidence to 95% unless we specify otherwise.

- **ACM DEBS 2015 Challenge dataset (DEBS):** For this dataset we used one of the challenge's aggregate operations, which features a 30 minute sliding window, with a 15 minute slide, for calculating the average fare amount for each route. Routes are divided into a 300 by 300 cell grid that covers the entire New York City.
- **Google Cluster Monitoring dataset (GCM):** For this dataset we used part of the *task-events* table and performed Query 1 from [82], which requires the average CPU time requested by different scheduling classes. In order to examine SPEAr's performance on larger windows, we set the sliding window size to 60 minutes and the slide to 30 minutes. Nevertheless, we investigated the sensitivity of SPEAr's performance improvements with smaller windows of sizes of 30 and 15 minutes, with slides of 15 and 7.5 minutes.
- **DEC Network Monitoring (DEC):** For this dataset we used time-based sliding windows of 45 seconds with a slide of 15 seconds (as the ones used in [24]). For DEC, we used two different scalar aggregate operations (i.e., without a group-by statement): (i) the average, and (ii) the median TCP packet size. For the median, we set the relative error to 10%, with a confidence of 99%, since the approximation algorithm presented in [91] presents budget values for a confidence value $\geq 99\%$.

[3] https://github.com/addthis/stream-lib

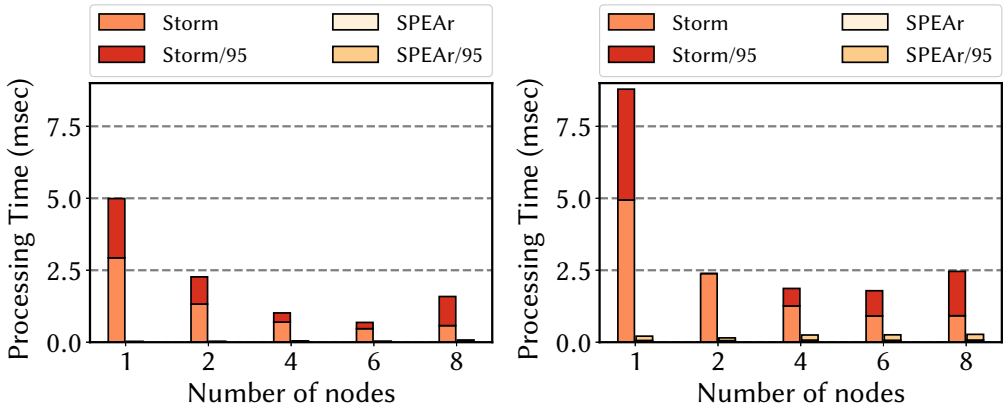
6.8 EXPERIMENTAL RESULTS

Our aim is to experimentally evaluate the merits of using SPEAr, by measuring performance improvement in terms of processing time, approximate results' accuracy, the success of SPEAr's accuracy estimation mechanism, and the total percentage of data processed.

Unless specified otherwise, we set SPEAr's budget b to a value that would cause it to accelerate processing in the majority of windows. In order to identify the proper b value for each dataset, we analyze their data characteristics offline, and then hard-code those values in the CQs submitted to SPEAr. For DEC, GCM, and DEBS, we set b to 1,000 for the average aggregate operation (150 for median), 4,000, and 2,000 tuples, respectively. Those b values amount to 2.1% (and 0.3% for the median), 5%, and 20% of the average window size for DEC, GCM, and DEBS (as shown in Table 8). We have to mention that the reason we set b to be a large percentage of the window size for DEBS is because this dataset is sparse and most distinct routes appear once or twice per window. Our analysis indicates that for an average window size of 9893 tuples, 4938 distinct routes appear on average. As a result, SPEAr needs to be able to accommodate at least a tuple from each group to build a stratified sample, and we discovered that setting $b = 2000$ tuples per worker allowed the acceleration of most windows in the dataset (at least 98.2% of the windows are accelerated).

6.8.1 Scalability (Figures 45-46)

The first batch of experiments aims at identifying SPEAr's scalability in terms of window processing time and the amount of memory used by each worker for processing. To this end, we measured Storm's and SPEAr's processing time and mean memory consumption per worker on five different levels of parallelism for both aggregate operations on the DEC workload (i.e., the average and the median TCP packet size). We chose DEC for our scalability experiment for two reasons: (a) scalar aggregate operations do not require additional memory, since they produce a single result for the whole window (i.e., processing time reflects the time it takes to process a set of tuples only); (b) DEC is the only dataset that could be executed on a single node in a reasonable amount of time.

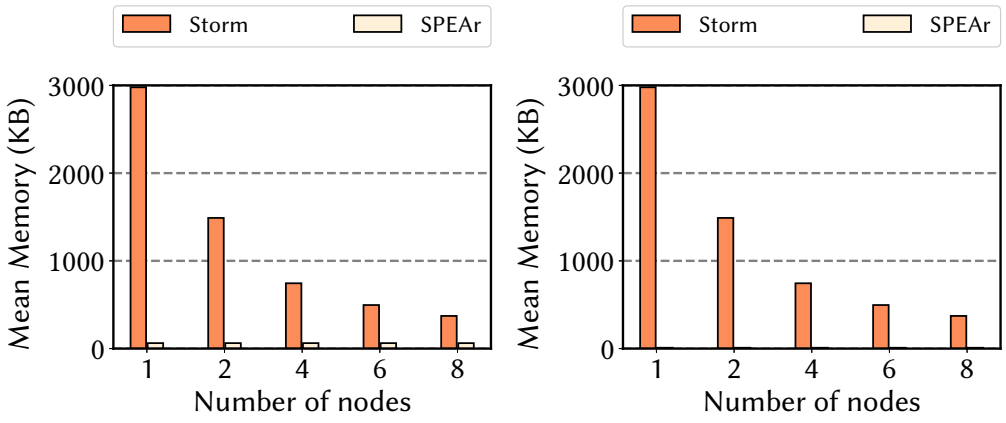


a: Mean Query

b: Median Query

Figure 45: Processing time of Storm and SPEAr on the DEC dataset.

Figure 45 shows the window processing time (both mean and 95-percentile) when one, two, four, six, and eight worker nodes are used (the average processing time among all workers is presented). Starting from the query for the average TCP packet size (Figure 45a), SPEAr is up to 293 times faster compared to Storm in terms of the average window processing time. Similarly, SPEAr is up to 166 times faster compared to Storm in terms of the 95 percentile window processing time. This happens because SPEAr has to process up to a 1,000 tuples per window (which is the budget size we set for this query) and achieves an error $\leq 10\%$ for at least 95% of the windows. On the other hand, Storm has to process many more tuples per window, and the average window size for DEC is 47,000 tuples). This drop in the processed tuples is the reason that processing times are much lower for SPEAr. We documented a similar outcome for the median TCP packet size aggregate operation (Figure 45b). SPEAr is up to 82 times faster compared to Storm in terms of the average window processing time; and up to 41 times faster compared to Storm in terms of the the 95 percentile window processing time. Despite the fact that SPEAr uses a budget of $b = 150$ tuples for the median, the improvement in window processing time is not as big as the one measured for the average processing time is because the approximation algorithm that SPEAr uses requires sorting the



a: Mean Query

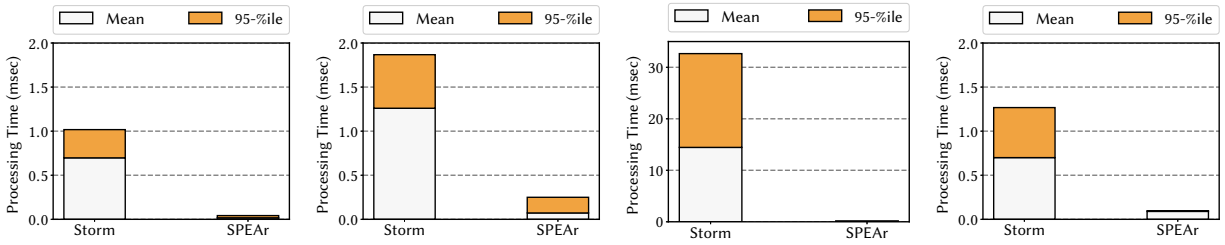
b: Median Query

Figure 46: Average window worker memory of Storm and SPEAr on the DEC dataset.

values maintained in the budget space.

In addition, we measured the average size of memory used for producing the result by each worker. Figure 46 illustrates the memory used by Storm and SPEAr on both the average and the median TCP packet size for each setting. As can be seen in Figures 46a and 46b, SPEAr uses a constant amount of memory for both aggregate operations, which is equal to the budget set (in this experiment we set the budget b to allow for acceleration of every window in this dataset). As a result, SPEAr uses from 5.97 to 47 times less memory per worker on average for the mean TCP packet size aggregate operation compared to Storm (Figure 46a). Similarly, SPEAr uses from 38.77 to 310 times less memory per worker on average for the median TCP packet size aggregate operation (Figure 46b).

Take-away: SPEAr achieves significantly lower processing time compared to Storm with only a fraction of the cluster nodes. Also, SPEAr requires much less memory per worker to produce a result within the specified accuracy. As a result, SPEAr will avoid spilling data on secondary storage in the event that the about-to-be-processed set of tuples can not fit in main memory.



a: DEC (Mean). b: DEC (Median). c: GCM. d: DEBS.

Figure 47: Average and 95-percentile processing time.

6.8.2 Performance (Figure 47 and Table 9)

For our second batch of experiments, we set our cluster to use up to four executor threads, each one running on a single node in isolation. We measure the average and 95-percentile window processing time for all datasets. On top of that, for GCM and DEBS we executed the grouped aggregate operations with the use of a CountMin sketch to measure the window processing time achieved with a state-of-the-art sketch. We allocate enough space for the CountMin sketch to achieve a confidence of 95% and an error of up to 10%. Figure 47 illustrates the mean and the 95-percentile window processing times of Storm and SPEAr.

Starting from the DEC workload (Figures 47a and 47b), SPEAr reduces the window processing time for the mean TCP packet size aggregate operation by up to 39.85 and 23.94 times compared to Storm on the mean and the 95-percentile case respectively. For the median TCP packet size aggregate operation, SPEAr reduces window processing time by up to 17.38 and 7.47 times for the mean and 95-percentile case respectively. For this set of experiments, SPEAr's budget for the mean query was set to 1,000 and for the median query to 150 tuples accordingly. Next, on GCM, SPEAr reduces window processing time by more than 120 times compared to Storm for both the mean and the 95-percentile case respectively (Figure 47c). This is due to the fact that GCM presents a small number of distinct groups, whose results can be approximated with only a small portion of the window's tuples. For

Table 9: Processing time (msec) of SPEAr and Count-Min sketch.

	Mean		95-%ile	
	SPEAr	CountMin	SPEAr	CountMin
GCM	.12	40.26	.04	107.6
DEBS	.09	3.7	.098	5.5

GCM we set SPEAr's budget to 4,000 tuples per worker, which as we explained earlier, we found to be enough for accelerating each one of the windows and offering an average error among groups less than 10% (95% of the time).

Finally, Figure 47d illustrates the processing time of Storm and SPEAr on the DEBS dataset. For this experiment, we set SPEAr's budget b to 2,000 tuples, which amounts for 20% of the average window size. As mentioned earlier, DEBS is a sparse dataset since a large portion of distinct groups appear on a window less than three times. As a result, b needs to be set at a high enough value to accommodate a stratified sample representing all distinct groups. Our offline analysis of DEBS concluded that a 2,000 tuple budget allows SPEAr to reduce window processing time by 7.77 and 13 times for the mean and the 95-percentile case. With this budget setting, SPEAr is able to accelerate at least 98% of the windows on each worker, and avoid processing at least 25% of the total number of tuples across all windows.

As far as the CountMin sketch is concerned, we compared SPEAr against a CQ that uses CountMin to produce the results of the grouped aggregate operations of GCM and DEBS. To this end, we used a CountMin sketch for counting the sum of values and the frequency of appearance of each distinct group for the group aggregate mean operation of GCM and DEBS. Table 9 presents the average and 95-percentile window processing times of SPEAr and Storm with a CountMin sketch. It is apparent that SPEAr is able to offer much lower processing time compared to the CountMin sketch setting on both datasets. In fact, the usage of a sketch causes performance to degrade compared to normal execution, which is justified by the application of the computation-heavy hash functions required by the sketch.

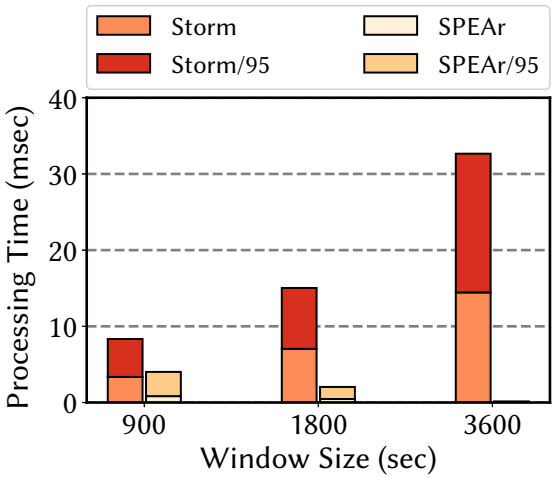


Figure 48: Processing Time on GCM with different window sizes.

With SPEAr processing time is reduced by at least 9.89 times for both the mean and for the 95-percentile case. In essence, the usage of a CountMin sketch increases the processing cost significantly, due to the additional processing required by this sketch technique (compared to normal execution it uses less memory).

Take-away: SPEAr outperforms both Storm and Strom with a CountMin sketch by more than 7 times in both the mean and the 95-percentile window processing time. This is an outcome of SPEAr's ability to approximate windows' results by processing less elements, without introducing additional overhead. Our experimental results on all three datasets indicate that the performance benefit is significant compared to both normal and sketch-based execution.

6.8.3 Window Size Sensitivity Analysis (Figure 48)

The third batch of experiments aimed at measuring the effect of window size (in terms of time) on the performance improvement offered by SPEAr. For this set of experiments, we picked GCM and measured the average and 95-percentile window processing time with different window definitions. To this end, we set SPEAr's budget to 4,000 tuples, and execute

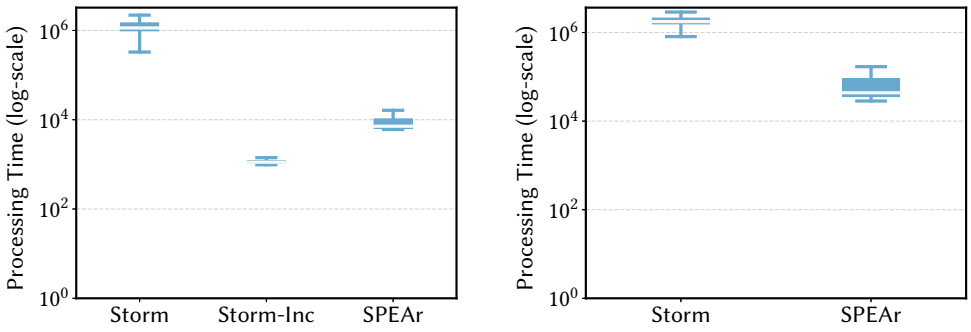
the GCM aggregate operation on three window size settings: 900, 1,800, and 3,600 seconds (with a slide of 450, 900, and 1,800 seconds). Those translate to an average window size of 84,000, 164,588, and 320,000 tuples respectively. The reason we picked GCM for measuring SPEAr's sensitivity analysis is because it features a grouped aggregate operation, and tuples for each distinct group appear multiple number of times per window.

Figure 48 depicts the average and 95-percentile processing time achieved by Storm and SPEAr. When the window size is set to 900 seconds, SPEAr is able to achieve at least 2 times better processing time compared to Storm. However, the budget of 4,000 tuples is not enough to accelerate all of the windows (i.e., approximate). In fact, SPEAr accelerates only 68% of the total windows, and this is a result of SPEAr's accuracy estimation mechanism, which for the remaining windows identifies that the approximate result is likely to have an average relative error $\geq$ 10%. As a result, the average and the 95-percentile window processing time reported contains the processing times of windows that are fully processed. When, the window size is set to 1,800 seconds, SPEAr is able to accelerate 88% of the total number of windows. Consequently, the improvement in processing time compared to Storm increases further. Finally, when the window size is set to 3,600 seconds, SPEAr is able to accelerate all windows and achieve more than two orders of magnitude lower window processing time compared to Storm, with an average window relative error of less than 10%.

Take-away: SPEAr is able to identify windows that are not safe to be "accelerated". In those cases, SPEAr improves performance up to 2 times compared to Storm. On larger windows, SPEAr is able to improve performance further.

6.8.4 SPEAr compared to Incremental Processing (Figure 49)

Next, our goal was to compare SPEAr against incremental processing techniques. Storm v1.2 supports incremental processing by separating a window's tuples in new (i.e., tuples that are pushed for processing for the first time), and expired (i.e., tuples that belong to a window that has been processed). However, this functionality is not equivalent with the incremental techniques that have been previously presented and offer a constant update cost at watermark arrival. To this end, we modified Storm's `WindowManager` class to offer incre-



a: Mean Query

b: Median Query

Figure 49: Processing time (nsec) on DEC between Storm, Storm with incremental processing, and SPEAr (without the incremental optimization).

mental processing, by updating each window's result at tuple arrival; and when a watermark arrives, Storm produces the result. This functionality is feasible only for associative and distributive/algebraic aggregate operations.

We conducted the comparison experiment on the DEC dataset and compared the window processing time between Storm, incremental processing with Storm (tagged as "Storm-Inc"), and SPEAr. Figure 49 presents the box plot for the window processing times of each technique in a logarithmic scale. Storm maintains the slowest processing time, and at least one order of magnitude slower compared to SPEAr and Storm-Inc. SPEAr is slower compared to Storm-Inc, because we did not enable SPEAr's optimization for associative operations (i.e., sum and count), and it still iterates over the tuples of the sample. As a result, SPEAr presents a higher runtime compared to incremental processing (i.e., Storm-Inc). However, when we enable SPEAr's optimization for associative aggregate operations, its runtime is identical to Storm-Inc's, and constant across windows.

One problem with incremental processing is that is not applicable to holistic aggregate operations. To this end we highlight SPEAr's performance benefits in a holistic aggregate operation, such as the median. To this end, we measured the performance experiment on

Figure 50: Relative error per window on DEC. The red (flat) line indicates the user-defined acceptable error (i.e., 10% at 95% confidence). The blue line indicates the error achieved by SPEAr.

a topology that produces the median of TCP packet sizes for the DEC dataset. To the extent of our knowledge, there exists no incremental technique for calculating the median of non-distinct values with a linear cost. As a result, we omitted Storm-Inc setup from this experiment. For SPEAr, we calculated the median with $\epsilon = 10\%$ and 99% confidence.

Figure 49b illustrates the window processing time for each window. It is clear that SPEAr achieves almost an order of magnitude better performance compared to Storm. This is because for DEC the average window size is about fifty thousand tuples. For SPEAr's quantile estimation algorithm, the median is approximated with a uniform sample of 150 tuples per window. As a result, the processing time is much lower.

Take-away: SPEAr presents identical processing time compared to incremental processing techniques. At the same time, SPEAr is able to accelerate a broader variety of aggregate operations, and achieves up to an order of magnitude better performance compared to Storm on quantile approximation.

6.8.5 Error (Figure 50)

Our next batch of experiments focused on SPEAr's ability to identify acceleration opportunities, and the success rate of its error estimation technique. We set the budget b to three different values: 250, 500, and 1000 tuples. The rationale for selecting those values, is a setting that is too low that will make SPEAr avoid acceleration (250), a setting that is relatively low and will cause SPEAr to produce error estimations (500), and a setting which will cause acceleration and acceptable accuracy on all windows (1000). The expected behavior of SPEAr is to simulate the aforementioned behavior and avoid accelerating execution, at the expense of quality. We came up with those values after we executed SPEAr on the DEC dataset and identified situations where the error would exceed the user defined error.

Figure 50 presents the error of SPEAr when compared with the actual result on the DEC dataset. When $b = 250$, SPEAr does not accelerate processing often (an error of 0 indicates that SPEAr performs normal processing). This happens because SPEAr's confidence interval estimation mechanism indicates that the interval is wider than 10% of the estimated value. As a result, SPEAr chooses to process the whole window. In fact, only 15% of windows are accelerated, and four out of 37 produce a result which diverges more than 10% from the actual answer. When $b = 500$, SPEAr accelerates 98% of the windows, and its result diverges from the actual answer by more than 10% in six windows (which accounts for 2.5% of windows-satisfies the 95% confidence property). Finally, when $b = 1000$ SPEAr accelerates all windows, and no windows have an error more than 10%. In this case, SPEAr identifies that the budget b is enough to safely accelerate a window.

Take-away: SPEAr's accuracy estimation mechanism makes correct decisions and accelerates processing, for appropriate budget values.

6.8.6 Data Volume Reduction (Figure 51)

Finally, we measured the total data volume processed by SPEAr when all windows are accelerated. This is an important metric because through it we can examine how SPEAr maintains the processing pipeline of an SPE less busy with processing and less crowded in terms of memory used. As a result, the effective throughput of the overall system increases.

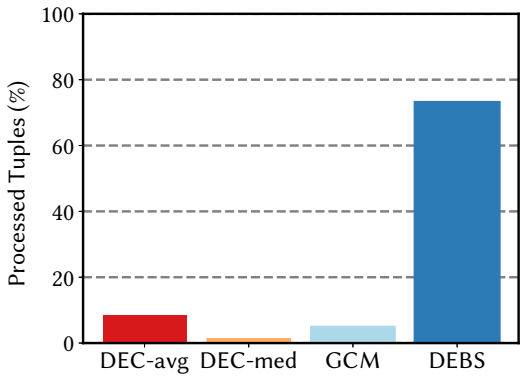


Figure 51: Percentage of total tuples processed by SPEAr.

To this end, we set the budget to 1,000, 150, 4,000, and 2,000 tuples per worker for DEC (mean query), DEC (median query), GCM, and DEBS. Figure 51 indicates that SPEAr processes less than 10% of the total data, including overlapping fragments among consecutive sliding windows for the DEC and GCM datasets. This means that from the total volume of tuples that need to be processed, SPEAr identifies that overall less than 10% of it needs to be processed, in order to produce results acceptable by the user. Turning to the DEBS dataset, it has to process about 74% of the total data due to the fact that DEBS features sparse data. However, even in this unfortunate scenario, SPEAr is able to avoid processing one fourth of the total data.

Take-away: By accelerating execution, only a small fraction of the whole data is processed, as low as 1.3% in our experiments with real datasets.

6.9 RELATED WORK

In this section, we cover work done in related fields for SPEs and stream processing. SPEAr represents a novel prototype, and the first of its kind to identify opportunities for acceleration. As a result, some of the related work discussed in this section might not be a direct

comparison to SPEAr, but rather a review of the design space for expedited stream processing. In Table 7, we have presented some of the features that we believe are important for improving the performance of SPEs.

6.9.1 Incremental Processing

To the extent of our knowledge, the first work introducing differential (or incremental) evaluation of CQs is the work by Liu et al [87]. In this work, a differential evaluation framework for select, project, and join queries is presented. In addition, they employ associative aggregate operations (e.g., `SUM`) for monitoring the satisfaction of conditions for triggering differential evaluation. However, their work fails to provide support for holistic aggregate operations. Next, the work of Arasu et al [20] presents efficient ways of accommodating multiple CQs in the same SPE. The different CQs that they consider have to be on the same input stream(s), and share the same windowing conditions. Their goal is to avoid using additional memory for each CQ, and at the same time save on computation. The framework they present covers a wide spectrum of window types and aggregations, along with quantile holistic aggregations. However, their algorithm for quantiles bears a logarithmic update time, and a costly (in terms of complexity) lookup time. SPEAr features a constant update time, a constrained memory update (b), and a constant lookup time. Similar work presented in [84] presents a framework for saving both in memory and computation when multiple queries of distributive and algebraic aggregate operations are used, which is an impact similar to the work presented in [122, 121, 113, 114]. All these improve on the amortized cost of updates and lookups for associative and (some) invertible operations. However, they do not address holistic aggregate operations. Finally, the work of Wesley et al [135] presents a solution for incremental processing of distinct value quantiles. The algorithm used by SPEAr is not restricted in the frequency of appearances of each value, i.e., would work even not for distinct value quantiles.

6.9.2 Load Shedding

Even though load shedding does not aim at providing tight accuracy guarantees, previous work strived to minimize error [124, 24, 96, 106]. The work of Babcock et al. [24] focused on

identifying shedding rates that maintain a low error. However, there are no guarantees that this error will be sustained when the input rate increases. *Semantic shedding* [124] limits the error when the user provides a utility graph. But, this approach is not feasible in queries that mine patterns (e.g., top-K). Finally, [96] presents an optimal shedding algorithm, but without tight accuracy guarantees.

6.9.3 Sketches

A lot of work has been done on sketch-based techniques. *Hyperloglog* [52] and *CountMin* sketch [43] are two of the most widely used sketches for keeping information about a stream. Despite the fact that those can provide tight error guarantees, they do not react to data changes and cannot be used in windows efficiently (a sketch is not aware of the window boundary). Recently, *Ada-Sketches* [115] showed that they can enhance the count using a novel method for adopting counters as time progresses. However, those cannot provide information on elements at the window boundary, and it is not clear how they would operate on a sliding window paradigm, when the window slides and some values no longer exist. In addition, when the groups are unknown, the memory benefit of the aforementioned structures is diminished, and (as shown in Section 6.8) all of the tuples need to be processed.

6.9.4 Elastic SPEs

Finally, a lot of SPEs with the ability to add additional workers have been presented [66, 141, 67, 68, 34, 86, 110, 107, 136]. All of those are able to maintain the performance of an SPE in acceptable levels, without adding a significant overhead to the execution. We consider this line of work orthogonal to SPEAr's goals, since SPEAr can be extended to support online reconfiguration of its resources.

6.10 SUMMARY

In this chapter we presented SPEAr, a prototype SPE built by combining principles from AQP and differential processing for SPEs. SPEAr's goal is to detect opportunities for accelerating *stateful* processing with accuracy guarantees. To the extent of our knowledge, SPEAr is the first system of its kind that automatically detects opportunities to accelerate processing, by applying AQP principles in streaming. Our experiments highlight the practical significance of SPEAr, with impacts on performance (Section 6.8.2), resource savings (Section 6.8.1), accuracy (Section 6.8.5), and effective data volume reduction (Section 6.8.6).

7.0 SHED-PARTITION: THE POWER OF TWO TECHNIQUES IN ONE

In the previous chapters we presented our own techniques for stream *partitioning*, load shedding, stream *re-partitioning*. In the last chapter, we presented SPEAr as a novel system model for automatically accelerating processing. An important part of SPEAr is its mechanism to detect opportunities for approximating a window's result at the worker level: each worker thread examines the possibility of processing a subset of its windows' tuples to expedite the production of a result, given specific accuracy requirements.

As discussed in Sections 1 and 3.8, there are circumstances when a single technique is not adequate to overcome an overloaded situation. In this chapter, we present a novel hybrid technique that can further improve load allocation among multiple workers of an SPE, and in turn improve performance. This technique operates similarly to SPEAr, and detects approximation opportunities at a distinct group level granularity.

Section 7.1 introduces the need for combining adaptation methods, and is followed by Section 7.2, which formalizes the stream *partitioning* problem. Next, Section 7.3 presents the hybrid model that combines *partitioning* with load shedding. Section 7.4 presents the details of our ShedPart operator, and is followed by our experimental evaluation (Section 7.5).

7.1 THE NEED FOR TIGHTER INTEGRATION OF ADAPTATION TECHNIQUES

Each one of the three adaptation techniques (stream *partitioning*, load shedding, and stream *re-partitioning*) is targeted to be used under specific circumstances. In our experience with stream processing, the expected sequence of actions performed by a user, who submits a CQ

and monitors its execution, to maintain an SPE's performance are the following:

Step 1 (Adaptive Stream-Partition): If the performance of an SPE drops due to imbalanced load allocation, change the stream partitioning. Apply this approach as long as each worker does not exceed its processing capacity. This technique is applicable when accuracy cannot be compromised. However, as we have shown in Chapter 3 there is no single stream partitioning algorithm that offers the best outcome in terms of both *imbalance* and *aggregation* cost. Also, input load can become exorbitant for existing resources. Therefore, the need for further action arises.

Step 2 (Load Shed): If the total input load exceeds available resources' capacity, and the increase in load is expected to last temporarily, then the user initiates load shedding. This technique is applicable when a CQ allows for *approximate* execution, and the load is expected to regress to normal levels shortly. Techniques to maintain results' accuracy, like SPEAr, can be applied proactively as well. However, both load shedding and SPEAr-like solutions can fail if the data are not fit for approximation (e.g., wide confidence intervals), long-lasting load increase, or data volume is enormous for available resources. Under such circumstances, additional infrastructure needs to be added.

Step 3 (Stream Re-Partition): If the input spike persists, and/or the accuracy of the results is violating a CQ's accuracy requirements, then additional resources need to be added. Stream *re-partitioning* has to occur, which entails provisioning more resources, re-partitioning state, and suspending operation momentarily. *Re-partitioning* can be a "pharmakon" (i.e., both a poison and a cure) due to its short-term disadvantages for long-term performance benefits [34, 63]. Even if no-state protocols, like UniMiCo are used, performance degradation will last until all active windows have been processed [107]. Also, scaling-out *stateful* aggregate operations can result in additional hops (i.e., edges) in the physical execution plan (e.g., an *aggregation* step needs to be added for combining partial results).

The aforementioned steps have been a road-map on when to use each technique, and it is apparent that there is no silver bullet for achieving fully adaptive stream processing. To compensate for the shortcomings of each adaptation technique, the combination of two

or more has been used in the past to alleviate the side-effects of each one. Two techniques have been applied in tandem to remedy situations when a single approach can not solve the performance issue [111, 86, 50]. In particular, stream partitioning algorithms and stream re-partitioning techniques have been combined to handle skewed data. In the event that a partitioning algorithm is not able to overcome long-term skewed load allocations, additional resources are added in an online fashion. In the case of Flux [111], an upstream partition operator would monitor each worker's load, and if the load distribution is skewed, Flux introduces additional workers, and re-distributes load. A similar approach for scalable stream join operators was presented in [86, 50] which was shown to maintain acceptable performance in highly skewed environments. Nevertheless, both solutions were not demonstrated in scenarios which required a global window result (i.e., a final aggregation step). Therefore, it is not clear if either would be suitable for CQs that feature grouped or scalar aggregate operations.

This motivated us to investigate the combination of stream partitioning and load shedding, to overcome the limitations of previous techniques that combined stream *partitioning* and *re-partitioning*. Our goal is to avoid having to "split" highly frequent groups, which introduces an aggregation step. In contrast, we propose to "trim" the excess tuples when the *exact* processing is not required by the application. Another observation that motivated our investigation is the pathology of *1-choice* partition algorithms in terms of *imbalance* minimization (Section 3.3). As far as *1-choice* partition algorithms are concerned, they sacrifice *imbalance* in order to avoid an aggregation step.

In this chapter, we introduce a solution that (i) improves an SPE's performance by reducing *imbalance* among workers, and (ii) does not carry the costly overhead of stream *re-partitioning*. ShedPart is our lightweight operator, which works alongside a stream *partitioning* algorithm, and "trims" excess tuples from individual groups. ShedPart's operation relies on incremental statistics, and at window completion (i.e., watermark arrival) it identifies the appropriate number of tuples that need to be processed for the most frequently appearing groups. Its operation is similar to SPEAr's, but it operates at a finer granularity to achieve higher accuracy while reducing *imbalance*. In essence, ShedPart's error estimation takes place on a distinct group rather than a worker level. This way, ShedPart can maintain

full accuracy on some groups and sacrifice the accuracy of specific groups only.

7.2 PROBLEM STATEMENT

In the work presented in Chapter 3, a window S_i^w is partitioned among $\mathcal{V}$ workers based on a stream partitioning algorithm P, and our exploration for stream partitioning algorithms was limited to exact processing solutions. In other words, we investigated partitioning algorithms when the exact result needs to be produced. The success of P is measured using the *imbalance* metric, which in its simplest form is measured by Equation 3.1. This entails that if in S_i^w exist $\mathcal{G}$ distinct groups (i.e., $\|S_i^w\| = \mathcal{G}$), then the tuples of S_i^w can be partitioned in $\mathcal{G}$ sub-groups based on their k attribute values. In essence, if N_g is the set of tuples of S_i^w that belong to distinct value g, then $N_1 \wedge \ldots \wedge N_{\mathcal{G}} = S_i^w$. Turning to *imbalance*, its first operand is the total number of tuples of the most loaded worker w. Essentially, the most loaded worker's total load is equal to $L_{S_i^w}^w = \sum_j |N_j|$, where $\{j | t \in S_i^w \wedge k_t = \{j\}\}$ and $P(N_j) \rightarrow \{w, \ldots, w\}$.

Essentially, to limit the *imbalance* in this setting, w's load need to be reduced, which is proved by the following lemma:

Lemma 1. *For a given window S_i^w, $\mathcal{V}$ number of workers, and a partition algorithm P,* imbalance $I(P(S_i^w))$ *is reduced if and only if the load of the most loaded worker w is reduced.*

Proof. The average load among all $\mathcal{V}$ workers is $\bar{l} = \frac{1}{|\mathcal{V}|} \sum_{j=1}^{\mathcal{V}} L_{S_i^w}^j$. This can be broken down to $\bar{l} = \frac{L_{S_i^w}^w}{\mathcal{V}} + \frac{1}{|\mathcal{V}|} \sum_{j=1,\ldots,\mathcal{V} \wedge j \neq w} L_{S_i^w}^j$. From Equation 3.1, the *imbalance* for window S_i^w is $I(P(S_i^w)) = L_{S_i^w}^w - \bar{l} \Rightarrow I(P(S_i^w)) = L_{S_i^w}^w - (\frac{L_{S_i^w}^w}{\mathcal{V}} + \frac{1}{|\mathcal{V}|} \sum_{j=1,\ldots,\mathcal{V} \wedge j \neq w} L_{S_i^w}^j)$. This can be written as: $I(P(S_i^w)) = L_{S_i^w}^w (1 - \frac{1}{\mathcal{V}}) - \frac{1}{|\mathcal{V}|} \sum_{j=1,\ldots,\mathcal{V} \wedge j \neq w} L_{S_i^w}^j$. The terms $(1 - \frac{1}{\mathcal{V}})$ and $\frac{1}{|\mathcal{V}|} \sum_{i=j,\ldots,\mathcal{V} \wedge j \neq w} L_{S_i^w}^j$ are constant given the partition algorithm P. Also, the term $(1 - \frac{1}{\mathcal{V}}) > 1$ (*imbalance* has no value when defined for a single worker). Then, $I(P(S_i^w))$ is directly proportional to $L_{S_i^w}^w$, which entails that if the latter is reduced, $I(P_{S_i^w})$ is reduced as well. □

Lemma 1 indicates that the goal to reducing imbalance, and in turn the time it takes

to perform the parallel step of a stateful operation (see Section 3.1), can only be reduced if the load of w is reduced. Therefore, the focal point of any technique aiming to reduce *imbalance* is to reduce the maximum load. w's load is the sum of the disjoint sets of tuples that belong to the groups assigned to this worker. Thus, to reduce $L^w_{S^w_i}$ one needs to reduce the individual disjoint sets. This can be achieved by shedding tuples for each individual group.

Dropping tuples will result in approximate results and deviate from previously proposed techniques of *exact* stream partitioning. Consequently, there is a need of a query model that includes an accuracy requirement, similar to the one used by SPEAr (Section 6.4). Hence, a CQ will be submitted with an error ϵ and a confidence α definition, as presented in Section 2.5. Given a stateful operation, the approximate window result $\hat{R}_w$ should not deviate by more than ϵ compared to the exact result R_w.

7.3 PARTITION MODEL

As discussed above, in order to reduce *imbalance* we need to reduce the load of the most loaded worker w by maintaining the accuracy of the result within the accuracy requirement. In essence, given a stateful aggregation f, the partition model should iterate over w's assigned groups, and examine if any of them can have its aggregate result R_w be approximated. This entails that for every group g assigned to w, for which $N_g \subseteq S^w_i$, we need to identify a simple random sample $n_g \subset N_g$ so that:

1. $|n_g| < |N_g|$
2. if $R_w = f(N_g)$ and $\hat{R}_w = f(n_g)$, then $\Pr(e(R_w, \hat{R}_w) \leq \epsilon) = \alpha$ (e can be any of the error metrics presented in Section 2.5.1)

For example, for the query of Figure 8, if w is assigned a *route* group with 1000 tuples and an average fare value of \$10, the goal is to find simple random sample of size < 1000 that will result in an average fare value of $\$10 \pm 1$. This process needs to take place until a group g is found that reduces w's load and keep repeating this process as long as *imbalance*

reduces.

7.3.1 Sample Size Estimation

The sample size n_g is a well-studied problem in statistics literature and can be estimated by tracking the attribute values that participate in f. In the example of Figure 8, this means that for a *route* group, the distribution of *fare* values needs to be tracked. Similar to SPEAr, our sample size estimation model employs normal approximation to estimate the expected error of an approximate result (Section 6.6).

According to [42], there are various approaches to the problem of estimating the proper sample size, given ϵ and α. A pivotal step is to estimate the total population's variance (S^2), and this is achieved by taking the sample in two steps: first by creating a simple random sample of size n_1 from which the value of s^2 is gained; then n_g is estimated using the s^2 as an estimation for S^2. For instance, if f is the arithmetic mean, s_1^2 is the variance estimated from the initial simple random sample of size n_1 and $V = \frac{\epsilon^2}{t^2}$, where t is the inverse cumulative probability of the standard normal distribution at α. n_g is estimated as:

$$n_g = \frac{s_1^2}{V}(1 + \frac{2}{n_1}) \tag{7.1}$$

The distribution of the attribute values is assumed to be approximately normal, therefore n_1 needs to be fairly large. If the population's variance S^2 is known exactly, the required sample size is $\frac{S^2}{V}$.

As a result, any attempt to reduce *imbalance* given the constraints of ϵ and α, requires the knowledge of the variance S^2 (or an estimate) of the attribute values being used by f, for each group g appearing in S_i^w.

7.4 SHEDPART

In this section we present our hybrid load **Shed**ding-***Part****itioning*** operator named ShedPart, which targets grouped aggregate operations like SPEAr. ShedPart's goal is to reduce *imbalance* among workers, by searching for groups that qualify for approximating their aggregate

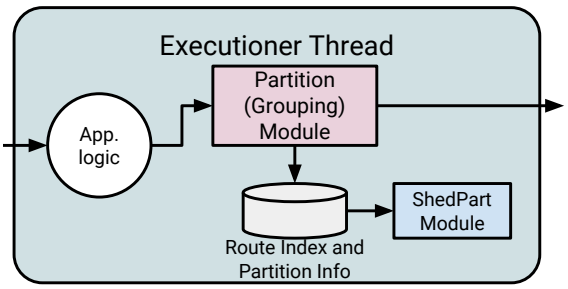


Figure 52: The internal architecture of a worker thread in modern SPEs, with the ShedPart module.

value given an accuracy requirement (ϵ, α). ShedPart is different from SPEAr's model in the following manner: SPEAr attempts to "trim" complete windows and produce approximate answers within the specified accuracy at the worker (thread) level; whereas ShedPart attempts to "trim" distinct groups and selectively produce approximate answers for those within the specified accuracy. The system model we adopt for this work is the one presented in Section 2.1, and the types of queries that ShedPart receives are similar to the query types submitted to SPEAr without a budget specification (Figure 44). In terms of error metrics, we adopt the formulas presented in Section 2.5. As we discuss below, ShedPart has been built and tested for Storm, whose architecture and runtime was presented in Section 2.6.2.

7.4.1 ShedPart Architecture

A worker's threads internal architecture appears in different variations on modern SPEs. In most SPEs, each worker maintains a module that applies the operator's logic, which is part of a broader CQ. In addition, each thread maintains a module that creates different partitions based on the operator's logic output exists as well. In Storm, each `Executioner` thread maintains an application logic module and a partition module (or `Grouper` in Storm's jargon). Every time the application logic produces a tuple, it is passed to the partition module, which in turn makes a decision on the destination of this tuple (i.e., receiving thread). Figure 52 depicts this architecture. The partition module has to maintain some

additional information in the event it supports load-balancing functionalities. For instance, if the partition module applies a load-aware stream partition algorithm like AM/cAM, then the *Route Index* and the *Partition Info* will include the number of tuples and the number of distinct groups sent to each downstream worker.

ShedPart is part of the `Executioner` thread's memory space and it lies between the partition module and the output. Its goal is to keep track of routing information, which can access through the partition module. At tuple arrival, it monitors the decisions made by the partition module, and incrementally updates information that it uses in the sample size estimation for different groups. At watermark arrival, it explores the distinct groups for opportunities to limit *imbalance*. By the time it concludes its operation, it allows the watermark to reach the downstream operators, along with information regarding individual groups.

As explained in Section 7.3.1, to reduce *imbalance* there is a need to keep track of each group's variance, in addition to each group's frequency. Also, to calculate *imbalance* the load of each worker needs to be available, along with the routing index. Apart from each group's attribute values' variances, the rest of the information are readily available by stream partition algorithms like AM and cAM.

7.4.2 ShedPart Algorithm

The operations of ShedPart are different during (a) tuple arrival (i.e., tuple partition), and (b) watermark arrival (i.e., window completion).

7.4.2.1 Tuple Arrival: Every time a tuple is produced by the application logic and is partitioned by the partition module, ShedPart needs to maintain some additional information. In detail, for each group g that appears in a window, ShedPart needs to keep track of the variance of the values that appear in the result used in the CQ. For instance, for the CQ of Figure 8, a group is defined by a tuple's $k = \{route\}$ and the payload $p = \{fare\}$. As a result, ShedPart needs to maintain a variance of *fare* values for each distinct *route* in a window. This information is needed in order to establish the proper sample size to estimate

Algorithm 10 Shed-Part algorithm

```
1: procedure ShedPart(L, R, GI)
2:     conclude = False, black_list = ∅, trim_info = INIT_MAP(∅)
3:     sorted_group_map = SORT_GROUPS(R, GI)
4:     repeat
5:         conclude = True
6:         w= MOST_LOADED_INDEX(L)
7:         G_w=sorted_group_map.get(w)
8:         if G_w ≠ ∅ then
9:             for all g ∈ G_w do
10:                if g ∉ trim_info ∧ g ∉ black_list then
11:                    s = SAMPLE_SIZE(GI.freq(g), α, ϵ, GI.var(g))
12:                    if s < GI.freq(g) then
13:                        conclude=False
14:                        UPDATE_LOAD(w, g, L, s)
15:                        trim_info.add(g,s)
16:                    else
17:                        black_list.add(g)
18:        else
19:            return trim_info
20:    until ¬conclude
21:    return trim_info
```

a population's characteristics (as explained in Section 7.3.1).

Variance requires constant memory for evaluating as it can be incrementally calculated as [81]:

$$\begin{aligned} M_1 &= x_1, \quad M_k = M_{k+1} + (x_k - M_{k-1})/k \\ S_1 &= 0, \quad S_k = S_{k-1} + (x_k - M_{k-1}) \times (x_k - M_k) \end{aligned} \tag{7.2}$$

The values x_k indicate the values that are of interest, M_k is the mean value after the x_k value, and S_k is the standard deviation after the x_k value. The variance is S_k^2. In the example query of Figure 8, x_k are the *fare* attribute values for each *route*.

In addition, as presented in Section 7.3.1, for the estimation of the variance of attribute values only a sample is required. This will allow for maintaining the overhead for ShedPart low as the values of only a subset of values will be considered for variance estimation.

7.4.2.2 Watermark Arrival: When a watermark gets generated by the source of the input data of the `Executioner` thread, it marks the completion of a window. At this point, ShedPart explores S_i^w's groups for possible opportunities in reducing *imbalance*. The basic ShedPart algorithm is depicted in Algorithm 10.

ShedPart's algorithm consists of a main loop (Lines 4- 20), which repeats as long as the most loaded worker w can not have its load reduced further. This outer loop is justified by Lemma 1, which indicates that by reducing w's load with reduce *imbalance*. The main loop can conclude when either all of w's groups do not allow for further reduction (i.e., a resulting sample size $n_g \geq N_g$), or because all of w's groups have already been marked as "trimmed" (i.e., a sample size has been established). At this point, we have to point out that on every iteration, the most loaded worker w might change.

Algorithm's 10 has the following input: L which is an array integers of size $\mathcal{V}$ with the total number of tuples assigned to each one of the $\mathcal{V}$ workers. This array is the same array (L) used by PK (Section 3.3.2.3), AM (Algorithm 3), cAM (Algorithm 4), and LM (Algorithm 5). The next argument of ShedPart is the route index $R : g \rightarrow \{w_1, \ldots, w_m\}$, which maps each distinct group g to the destination worker(s) it has been sent to. The destination worker(s) can be more than one, in case an *m-choice* partition algorithm is used (e.g., PK and CM). Most of the times, R can be the hash function(s) used for partitioning each group g. Finally,

a map $GI : g \rightarrow (N_g, s_g^2)$ that for each distinct group g returns its frequency (N_g) and the estimated variance of its attribute values s_g^2. Essentially, GI is the only additional structure that ShedPart requires and has an amortized memory overhead of $O(\mathcal{G})$. Fortunately, there are sketching techniques that can reduce this memory overhead [43, 115], and GI can be built incrementally using Equations 7.2. The output of Algorithm 10 is a map trim_info : $g \rightarrow s_g$, which for a subset of distinct groups g returns a sample size s_g (Line 21). This value indicates that tuples that belong to group g can have their aggregate result be approximated by a simple random sample of size s_g.

In the beginning of the algorithm, an index named sorted_group_map is created, which maps to each worker a list of of groups in descending order in terms of frequency (Line 3). The creation of this index allows for faster execution of the main loop its creation carries a time complexity of $\mathrm{O}(\mathcal{G} \log \mathcal{G})$ in the worst case. Also, one more auxiliary structure is initialized, named black_list, which is a set structure maintaining the groups that have been already examined and are not fit for trimming.

At every iteration of the inner loop, the most loaded worker w is located by scanning array L and identifying the worker with the maximum number of tuples assigned to it (Line 6). Then, w's sorted list of assigned groups (in descending frequency order) are retrieved from the sorted_group_map (Line 7). In the event that the remaining groups $\mathcal{G}_w$ are empty, ShedPart concludes execution (Line 19). Otherwise, each group g of $\mathcal{G}_w$ is scanned (Line 9). If a group g is neither part of trim_info nor black_list, then its sample size s is estimated (Line 11). If s is smaller than N_g (i.e., the recorded frequency in GI), then w's load is updated (Line 14), a record for g is created in trim_info (Line 15), and the outer loop's condition is updated (Line 13). On the other hand, if s is greater or equal to N_g, the group g is added to the black_list structure so that it is not revisited in the future (Line 17).

Overall, the algorithm's runtime complexity is relevant to the distribution of groups per worker, and the variance of attribute values that are part of the aggregate operation. The worst runtime complexity materializes when all distinct group values are visited. In this case the main loop will have a runtime of $\mathrm{O}(\mathcal{G})$. In addition, the memory overhead is in the worst case $\mathrm{O}(\mathcal{G})$, which is when exact data structures are used.

7.4.3 ShedPart Optimizations

The runtime and the storage complexity of ShedPart can become very high. In fact, both can end up being $O(\mathcal{G})$. This can result in significant performance degradation in congested environments. To this end, we implemented two optimizations, which try to alleviate the complexity of time and storage.

7.4.3.1 Storage Optimization As mentioned earlier, ShedPart maintains a tuple (N_g, s_g^2) for each group g that appears in a window S_i^w. As a result, the storage cost becomes proportional to $\mathcal{G}$ and will get very big if $\mathcal{G} \to \infty$. To this end, we expand ShedPart's internal structure to separate group tracking based on each group's frequency. In essence, ShedPart receives a parameter T_g, which is a frequency threshold used to separate distinct groups of S_i^w into two categories: the *heavy hitters* (i.e., groups that $N_g \geq T_g$), and the *cold groups* (i.e., groups that $N_g < T_g$). This optimization is similar to the one proposed for ColdFilter [140], but ShedPart uses this to limit the storage overhead.

Internally, ShedPart maintains an approximate data structure of, constant size, to keep track of *cold groups'* frequencies. Currently, ShedPart uses a CountMin sketch [43], whose size is controlled by the desired level of accuracy. In addition, ShedPart maintains a hash table for each group g's tuple (N_g, s_g^2) $\mathcal{H}$. At tuple arrival, the tuple's group g is extracted, and $\mathcal{H}$ is checked for a record for group g. If one does not exist, the estimated count for g is retrieved from the CountMin sketch, $\hat{N}_g$. If $\hat{N}_g = 0$ then g's record in the CountMin sketch is incremented. Otherwise, ShedPart checks if $\hat{N}_g \geq T_g - 1$. If the previous check is true, g is promoted to the *heavy hitters* groups, and a record is created in $\mathcal{H}$. Otherwise, the CountMin sketch increments g estimated frequency.

In our experience, we found that setting $T_g \geq \frac{|S_i^w|}{\mathcal{G}}$ provides the best results. ShedPart aims to improve *imbalance*, which appears in skewed datasets. The former is not the case with uniform datasets, since the stream partitioning algorithm manages to balance the workload among workers (see Chapter 3). Highly frequent groups are more likely to cause *imbalance*, and by setting $T_g \geq \frac{|S_i^w|}{\mathcal{G}}$ ShedPart focuses on those.

Until a group g makes it to $\mathcal{H}$ the attribute values needed for s_g^2 will be lost. However,

this does not impact the sample size estimation since the Equation 7.1 requires the variance of a subset of the population, which in this case would be all the values of group g for S_i^w.

Finally, this optimization is used to limit the time complexity of Algorithm 10 as well. By considering only *heavy hitter* groups during exploration, only those will be examined for down-sampling. Hence, the *cold* groups will not slow-down execution, and potentially impact the performance benefits of ShedPart.

7.4.3.2 Incremental Variance Monitoring In our experience with real datasets, we encountered situations where the overall performance improvement by using ShedPart was minimal or non-existent. Often, this happened because the attribute values' did not allow for approximation. The previous occurred when s^2 was very wide, Equation 7.1 resulted in a sample size equal to the actual size of a group. Thus, executing ShedPart's algorithm would slow-down an SPE by increasing the total window processing time.

To this end, we enhanced ShedPart's algorithm to keep track of the tuple savings incrementally. When a *heavy hitter* group's variance is updated, its required sampling size $\hat{n}_g$ is updated as well. At watermark arrival, the percentage $\frac{|D-\hat{D}|}{|S_i^w|}100$ is calculated, where D is the sum of frequencies of all *heavy hitter* groups[1] and $\hat{D}$ is the sum of the estimated sample sizes $\hat{n}_g$. If this percentage is lower than $\mathcal{T}$, which is ShedPart's percentage parameter, then ShedPart does not execute Algorithm 10. As we will show in our experimental evaluation (Section 7.5) this optimization protects ShedPart from unnecessary explorations, which would cause performance degradation.

7.5 SHEDPART EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation of ShedPart. Our aim is to document the merits of using ShedPart in real-world and synthetic datasets, and specifically identify under which circumstances ShedPart will offer performance improvements, what is the impact in *imbalance* and maximum parallel step reduction, and overall performance improvements

[1] Each one incremented by T_g.

in real-world datasets.

7.5.1 Experimental Setup

We have implemented a simulator of ShedPart on Storm v1.2. The reason that we did not provide a full-fledged implementation of ShedPart is because of Storm's watermark mechanism. Specifically, Storm has each `Executioner` thread generate its own watermarks, by spawning a background Java thread. This thread wakes up periodically and produces a watermark tuple, which is fed to the `WindowManager` module of the thread. This does not allow for a centralized control of the watermark generation in the upstream node.

To overcome this limitation, we implemented ShedPart as an intermediate `BaseWindowedBolt` that sits between a data generator bolt, and the actual `BaseWindowedBolt` performing the *stateful* aggregate operation. The latter's watermark generation mechanism is disabled. When the watermark mechanism is triggered, our ShedPart bolt executes Algorithm 10 (selectively) and generates the sample sizes for each group for the current window. It forwards this information to the downstream bolt to down-sample for the active window accordingly, and also produces a watermark to trigger execution.

Our experimental setup is the Amazon EC2 cluster explained in Section 6.7.2. In terms of datasets, we used two of the datasets described in Section 2.6.1: Namely, the DEBS and the GCM datasets. For the sensitivity analysis of ShedPart we introduced synthetic datasets, in order to explore a wider spectrum of distributions that allowed us to map the circumstances under which ShedPart provides significant performance improvements.

7.5.2 Experimental Results

In Section 7.5.2.1 we investigate under which circumstances ShedPart achieves performance improvements. Next, in Section 7.5.2.2 we measure the impact of ShedPart in *imbalance* and maximum parallel load. Finally, in Section 7.5.2.3 we measure overall processing time when ShedPart is used.

7.5.2.1 Sensitivity Analysis In order to establish under which circumstances ShedPart allows for performance improvements, we conducted a sensitivity analysis using synthetic data. In our analysis, we allowed the error to get any of the following values $\epsilon = \{1, 5, 10, 15\}\%$ and the confidence $\alpha = \{85, 90, 95, 99\}\%$. Those values are often used for estimating population's aggregate values. In addition, for each of the distributions that we present below, we generated 10 million values. The distributions that we have used to generate data are the following:

1. Normal Distributions $\mathcal{N}(\mu, \sigma)$ from the work presented in [26]. The normal distributions used in this work vary in the mean value $\mu = (10, 40, 70, -10, 85)$, and in the standard deviation value $\sigma = (10, 15)$. In our analysis we wanted to experiment with normal distributions with wider tails, so we added the values 20 and 25 for σ.
2. Log-Normal distribution is documented as the most appropriate distribution for characterizing user behavior in social networks [27]. In detail, we produced two distributions: $L_1(\mu_1 = 1.789, \sigma_1 = 2.366)$ and $L_2(\mu_2 = 2.245, \sigma_2 = 1.133)$.
3. Zipf Distributions with an exponent s ranging between 0.5 to 4.

For generating distribution values we used Apache Commons Math Java Library v3.6.1. In this set of experiments, our goal is to measure the sample size estimated by ShedPart (using Equation 7.1). This will aid us in understanding the potential benefits of ShedPart for different value distributions.

As far as $\mathcal{N}$ distributions are concerned, our analysis revealed that σ is the parameter that affects the sample size detected the most. In detail, for all $\sigma < 15$, the resulting sampling rate required ranges from 54% to 0.04%, regardless of μ's value. Those numbers translate to 5,400,000 and 400,000 accordingly, and indicate a significant improvement in data volume. When $\sigma = 15$, $\epsilon = 1\%$, and $\alpha = 99\%$ the sample needs to be equal to the window. However, for the rest of the error values, the sample rate required significantly drops. Figure 53 presents the resulting sampling rate. For confidence values smaller than 99%, the sampling rate starts from 60% and quickly drops down to 0.1%. As a result, when attribute values follow a Normal distribution, ShedPart will detect opportunities to reduce the processing load. Turning to the L_1 and L_2 distributions, ShedPart is not able to reduce

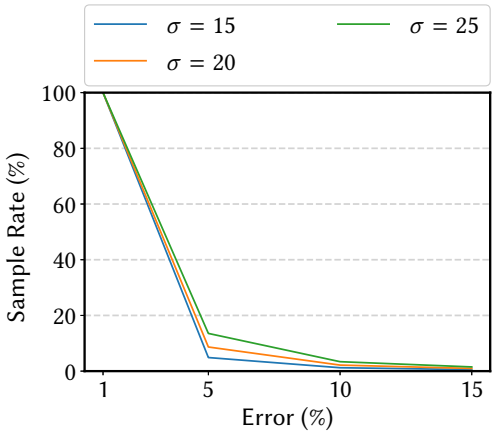


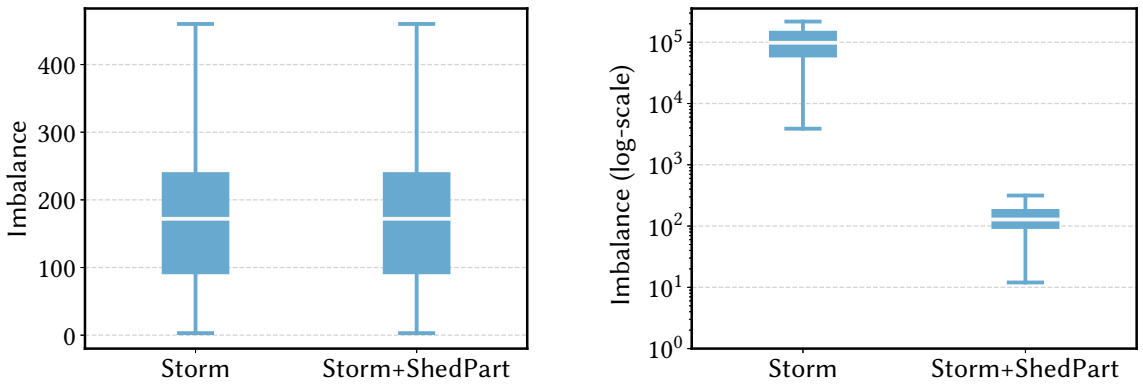
Figure 53: Sample rate estimated by Equation 7.1 for 99% confidence.

the processing volume of both, unless the confidence $\alpha \leq 50\%$. This is due to the variance of both distributions, which is very high due to the long tails of both L_1 and L_2.

As far as Zipf distributions are concerned, ShedPart is not able to reduce the processing amount for $s < 1.75$. When $s = 1.75$, ShedPart is able to reduce the processing amount for $\epsilon \geq 10\%$ and $\alpha \leq 90\%$. In this case, the sample rate starts from 65% and drops. For $s = 2$, the sample rate starts from 67.3% when $\epsilon \geq 5\%$ and $\alpha \leq 95\%$ and drops. For $s \geq 3$ the sampling rate drops significantly for any combination of (ϵ, α) values.

Take-away: Our analysis on ShedPart's ability to reduce load indicates that the variance of values is the dominant factor. In detail, the variance of attribute values $s_2 < N_g$, where N_g is the number of tuples that belong to a group g.

7.5.2.2 Impact on Imbalance Next, we examined ShedPart's impact on *imbalance*. As mentioned earlier in this chapter, the main goal of ShedPart is to achieve a more balanced allocation among downstream workers. For this set of experiments, we used the DEBS and GCM datasets. Both of those come with group aggregation queries that feature the calculation of an arithmetic mean for different groups. Those two datasets present significant differences in the number of groups per window (up to four for GCM and 10 thousand in average for DEBS), and in the variance in values. As a result, ShedPart is expected to face



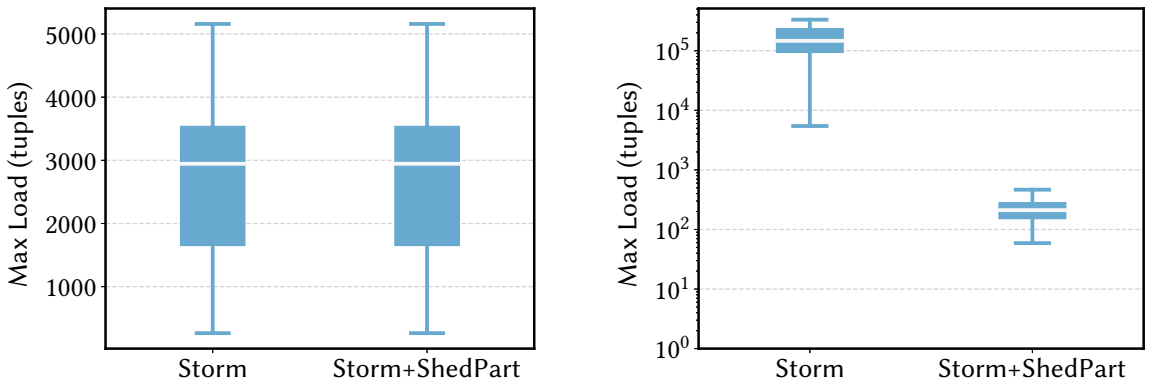
a: DEBS

b: GCM

Figure 54: Imbalance.

different opportunities to balance the load on each dataset. For this set of experiments we set $\epsilon = 10\%$ and $\alpha = 95\%$ for both DEBS and GCM. In addition, we set T_g to three, and $\mathcal{T}$ to 15%. For the application logic bolts, we set the number of workers (i.e., parallelism hint) to 4.

Figure 54 depicts the *imbalance* achieved when only Storm's partition algorithm is used, and when ShedPart is used. For Storm, we used FLD grouping, which proved to be equivalently good for those datasets (as shown also in Section 3.7). As far as DEBS is concerned, ShedPart fails to improve on *imbalance*. This happens because the variance of attribute values for DEBS data (i.e., the *fare* value) receives a wide range of values. In addition, the majority of routes appear less than 10 times per window (on average), and very infrequent *route* groups cover a significant portion of the window. As a result, ShedPart is not able to approximate the mean *fare* value with smaller samples, and the *imbalance* is similar to vanilla Storm (as illustrated in Figure 54a). In essence, DEBS is similar to the case of L_1, L_2, and Zipf with $s \leq 1$ distributions, which feature long tails. However, this is not the case with GCM.



a: DEBS

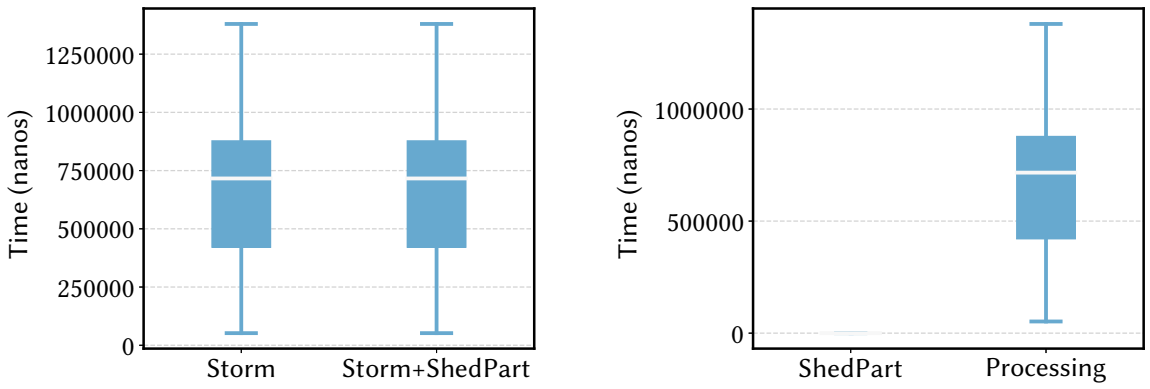
b: GCM

Figure 55: Maximum worker load.

The GCM dataset features up to four groups (i.e., scheduling classes) per window, and the mean CPU request values do not carry a high variance. Therefore, ShedPart is able to approximate the mean CPU request of each scheduling class by a small set of tuples. This results in significant improvement in terms of *imbalance*, as is shown in Figure 54b. GCM resembles the case of $\mathcal{N}$, and Zipf with $s \geq 3$ distributions, which feature low variance in their values.

Another dimension in which ShedPart can contribute is in terms of the most loaded worker's load per window. In essence, this affects the time of the partial-evaluation step of a stateful aggregate operation (Section 3.1). Figure 55 illustrates the box-plots of the load of the most loaded worker per window, for DEBS and GCM. As far as DEBS is concerned, ShedPart is unable to limit the maximum load (Figure 55a). This is expected since ShedPart was unable to effectively improve performance of *imbalance* on DEBS.

Turning to GCM, the picture is very different. As can be seen in Figure 55b, ShedPart is able to improve performance by almost two orders of magnitude. This happens due to the low variance of data, and the fact in GCM there are not long tails (i.e., many groups of



a: Total window runtime.

b: Breakdown of total time.

Figure 56: Window runtime analysis for ShedPart on DEBS.

infrequent data). As a result, GCM is able to reduce the load by approximating the mean CPU requested by each scheduling class effectively.

Take-away: ShedPart is able to significantly reduce *imbalance* and the maximum load on the parallel step of a stateful operation, when data are fit for approximation. ShedPart can reduce *imbalance* and the partial evaluation step by more than an order of magnitude.

7.5.2.3 Overall Performance In this set of experiments, we measure the overall window processing time for each window of DEBS and GCM. Our goal is to measure ShedPart's ability to avoid wasting time in the exploration, and the performance acceleration in terms of total processing time. Similar to the previous experiments on those datasets we set $\epsilon = 10\%$, $\alpha = 95\%$, $T_g = 3$, $\mathcal{T} = 15\%$, and the number of workers to 4. The reported times represent the sum of ShedPart's execution and the processing time for each window. In addition, we execute each experiment seven times, and we report average times among all runs after subtracting the maximum and the minimum.

Starting from DEBS, the main challenge for ShedPart is to avoid wasting time in the exploration phase. This is crucial because DEBS features tens of thousands of groups per

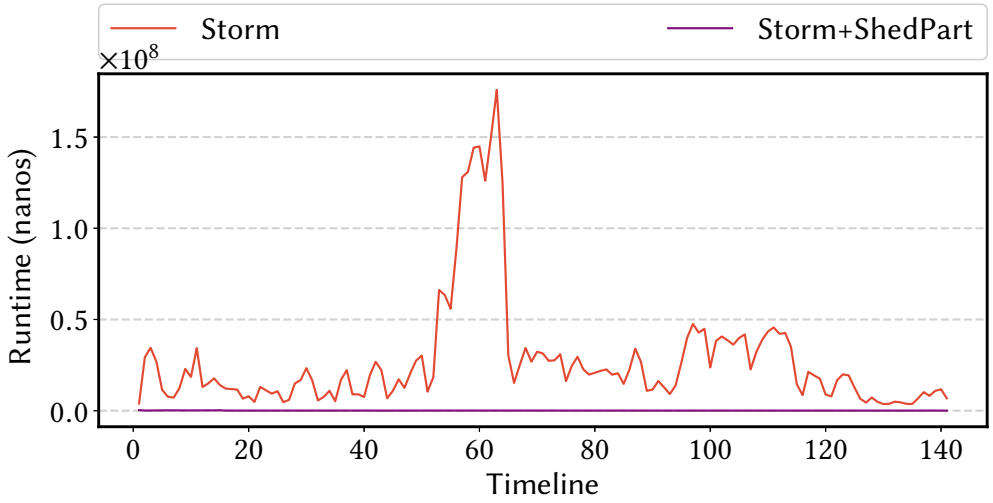


Figure 57: Max parallel step runtime on GCM.

window, and the iterations can impact performance. Also, DEBS is not able to be accelerated because of the values and the frequency of appearance of its groups (as explained in analysis of Section 7.5.2.2). Figure 56 presents the box plots of the total window runtimes (Figure 56a) of Storm and Storm with ShedPart, and the break-down of Storm and ShedPart in ShedPart execution and processing of tuples (Figure 56b).

It can be seen that ShedPart's optimizations mechanisms are able to protect ShedPart from unnecessary overhead, when the data are unlikely to be able to be trimmed. For this experiment, ShedPart's mechanism is configured to check for at least a 15% reduction in the processing volume of the window (i.e., $\mathcal{T} = 15\%$). During watermark arrival, our ShedPart implementation identifies that this is not feasible. Thus, it skips the execution of the ShedPart algorithm and jumps directly to processing. Figure 56a depicts the box plot of total window runtime for Storm and for Storm with ShedPart. Both box plots are identical, which is expected since DEBS is not fit for approximating groups based on the (ϵ, α) specification (which is also shown in the experiments on *imbalance*-Section 7.5.2.3). In addition, ShedPart spends (almost) no time in exploration for approximation opportunities (as shown in Figure 56b).

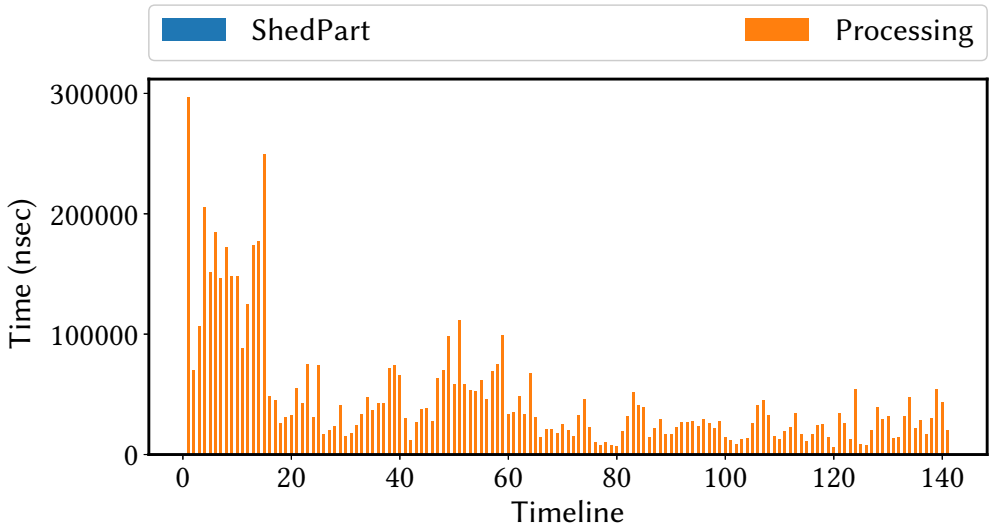


Figure 58: Breakdown of time spent on each window, in terms of ShedPart and processing (GCM).

Turning to GCM, ShedPart is more effective and can lead to significant performance improvements. Figure 57 illustrates the total runtime for Storm and Storm with ShedPart. As shown in Figure 55b, ShedPart is able to significantly reduce the load on the most loaded worker of each window. As a result, the total runtime drops significantly. At the same time, as shown in Figure 57, in the windows between 50 and 65 there is a temporary increase in the frequency of a particular scheduling class. Storm has to sustain this temporal skew in input. On the other hand, with ShedPart this temporary *imbalance* disappears.

Figure 58 presents the breakdown of Storm with ShedPart. GCM features up to four distinct groups per window, which results in insignificant time spent for ShedPart execution. As shown in Figure 58, all time is spent in processing the result.

Take-away: ShedPart is able to significantly improve the performance of an SPE, when data are fit for approximation. In addition, when data are not able to be efficiently approximated, ShedPart does not introduce additional overhead to the total runtime.

7.6 SUMMARY

In this chapter we presented ShedPart, which is a hybrid method that combines stream partitioning and load shedding. ShedPart is motivated by the shortcomings of stream *partitioning* solutions under certain circumstances to balance load accordingly. Previous solutions of this dissertation followed holistic approaches to approximation. In detail, SPEAr follows a decentralized approach to approximate a window's result by processing a subset of the tuples of each distinct group. SPEAr makes a decision based on the local knowledge that it maintains and its goal is to provide an average error among distinct groups, without incurring additional overhead to the workflow triggered by a watermark.

In contrast, ShedPart prioritizes *imbalance* and addresses approximation in a global level and at the finest approximation granularity. ShedPart operates at the source level and iterating over all tuples of the window. In addition, ShedPart approximates a result on the distinct group level, which is not the case with SPEAr's approach. As a result, ShedPart approximates only the groups that appear very frequently, and leaves the infrequent groups intact. This leads to a lower approximation error for each window. However, ShedPart incurs additional overhead, which can lead to either excess memory usage or a higher window processing time compared to normal execution.

Our proposed optimizations work as safeguards against introduction of additional overhead. Despite the fact that ShedPart relies heavily on the underlying data, our design avoids processing overheads, which emanate from futile exploration of acceleration opportunities. Our experimental analysis indicates that ShedPart can reduce both *imbalance* and the maximum load of a partial evaluation step in a stateful aggregate operation. In addition, ShedPart improves performance and alleviates temporal skewness on certain workloads.

8.0 EPILOGUE

In this chapter, we conclude this dissertation by summarizing our contributions and stating their broader impact (Section 8.1). Furthermore, we list a number of future research directions, which we identified as important for improving the applicability of our techniques (Section 8.2). To put our contributions in context, we show in Figure 59 a taxonomy of existing SPE adaptation techniques. In it, each technique is categorized based on its expected usage: either for *exact* or *approximate* processing. In addition, Figure 59 presents the separation for *approximate* processing, which divides adaptation techniques based on their ability to provide accuracy guarantees.

8.1 SUMMARY OF CONTRIBUTIONS

As far as adaptivity techniques for *exact* processing are concerned, our work's contributions span both stream *partitioning* and *re-partitioning*. In detail, two of our proposed stream *partitioning* algorithms, namely AM and cAM, achieve up to an order of magnitude better throughput (Chapter 3). Turning to stream *re-partitioning*, we have presented the design of *Synefo*, which is an SPE that offers *re-partitioning* capabilities without requiring changes in the underlying fabric (Chapter 4). In addition, UniMiCo, our proposed state migration protocol for stream *re-partitioning* can migrate CQs without causing performance degradation for a broad range of window semantics (Chapter 4). In Figure 59, stream *partitioning* and *re-partitioning* are part of techniques aimed for applications requiring *exact* processing. Our aforementioned contributions (i.e., AM, cAM, *Synefo*, UniMiCo) fall into this category, and each of them are illustrated as part of their corresponding technique's circle. Our *partition*

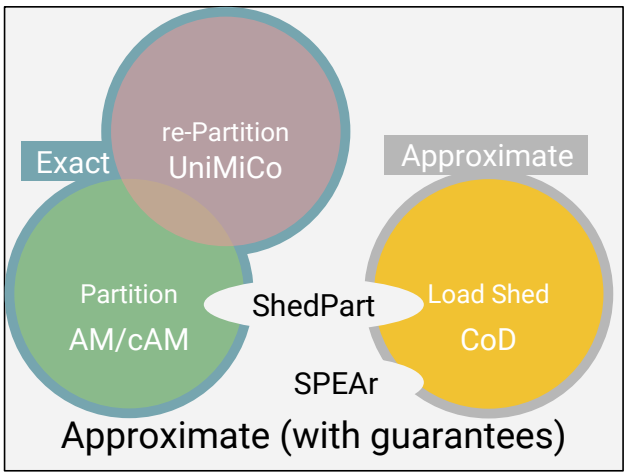


Figure 59: Design Space of techniques for adaptability techniques.

algorithms and state migration protocol are aimed to impact the efficiency of existing SPEs, in terms of adaptability when input increases. Our algorithms will react to workers' loads and consider an active CQ's semantics. As a result, stream processing will become more efficient and result in lower operational costs and energy consumption.

Turning to adaptation techniques for *approximate* processing without guarantees, we contributed CoD, which is a load shedding technique for data streams with concept drift (Chapter 5). CoD can achieve more than an order of magnitude higher accuracy compared to the state of the art, and at the same time not introduce performance overhead for exploring each window's concept. In Figure 59, load shedding is part of the techniques for *approximate* processing without guarantees. In turn, CoD is part of this set of techniques, since it is not designed for providing accuracy guarantees. CoD's impact lies in the fact that load shedding can be applied in applications that aim at real-time pattern extraction, without deteriorating a CQ's accuracy.

Turning to *approximate* processing with accuracy guarantees, we have shown that it is feasible in current SPEs and leads to significant gains. Those gains become apparent both in performance, scalability, and reduction of operational costs. In detail, we presented a novel system model, which automatically detects opportunities for accelerating execution. SPEAr,

our proposed SPE, crafts appropriate samples and recognizes whether an approximate result is within a user-specified accuracy requirement. If the former holds true, SPEAr skips the processing phase of a window, which leads to orders of magnitude better performance. Our experimental evaluation indicates that SPEAr is able to cut down operational resources and reduce total processing time significantly, when compared to *exact* and *approximate* processing solutions (Chapter 6). In Figure 59, SPEAr is part of the techniques for *approximate* processing with accuracy guarantees. Our work on SPEAr can impact the efficiency of SPEs, the pricing models that are used for current SPE-as-a-Service offerings, and the way users perceive stream processing in general (more details in Section 8.2). The most important aspect of SPEAr's impact is that is has illustrated that acceptable result quality can come with frugal resource allocations.

Last but not least, we contributed a hybrid technique, named ShedPart, which comes from the combination of *approximate* stream processing with the goal of reducing load *imbalance*. ShedPart investigates opportunities to approximate a subset of a window result, in order to achieve a balanced load allocation among workers (Chapter 7). Compared to SPEAr, ShedPart follows a global view of load distribution and selectively approximates large groups. Our experimental evaluation indicates that ShedPart is able to significantly improve both *imbalance* and performance. In addition, ShedPart is enhanced to identify situations which are not likely to reduce *imbalance*. In Figure 59, ShedPart lies mostly in the set of techniques for *approximate* processing with accuracy guarantees. However, parts of it overlap with stream *partitioning* and load shedding, since it has been created by combining fundamental principles of those techniques. ShedPart's broader impact is similar to SPEAr's and those two differ in the granularity level of *approximation*. The former examines each group individually; whereas the latter is a solution for approximating the total window. As a result, ShedPart can lead to insignificant performance gains, and is able to achieve better approximation accuracy.

8.2 FUTURE RESEARCH DIRECTIONS

This dissertation has revealed promising research directions for SPEs. In detail, our very promising experimental results with SPEAr, and our experience with ShedPart reveal huge potential in further exploring this line of research and address the following matters:

Combination of Load-shedding and Re-partitioning: ShedPart has revealed that ad-hoc approximation on a sub-set of input data can lead to significant performance improvements. In Figure 59, there exists no technique that connects stream *re-partitioning* and load shedding, while offering accuracy guarantees. A simple application of ShedPart to selectively send subsets of the migrated state can lead to a reduced state migration time. In turn, such a thing will limit the impact of checkpoint-based techniques to an SPE's response time during *re-partitioning*.

Combination of all three techniques: To the extent of our knowledge, an SPE enhanced with a decision framework, which is able to combine all three techniques, does not exist. In real world scenarios, the decisions on enabling each technique follow a "human-in-the-loop" approach. Concretely, an administrator monitors an SPE's performance and makes appropriate decisions. However, we gradually see the merits of automating the configuration of DBMSs [133, 90], thus we believe that this is a very promising line of research.

Dynamic Budget Allocation: In the current version of SPEAr, the budget is statically allocated, and identifying the proper value for a stream is a daunting task. In our view, this approach is unsuitable for the end user, who would not know the appropriate budget for a CQ-stream-fabric combination. As a result, tuning SPEAr to achieve reasonable acceleration from the start can be very hard. This reveals a promising research direction in developing active budget learning techniques, so that SPEAr is able to detect a suitable budget that will provide a reasonable trade off between accuracy and processing time.

Approximate Join Support: Joining streams before aggregation is a common use-case for SPEs (especially equality join operations). Window joins between streams is a challenging task and it has been widely studied in the past [23, 60, 127, 50, 86, 45, 74, 119]. However, SPEAr (in its current form) does not support joins and an interesting research direction is to examine how its model can be expanded for streaming joins with accuracy guarantees.

New User APIs: We observed that analysts' ability to identify patterns can be hindered by "noise", which often materializes through rare groups appearing in sliding windows. However, analysts are mostly interested in persisting trends, and are willing to sacrifice up to a specific percentage of groups that are part of this "noise". This raises the opportunity to investigate novel interfaces for approximate stream processing, which can improve both analysts' productivity and an SPE's performance.

Pricing Models: SPEs-as-a-Service are offered by many Cloud Service Providers (CSPs) (e.g., AWS Kinesis, Azure Streams etc.). These services follow pricing models that are related to the processing capacity associated with a user's query. If the option for accelerating execution is available, then numerous challenges arise in the pricing model offered. For instance, a user needs to have guaranteed monetary savings from the CSP, if the former opts in using accelerated processing. Similarly, the CSP has to offer such guarantees, if input data follow specific properties. Those challenges need to be addressed in order to harness practical monetary/resource impact for both the CSP and the users.

Bibliography

[1] Aqsios: Algorithms and metrics for new generation data stream management systems. http://db.cs.pitt.edu/group/projects/aqsios, 2010.

[2] Amazon kinesis. https://aws.amazon.com/kinesis/, 2018.

[3] Apache apex. https://apex.apache.org/, 2018.

[4] Apache samza. http://samza.apache.org/, 2018.

[5] Azure stream analytics. https://azure.microsoft.com/en-us/services/stream-analytics/, 2018.

[6] Kafka streams. https://kafka.apache.org/, 2018.

[7] D. J. Abadi, Y. Ahmad, M. Balazinska, et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[8] D. J. Abadi, D. Carney, U. Çetintemel, et al. Aurora: A new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.

[9] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A fast decision support systems using approximate query answers. In *PVLDB*, pages 754–757, 1999.

[10] S. Acharya, P. B. Gibbons, et al. Join synopses for approximate query answering. In *ACM SIGMOD*, pages 275–286, 1999.

[11] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498, 2000.

[12] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *SIGMOD*, pages 574–576, 1999.

[13] S. Agarwal, B. Mozafari, A. Panda, et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[14] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[15] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[16] R. Ananthanarayanan, V. Basker, S. Das, et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.

[17] H. Andrade, B. Gedik, K. L. Wu, and P. S. Yu. Processing high data rate streams in system s. *J. Parallel Distrib. Comput.*, 71(2):145–156, 2011.

[18] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *VLDBJ*, 15(2):121–142, 2006.

[19] A. Arasu, M. Cherniack, E. Galvez, D. Maier, et al. Linear road: A stream data management benchmark. In *PVLDB*, pages 480–491, 2004.

[20] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *PVLDB*, pages 336–347, 2004.

[21] Y. Azar, A. Z. Broder, et al. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[22] B. Babcock, S. Babu, M. Datar, et al. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[23] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 333–353, 2003.

[24] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[25] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[26] P. Bailis, E. Gan, S. Madden, et al. Macrobase: Prioritizing attention in fast data. In *SIGMOD*, pages 541–556, 2017.

[27] F. Benevenuto, T. Rodrigues, et al. Characterizing user behavior in online social networks. In *SIGCOMM*, pages 49–62, 2009.

[28] F. Bin, D. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88, 2014.

[29] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[30] P. Carbone et al. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[31] P. Carbone et al. State management in apache flink: Consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[32] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, pages 1201–1210, 2016.

[33] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *VLDB*, pages 215–226, 2002.

[34] R. Castro Fernandez, M. Migliavacca, et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.

[35] B. Chandramouli and J. Goldstein. Shrink - prescribing resiliency solutions for streaming. In *PVLDB*, 2017.

[36] B. Chandramouli, J. Goldstein, M. Barnett, et al. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2015.

[37] S. Chandrasekaran, O. Cooper, A. Deshpande, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[38] S. Chandrasekaran and M. J. Franklin. Psoup: A system for streaming queries over streaming data. *VLDBJ*, 12(2):140–156, 2003.

[39] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 32(2), 2007.

[40] G. J. Chen, J. L. Wiener, S. Iyer, et al. Realtime data processing at facebook. In *SIGMOD*, pages 1087–1098, 2016.

[41] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[42] W. G. Cochran. *Sampling Techniques*. John Wiley & Sons, 3rd edition, 1977.

[43] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[44] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI*, pages 317–318, 2001.

[45] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[46] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[47] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 10–10, 2004.

[48] M. Durand et al. Loglog counting of large cardinalities. In *ESA*, pages 605–617, 2003.

[49] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, pages 301–312, 2011.

[50] M. Elseidy, A. Elguindy, V. A., and C. Koch. Scalable and adaptive online joins. In *PVLDB*, pages 441–452, 2014.

[51] P. Flajolet et al. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, pages 182–209, 1985.

[52] P. Flajolet et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[53] T. A. S. Foundation. Apache zookeeper. https://zookeeper.apache.org/, 2018.

[54] M. Fu, S. Mittal, V. Kedigehalli, K. Ramasamy, et al. Streaming@twitter. *IEEE Data Eng. Bull.*, 38(4):15–27, 2015.

[55] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. In *PVLDB*, pages 1142–1153, 2017.

[56] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, 2014.

[57] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management*. Springer, 2016.

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

[59] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *CIKM*, 2005.

[60] L. Golab and T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *PVLDB*, pages 500–511, 2003.

[61] Google. Google cluster monitoring trace data. https://bit.ly/2Kgj6B4, 2015.

[62] J. Gray, S. Chaudhuri, A. Bosworth, et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[63] V. Gulisano et al. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 23(12):2351–2365, 2012.

[64] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[65] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[66] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *DEBS*, pages 318–321, 2014.

[67] T. Heinze, L. Roediger, A. Meister, et al. Online parameter optimization for elastic data stream processing. In *SoCC*, 2015.

[68] T. Heinze, M. Zia, R. Krahn, et al. An adaptive replication scheme for elastic data stream processing systems. In *DEBS*, pages 150–161, 2015.

[69] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182. ACM, 1997.

[70] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, 2013.

[71] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *DEBS*, 2015.

[72] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *SIGMOD*, pages 541–553, 2016.

[73] S. Kandula et al. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, pages 631–646, 2016.

[74] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

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

[76] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. Concept-driven load shedding: Reducing size and error of voluminous and variable data streams. In *IEEE BigData*, pages –, 2018.

[77] N. R. Katsipoulakis, C. Thoma, E. Gratta, et al. Ce-storm: Confidential elastic processing of data streams. In *ACM SIGMOD*, pages 859–864, 2015.

[78] N. R. Katsipoulakis, C. Thoma, E. A. Gratta, et al. Ce-storm: Confidential elastic processing of data streams. In *SIGMOD*, pages 859–864, 2015.

[79] N. R. Katsipoulakis, K. Tsakalozos, and A. Delis. Adaptive live VM migration in share-nothing iaas-clouds with livefs. In *IEEE Cloudcom*, pages 293–298, 2013.

[80] R. Klinkenberg and T. Joachims. Detecting concept drift with support vector machines. In *ICML*, 2000.

[81] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[82] A. Koliousis et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, pages 555–569, 2016.

[83] S. Kulkarni, N. Bhagat, M. Fu, et al. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[84] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[85] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast json parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[86] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.

[87] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. In *ICDCS*, pages 27–30, 1996.

[88] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *TKDE*, 11(4):610–628, 1999.

[89] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, 2014.

[90] L. Ma, D. Van Aken, A. Hefny, et al. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, pages 631–645, 2018.

[91] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998.

[92] S. Manku and R. Motwani. Approximate frequency counts over data streams. *PVLDB*, pages 346–357, 2002.

[93] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.

[94] Microsoft. Intelligent query processing in sql databases. https://docs.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing?view=sql-server-2017, 2018.

[95] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, et al. Elastic scale-out for partition-based database systems. In *ICDEW*, pages 281–288, 2012.

[96] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *IEEE ICDE*, pages 76–88, 2010.

[97] D. G. Murray, F. McSherry, R. Isaacs, et al. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[98] M. A. Nasir, G. Morales, D. Garcia-Sorano, et al. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pages 137–148, 2015.

[99] M. A. Nasir, G. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *ICDE*, pages 589–600, 2016.

[100] P. Neophytou, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Power-aware operator placement and broadcasting of continuous query results. In *MobiDE*, pages 49–56, 2010.

[101] P. Neophytou, J. Szwedko, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimizing the energy consumption of continuous query processing with mobile clients. In *MDM*, pages 98–103, 2011.

[102] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *SIGMOD*, pages 511–526, 2016.

[103] Oracle. Approximate query processing in oracle database 12c. https://oracle-base.com/articles/12c/approximate-query-processing-12cr2, 2018.

[104] A. Parameswaran. Visual data exploration: A fertile ground for data management research. http://wp.sigmod.org/?p=2342, 2018.

[105] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *SIGMOD*, pages 1461–1476, 2018.

[106] T. Pham, P. Chrysanthis, and A. Labrinidis. Avoiding class warfare: managing continuous queries with differentiated classes of service. *VLDBJ*, 25(2):197–221, 2016.

[107] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis. Uninterruptible migration of continuous queries without operator state migration. *SIGMOD Record*, 46(3):17–22, 2017.

[108] N. Rivetti, E. Anceaume, Y. Busnel, et al. Online scheduling for shuffle grouping in distributed stream processing systems. In *Middleware*, pages 11:1–11:12, 2016.

[109] K. Rong and P. Bailis. ASAP: prioritizing attention via time series smoothing. *PVLDB*, 10(11):1358–1369, 2017.

[110] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu. Elastic scaling of data parallel operators in stream processing. In *IEEE IPDPS*, pages 1–12, 2009.

[111] M. Shah, M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[112] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Efficient scheduling of heterogeneous continuous queries. In *PVLDB*, pages 511–522, 2006.

[113] A. Shein, P. Chrysanthis, and A. Labrinidis. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*, pages 5:1–5:12, 2017.

[114] A. Shein, P. Chrysanthis, and A. Labrinidis. Slickdeque: High throughput and low latency incremental sliding-window aggregation. In *EDBT*, pages 397–408, 2018.

[115] A. Shrivastava, A. C. Konig, and M. Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *SIGMOD*, pages 1417–1432, 2016.

[116] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys*, pages 239–249, 2004.

[117] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 6th edition, 2010.

[118] R. L. Smith. *Extreme Value Theory. Handbook of Applicable Mathematics*. John Wiley, 190.

[119] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. *PVLDB*, 30, 2004.

[120] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[121] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *DEBS*, pages 66–77, 2017.

[122] K. Tangwongsan, M. Hirzel, S. Schneider, and K. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[123] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. *PVLDB*, 2007.

[124] N. Tatbul, U. Cetintemel, S. Zdonik, et al. Load shedding in a data stream manager. *PVLDB*, 29:309–320, 2003.

[125] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. *PVLDB*, 2006.

[126] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[127] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, pages 625–636, 2011.

[128] Πλούταρχος. *Βίοι Παράλληλοι*. 1470.

[129] C. Thoma, A. J. Lee, and A. Labrinidis. Polystream: Cryptographically enforced access controls for outsourced data stream processing. In *SACMAT*, pages 227–238, 2016.

[130] D. Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD*, pages 1129–1140, 2018.

[131] A. Toshniwal, S. Taneja, A. Shukla, et al. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[132] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. *PVLDB*, 2006.

[133] D. Van Aken, A. Pavlo, et al. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[134] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *KDD*, pages 226–235, 2003.

[135] R. Wesley and F. Xu. Incremental computation of common windowed holistic aggregates. 9(12):1221–1232, 2016.

[136] Y. Wu and K. L. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734, 2015.

[137] Y. Xing, S. Zdonik, and J. H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[138] M. Zaharia, T. Das, H. Li, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[139] E. Zgraggen, R. C. Zeleznik, and S. M. Drucker. Panoramicdata: Data analysis through pen and touch. *IEEE TVCG*, 20:2112–2121, 2014.

[140] Y. Zhou, T. Yang, J. Jiang, B. Cui, et al. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD*, pages 741–756, 2018.

[141] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.