

# Ad-Hoc Stream Query Processing

vorgelegt von  
M.Comp.Sc  
Jeyhun Karimov

von der Fakultät IV - Elektrotechnik und Informatik  
der Technischen Universität Berlin  
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften  
- Dr. rer. nat. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Manfred Hauswirth

Gutachter: Prof. Dr. Volker Markl, Technische Universität Berlin

Gutachter: Prof. Dr. Tilmann Rabl, HPI, University of Potsdam

Gutachter: Prof. Dr. Gustavo Alonso, ETH Zurich

Gutachter: Prof. Dr. Stratos Idreos, Harvard University

Tag der wissenschaftlichen Aussprache: 18. Dezember 2019

Berlin 2020



## Zusammenfassung

Eine Vielzahl moderner Anwendungen setzen die Echtzeitverarbeitung großer Datenmengen voraus. Aus diesem Grund haben neuerdings verteilte Systeme zur Verarbeitung von Datenströmen (sog. Datenstrom-Verarbeitungssysteme, abgek. "DSV") eine wichtige Bedeutung als neue Kategorie von Massendaten-Verarbeitungssystemen erlangt. Das zentrale Entwurfsprinzip dieser DSVs ist es, Anfragen, die potenziell unendlich lange auf einem Datenstrom laufen, jeweils Eine nach der Anderen zu verarbeiten (Englisch: "query-at-a-time model"). Das bedeutet, dass jede Anfrage eigenständig vom System optimiert und ausgeführt wird. Allerdings stellen vielen reale Anwendungen nicht nur lang laufende Anfragen auf Datenströmen, sondern auch kurz laufende Spontananfragen. Solche Anwendungen können mehrere Anfragen spontan und zeitgleich erstellen und entfernen. Das bewährte Verfahren, um Spontananfragen zu bearbeiten, zweigt den eingehenden Datenstrom ab und belegt zusätzliche Ressourcen für jede neue Anfrage. Allerdings ist dieses Verfahren ineffizient, weil Spontananfragen damit redundante Berechnungen und Daten-Kopieroperationen verursachen.

In dieser Arbeit legen wir das Fundament für die effiziente Verarbeitung von Spontananfragen auf Datenströmen. Wir schließen in den folgenden drei Schritten die Lücke zwischen verteilter Datenstromanfrage-Verarbeitung und Spontananfrage-Verarbeitung.

Erstens stellen wir ein Benchmark-Framework zur Analyse von modernen DSVs vor. In diesem Framework stellen wir eine neue Definition für die Latenz und den Durchsatz von zustandsbehafteten Operatoren vor. Zudem unterscheiden wir genau zwischen dem zu testenden System und dem Treibersystem, um das offene-Welt Modell, welches den typischen Anwendungsszenarien in der Datenstromverarbeitung entspricht, korrekt zu repräsentieren. Diese strikte Unterscheidung ermöglicht es, die Systemleistung unter realen Bedingungen zu messen. Unsere Lösung ist damit das erste Benchmark-Framework, welches die dauerhaft durchhaltbare Systemleistung von DSVs definiert und testet. Durch eine systematische Analyse aktueller DSVs stellen wir fest, dass aktuelle DSVs außerstande sind, Spontananfragen effizient zu verarbeiten.

Zweitens stellen wir das erste verteilte DSV zur Spontananfrageverarbeitung vor. Wir entwickeln unser Lösungskonzept basierend auf drei Hauptanforderungen: (1) Integration: Spontananfrageverarbeitung soll ein modularer Baustein sein, mit dem Datenstrom-Operatoren wie z.B. Join, Aggregation, und Zeitfenster-Operatoren erweitert werden können; (2) Konsistenz: die Erstellung und Entfernung von Spontananfragen müssen konsistent ausgeführt werden, die Semantik für einmalige Nachrichtenzustellung erhalten, sowie die Korrektheit des Anfrage-Ergebnisses sicherstellen; (3) Leistung: Im Gegensatz zu modernen DSVs sollen DSVs zur Spontananfrageverarbeitung nicht nur den Datendurchsatz, sondern auch den Anfragedurchsatz maximieren. Dies ermöglichen wir durch inkrementelle Kompilation und der Ressourcenteilung zwischen Anfragen.

Drittens stellen wir ein Programmiergerüst zur Verarbeitung von Spontananfragen auf Datenströmen vor. Dieses integriert die dynamische Anfrageverarbeitung und die Nachoptimierung von Anfragen mit der Spontananfrageverarbeitung auf Datenströmen. Unser Lösungsansatz besteht aus einer Schicht zur Anfrageoptimierung und einer Schicht zur Anfrageverarbeitung. Die Optimierungsschicht optimiert

---

periodisch den Anfrageverarbeitungsplan nach, wobei sie zur Laufzeit Joins neu anordnet und vertikal sowie horizontal skaliert, ohne die Verarbeitung anzuhalten. Die Verarbeitungsschicht ermöglicht eine inkrementelle und konsistente Anfrageverarbeitung und unterstützt alle zuvor beschriebenen Eingriffe der Optimierungsschicht in die Anfrageverarbeitung.

Zusammengefasst ergeben unsere zweiten und dritten Lösungskonzepte eine vollständige DSV zur Spontananfrageverarbeitung. Wir verwenden hierzu unseren ersten Beitrag nicht nur zur Bewertung moderner DSVs, sondern auch zur Evaluation unseres DSVs zur Spontananfrageverarbeitung.

## Abstract

Many modern applications require processing large amounts of data in a real-time fashion. As a result, distributed stream processing engines (SPEs) have gained significant attention as an important new class of big data processing systems. The central design principle of these SPEs is to handle queries that potentially run forever on data streams with a query-at-a-time model, i.e., each query is optimized and executed separately. However, in many real applications, not only long-running queries but also many short-running queries are processed on data streams. In these applications, multiple stream queries are created and deleted concurrently, in an ad-hoc manner. The best practice to handle ad-hoc stream queries is to fork input stream and add additional resources for each query. However, this approach leads to redundant computation and data copy.

This thesis lays the foundation for efficient ad-hoc stream query processing. To bridge the gap between stream data processing and ad-hoc query processing, we follow a top-down approach.

First, we propose a benchmarking framework to analyze state-of-the-art SPEs. We provide a definition of latency and throughput for stateful operators. Moreover, we carefully separate the system under test and the driver, to correctly represent the open-world model of typical stream processing deployments. This separation enables us to measure the system performance under realistic conditions. Our solution is the first benchmarking framework to define and test the sustainable performance of SPEs. Throughout our analysis, we realize that the state-of-the-art SPEs are unable to execute stream queries in an ad-hoc manner.

Second, we propose the first ad-hoc stream query processing engine for distributed data processing environments. We develop our solution based on three main requirements: (1) Integration: Ad-hoc query processing should be a composable layer that can extend stream operators, such as join, aggregation, and window operators; (2) Consistency: Ad-hoc query creation and deletion must be performed consistently and ensure exactly-once semantics and correctness; (3) Performance: In contrast to modern SPEs, ad-hoc SPEs should not only maximize data throughput but also query throughput via incremental computation and resource sharing.

Third, we propose an ad-hoc stream join processing framework that integrates dynamic query processing and query re-optimization techniques with ad-hoc stream query processing. Our solution comprises an optimization layer and a stream data processing layer. The optimization layer periodically re-optimizes the query execution plan, performing join reordering and vertical and horizontal scaling at runtime without stopping the execution. The data processing layer enables incremental and consistent query processing, supporting all the actions triggered by the optimizer.

The result of the second and the third contributions forms a complete ad-hoc SPE. We utilize the first contribution not only for benchmarking modern SPEs but also for evaluating the ad-hoc SPE.



## Acknowledgements

First and foremost, I would like to thank my advisors Tilmann Rabl and Volker Markl, who introduced me to the academic world, gave me both, guidance and freedom, to conduct my research and lots of very valuable advice. Also, I want to express my appreciation to Gustavo Alonso and Stratos Idreos for agreeing to review this thesis.

During my time as a Ph.D. student, many people at Database and Information Systems Group of TU Berlin and Intelligent Analytics for Massive Data Group of DFKI accompanied and worked with me to advance my research. Especially, I would like to thank my colleagues Alireza Rezai Mahdiraji, Clemens Lutz, Bonaventura Del Monte, Behrouz Derakhshan, Asterios Katsifodimos, Gabor Gevay, Andreas Kuntz, Kaustubh Beedkar, Jonas Traub, Ankit Chaudhary, Viktor Rosenfeld, and Ariane Ziehn. Furthermore, I would like to thank all my co-authors: Tilmann Rabl, Volker Markl, Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, Alireza Rezaei Mahdiraji, Harry Xuegang Huang, and Christian Thomsen. Also, I would like to thank Phil Bernstein and Walter Cai, who helped me with feedback, advice, and discussions.

I am also very grateful to my previous advisor Murat Ozbayoglu and Bulent Tavli from TOBB ETU. They encouraged me to start a PhD. They were the one to guide me in my very first steps in research.

Finally, I would like to thank my family and friends. My friends Yusuf and Karim, Schubert family (Kristine, Michael, and Philipp), my parents Haqiqat and Oruj, and my wife Sevinj have always supported me in any possible way. Thank you for everything you have done for me.





# Table of Contents

|  |            |
|--|------------|
| Title Page   | <b>i</b>   |
| Zusammenfassung  | <b>iii</b> |
| Abstract   | <b>v</b>   |
| List of Figures  | <b>xv</b>  |
| List of Tables   | <b>xix</b> |
| <b>1 Introduction</b>  | <b>1</b>   |
| 1.1 Motivation   | 1          |
| 1.2 Challenges and Contributions   | 3          |
| 1.2.1 Analyzing SPEs Based on Real-life Streaming Scenarios                      | 3          |
| 1.2.2 Composable Ad-hoc Stream Query Processing                                  | 4          |
| 1.2.3 Enriching Ad-hoc Query Processing Layer with Reoptimization and Dynamicity | 5          |
| 1.2.4 Ad-hoc Query Processing with Traditional DBMS                              | 6          |
| 1.2.5 Distributed vs. Single-node Ad-hoc Query Processing                        | 6          |
| 1.3 Impact of Thesis Contributions   | 7          |
| 1.4 Structure of the Thesis  | 7          |
| <b>2 Background</b>  | <b>9</b>   |
| 2.1 Fundamentals of Stream Data Processing                                       | 9          |
| 2.1.1 Event-time vs. Processing-time Stream Data Processing                      | 9          |
| 2.1.2 Windowed Stream Processing   | 11         |
| 2.1.3 Delivery Semantics   | 11         |
| 2.1.4 Backpressure   | 12         |
| 2.2 Distributed Stream Data Processing Engines                                   | 12         |
| 2.2.1 Apache Storm   | 12         |
| 2.2.2 Apache Spark   | 13         |
| 2.2.3 Apache Flink   | 14         |
| 2.3 Query Optimization   | 14         |
| <b>3 Benchmarking Distributed Stream Data Processing Engines</b>                 | <b>17</b>  |
| 3.1 Introduction   | 19         |
| 3.2 Related Work   | 19         |
| 3.2.1 Batch Processing   | 20         |
| 3.2.2 Stream Processing  | 20         |
| 3.3 Benchmark Design Decisions   | 21         |

## TABLE OF CONTENTS

---

|          |   |           |
|----------|---|-----------|
| 3.3.1    | Simplicity is Key . . . . .   | 21        |
| 3.3.2    | On-the-fly Data Generation vs. Message Brokers . . . . .                  | 21        |
| 3.3.3    | Queues Between Data Generators and SUT Sources . . . . .                  | 22        |
| 3.3.4    | Separation of Driver and the SUT . . . . .                                | 22        |
| 3.4      | Metrics . . . . .   | 23        |
| 3.4.1    | Latency . . . . .   | 23        |
| 3.4.1.1  | Event-time vs. Processing-time Latency . . . . .                          | 24        |
| 3.4.1.2  | Event-time Latency in Windowed Operators . . . . .                        | 24        |
| 3.4.1.3  | Processing-time Latency in Windowed Operators . . . . .                   | 25        |
| 3.4.2    | Throughput . . . . .  | 26        |
| 3.4.2.1  | Sustainable Throughput. . . . .   | 26        |
| 3.5      | Workload Design . . . . .   | 28        |
| 3.5.1    | Dataset . . . . .   | 28        |
| 3.5.2    | Queries . . . . .   | 28        |
| 3.6      | Evaluation . . . . .  | 29        |
| 3.6.1    | System Setup . . . . .  | 29        |
| 3.6.1.1  | Tuning the Systems . . . . .  | 29        |
| 3.6.2    | Performance Evaluation . . . . .  | 30        |
| 3.6.2.1  | Windowed Aggregations . . . . .   | 30        |
| 3.6.2.2  | Windowed Joins . . . . .  | 32        |
| 3.6.2.3  | Unsustainable Throughput . . . . .  | 35        |
| 3.6.2.4  | Queries with Large Windows . . . . .                                      | 35        |
| 3.6.2.5  | Data Skew . . . . .   | 35        |
| 3.6.2.6  | Fluctuating Workloads . . . . .   | 36        |
| 3.6.2.7  | Event-time vs. Processing-time Latency . . . . .                          | 36        |
| 3.6.2.8  | Observing Backpressure . . . . .  | 37        |
| 3.6.2.9  | Throughput Graphs . . . . .   | 37        |
| 3.6.2.10 | Resource Usage Statistics . . . . .                                       | 38        |
| 3.6.2.11 | Multiple Stream Query Execution . . . . .                                 | 39        |
| 3.6.3    | Discussion . . . . .  | 39        |
| 3.7      | Conclusion . . . . .  | 40        |
| <b>4</b> | <b>AStream: Ad-hoc Shared Stream Processing</b> . . . . .                 | <b>41</b> |
| 4.1      | Introduction . . . . .  | 43        |
| 4.1.1    | Motivating Example . . . . .  | 43        |
| 4.1.2    | Ad-hoc Stream Requirements . . . . .                                      | 44        |
| 4.1.2.1  | Integration . . . . .   | 44        |
| 4.1.2.2  | Consistency . . . . .   | 44        |
| 4.1.2.3  | Performance . . . . .   | 44        |
| 4.1.3    | AStream . . . . .   | 44        |
| 4.1.4    | Sharing Limitations in State-of-the-Art Data Processing Systems . . . . . | 45        |
| 4.1.5    | Contributions and Chapter Organization . . . . .                          | 45        |
| 4.2      | System Overview . . . . .   | 46        |
| 4.2.1    | Data Model . . . . .  | 46        |
| 4.2.1.1  | Query-set . . . . .   | 46        |
| 4.2.1.2  | Changelog . . . . .   | 47        |
| 4.3      | Implementation Details . . . . .  | 48        |

|          |   |           |
|----------|---|-----------|
| 4.3.1    | Ad-hoc Operators . . . . .                              | 48        |
| 4.3.1.1  | Shared Session . . . . .                                | 48        |
| 4.3.1.2  | Shared Selection . . . . .                              | 48        |
| 4.3.1.3  | Window Slicing . . . . .                                | 50        |
| 4.3.1.4  | Shared Join . . . . .                                   | 50        |
| 4.3.1.5  | Shared Aggregation . . . . .                            | 50        |
| 4.3.1.6  | Router . . . . .  | 51        |
| 4.3.2    | Optimizations . . . . .                                 | 51        |
| 4.3.2.1  | Incremental Query Processing . . . . .                  | 51        |
| 4.3.2.2  | Data Copy and Shuffling . . . . .                       | 51        |
| 4.3.2.3  | Memory Efficient Dynamic Slice Data Structure . . . . . | 51        |
| 4.3.2.4  | Changelog-set Size . . . . .                            | 51        |
| 4.3.3    | Exactly-once Semantics . . . . .                        | 51        |
| 4.3.4    | QoS . . . . .   | 52        |
| 4.4      | Experiments . . . . .                                   | 52        |
| 4.4.1    | Experimental Design . . . . .                           | 52        |
| 4.4.2    | Generators . . . . .                                    | 53        |
| 4.4.2.1  | Data Generation . . . . .                               | 53        |
| 4.4.2.2  | Selection Predicate Generation . . . . .                | 53        |
| 4.4.2.3  | Join and Aggregation Query Generation . . . . .         | 53        |
| 4.4.3    | Metrics . . . . .                                       | 53        |
| 4.4.4    | Setup . . . . .   | 54        |
| 4.4.4.1  | Workloads . . . . .                                     | 54        |
| 4.4.5    | Workload Scenario 1 . . . . .                           | 54        |
| 4.4.6    | Workload Scenario 2 . . . . .                           | 56        |
| 4.4.7    | Complex Queries . . . . .                               | 57        |
| 4.4.8    | Sharing Overhead . . . . .                              | 58        |
| 4.4.9    | Discussion . . . . .                                    | 60        |
| 4.5      | Integration . . . . .                                   | 60        |
| 4.6      | Related Work . . . . .                                  | 61        |
| 4.6.1    | Query-at-a-time Processing . . . . .                    | 61        |
| 4.6.2    | Stream Multi-query Optimization . . . . .               | 61        |
| 4.6.3    | Adaptive Query Optimization . . . . .                   | 62        |
| 4.6.4    | Batch Ad-hoc Query Processing Systems . . . . .         | 62        |
| 4.6.5    | Stream Query Sharing . . . . .                          | 63        |
| 4.7      | Conclusion . . . . .                                    | 63        |
| <b>5</b> | <b>AJoin: Ad-hoc Stream Joins at Scale</b>              | <b>65</b> |
| 5.1      | Introduction . . . . .                                  | 67        |
| 5.1.1    | Motivation . . . . .                                    | 67        |
| 5.1.2    | Sharing Limitations in Ad-hoc SPEs . . . . .            | 68        |
| 5.1.2.1  | Missed Optimization Potential . . . . .                 | 68        |
| 5.1.2.2  | Dynamicity . . . . .                                    | 69        |
| 5.1.3    | AJoin . . . . .   | 69        |
| 5.1.3.1  | Efficient Distributed Join Architecture . . . . .       | 69        |
| 5.1.3.2  | Dynamic Query Processing . . . . .                      | 69        |
| 5.1.3.3  | AJoin and AStream: Complete Ad-hoc SPE . . . . .        | 69        |

## TABLE OF CONTENTS

---

|          |  |           |
|----------|--|-----------|
| 5.1.4    | Contributions and Chapter Organization . . . . .             | 70        |
| 5.2      | Related Work . . . . .                                       | 70        |
| 5.2.1    | Shared Query Processing . . . . .                            | 70        |
| 5.2.2    | Adaptive Query Processing . . . . .                          | 71        |
| 5.2.3    | Query Optimization . . . . .                                 | 71        |
| 5.2.4    | Mini-batch Query Processing . . . . .                        | 71        |
| 5.3      | System Overview and Example . . . . .                        | 71        |
| 5.3.1    | Data Model . . . . .   | 73        |
| 5.3.1.1  | Bucket . . . . .   | 73        |
| 5.3.1.2  | Changelog . . . . .  | 73        |
| 5.3.2    | Join Operation . . . . .                                     | 73        |
| 5.4      | Optimizer . . . . .  | 74        |
| 5.4.1    | Query Grouping . . . . .                                     | 74        |
| 5.4.2    | Join Reordering . . . . .                                    | 76        |
| 5.4.3    | Vertical and Horizontal Scaling . . . . .                    | 78        |
| 5.5      | Implementation Details . . . . .                             | 78        |
| 5.5.1    | Join Phases . . . . .  | 79        |
| 5.5.1.1  | Bucketing . . . . .  | 79        |
| 5.5.1.2  | Partitioning . . . . .                                       | 79        |
| 5.5.1.3  | Join . . . . .   | 79        |
| 5.5.1.4  | Materialization . . . . .                                    | 80        |
| 5.5.2    | Exactly-once Semantics . . . . .                             | 80        |
| 5.5.3    | Optimizer . . . . .  | 80        |
| 5.6      | Runtime QEP changes . . . . .                                | 81        |
| 5.6.1    | Consistency Protocols . . . . .                              | 81        |
| 5.6.2    | Vertical Scaling . . . . .                                   | 82        |
| 5.6.3    | Horizontal Scaling . . . . .                                 | 82        |
| 5.6.4    | Join Reordering . . . . .                                    | 84        |
| 5.7      | Experiments . . . . .  | 85        |
| 5.7.1    | Experimental Design . . . . .                                | 85        |
| 5.7.2    | Metrics and Data Generation . . . . .                        | 85        |
| 5.7.3    | Workload . . . . .   | 86        |
| 5.7.4    | Setup . . . . .  | 87        |
| 5.7.5    | Scalability . . . . .  | 87        |
| 5.7.6    | Distinct Keys . . . . .                                      | 89        |
| 5.7.7    | Dynamicity . . . . .   | 89        |
| 5.7.7.1  | Latency . . . . .  | 89        |
| 5.7.7.2  | Breakdown . . . . .  | 90        |
| 5.7.7.3  | Throughput . . . . .   | 91        |
| 5.7.7.4  | Impact of Each Component . . . . .                           | 92        |
| 5.7.7.5  | Cost of Sharing . . . . .                                    | 93        |
| 5.7.7.6  | Impact of the Latency Threshold Value . . . . .              | 94        |
| 5.7.7.7  | Impact of the Query Reoptimization Threshold Value . . . . . | 94        |
| 5.8      | Conclusion . . . . .   | 94        |
| <b>6</b> | <b>Additional Contributions</b>                              | <b>95</b> |

|   |           |
|---|-----------|
| <b>7 Conclusion and Future Research</b> | <b>97</b> |
| 7.1 Future Research . . . . .           | <b>97</b> |
| <b>References</b>                       | <b>99</b> |



# List of Figures

|      |   |    |
|------|---|----|
| 1.1  | Use-case for an ad-hoc stream query processing . . . . .  | 2  |
| 1.2  | Scope of this thesis represented as a puzzle . . . . .  | 3  |
| 1.3  | Number of ingested tuples per time unit . . . . .   | 4  |
| 2.1  | Scope of Chapter 2 - Background . . . . .   | 10 |
| 2.2  | Optimizing a 5-way join query with IDP ( $k=3$ ) . . . . .  | 15 |
| 3.1  | Scope of Chapter 3: Performance Analysis of modern SPEs . . . . .   | 18 |
| 3.2  | Benchmark designs to connect the data generator (on the right) and SUT (on the left) . . . . .                                    | 22 |
| 3.3  | Design of our benchmarking framework . . . . .  | 23 |
| 3.4  | An example scenario for coordinated and realistic data generation . . . . .   | 24 |
| 3.5  | End-to-end example of an aggregation query . . . . .  | 25 |
| 3.6  | End-to-end join of two streams . . . . .  | 26 |
| 3.7  | Impact of sustainable and unsustainable throughput to the latency, data generation speed, and data ingestion throughput . . . . . | 27 |
| 3.8  | Query templates used by our workloads . . . . .   | 29 |
| 3.9  | Windowed aggregation latency distributions in time series . . . . .   | 32 |
| 3.10 | Windowed aggregation latency distributions in time series with 90% sustainable throughput . . . . .                               | 33 |
| 3.11 | Windowed join latency distributions in time series with maximum sustainable throughput . . . . .                                  | 34 |
| 3.12 | Windowed join latency distributions in time series with 90% sustainable throughput . . . . .                                      | 34 |
| 3.13 | Event-time latency on stream aggregation workloads with fluctuating data arrival rate . . . . .                                   | 36 |
| 3.14 | Event-time latency on stream join workloads with fluctuating data arrival rate . . . . .  | 36 |
| 3.15 | Comparison between event (top row) and processing-time (bottom row) latency . . . . .   | 37 |
| 3.16 | Comparison between event- and processing-time latency of Spark with unsustainable throughput . . . . .                            | 37 |
| 3.17 | Throughput graphs of systems under test . . . . .   | 37 |
| 3.18 | Network usages of the SUTs in a 4-node cluster . . . . .  | 38 |
| 3.19 | CPU usages of the SUTs in a 4-node cluster . . . . .  | 38 |
| 3.20 | Scheduler delay (top row) vs. throughput (bottom row) in Spark . . . . .  | 38 |
| 3.21 | Query templates used for multiple stream query workloads. PARAM_VAL $n$ is a parameter value given by the user. . . . .           | 39 |
| 4.1  | Scope of Chapter 4: Ad-hoc Shared Stream Processing . . . . .   | 42 |
| 4.2  | Ad-hoc stream queries in online gaming scenarios . . . . .  | 44 |
| 4.3  | AStream architecture . . . . .  | 46 |
| 4.4  | AStream and naive data model . . . . .  | 47 |
| 4.5  | End-to-end ad-hoc query example . . . . .   | 49 |

## LIST OF FIGURES

---

|      |   |    |
|------|---|----|
| 4.6  | Design of the driver for the experimental analysis . . . . .  | 52 |
| 4.7  | Join query template. VAL $n$ is a random number, VAL5 and VAL6 are less than $ fields =5$ . . . . .   | 53 |
| 4.8  | Aggregation query template. VAL $n$ is a random number, VAL4 is less than $ fields =5$ . . . . .  | 53 |
| 4.9  | Two scenarios for ad-hoc query processing environments . . . . .  | 54 |
| 4.10 | Slowest and overall data throughputs for SC1, 4- and 8-node cluster configurations. $n$ q/s $m$ qp indicates $n$ queries per second until $m$ query parallelism . . . . . | 55 |
| 4.11 | AStream performance for SC1 . . . . .   | 56 |
| 4.12 | Query deployment latency, one query per second, up to 20 queries . . . . .  | 56 |
| 4.13 | AStream performance for SC2 . . . . .   | 57 |
| 4.14 | Slowest data throughput (upper), event-time latency (middle), and query count graphs (bottom) for complex ad-hoc queries, with the same $x$ axis values . . . . .         | 58 |
| 4.15 | Input data throughput for different levels of query parallelism in SC1 . . . . .  | 59 |
| 4.16 | Overhead of AStream . . . . .   | 59 |
| 4.17 | Effect of new ad-hoc join queries on existing long-running queries. x-axis shows the number of long-running queries and the workload scenario . . . . .                   | 60 |
| 4.18 | Scalability with the number of queries . . . . .  | 60 |
| 5.1  | Scope of Chapter 5 . . . . .  | 66 |
| 5.2  | Ad-hoc stream join queries. $T_{iC}$ and $T_{iD}$ show creation and deletion times of $i$ th query, respectively. . . . .   | 68 |
| 5.3  | AJoin and AStream: Complete Ad-hoc SPE . . . . .  | 69 |
| 5.4  | AJoin architecture . . . . .  | 72 |
| 5.5  | Executing Q1, Q2, and Q3 in AJoin between time $T_{4C}$ and $T_{1D}$ . . . . .  | 72 |
| 5.6  | Optimization process . . . . .  | 73 |
| 5.7  | Cost of shared and separate join execution for Q4 and Q5. Q.S means the stream S of the query Q. . . . .  | 75 |
| 5.8  | Calculation of query groups . . . . .   | 76 |
| 5.9  | Join reordering . . . . .   | 77 |
| 5.10 | Example partitioning of the bucket described in Figure 5.5e . . . . .   | 79 |
| 5.11 | Ad-hoc join example. The join operation is performed between $T_{1C}$ and $T_{2D}$ . . . . .  | 80 |
| 5.12 | 3-phase atomic protocol . . . . .   | 81 |
| 5.13 | Scale up operation . . . . .  | 82 |
| 5.14 | Partition function change operation. PF refers to the partitioning function . . . . .   | 83 |
| 5.15 | Join reordering . . . . .   | 84 |
| 5.16 | Formal definition of join reordering . . . . .  | 85 |
| 5.17 | Query template used in experiments . . . . .  | 86 |
| 5.18 | Two scenarios for ad-hoc query processing environments . . . . .  | 87 |
| 5.19 | Overall data throughput of AJoin, AStream, Spark, and Flink . . . . .   | 87 |
| 5.20 | Buffer space used for tuples and indexes inside a 1-second bucket . . . . .   | 88 |
| 5.21 | The effect of the number of distinct keys in stream sources and the selectivity of selection operators on the performance of AJoin, AStream, Spark, and Flink . . . . .   | 89 |
| 5.22 | Average event-time latency of stream tuples with min and max boundaries for SC1 . . . . .   | 90 |
| 5.23 | Deployment latency for SC1 . . . . .  | 90 |
| 5.24 | Breakdown of AJoin components in terms of percentage for SC1 . . . . .  | 91 |
| 5.25 | Throughput measurements for AJoin, AStream, Spark, and Flink . . . . .  | 92 |
| 5.26 | Impact of AJoin components in terms of percentage . . . . .   | 93 |
| 5.27 | Cost of data sharing and the impact of the latency threshold value with 3-way join queries . . . . .  | 93 |



5.28 Impact of the threshold value of query reoptimization on the performance of AJoin . . . . 94



# List of Tables

|     |  |           |
|-----|--|-----------|
| 3.1 | Sustainable throughput for windowed aggregations . . . . .   | <b>30</b> |
| 3.2 | Latency statistics, avg, min, max, and quantiles (90, 95, 99) in seconds for windowed aggregations . . . . . | <b>31</b> |
| 3.3 | Sustainable throughput for windowed joins . . . . .  | <b>33</b> |
| 3.4 | Latency statistics, avg, min, max and quantiles (90, 95, 99) in seconds for windowed joins                   | <b>34</b> |



# 1

## Introduction

### This Chapter contains:

|       |  |   |
|-------|--|---|
| 1.1   | Motivation . . . . .   | 1 |
| 1.2   | Challenges and Contributions . . . . .   | 3 |
| 1.2.1 | Analyzing SPEs Based on Real-life Streaming Scenarios . . . . .                      | 3 |
| 1.2.2 | Composable Ad-hoc Stream Query Processing . . . . .                                  | 4 |
| 1.2.3 | Enriching Ad-hoc Query Processing Layer with Reoptimization and Dynamicity . . . . . | 5 |
| 1.2.4 | Ad-hoc Query Processing with Traditional DBMS . . . . .                              | 6 |
| 1.2.5 | Distributed vs. Single-node Ad-hoc Query Processing . . . . .                        | 6 |
| 1.3   | Impact of Thesis Contributions . . . . .   | 7 |
| 1.4   | Structure of the Thesis . . . . .  | 7 |

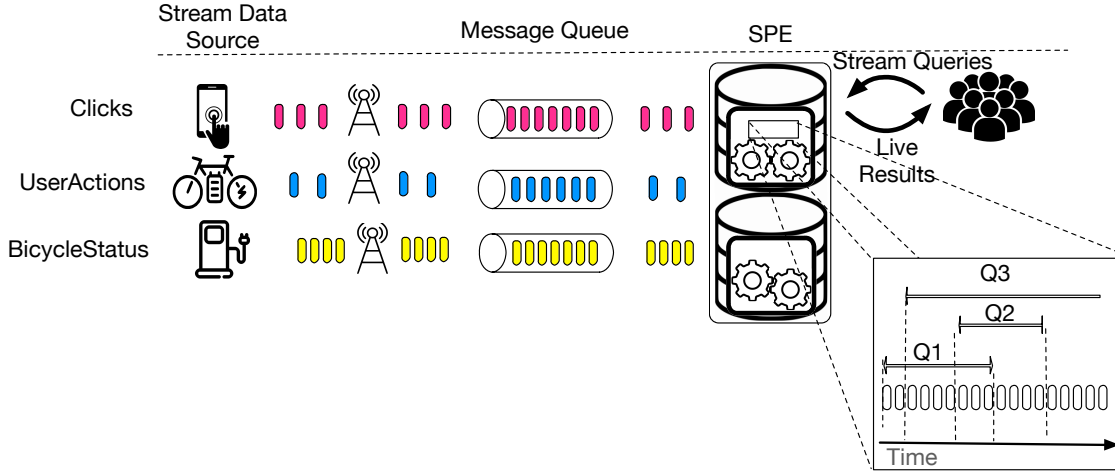
---

### 1.1 Motivation

The goal of streaming applications is to query continuous data streams with low-latency, i.e., within a small time period from the time of receiving the data until the query result is computed. Due to increasing data volume (e.g., in telecommunications, finance, and internet of things (IoT)), streaming applications have become ubiquitous. For example, the IoT market is expected to grow from an installed base of 15.4 billion devices in 2015 to 30.7 billion devices in 2020 and 75.4 billion in 2025 [1]. Streaming applications process large volumes of data generated from such devices that create a continuous stream of information.

Various academic and industrial communities have developed programming models for distributed stream data processing. Although the proposed models differ both at the language level and at the system level, they represent streaming applications as a data flow graph of data streams and operators. A vertex of the graph represents stream operators, and an edge denotes a data stream. The stream operators implement transformations on a data stream (e.g., filtering, aggregating, joining). After all transformations are performed via stream operators, the resulting data tuples are pushed to external output channels. Apache Storm [2], Apache Spark [3, 4], and Apache Flink [5] are examples of distributed stream processing engines (SPEs) with significant adoption in industry and the research community.

Cloud computing has gained significant attention as an emerging paradigm for developing and delivering computing services. Derived from mainframe computing, it has advanced to an on-demand and virtualized



**Figure 1.1:** Use-case for an ad-hoc stream query processing

delivery of computing power. As a result, one does not need to make large upfront investments in hardware and spend time on managing that hardware.

With the advance of cloud computing, several service models have been developed, such as software as a service (SaaS), infrastructure as a service, and platform as a service. These models allow a third-party provider to host applications, infrastructure, and platforms, and to make them available to customers over the Internet. Meanwhile, multi-tenant systems were developed or existing systems extended to support multi-tenancy. Multi-tenancy is an instance of software and its supporting infrastructure serving multiple customers, as an extension of the SaaS model. The main idea of multi-tenancy is that sharing resources among multiple tenants leads to lower costs.

Although multi-tenancy (serving multiple concurrent user queries) has been extensively adopted by relational database management systems and batch data processing systems, adapting multi-tenancy for stream processing workloads is challenging. Unlike batch data processing systems, in SPEs ad-hoc queries target potentially different data tuples, depending on query creation and deletion. An **ad-hoc stream query** is a query that is created and deleted on demand. An **ad-hoc SPE** is a system that is able to execute concurrent ad-hoc stream queries. The goal of this thesis is to bridge the gap between ad-hoc query processing and distributed stream data processing.

Figure 1.1 shows a use-case for ad-hoc stream query processing. Electrical bicycle sharing is widely spread to promote green transportation [6] [7]. In the example scenario in Figure 1.1 electrical bicycles (🚲) regularly send information (🔴) about user actions, such as (un)locking the bicycle to start (finish) a journey. In order to use an electrical bicycle, the user installs the related app on her mobile phone and makes a payment. The mobile phone (📱) also periodically emits click stream (🔵). Meanwhile, electric charging stations (🔌) dispatch information (🟡) about the technical status of charging bicycles. In industrial setups the transmitted information is saved at message queues [8]. Users create and delete ad-hoc stream queries (Q1 and Q2) or submit long-running stream queries (Q3). For example, a user might want to enrich Clicks stream with UserActions stream emitted at rush hours. Another user performs a similar computation for customers older than 25 years. After the rush hour is finished, both queries are deleted. Assuming that the queries share at least one stream data source, the main challenge is to share computation, minimize data copy, and maximize the amount of served stream queries.

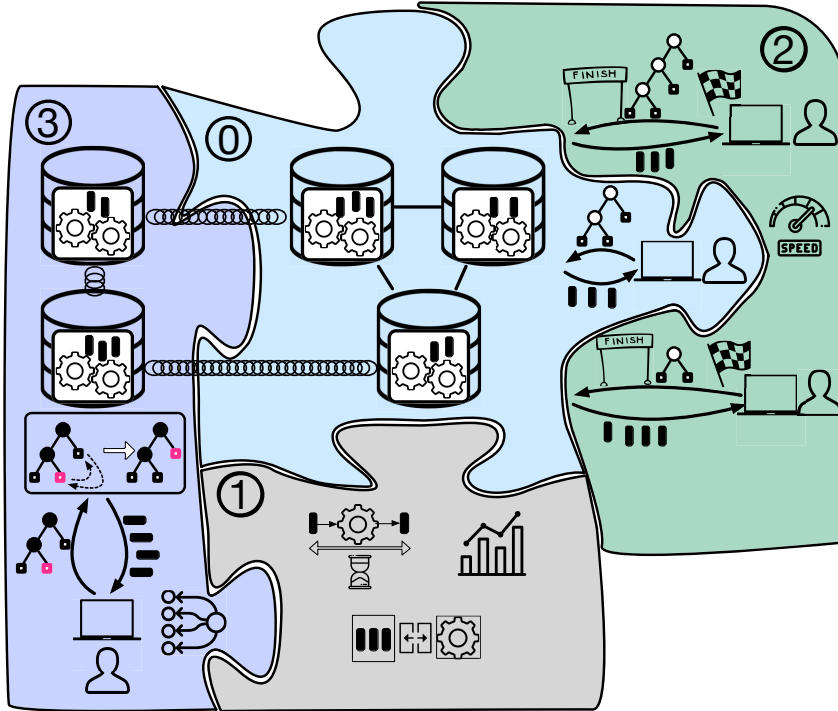


Figure 1.2: Scope of this thesis represented as a puzzle

## 1.2 Challenges and Contributions

In this section, we explain the challenges and state our contributions. We summarize the scope and provide a high level overview of this thesis in Figure 1.2. We build this thesis on top of the piece ① of the puzzle in the figure. The piece ① includes a modern SPE that is optimized for single long-running stream queries. The SPE ingests input tuples (📦) and processes them in a distributed manner (🏠). A user (👤) submits (↩️) the query (📊) to the system and receives (↪️) output results. We describe each challenge and our contribution by stating the challenge, providing an example scenario, stating short description of our contribution, and explaining the related piece of the puzzle from Figure 1.2.

### 1.2.1 Analyzing SPEs Based on Real-life Streaming Scenarios

A thorough analysis of SPEs is essential to discover potential limitations in stream data processing. We have realized that there are numerous challenges in benchmarking SPEs based on real-life streaming scenarios. The piece ① in Figure 1.2 indicates our contributions related to analysis and benchmarking of SPEs.

**Challenge 1: Accurate and objective metric calculations.** The metric calculation should have minimum impact on the performance of the system under test (SUT). Besides, the calculation semantics must be the same among all SUTs to ensure fairness. Designing a benchmarking framework to ensure accurate and objective metric calculations for all SPEs is a challenge.

**Example.** Modern SPEs feature a set of performance metrics, such as latency, throughput, and resource usage, to monitor applications. Relying on these metrics while analyzing different SPEs might lead to incorrect results because of different metric calculation semantics, or even because of different engine design semantics. For example, Flink measures the latency using latency markers, and Spark measures the runtime of each mini-batch computation. In addition, none of the modern SPEs consider

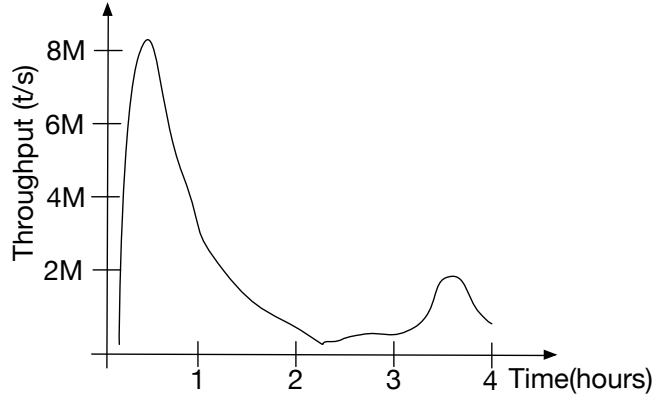


Figure 1.3: Number of ingested tuples per time unit

the additional time an input tuple spent outside the SPE (e.g., time spent in the message queue in Figure 1.1).

**Contribution 1.** We perform a complete separation of the SUT and the test driver (🔧🔧🔧). This enables us to *i*) isolate the data computation and metric calculation and *ii*) unify the benchmarking process among all SUTs objectively.

**Challenge 2: Latency of stateful stream operators.** Latency is one of the main performance metrics for SPEs. Yet, the definition of latency is missing for stateful stream operators, such as windowed aggregations and joins.

**Example.** Assume that a user in Figure 1.1 executes a windowed aggregation query on the Clicks stream to calculate the average number of clicks for each user. Each window computes a single aggregate value and outputs it. Computing the latency of the outputted tuple is nontrivial because potentially many input tuples inside the window contribute to the value of the output tuple.

**Contribution 2.** We provide the definition of latency (🔧🔧🔧) for stateful stream operators. We apply the proposed definition to various use-cases, such as windowed aggregations and joins.

**Challenge 3: Throughput measurement.** Current stream benchmarks either adopt throughput measurement techniques from batch data processing systems (overall number of tuples divided by the runtime) or utilize min, max, or average throughput. However, none of these metrics measure the throughput that can be achieved in a production setting.

**Example.** Figure 1.3 shows the number of ingested tuples per time unit. The maximum throughput (8.1M t/s) can be interpreted in two ways. One way is that, the system successfully ingests and processes input tuples during the high workload. Then, the workload decreases leading to a lesser number of processed tuples. Another scenario is that the system buffers too many tuples because it cannot keep up with the data arrival rate. Thus, the SUT realizes the backpressure, and all upstream operators slow down their data ingestion rate. Looking at the figure, it is difficult to identify which of these scenarios happened.

**Contribution 3.** We measure the maximum sustainable throughput of an SPE (🏠), i.e., the highest load of event traffic that a system can handle. Our benchmarking framework handles system specific features like backpressure to measure the maximum sustainable throughput.

## 1.2.2 Composable Ad-hoc Stream Query Processing

Analyzing SPEs is an important approach to identify possible limitations. During our analysis, one of our takeaways (among others) was that state-of-the-art SPEs are not able to process ad-hoc stream



queries, which is the main query processing model for multi-tenant cloud architectures. Thus, we provide a foundation for ad-hoc stream query processing. We indicate this work as the piece ② in Figure 1.2



**Challenge 4: Composability.** Composability is a system design principle in which components of the system can be assembled and dismantled to satisfy specific user requirements. The challenge is to *i*) avoid re-implementing an existing set of SPE features, such as out-of-order stream processing, event-time processing, and fault tolerance, and *ii*) design the ad-hoc query processing layer to be easily integrable with any SPE.

**Example.** Assume that the service owner decides to replace the existing SPE with another one because the latter one provides a new set of required features or its performance is higher than the former. Usually, significant software development effort is required if the new engine does not support ad-hoc stream queries by default.

**Contribution 4.** We design the ad-hoc query processing tier to be a composable layer of an underlying SPE. The idea is that the piece ② (Figure 1.2) is pluggable not only to the piece ① but also to other pieces of a similar type. The composable layer supports ad-hoc stream query processing for stream operators, such as filter, join, aggregation, and window operators.





**Challenge 5: Consistency.** In the presence of multiple ad-hoc stream queries, adding and removing queries consistently while ensuring the correctness of results is a challenge.

**Example.** In Figure 1.1, Q1 and Q2 are ad-hoc stream queries, while Q3 is a long-running stream query. An ad-hoc SPE must ensure that all the queries are created and deleted in a consistent manner. For example, in Figure 1.1 Q1 must process only tuples between creation and deletion time of Q1.

**Contribution 5.** We provide consistent query creation () and deletion (), and ensure the correctness of results for all running ad-hoc queries.

**Challenge 6: Performance.** A main objective of modern SPEs is to maximize input data throughput and minimize data latency. In the presence of ad-hoc stream queries, the challenge is to maximize the query throughput (number of created and deleted ad-hoc queries per time unit) in addition to the aforementioned objectives.

**Example.** In Figure 1.1 the ad-hoc SPE serves three parallel user queries. The objective is to maximize the number of users served at the same time. Q3 shares data with Q1 and Q2. Also, Q1 shares data with Q2. Depending on the queries submitted, Q1, Q2, and Q3 might also share computation. In this case, the system computes the computation shared among queries and reuses it.


**Contribution 6.** We provide a set of incremental computation and optimization techniques to achieve high performance. Also, we provide a rule-based optimization technique to determine whether sharing data and computation is beneficial. The piece ② of the puzzle shows that the ad-hoc () query processing layer enables us to serve multiple user queries ( ) and ensures high performance ()

### 1.2.3 Enriching Ad-hoc Query Processing Layer with Reoptimization and Dynamicity


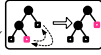
Stream data processing with a single-query workload is challenging [9], as it is generally accepted that stream workloads are unpredictable [10]. With multiple ad-hoc stream queries, the workload is even more unpredictable, and processing ad-hoc stream queries is more challenging. Thus, we enrich ad-hoc stream query processing with dynamic and incremental query processing techniques. The piece ③ in Figure 1.2 shows the high-level overview of this part of the thesis.

**Challenge 7: Lack of Dynamicity.** State-of-the-art SPEs adopt static query execution plans (QEPs). However, in the presence of ad-hoc stream queries and fluctuating query and data workloads, a static QEP might be suboptimal. Also, state-of-the-art stream optimizers adopt rule-based strategies that

optimize input queries at compile-time. However, due to the nature of ad-hoc stream query processing, compile-time and rule-based query optimization often lead to suboptimal QEPs.


**Example.** Assume that  $Q1 = \text{Clicks} \bowtie \text{UserActions}$  ( $\text{Window} = 5\text{sec}$ ) in Figure 1.1. All the resources (two ) are allocated to the SPE operators executing Q1. Then, Q2 and Q3 are created. For all the queries to run smoothly, rescheduling might be required (scale down or scale in) for Q1. A similar condition might arise for scaling out or scaling up.

Also, assume that  $Q2 = \text{UserActions} \bowtie \text{BicycleStatus}$  ( $\text{Window} = 1\text{sec}$ ) and  $Q3 = \text{Clicks} \bowtie \text{UserActions} \bowtie \text{BicycleStatus}$  ( $\text{Window} = 2\text{sec}$ ) in Figure 1.1. When Q3 is created, the optimizer assigns  $(\text{Clicks} \bowtie \text{UserActions}) \bowtie \text{BicycleStatus}$  QEP to it in order to benefit from the sharing opportunity  $(\text{Clicks} \bowtie \text{UserActions})$  with Q1. After some time, Q1 is deleted, and Q2 is created. Then, the QEP of Q3 might be suboptimal.

**Contribution 7.** We provide dynamicity at two layers. Dynamicity at the optimization layer means that the optimization layer performs regular reoptimization () , such as join reordering and horizontal and vertical scaling. Dynamicity at the data processing layer means that the layer is able to perform all the actions triggered by the optimizer at runtime, without stopping the QEP ().

**Challenge 8: Missed Optimization Potential.** First, to the best of our knowledge, there is no ad-hoc SPE providing ad-hoc stream QEP optimization. Second, join operator structure is prone to be the bottleneck to the whole QEP, because the computation distribution of a join operation is rather skewed among different stream operators.

**Example.** The source operator of the SPE is responsible for pulling stream tuples from the message queue. To execute a join query, e.g., Q1, the join operator buffers stream tuples in a window, finds matching tuples and builds resulting tuples by assembling the matching tuples. The join operator also implements all the functionalities of a windowing operator. The sink operator pushes the resulting tuples to output channels provided by the user. Because most of the computation is performed in the join operator, it can easily become a bottleneck.

**Contribution 8.** We provide an incremental optimization technique for ad-hoc queries. Also, we redesign the join operator structure to exploit the pipeline parallelism ().

### 1.2.4 Ad-hoc Query Processing with Traditional DBMS

Traditional DBMSs are designed to handle ad-hoc queries by default. They perform scan and computation sharing to execute multiple ad-hoc queries [11]. It is also possible to reuse all sophisticated algorithms and techniques that traditional DBMSs adopt to handle ad-hoc queries for streaming workloads [12] [13].

The ad-hoc query processing techniques discussed in this thesis (Chapters 4 and 5) can be applied to any data processing system that can handle streaming workloads and are not specific to SPEs. The main contributions of this thesis are tightly coupled with streaming workloads, not with SPEs.

### 1.2.5 Distributed vs. Single-node Ad-hoc Query Processing

The contributions of this thesis are designed for distributed data processing environments. We design our solution to be a composable layer over existing SPEs (Contribution 4). The proposed ad-hoc SPE (Chapters 4 and 5) does not utilize any centralized computing structure. Our solution adopts dynamicity and progressive optimization (Contribution 7), which are more essential in distributed environments. To ensure the correctness of query results, our solution utilizes distributed consistency protocols. Also, the ad-hoc SPE exploits pipeline-parallelism (Contribution 8). In a single-node environment, however, task-fusion is more beneficial [14]. Other contributions of our work are not specific to distributed data processing environments.

## 1.3 Impact of Thesis Contributions

**Research Publications.** The primary results of this thesis have been published in the following peer-reviewed publications at international top-tier venues:

1. **Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, Volker Markl:** *Benchmarking Distributed Stream Data Processing Systems* In Proceedings of IEEE International Conference on Data Engineering (ICDE), 2018.
2. **Jeyhun Karimov:** *Stream Benchmarks* In Proceedings of Encyclopedia of Big Data Technologies, 2018.
3. **Jeyhun Karimov, Tilmann Rabl, Volker Markl:** *AStream: Ad-hoc Shared Stream Processing* In Proceedings of ACM International Conference on Management of Data (SIGMOD), 2019.
4. **Jeyhun Karimov, Tilmann Rabl, Volker Markl:** *AJoin: Ad-hoc Stream Joins at Scale* In Proceedings of VLDB Endowment, 2019.

**Research Talks.** Parts of the work on benchmarking distributed stream data processing systems in Chapter 3 have been presented at 4th International Workshop on Performance Analysis of Big data Systems (PABS) [15]. Also, parts of the work on ad-hoc stream query processing in Chapter 4 have been presented at the FlinkForward Berlin conference 2019 [16]. FlinkForward is a conference for Apache Flink and stream processing communities, consisting of industrial experiences, best practices, and research sessions. We believe that our talk at FlinkForward will promote the adoption of our research contributions in the industry.

**Summary.** Our contributions present a realistic way of performance analysis for SPEs, which is essential for all streaming systems. Also, the contributions made in this thesis provide a foundation for ad-hoc stream query processing. Our examples show great potential with respect to shared resource utilization, dynamicity, and query (re)optimization. We believe that our contributions will lead to a new generation of SPEs to support various industrial use-cases on ad-hoc stream query processing.

## 1.4 Structure of the Thesis

**Chapter 2:** Chapter 2 provides background information for the subsequent chapters. We explain the fundamental concepts of stream data processing. Also, we explain the modern SPEs in detail, including their computation semantics and differences and similarities between them. In addition, we present existing query optimization techniques, which we adopt and enhance to support ad-hoc query optimization.

**Chapter 3:** Chapter 3 lays the basis to explore modern SPEs and analyze their strengths and limitations. We also show drawbacks of existing performance evaluation techniques for SPEs. One outcome of this work is that modern SPEs are not able to execute ad-hoc stream queries.

**Chapter 4:** Chapter 4 presents the fundamentals of ad-hoc shared stream query processing. We propose the first ad-hoc SPE and design our solution based on three principles: ease of integration, consistency, and performance.

**Chapter 5:** Chapter 5 bridges the gap between ad-hoc stream query processing, incremental query processing, and dynamic query processing. We enhance existing ad-hoc stream query processing techniques with cost-based ad-hoc query (re)optimization techniques and dynamicity.

**Chapter 6:** Chapter 6 lists additional related research contributions of the author. These contributions are not covered in the above chapters, but have been accomplished while working on this thesis.

**Chapter 7:** Chapter 7 concludes the thesis and provides an outlook to future work.

# 2

## Background

**This Chapter contains:**

|       |   |    |
|-------|---|----|
| 2.1   | Fundamentals of Stream Data Processing . . . . .                | 9  |
| 2.1.1 | Event-time vs. Processing-time Stream Data Processing . . . . . | 9  |
| 2.1.2 | Windowed Stream Processing . . . . .                            | 11 |
| 2.1.3 | Delivery Semantics . . . . .                                    | 11 |
| 2.1.4 | Backpressure . . . . .  | 12 |
| 2.2   | Distributed Stream Data Processing Engines . . . . .            | 12 |
| 2.2.1 | Apache Storm . . . . .  | 12 |
| 2.2.2 | Apache Spark . . . . .  | 13 |
| 2.2.3 | Apache Flink . . . . .  | 14 |
| 2.3   | Query Optimization . . . . .                                    | 14 |

---

### 2.1 Fundamentals of Stream Data Processing

The main goal of stream data processing applications is to process high volume, continuous feeds from live data sources, analyze these feeds, and produce near real-time insights with low latency. Dataflow [17] and MilWheel [18] can be regarded as one of the first SPEs used in production at web scale. The Dataflow model is a data processing paradigm proposed by Google. The main idea of this model is to deal with sophisticated requirements, such as event-time ordering, event-time windowing, and low latency. The model avoids to groom unbounded datasets into finite pools of information, wait until the pools are complete, and process resulting pools as a batch. Instead, the Dataflow model assumes that we will never know if or when we have seen all of our data. The model provides principled abstractions that allow the practitioner to select the appropriate tradeoffs along the axes of interest: correctness, latency, and cost. Modern SPEs, such as Apache Flink [5] adopt the Dataflow model in their implementation.

#### 2.1.1 Event-time vs. Processing-time Stream Data Processing

Event-time is the time at which the event itself actually occurred. This time is typically embedded within stream tuples as a separate attribute and is extracted from the tuple inside an SPE. Event-time data

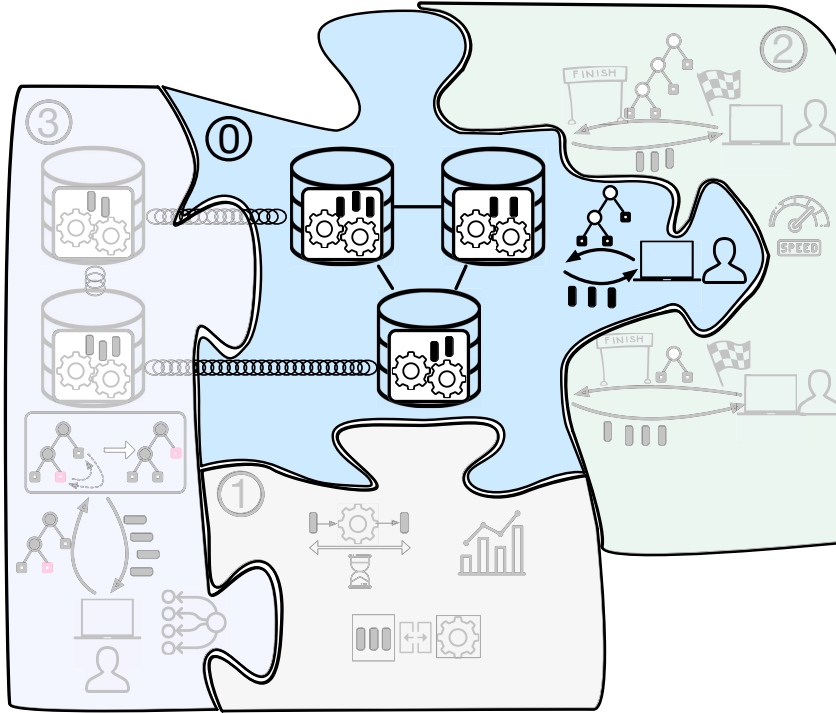


Figure 2.1: Scope of Chapter 2 - Background

processing allows time-based operations, such as time windows, to be computed deterministically. For example, a 10-second event-time window contains all stream tuples with event timestamp that fall into the respective 10 second time slot. The semantics of event-time data processing is agnostic to the arrival time or the arrival order of stream tuples. In contrast to the processing-time stream data processing, in the event-time stream data processing, the progress of time depends on the data, not on any wall clock. Event-time data processing is helpful especially when a failure occurs and some portion of the stream is replayed, or when events arrive late or in an out-of-order manner.

Processing-time is the system time of the machine executing the respective operation. The main difference between the processing-time and event-time is that event-time for a given event never changes, but processing-time changes depending on the system clock of the machines that run the respective operator. Stream operators implementing processing-time semantics, such as time windows, use the system clock of the machines that run the respective operator. For example, a 10-second processing-time window includes all stream tuples that arrive at the system, which is measured by the system clock, in the respective 10-seconds time slot.

Compared with event-time, processing-time is a simpler notion of time. With processing-time, no coordination is needed between streams and machines. Thus, stream operators implementing processing-time semantics provide a better performance and a lower latency than the ones with event-time semantics. The main limitation of processing-time semantics is lack of determinism. The main reason is that processing-time is highly susceptible to the speed at which tuples arrive at the system, to the speed at which tuples flow between stream operators, to parallelism, scheduling, etc.

Ideally, when the time domain skew between processing-time and event-time is always zero, all events can be processed immediately as they happen. However, in reality, there is a skew between the two time domains, as there can be delays due to network, user activity, etc. Therefore, event-time stream data processing might lead to a certain latency, especially when waiting a certain time for late events and out-of-order events.

### 2.1.2 Windowed Stream Processing

One of the fundamental characteristics of streaming applications is the on-the-fly nature of the computation. That is, streaming applications do not require access to disk-resident data. Instead, the applications discretize continuous data and store the most recent history of streams in memory to perform necessary tasks. This discretized data is often managed using windows. All modern SPEs feature some form of windowing functionality. Windowing enables stream processing applications to perform blocking relational operations, such as windowed aggregations, windowed joins, and any user-defined operation.

Usually, stream windows are assigned a window function that is executed in parallel. A window operator encompasses all windowing features and computations in a single operator. Each parallel instance of the window operator is deployed on a specific partition of stream data. We call this window type *keyed windows*. Non-keyed windows or global windows collect all the stream tuples inside a single global window. Although the global windows are necessary for some use-cases, their non-parallel and centralized execution semantics might be a bottleneck for some workloads.

A window assignment operator inside an SPE defines how stream tuples are assigned to windows. A window is configured by its length and slide. The length parameter controls the duration of a window. The slide parameter controls how frequently a window is started. As a result, windows can be overlapping (tuples are assigned to multiple windows) if the slide is smaller than the length. Modern SPEs are shipped with pre-defined window assigners for the most common use-cases, such as tumbling windows, sliding windows, and session windows, along with generic user-defined window assigners. Windows can be constructed based on time, count, or some user-defined logic.

A tumbling window assigner assigns each stream tuple to a single window with a fixed size. Time-based tumbling windows collect tuples from upstream operators until the closing time of the window is reached. Then, the window is closed, and processing is performed on the stored data. Afterwards, the outcome of the computation is sent to the downstream operators, and all the data tuples inside the window are evicted.

Sliding windows can be regarded as a superset of tumbling windows. These windows continuously maintain the most recent tuples. Each stream tuple is assigned to one or more windows depending on the length and the slide of a window. When the window is full, the sliding window evicts only the oldest tuples instead of all the tuples inside the window.

In session windows, input stream tuples are assigned to windows based on their frequency. Unlike sliding and tumbling windows, session windows do not overlap and have a dynamic length, defined at runtime. A session window is regarded as full when the assigner does not receive stream tuples for a certain period of time. When the session window is full, all the elements are processed. Afterwards, they are removed from the window.

### 2.1.3 Delivery Semantics

In a distributed data processing environment, the computers that make up an SPE can always fail independently of one another. Depending on the action the SPE takes to handle such a failure, the resulting delivery semantics will differ.

Assume that a producer receives an acknowledgment message from its downstream consumer for every message that has been sent. We refer to any two operators that exchange data, whether these operators are within a single system or not, as a producer and consumer. In case of a failure, such as a network failure, the producer acknowledgment times out or leads to an error. In this case, the producer might retry sending the message several times, such that at least one attempt succeeds. This might result in duplicated messages on the consumer side; however, no message is lost. This delivery option is called at-least-once delivery semantics.

## 2. Background

---

When the failure occurs or acknowledgment time exceeds, the producer might avoid sending multiple messages to the consumer. In this case, the message is delivered zero or one times; meaning messages may be lost. This delivery option is called at-most-once delivery semantics.

In exactly-once-semantics, even if the producer attempts to send a message to its consumer multiple times, the message is delivered to the consumer exactly once. This delivery semantics is the most desired one because it guarantees that the message can neither be lost nor duplicated, and results are correct.

### 2.1.4 Backpressure

In stream data processing scenarios the input data throughput might fluctuate over time. Source operator instances (the most upstream operator instances of an SPE) periodically pull input data from external sources, such as message queues. When an SPE receives data at a higher rate than it can process (e.g., during a temporary load spike), it initiates backpressure. Backpressure can occur due to various reasons. For example, garbage collection stalls or resource bottlenecks, such as CPU, memory, network bottlenecks, or fluctuating input data throughput might cause an operator to compute at a lower speed than the output rate of its upstream operator.

There are three main ways to handle backpressure. The first way is that the SPE, which cannot keep up with the input data rate, drops data tuples. Although this is a reasonable solution for a wide variety of use-cases, such as approximate computing applications, for some stream use-cases it is not acceptable. The second way is that the SPE ingests and accumulates all input data, although it cannot sustain the workload. However, this will eventually result in a shortage of resources, such as lack of memory. The third way is that the SPE automatically adjusts the data flow rate throughout all stream operators. The SPE initially detects the backpressure. Then it takes necessary actions to handle it. We discuss specific backpressure implementations, along with other details, for the state-of-the-art SPEs (Apache Storm, Apache Spark, and Apache Flink) below.

## 2.2 Distributed Stream Data Processing Engines

In this section, we provide background information about the state-of-the-art SPEs and their features used in this thesis. We analyze Apache Storm, Apache Spark, and Apache Flink as they are the most mature and accepted ones in both academia and industry.

Unlike stream data processing, which performs real-time data analysis, batch data processing collects newly arriving data elements in groups and processes the whole group at a future time. In other words, stream processing processes data as they come in and spreads the processing over time, while batch processing lets the data build up and try to process them at once. While some systems, such as Spark Streaming [3], inherit a batch data processing architecture to execute streaming workloads, some systems, such as Flink [5], adopt a stream data processing architecture and implement batch data processing as a special case of stream data processing.

### 2.2.1 Apache Storm

Apache Storm is a distributed stream processing framework, which was open sourced after being acquired by Twitter [19]. Storm operates on tuple streams and provides tuple-at-a-time stream processing. It supports an at-least-once processing semantics and guarantees all tuples to be processed. In case of a failure, events are replayed. Storm also supports exactly-once processing semantics with its Trident abstraction [20].

Stream processing programs of Storm are represented by a computational topology, which consists of spouts and bolts. Spouts are source operators, and bolts are processing and sink operators. A Storm



topology forms a directed acyclic graph (DAG), where the edges are tuple streams and vertices are operators (bolts and spouts). When a spout or bolt emits a tuple, the bolts that are subscribed to this spout or bolt receive input.

Storm’s lower level APIs provide little support for automatic memory and state management. Therefore, choosing the right data structure for state management and utilizing memory efficiently by making computations incrementally is up to the user. Storm supports caching and batching the state transition. However, the efficiency of these operations degrades as the size of the state grows.

Storm has built-in support for windowing. Although the information of expired, newly arrived, and total tuples within a window is provided through APIs, the incremental state management is not transparent to the users. Trident, on the other hand, has built-in support for partitioned windowed joins and aggregations. Storm supports sliding and tumbling windows with processing- and event-time semantics.

Any worker process in the Storm topology sends an acknowledgment to its upstream executor for a processed tuple. In case of failure, Storm sends the tuple again. One of the downsides of Storm’s use of acknowledgments is that the tuples can only be acknowledged once a window operator completely flushes them out of a window. This can be an issue on windows with large length and a small slide.

Storm supports backpressure although the feature is not mature yet [21]. This was confirmed throughout our experiments as well. Storm uses an extra backpressure thread inside the system. Once the receiver queue of an operator is full, the backpressure thread is notified. This way Storm can notify all workers that the system is overloaded. Due to its high complexity and centralized nature, Storm’s backpressure feature can stall the system and, therefore, it is not enabled by default (version 1.0.2).

### 2.2.2 Apache Spark

Apache Spark is an open source big data processing engine, originally developed at the University of California, Berkeley [22]. Unlike Storm and Flink, which support tuple-at-a-time, Spark Streaming inherits its architecture from batch processing, which supports processing tuples in micro-batches. The Resilient Distributed Dataset (RDD) is a fault-tolerant abstraction of Spark, which enables in-memory, parallel computation in distributed cluster environments [23].

Spark supports stage-oriented scheduling. Initially, it computes a DAG of stages for each submitted job. Then it keeps track of materialized RDDs and outputs from each stage and finally finds a minimal schedule. Unlike Flink and Storm, which also work based on DAG execution graphs, Spark’s computing unit in a graph (edge) is a data set rather than streaming tuples, and each vertex in a graph is a stage rather than individual operators.

Spark has improved its memory management significantly in the recent releases (we use Spark v2.0.1 in our experiments). The system shares the memory between execution and storage. This unified memory management supports dynamic memory management between the two modules. Moreover, Spark supports dynamic memory management throughout the tasks and within operators of each task.

Spark has built-in support for windowed calculations. With its DStream abstraction [3], it supports only windows defined by processing-time. The window size must be a multiple of the batch interval because a window keeps a particular number of batches until it is purged. Choosing the batch interval can heavily affect the performance of window-based analytics. First, the latency and response time of windowed analytics is strongly relying on the batch interval. Second, supporting only processing-time windowed analytics, can be a severe limitation for some use-cases.

Spark also supports backpressure. It handles backpressure by putting a bound to block size. Blocks are created in data source operators per each predefined time unit. Depending on the duration and load of each mini-batch job, the effectiveness of backpressure signal handling from source to destination may

vary. To detect the backpressure, Spark implements a contract that listens to mini-batch completion updates from the related operators and maintains a rate limit, i.e. an estimate of the speed at which the engine should ingest tuples. With every completed mini-batch update event, Spark calculates the current processing rate and estimates the optimal data ingestion rate.

### 2.2.3 Apache Flink

Apache Flink started off as an open-source big data processing system at TU Berlin, leveraging major parts of the Stratosphere project [24]. At its core, Flink is a distributed data flow engine. Like in Storm, a Flink runtime program is a DAG of operators connected with data streams. Flink’s runtime engine supports unified processing of batch (bounded) and stream (unbounded) data, considering former as being the special case of the latter.

Flink provides its own memory management to avoid long-running JVM’s garbage collector stalls by serializing data into memory segments. The data exchange in distributed environments is achieved via buffers. A producer takes a buffer from the pool and fills it up with data. Then, the consumer receives the data and frees the buffer informing the memory manager. Flink provides a wide range of high level and user-friendly APIs to manage state. Incremental state update, managing the memory, or checkpointing with big states are performed automatically and transparently to the user.

Flink has a strong feature set for building and evaluating windows on data streams. With a wide range of pre-defined windowing operators, it supports user-defined windows with custom logic. The engine provides processing-time, event-time, and ingestion-time data processing semantics. Like in Storm, the timestamps must be attached to each tuple as a separate field. At ingestion time, the system processes tuples with event-time semantics on these timestamps. Flink provides support for out-of-order streams. Flink also supports backpressure. It uses blocking queues. Once the congestion is detected, this information is automatically transferred to upstream operators with negligible cost.

## 2.3 Query Optimization

We utilize rule- and cost-based query optimization techniques to empower ad-hoc stream query processing, in Chapters 4 and 5, respectively. To enrich our work with cost-based query optimization, we adopt the Iterative Dynamic Programming (IDP) technique [25] and enhance it for streaming workloads. Below, we provide background information about the original IDP technique.

Algorithms based on dynamic programming lay in the core of query optimization. While these algorithms produce good optimization results (i.e., good query execution plans), its high complexity can be restrictive for optimizing complex queries or multiple queries. Optimization algorithms that are based on the IDP principle propose several advantages to deal with highly complex queries. IDP-based algorithms include both dynamic and iterative programming techniques. Thus, these algorithms are adaptive and produce as good plans as dynamic programming based algorithms if dynamic programming is viable. If dynamic programming is not viable (e.g., the problem is too complex), then IDP variants still are able to produce as-good-as possible plans. Also, existing dynamic programming based query optimizers can be easily extended to their IDP counterparts. There are two main variants of the IDP approach: IDP1 and IDP2. In this thesis, we adopt and enhance IDP1. We explain this algorithm below and refer to it as IDP throughout the thesis.

The main idea behind IDP is to *i*) break the query into subqueries containing join trees with up to  $k$  relations, *ii*) calculate the cost of each tree, *iii*) greedily choose the cheapest plan, *iv*) replace the cheapest one by a compound relation, and *v*) start the process all over again. Figure 2.2 shows an example query optimization scenario with IDP. The example join query includes 5 relations with block size  $k=3$ . The

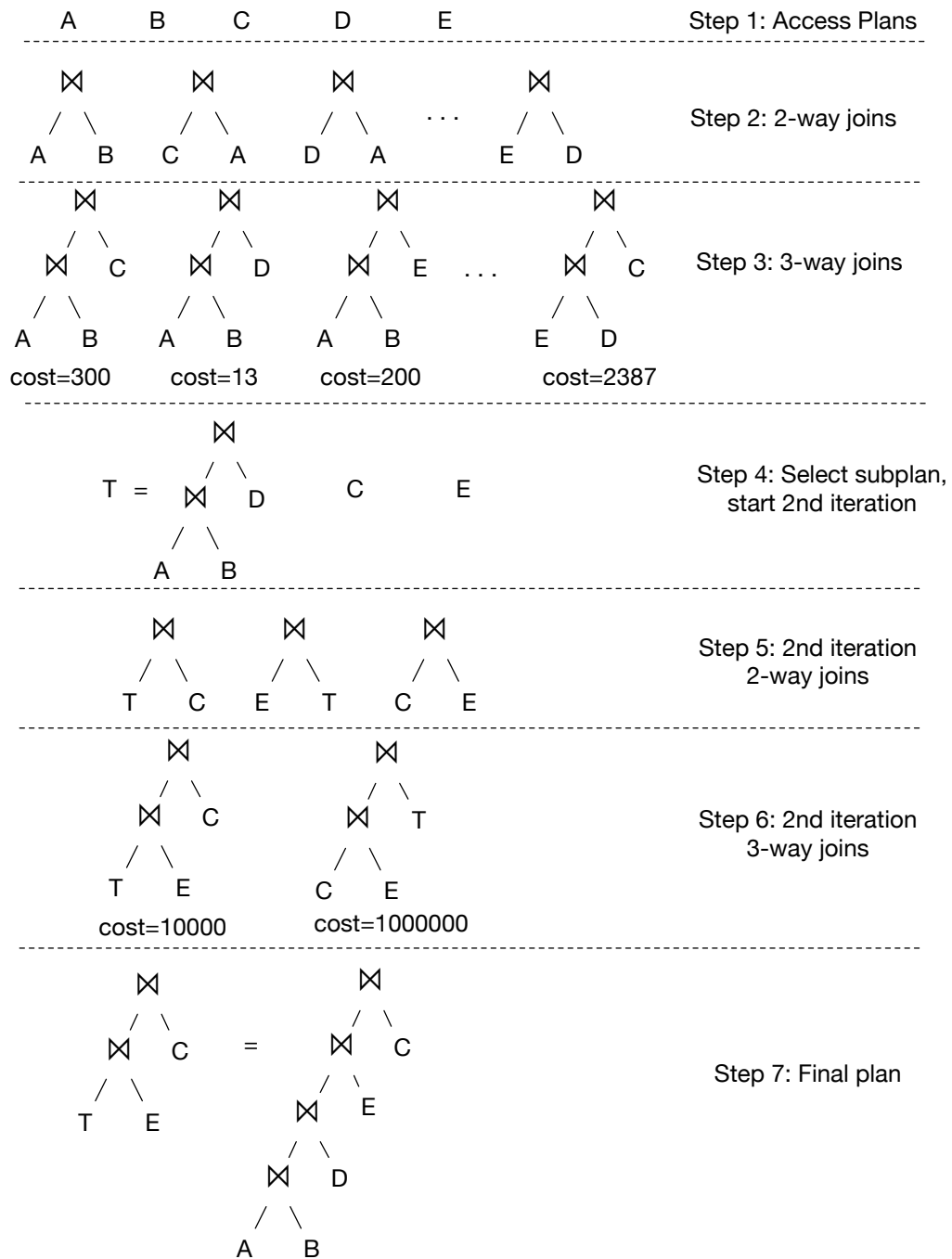


Figure 2.2: Optimizing a 5-way join query with IDP ( $k=3$ )

## 2. Background

---

first three steps are similar to classic dynamic programming, meaning that the algorithm generates access plans, 2-way and 3-way join plans, calculates the optimal QEP, and prunes suboptimal plans. Because we choose the block size to be 3 ( $k=3$ ), the algorithm breaks in Step 4, and greedily chooses the subplan with the lowest cost (T). All other plans containing one or more tables considered in the selected plan are discarded. In Step 5 IDP starts the second iteration with C, E, and T. This process continues until the final plan is computed (Step 7 in the example).

In the special case where  $k$  is equal to the number of relations in the input query (e.g., for smaller problems), IDP calculates the optimal solution. Thus, tuning  $k$  provides a good compromise between runtime and optimality. Because the algorithm combines greedy heuristics with dynamic programming, it is able to scale to large problems.

# 3

## Benchmarking Distributed Stream Data Processing Engines

This Chapter contains:

|         |  |    |
|---------|--|----|
| 3.1     | Introduction . . . . .                                   | 19 |
| 3.2     | Related Work . . . . .                                   | 19 |
| 3.2.1   | Batch Processing . . . . .                               | 20 |
| 3.2.2   | Stream Processing . . . . .                              | 20 |
| 3.3     | Benchmark Design Decisions . . . . .                     | 21 |
| 3.3.1   | Simplicity is Key . . . . .                              | 21 |
| 3.3.2   | On-the-fly Data Generation vs. Message Brokers . . . . . | 21 |
| 3.3.3   | Queues Between Data Generators and SUT Sources . . . . . | 22 |
| 3.3.4   | Separation of Driver and the SUT . . . . .               | 22 |
| 3.4     | Metrics . . . . .  | 23 |
| 3.4.1   | Latency . . . . .  | 23 |
| 3.4.1.1 | Event-time vs. Processing-time Latency . . . . .         | 24 |
| 3.4.1.2 | Event-time Latency in Windowed Operators . . . . .       | 24 |
| 3.4.1.3 | Processing-time Latency in Windowed Operators . . . . .  | 25 |
| 3.4.2   | Throughput . . . . .                                     | 26 |
| 3.4.2.1 | Sustainable Throughput. . . . .                          | 26 |
| 3.5     | Workload Design . . . . .                                | 28 |
| 3.5.1   | Dataset . . . . .  | 28 |
| 3.5.2   | Queries . . . . .  | 28 |
| 3.6     | Evaluation . . . . .                                     | 29 |
| 3.6.1   | System Setup . . . . .                                   | 29 |
| 3.6.1.1 | Tuning the Systems . . . . .                             | 29 |
| 3.6.2   | Performance Evaluation . . . . .                         | 30 |
| 3.6.2.1 | Windowed Aggregations . . . . .                          | 30 |
| 3.6.2.2 | Windowed Joins . . . . .                                 | 32 |
| 3.6.2.3 | Unsustainable Throughput . . . . .                       | 35 |

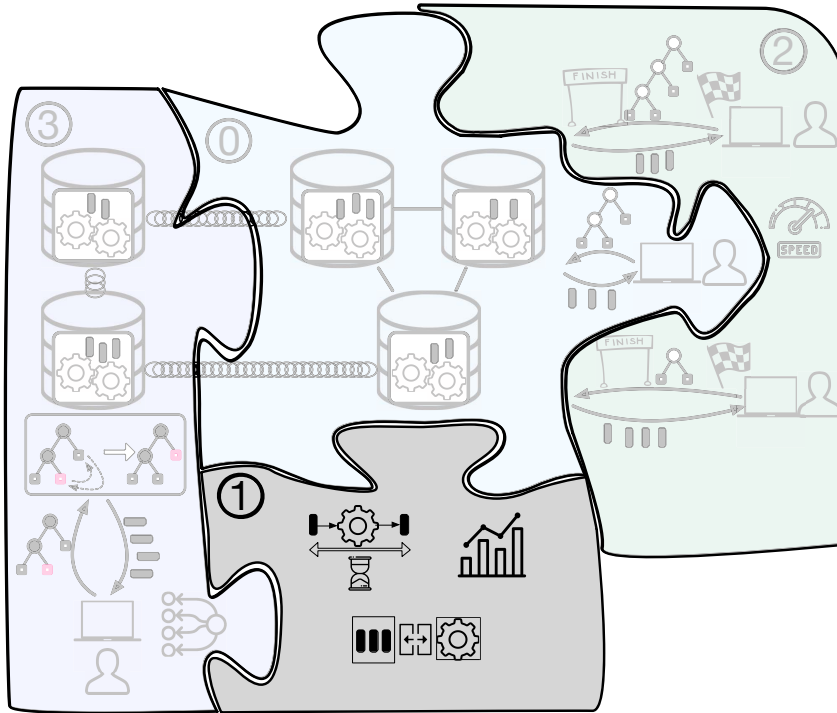


Figure 3.1: Scope of Chapter 3: Performance Analysis of modern SPEs

|          |  |           |
|----------|--|-----------|
| 3.6.2.4  | Queries with Large Windows . . . . .             | <b>35</b> |
| 3.6.2.5  | Data Skew . . . . .                              | <b>35</b> |
| 3.6.2.6  | Fluctuating Workloads . . . . .                  | <b>36</b> |
| 3.6.2.7  | Event-time vs. Processing-time Latency . . . . . | <b>36</b> |
| 3.6.2.8  | Observing Backpressure . . . . .                 | <b>37</b> |
| 3.6.2.9  | Throughput Graphs . . . . .                      | <b>37</b> |
| 3.6.2.10 | Resource Usage Statistics . . . . .              | <b>38</b> |
| 3.6.2.11 | Multiple Stream Query Execution . . . . .        | <b>39</b> |
| 3.6.3    | Discussion . . . . .                             | <b>39</b> |
| 3.7      | Conclusion . . . . .                             | <b>40</b> |

The need for scalable and efficient stream analysis has led to the development of many open-source SPEs with highly diverging capabilities and performance characteristics. While first initiatives try to compare the systems for simple workloads, there is a clear gap of detailed analyses of the systems’ performance characteristics. In this chapter, we present a framework for benchmarking distributed stream processing engines. We use our suite to evaluate the performance of three widely used SPEs in detail, namely Apache Storm, Apache Spark, and Apache Flink. Our evaluation focuses in particular on measuring the throughput and the latency of windowed operations, which are the basic type of operations in stream analytics. For this benchmark, we design workloads based on real-life, industrial use-cases inspired by the online gaming industry. The contribution of this chapter is threefold. First, we decouple the SUT from the test driver, in order to correctly represent the open-world model of typical stream processing deployments. This separation enables our benchmark suite to measure system performance under realistic conditions. Second, we give a definition of latency and throughput for stateful operators. Third, we propose the first benchmarking framework to define and test the sustainable performance of SPEs. Our detailed evaluation highlights the individual characteristics and use-cases of each system.

## 3.1 Introduction

Processing large volumes of data in batch is often not sufficient when the new data have to be processed fast. For that reason, stream data processing has gained significant attention. The most popular SPEs, with large-scale adoption in industry and the research community, are Apache Storm [2], Apache Spark [3], and Apache Flink [5]. As a measure of popularity, we consider the systems' community size, pull requests, number of contributors, commit frequency at the source repositories, and the size of the industrial community adopting the respective systems in their production environment.

An important application area of stream data processing is online video games. This application area requires the fast processing of large scale online data feeds from different sources. Windowed aggregations and windowed joins are two main operations that are used to monitor user feeds. A typical use-case is tracking the in-application-purchases per application, distribution channel, or product item (in-app products). Another typical use-case is the monitoring of advertising: making sure that all campaigns and advertisement networks work flawlessly, and comparing different user feeds by joining them. For example, monitoring the in-application-purchases of the same game downloaded from different distribution channels and comparing users' actions are essential in online video game monitoring.

In this work, we propose a benchmarking framework to accurately measure the performance of SPEs. For our experimental evaluation, we test three publicly available open-source engines: Apache Storm, Apache Spark, and Apache Flink. We use latency and throughput as the two major performance indicators. Latency, in SPEs, is the time difference between the moment of data production at the source (e.g., the mobile device) and the moment that the SPE has produced an output. Throughput, in this scenario, determines the number of ingested and processed tuples per time unit.

Even though there have been several comparisons of the performance of SPEs recently [26, 27, 28], they did not measure the latency and throughput that can be achieved in a production setting. One of the repeating issues in previous work is the missing definition and inaccurate measurement of latency in stateful operators (e.g., joins). Moreover, previous work does not clearly separate the SUT and the test driver. Frequently, the performance metrics are measured and calculated within the SUT, resulting in incorrect measurements.

In this chapter, we address the above mentioned challenges. Our proposed benchmarking framework is generic with a clear design and well-defined metrics, which can be applied to any SPE. The main contributions of this chapter are as follows:

- We accomplish the complete separation of the test driver from the SUT.
- We introduce a technique to accurately measure the latency of stateful operators in SPEs. We apply the proposed method to various use-cases.
- We measure the maximum sustainable throughput of SPEs. Our benchmarking framework handles system-specific features like backpressure to measure the maximum sustainable throughput of a system.
- We use the proposed benchmarking system for an extensive evaluation of Storm, Spark, and Flink with practical use-cases.

## 3.2 Related Work

Benchmarking parallel data processing systems has been an active area of research. Early benchmarking efforts have focused on batch processing and later on extended to stream processing.

#### 3.2.1 Batch Processing

HiBench [29] was the first benchmark suite to evaluate and characterize the performance of Hadoop. Later, it was extended with a streaming component [30]. HiBench includes a wide range of experiments ranging from micro-benchmarks to machine learning algorithms. SparkBench, features machine learning, graph computation, SQL queries, and streaming applications on top of Apache Spark [31]. BigBench [32] is an end-to-end benchmark with all major characteristics of big data systems. The BigDataBench [33] suite contains 19 scenarios covering a broad range of applications and diverse data sets. Marcu et al. [34] performed an extensive analysis on Apache Spark and Apache Flink with iterative workloads.

The above benchmarks either adopt batch processing systems and metrics used in batch processing systems or apply the batch-based metrics on SPEs. We, on the other hand, analyze SPEs with a new definition of metrics and show that adopting batch processing metrics for SPEs leads to biased benchmark results.

#### 3.2.2 Stream Processing

Recently, a team from Yahoo! conducted a series of experiments on three Apache projects, namely Storm, Flink, and Spark and measured their latency and throughput [26]. They used Apache Kafka [35] and Redis [36] for data retrieval and storage respectively. Perera et al. used the Yahoo Streaming Benchmark and Karamel [37] to provide reproducible batch and streaming benchmarks of Apache Spark and Apache Flink in a cloud environment [38]. Later on, it was shown that Kafka and Redis were the bottleneck in the experiments of the Yahoo! Streaming Benchmark [39].

In this chapter, we overcome these bottlenecks by *i*) generating the data on the fly with a scalable data generator (Section 3.3) instead of ingesting data from Kafka and *ii*) not storing data in a key-value store.

Lopez et al. [27] propose a benchmarking framework to assess the throughput performance of Apache Storm, Spark, and Flink under node failures. The key finding of their work is that Spark is more robust to node failures but it performs up to an order of magnitude worse than Storm and Flink. Compared to this work, we observed a large difference with respect to the throughput achieved by the same systems. The paper allows the SPEs to ingest data at maximum rate. Instead, we introduce the concept of sustainable throughput: in our experiments, we control the data ingestion rate (throughput) of an SPE, in order to avoid large latency fluctuations. We argue that sustainable throughput is a more representative metric which takes into account the latency of a system.

Shukla et al. [28] perform common IoT tasks with different SPEs and evaluate their performance. The authors define latency as the interval between the source operator’s ingestion time and the sink operator’s result emission time. As we discuss in Section 3.4, this approach leads to inaccurate measurements. The same issue is also present in the LinearRoad benchmark [40]. To alleviate this problem, we perform experiments measuring event-time latency. Additionally, Shukla et al. define throughput as the rate of output messages emitted from the output operators in a unit time. However, since the number of result-tuples can differ from the input-tuples (e.g., in an aggregation query) we measure the throughput of data ingestion and introduce the concept of sustainable throughput.

StreamBench [41], proposes a method to measure the throughput and latency of SPEs with the use of a mediator system between the data source and the SUT. In this work, we explain that such a mediator system is a bottleneck and/or affect the measurements’ accuracy. Finally, several SPEs implement their own benchmarks to measure the system performance without comparing them with any other system [42] [43] [3].

In summary, our benchmarking framework is the first to *i*) separate the SUT and driver, *ii*) use a scalable data generator and to *iii*) define metrics for system-, and event-time, as well as to *iv*) introduce and use the concept of sustainable throughput throughout experiments.



### 3.3 Benchmark Design Decisions

In this section, we discuss the main design decisions of our benchmarking framework. Simplicity is one of the key factors of our benchmarking framework. Also, we embrace on-the-fly data generation, instead of pulling the data from an external system, such as a message broker or a file system. In addition, we utilize queues between a data generator and an SPE. Finally, our benchmark design embraces the complete separation of the system under test and the test driver.

#### 3.3.1 Simplicity is Key

Modern SPE benchmarks, such as Yahoo Streaming Benchmark [26], comprise of complete and end-to-end industrial use-cases. On the one hand, the resulting setup simulates the real production use-case, with the test driver, the SUT, and a set of other systems. For example, Yahoo Streaming Benchmark setup includes Apache Kafka and Redis in addition to the driver and SUTs [26]. On the other hand, third-party systems in this design might affect the benchmark results. In other words, additional systems between the SPE and the driver are likely to add an extra latency for each tuple.

Our benchmark setup comprises of the test driver and SUT. Third-party systems are not part of the benchmarking framework. This design decision enables us to measure the performance of the SUT with minimum overhead from external factors.

#### 3.3.2 On-the-fly Data Generation vs. Message Brokers

SPEs nowadays typically pull the data from message brokers, such as Apache Kafka [35], instead of directly connecting to push-based data sources. The message broker persists the data coming from various sources [44], performs data replication, and makes the data available for other systems to use. The data exchange between the message broker and an SPE may easily become the bottleneck of a benchmark deployment for two main reasons. First, if the message broker’s data partitioning is not chosen wisely, data re-partitioning may occur before the data reaches the sources of the SPE. This can happen when data resides in a different machine in the cluster or the data is partitioned in a different way than the SPE requires it. Even if the data inside the message broker is pre-partitioned with respect to the SPE’s partitioning, the SPE might still shuffle the input data. For example, at the time of writing this thesis, Apache Flink did not support already partitioned sources, meaning it shuffles the input data even if the data have already been pre-partitioned at the input source. Second, the data needs to be persisted on disk before going through a de-/serialization layer between the SPE and the message broker.

In our benchmark design, we choose not to use a message broker, but rather, adopt a distributed in-memory data generator with configurable data generation rates. Each data generator retains a local queue. The data generator pushes each generated tuple into the local queue. The SUT pulls the tuple from the local queue for processing. The communication between the driver and the SUT is bounded only by the network bandwidth and the speed of the data ingestion by the SUT.

The data transfer, between the driver and SUT, utilizes a pull-based approach. The pull- and push-based data ingestion approaches perform theoretically the same with sustainable workloads. However, the pull-based approach is the one used by modern SPEs [5, 19, 45]. No matter how high the workload a system can sustain, it always has an upper limit. As a result, using push-based approaches leaves the user with no guarantee for possible high workloads.

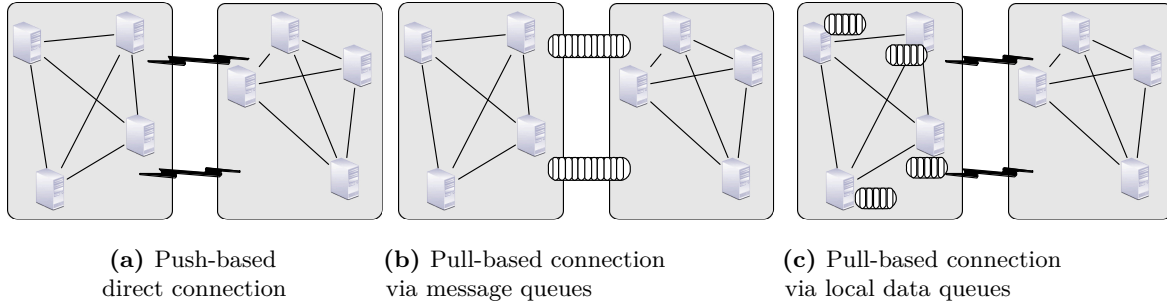


Figure 3.2: Benchmark designs to connect the data generator (on the right) and SUT (on the left)

#### 3.3.3 Queues Between Data Generators and SUT Sources

It is quite common that the throughput (data ingestion rate) of an SPE is not constant throughout the duration of an experiment. The fluctuations in the ingestion rate can be due to transient network issues, garbage collection in JVM-based engines, etc. To alleviate this problem, we add a queue between each data generator and the SUT’s source operators to even out the difference in the rates of data generation and data ingestion.

Figure 3.2 shows three possible cases to link the data generator with the SPE. The simplest design is to connect the SPE directly to the data generator as shown in Figure 3.2a. Although this is a perfectly acceptable design, it does not match real-life use-cases. In large scale setups, SPEs do not connect to push-based data sources, but pull data from distributed message queues. Figure 3.2b shows the pull-based design, where the data source and the SPE are connected through the message queues. A common bottleneck of this option is the throughput of the message queuing system. Also, this adds a de-/serialization layer between the SPE and the data sources. Therefore, we use the third option, which is a hybrid of the first two. As can be seen in Figure 3.2c we embed the local data queues as a separate module in the data generators. This way, the throughput is bounded only by the network bandwidth. Also, the systems work more efficiently as there are no de-/serialization overheads.

#### 3.3.4 Separation of Driver and the SUT

In previous work, the throughput was either measured inside the SUT or the benchmark leveraged internal statistics of the SUT. However, different systems can have very diverse definitions and computation semantics of latency and throughput. The computation semantics might also be unknown if the system is not open-source.

In our benchmarking framework, we isolate the test driver (i.e., the data generator, queues, and measurements) from the SUT (i.e., the SPE), to perform measurements out of the SUT. More specifically, we measure throughput at the queues between the data generator and the SUT and measure the latency at the sink operator of the SUT. Each pair of the data generator and the queue resides on the same machine to avoid any network overhead and to ensure data locality. The queue data is always kept in memory to avoid disk write/read overhead.

The data generator timestamps each tuple at generation time. It performs so, with a constant speed throughout the experiment. The event’s latency is calculated from the time instance that it is generated. So, the longer a tuple stays in a queue, the higher its latency is. We make sure that the driver and SUT instances do not share computational resources, as they might affect each other’s performance.

Figure 3.3 shows the overall architecture of our benchmarking framework. There are two main components of test deployment: the SUT and the driver. The driver consists of data generators, each of which maintains local data queues. The driver is responsible for generating and queuing the data. It is

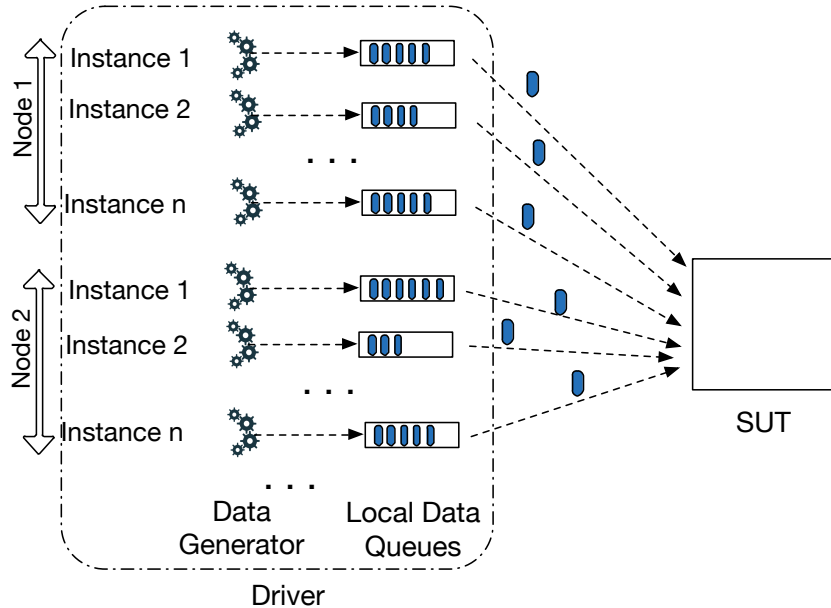


Figure 3.3: Design of our benchmarking framework

composed of a finite number of instances which are distributed evenly to the worker nodes. The driver nodes are separate from SUT nodes in the cluster deployment. Data generators and their corresponding data queues reside in the same machine to avoid any network overhead and to ensure data locality.

## 3.4 Metrics

SPEs are typically evaluated using two main metrics: throughput and latency. In this section, we make a distinction between two types of latency, namely event-time latency and processing-time latency. We then describe two types of throughput, namely maximum throughput and sustainable throughput.

### 3.4.1 Latency

Modern stream processing semantics distinguish two notions of time: *event-time* and *processing-time* [17]. The *event-time* is the time when an event is captured while the *processing-time* is the time when an operator processes a tuple. Similar to the nomenclature of these two notions of time, we distinguish between event- and processing-time latency.

**Definition 1 (Event-time Latency)** We define event-time latency to be the interval between a tuple’s event-time and its emission time from the SPE’s output operator.

For instance, in an ATM transaction, the event-time is the moment of a user’s action at the terminal. The event-time latency is the time interval between the moment that the user’s action took place and the moment that the event has been fully processed by the SPE.

**Definition 2 (Processing-time Latency)** We define processing-time latency to be the interval between a tuple’s ingestion time (i.e., the time that the event has reached the input operator of the SPE) and its emission time from the SPE’s output operator.

For instance, in an ATM transaction, the processing-time is the moment in which the transaction reaches the source operator of the SPE. The processing-time latency is the time between the moment that the transaction is reached at the SPE and the moment that it has been fully processed by the SPE.

3.4.1.1 Event-time vs. Processing-time Latency

Event- and processing-time latencies are equally important metrics. The event-time latency includes the time that a given event has spent in a queue, waiting to be processed, while processing-time latency is used to measure the time it took for the event to be processed by the SPE. In practical scenarios, event-time latency is very important as it defines the time in which the user interacts with a given system. Ideally, this time should be minimized. Clearly, the processing-time latency makes part of the event-time latency. We use both metrics to characterize a system’s performance.

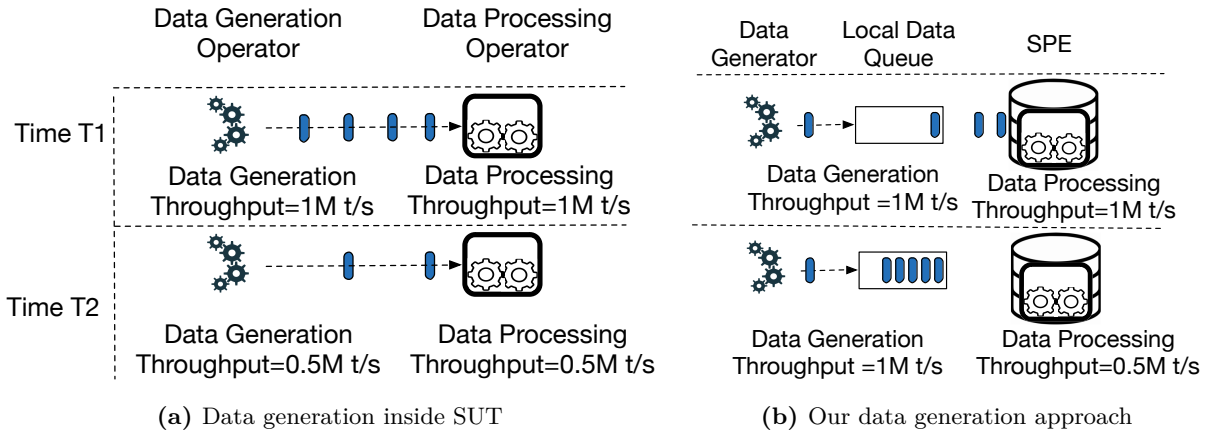


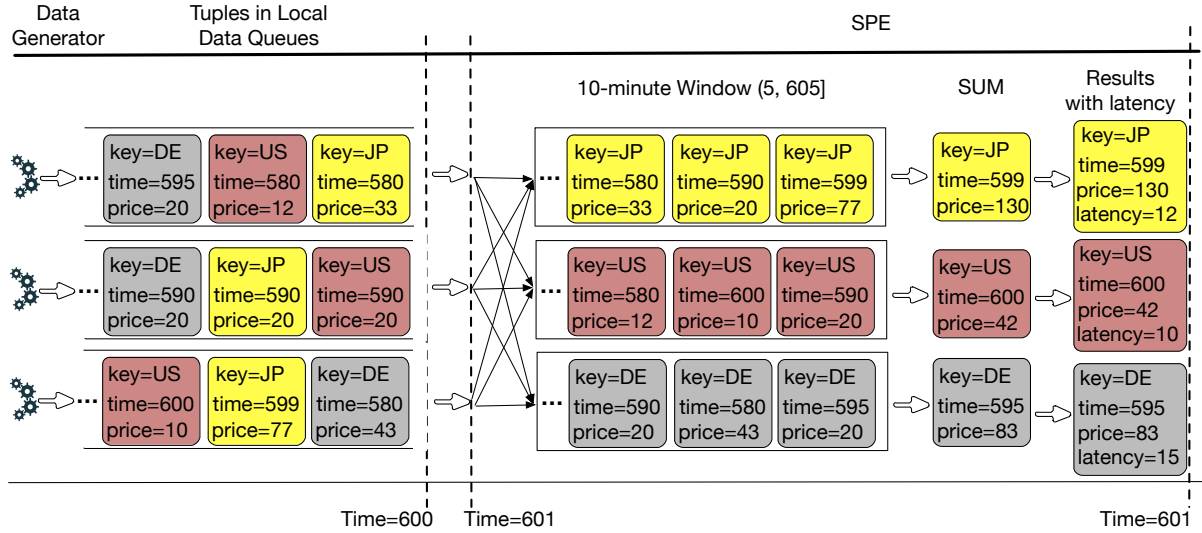
Figure 3.4: An example scenario for coordinated and realistic data generation

Not clearly differentiating the two metrics leads to the coordinated omission problem. In coordinated omission service time (i.e., processing-time) is measured at the SUT and any increasing queuing time, which is part of the *response time* (i.e., event-time), is ignored [46]. Friedrich et al. show that coordinated omission leads to significant underestimation of latencies [47]. Figure 3.4 shows the two data generation scenarios: data generation inside the SUT and our approach. In Figure 3.4a at time T1 the data generator produces 1 million tuples per second. At the same time, the downstream data processing operator processes all the generated tuples. At time T2 the data processing throughput diminishes to 0.5 million tuples per second. The data generation speed automatically is dropped at the data generator. Thus, the data is generated on demand, meaning the data generator and the data processing operator perform computation in a coordinated way.

In real industrial use-cases a data source emits data independently from a data processing system. For example, a video game player does not adjust the frequency of its actions (e.g., clicks) based on the performance of the underlying data processing system (e.g., SPE). Figure 3.4b shows our data generation approach. At time T1, the data generation and processing speeds are equal. Therefore, there are few tuples residing in data queues. At time T2, the data processing throughput drops. In this case, the data generation speed is still the same (1 million tuples per second), because it should not depend on the performance of the SPE. If the SPE cannot catch up with the data generation speed, then after some time the queue will be full. Also, each subsequent tuple inside the queue will have higher event-time latency. In this case, the SPE cannot sustain the workload.

3.4.1.2 Event-time Latency in Windowed Operators

Stateful operators, such as window aggregates (e.g., a sum aggregate over an hour’s worth of data), retain state and return results after having seen a number of tuples over some time. Measuring latency in such cases is non-trivial. The reason is that the latency of a given windowed operator is affected by the tuples’ waiting time until the window is formed completely.



**Figure 3.5:** End-to-end example of an aggregation query. The data generator produces tuples and timestamps their event-time (before `time=600`). After that, the SPE ingests the tuples, groups them by their key, and aggregates the tuples (SUM). The event-time latency of the output tuples equals to the maximum event-time latency of tuples in each window.

Figure 3.5 depicts the data generator and a set of tuples in three queues. The tuples are timestamped with their event-time when they are generated. The tuples are then grouped by their key and are put in a 10-minute window. Take, for example, the window containing the red tuples with `key=US`. The timestamps of these three tuples are 580, 590, and 600. When these tuples are aggregated into a new tuple (the sum of their values with a total of `value=42`), we need to assign an event-time to that output. That event-time is then used to calculate the event-time latency (in this case, `latency=10`). The main intuition is that in this way, we exclude the tuples' waiting time while the window is still buffering data. The event-time is defined more formally below.

**Definition 3 (Event-time of Windowed Events)** *The event-time of a windowed operator's output tuple, is the maximum event-time of all tuples that contributed to that output.*

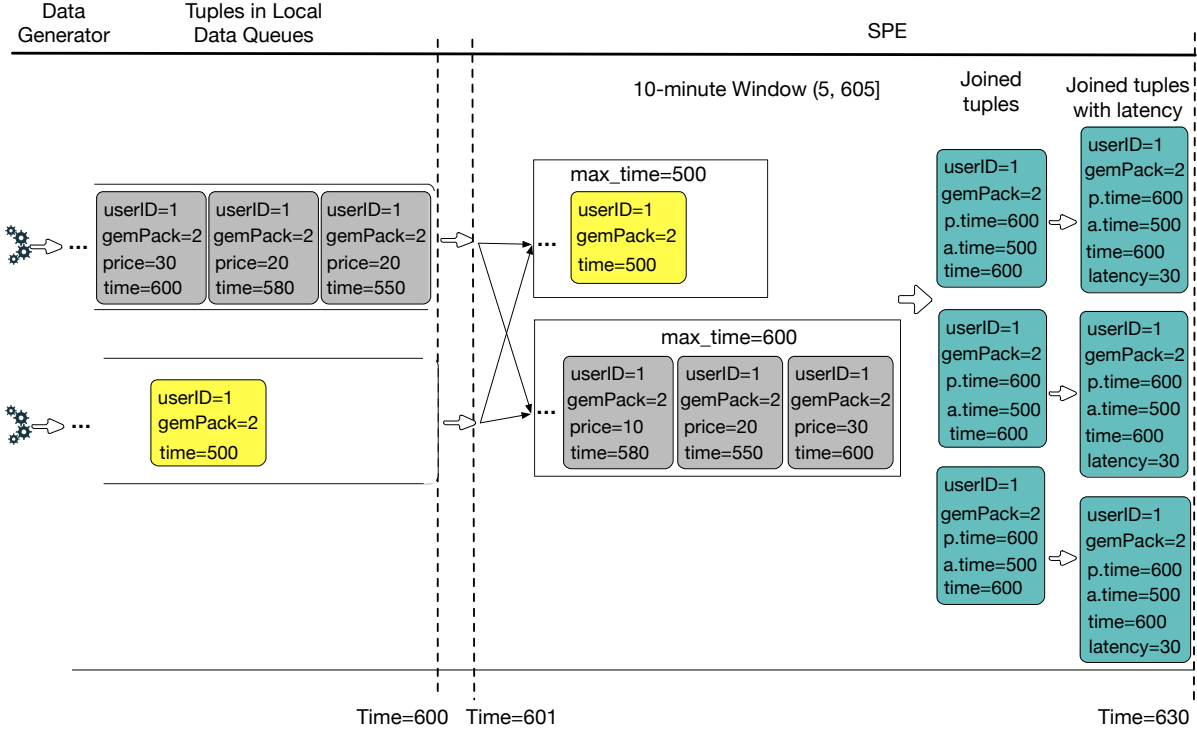
In a windowed join operation, the joined tuples' event-time is the maximum event-time of their window. Afterwards, each join output is assigned the maximum event-time of its matching tuples. As described in our example, in order to calculate the event-time latency of an output tuple, all we have to do is subtract the event-time of that tuple from the current system time. Figure 3.6 shows the main intuition behind this idea. We join ads (yellow) and purchases (gray) streams in a 10-minute window. The join operator is an equi-join that is based on the attribute `userID` and `gemPack`. The maximum timestamp of the ads stream tuples (`a.time`) in the window is 500. For the purchases stream, the maximum event-time timestamp (`p.time`) in the window is 600. We assign the event-time of the joined tuple as the maximum of these two values (`a.time` and `p.time`). We use the assigned event-time value to calculate the event-time latency of the tuple.

### 3.4.1.3 Processing-time Latency in Windowed Operators

Apart from event-time latency, we need to calculate the processing-time latency of tuples as well. We define the processing-time of a windowed stream tuple similarly to the event-time.

**Definition 4 (Processing-time of Windowed Events)** *The processing-time of a windowed operator's output event, is the maximum processing-time of all events that contributed to that output.*

### 3. Benchmarking Distributed Stream Data Processing Engines



**Figure 3.6:** End-to-end join of two streams. The SPE reads tuples and forms a 10-minute window. The tuples are joined, and the event-time of the result-tuples equals to the maximum event-time of tuples in their corresponding windows.

The processing-time latency is calculated in the same way as for event-time, with a small difference. Every tuple is enriched with an extra, processing-time field at its ingestion time (when the tuple reaches the first operator of the SPE). The processing-time field is the system clock of the machines that run the respective operator. In our example in Figure 3.5, this enrichment happens right after `time=601`. To calculate the processing-time latency, we simply subtract the processing-time of that tuple from the current system time.

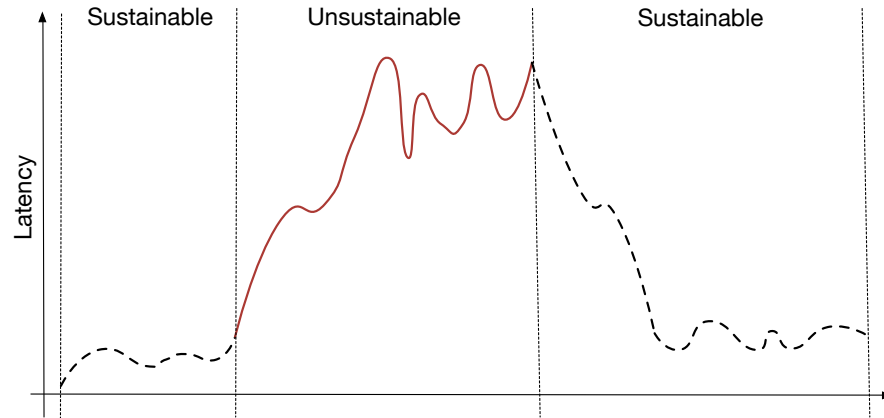
#### 3.4.2 Throughput

The throughput of a data processing system is defined as the number of events that the system can process in a given amount of time. In this context, the throughput and event-time latency often do not correlate. For instance, an SPE that batches tuples together before processing them can generally achieve higher throughput. However, the time spent on batching events affects the events' event-time latency.

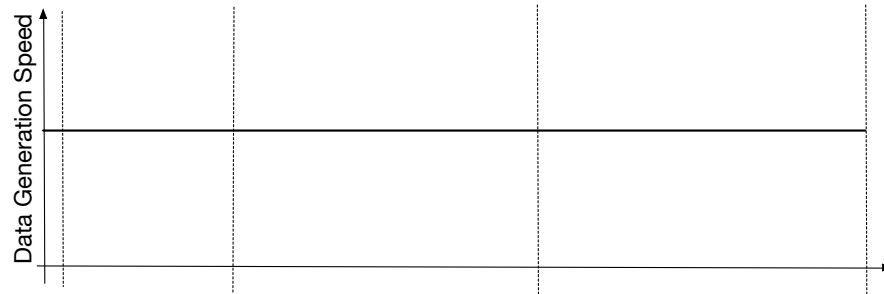
In practice, the deployment of an SPE has to take into account the arrival rate of data. When the data arrival rate increases, the system has to adapt (e.g., by scaling out) in order to handle the increased arrival rate and process tuples without exhibiting backpressure. To reflect this, we define the concept of *sustainable throughput* and discuss how we attain it in our experiments.

##### 3.4.2.1 Sustainable Throughput.

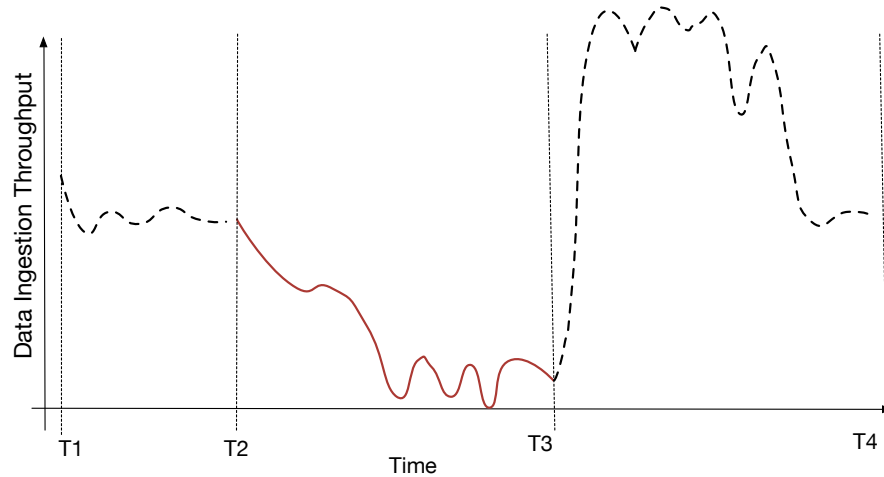
An SPE starts to build up backpressure (i.e., the system queues up new tuples in order to process the tuples that have already been ingested), when the amount of data is more than it can handle. As a result, from the moment that the backpressure mechanism is initiated, the event-time latency of all queued tuples increases. As we can see from Figure 3.7a, backpressure can be transient: as soon as the system catches up again with the tuples' arrival rate (time T3), the event-time latency will stabilize (between T3 and T4).



(a) Event-time latency over time



(b) Data generation speed over time



(c) Data ingestion throughput over time

**Figure 3.7:** Impact of sustainable and unsustainable throughput to the latency, data generation speed, and data ingestion throughput. All the figures share the same x-axis. The SPE in this figure sustains the given workload until time T2. The workload between T2 and T3 is unsustainable. The SPE catches up with the data generation speed and sustains the workload after time T3.

When the system’s throughput is larger than the tuples’ arrival rate (time T2), the event-time latency will decrease to the minimum (i.e., the processing-time). Otherwise, the latency increases continuously. In real-world applications, delays longer than a predefined threshold are unacceptable. For example, if the server is overloaded because of throughput spikes, skews, and etc., the player observes a loading screen, which might reduce customer satisfaction. Depending on the Quality of Service (QoS), the delay threshold can be different.

To avoid the coordinated omission [47] our data generation speed is clearly separated from the data processing throughput. For example, the data generation speed in Figure 3.7b is stable, although the data processing throughput drops in Figure 3.7c. The SPE in the figure manages to catch up with the data generation speed after time T3. Therefore, the system ingests all tuples queued between T2 and T3, along with the generated tuples at current time. After some time, the SPE finishes to process queued tuples (between T2 and T3); therefore, the data ingestion throughput diminishes at T4.

**Definition 5 (Sustainable Throughput)** *Sustainable throughput is the highest load of event traffic that an SPE can handle without exhibiting prolonged backpressure, i.e., without a continuously increasing event-time latency.*

In our experiments, we make sure that the data generation rate matches the sustainable throughput of a given deployment. To find the sustainable throughput of a given deployment, we execute each of the systems under test with a very high generation rate. Then, we decrease the data generation speed until the system can sustain that data generation rate. We allow for some fluctuation, i.e., we allow a maximum number of tuples to be queued, as soon as the queue does not continuously increase.

## 3.5 Workload Design

The workload for our benchmark is derived from an online video game application at Rovio<sup>1</sup>. Rovio continuously monitors user actions in games to ensure that their services work as expected. For instance, the quality assurance team continuously monitors the number of active users and generates alerts when this number has large drops. Moreover, once a game has an update or receives a new feature, the team monitors incoming events to check whether the newly added feature is working smoothly. Rovio also tracks the in-app purchases per game, and distribution channel (e.g., Apple’s AppStore, Google Play), and per in-app purchased item (e.g., a gem pack) and proposes gem packs to users as their game progresses.

### 3.5.1 Dataset

Listing 1 shows two data streams: *i*) the `purchases` stream, which contains tuples of purchased gem packs and *ii*) the `ads` stream, which contains a stream of proposals of gem packs to users. Both stream sources contain the time attribute. The purchase time shows the time the user bought the gem pack. The ads time shows the time the ad was shown to the user.

### 3.5.2 Queries

We adopt queries from Figure 3.8 as representative queries for stream processing especially in online gaming scenarios. The real-time sliding windowed aggregation and windowed join queries are the ones that distinguish SPEs from batch processing system. Moreover, the use-cases are taken from real scenarios.

The first query that we use for our evaluation is a windowed aggregation query. More specifically, the query calculates the revenue made from each gem pack with a sliding window. The template for this query can be found in Figure 3.8.

---

<sup>1</sup>Creator of the Angry Birds game: <http://www.rovio.com/>



```

# Streams
PURCHASES(userID, gemPack, price, time)
ADS(userID, gemPack, time)

# Windowed Aggregation Query
SELECT SUM(price)
FROM PURCHASES [Range r, Slide s]
GROUP BY gemPack

# Windowed Join Query
SELECT p.userID, p.gemPack, p.price
FROM PURCHASES [Range r, Slide s] as p,
ADS [Range r, Slide s] as a,
WHERE p.userID = a.userID AND
p.gemPack = a.gemPack

```

**Figure 3.8:** Query templates used by our workloads

The second query is a windowed join query that is a typical use-case of correlating advertisements with their revenue. As we can see in Listing 1, each user is presented with a specified proposal to buy a gem pack at a given time instant. We join this stream with the stream of purchases in order to find which of the proposed gems has been bought, as a result of proposing the gem to users.

## 3.6 Evaluation

In this section, we evaluate the performance of three SPEs, namely Storm 1.0.2, Spark 2.0.1, and Flink 1.1.3. Due to the large number of parameters and variables, it is not possible not include all the experimental results in this section. Instead, we present the most interesting results.

### 3.6.1 System Setup

Our experiments have different runtime durations. If one experiment runs shorter than another, it means the former experiment has higher throughput than the latter. We also fix the data generation speed in all driver instances. Each data generator produces 100 M events with constant speed. We generate events with normal distribution on the key field.

Our cluster consists of 20 nodes, each equipped with 2.40GHz Intel Xeon CPU E5620. For our experiments, we allocate 16 cores and 16GB RAM of each machine. The network bandwidth is 1Gb/s. We dedicate one node to the master node of the SPEs. All nodes' system clocks in the cluster are synchronized via a local NTP server.

We use 25% of the input data as a warmup. So, in all experiments, we exclude the first 25% of output. We enable backpressure in all SUTs. That is, we do not allow the systems to ingest more input than they can process and crash during the experiment. If the SUT drops one or more connections to the driver instance, then the driver halts the experiment with the conclusion that the SUT cannot sustain the given throughput. Similarly, in real-life if the SPE cannot sustain the user feed and drops the connection, this is considered as a failure.

#### 3.6.1.1 Tuning the Systems

Tuning configuration parameters of the SPEs is important to achieve good performance. There are several properties for each SPE, that need to be tuned and customized to the given use-case. We adjust the buffer size in Flink to ensure a good balance between throughput and latency. Although selecting low buffer size

### 3. Benchmarking Distributed Stream Data Processing Engines

---

|       | 2-node   | 4-node   | 8-node   |
|-------|----------|----------|----------|
| Storm | 0.4 M/s  | 0.69 M/s | 0.99 M/s |
| Spark | 0.38 M/s | 0.64 M/s | 0.91 M/s |
| Flink | 1.2 M/s  | 1.2 M/s  | 1.2 M/s  |

**Table 3.1:** Sustainable throughput for windowed aggregations

can result in a low processing-time latency, the event-time latency of tuples may increase. Because the buffer size is small, the majority of the tuples will be queued in the driver queues instead of the buffers inside the SPE.

We adjust the block interval in Spark for partitioning RDDs in Spark. The number of RDD partitions in a mini-batch is bounded by  $\frac{\text{Batch}}{\text{Block}} \frac{\text{Interval}}{\text{Interval}}$ . As the cluster size increases, decreasing the block interval increases the parallelism. One of the main reasons that Spark scales out very well is the partitioning of RDDs.

In Storm, the number of workers, executors, and buffer size are the configurations (among many others) that need to be tuned to get the best performance. Similar to Flink, tuning the buffer size is a key to balance between latency and throughput. For all systems, choosing the right level of parallelism is essential to balance between an efficient resource utilization and network or resource exhaustion.

Storm introduced the backpressure feature in recent releases; however, it is not mature yet. With high workloads, it is possible that the backpressure stalls the topology, causing spouts to stop emitting tuples. Moreover, we notice that Storm drops some connections to the data queue when tested with high workloads with backpressure disabled, which is not acceptable according to the real world use-cases. Dropping connections due to high throughput is considered a system failure.

#### 3.6.2 Performance Evaluation

In this section, we present a set of experiments to evaluate SUTs. We evaluate SUT’s performance on windowed aggregation and windowed join workloads. Also, we test SUTs’ performance with unsustainable throughput and with large windows. We also evaluate SUTs’ performance with skewed and fluctuating workloads.

##### 3.6.2.1 Windowed Aggregations

We use the aggregation query (Figure 3.8) with 8 seconds window length and 4 seconds window slide, for our first evaluations. Table 3.1 shows the sustainable throughput of the SPEs. We use a four-second batch-size for Spark, as it can sustain the maximum throughput with this configuration. We identify that Flink’s performance is bounded by network bandwidth with 4- or more node cluster configuration. Storm’s and Spark’s performance in terms of throughput are comparable, with Storm outperforming Spark by approximately 8% in all configurations.

Table 3.2 shows the latency measurements of windowed aggregations. We conduct experiments with maximum and 90%-workloads. The latency values shown in this table correspond to the workloads given in Table 3.1. In most cases, where the network bandwidth is not a bottleneck, we can see a significant decrease in latency when lowering the throughput by 10%. This shows that the sustainable throughput saturates the system.

Flink has the best *min* and *avg* latencies. Although its *max* latency is way above than its *min*, from quantile values we can conclude that those values can be considered as outliers. The main reason for having such a high *max* latency is associated with the buffer size. The large buffer size enables high throughput; on the other hand, it can cause some tuples to have high latencies. With 4- and more node cluster configurations, we can observe that there is a slight difference in Flink’s latency statistics between

|            | 2-node     |            |            |                             |
|------------|------------|------------|------------|-----------------------------|
|            | <i>avg</i> | <i>min</i> | <i>max</i> | <i>quantiles (90,95,99)</i> |
| Storm      | 1.4        | 0.07       | 5.7        | (2.3, 2.7, 3.4)             |
| Storm(90%) | 1.1        | 0.08       | 5.7        | (1.8, 2.1, 2.8)             |
| Spark      | 3.6        | 2.5        | 8.5        | (4.6, 4.9, 5.9)             |
| Spark(90%) | 3.4        | 2.3        | 8          | (3.9, 4.5, 5.4)             |
| Flink      | 0.5        | 0.004      | 12.3       | (1.4, 2.2, 5.2)             |
| Flink(90%) | 0.3        | 0.003      | 5.8        | (0.7, 1.1, 2)               |
|            | 4-node     |            |            |                             |
| Storm      | 2.1        | 0.1        | 12.2       | (3.7, 5.8, 7.7)             |
| Storm(90%) | 1.6        | 0.04       | 9.2        | (2.9, 4.1, 6.3)             |
| Spark      | 3.3        | 1.9        | 6.9        | (4.1, 4.3, 4.9)             |
| Spark(90%) | 2.8        | 1.6        | 6.9        | (3.4, 3.7, 4.8)             |
| Flink      | 0.2        | 0.004      | 5.1        | (0.6, 1.2, 2.4)             |
| Flink(90%) | 0.2        | 0.004      | 5.1        | (0.6, 1.3, 2.4)             |
|            | 8-node     |            |            |                             |
| Storm      | 2.2        | 0.2        | 17.7       | (3.8, 6.4, 9.2)             |
| Storm(90%) | 1.9        | 0.2        | 11         | (3.3, 5, 7.6)               |
| Spark      | 3.1        | 1.2        | 6.9        | (3.8, 4.1, 4.7)             |
| Spark(90%) | 2.7        | 1.7        | 5.9        | (3.6, 3.9, 4.8)             |
| Flink      | 0.2        | 0.004      | 5.4        | (0.6, 1.2, 3.9)             |
| Flink(90%) | 0.2        | 0.002      | 5.4        | (0.5, 0.8, 3.4)             |

**Table 3.2:** Latency statistics, avg, min, max, and quantiles (90, 95, 99) in seconds for windowed aggregations. For each system experiments are executed with maximum and 90% sustainable throughput.

the maximum and 90%-throughput. The reason is that this workload is not the maximum sustainable throughput, and it is bounded by the network bandwidth.

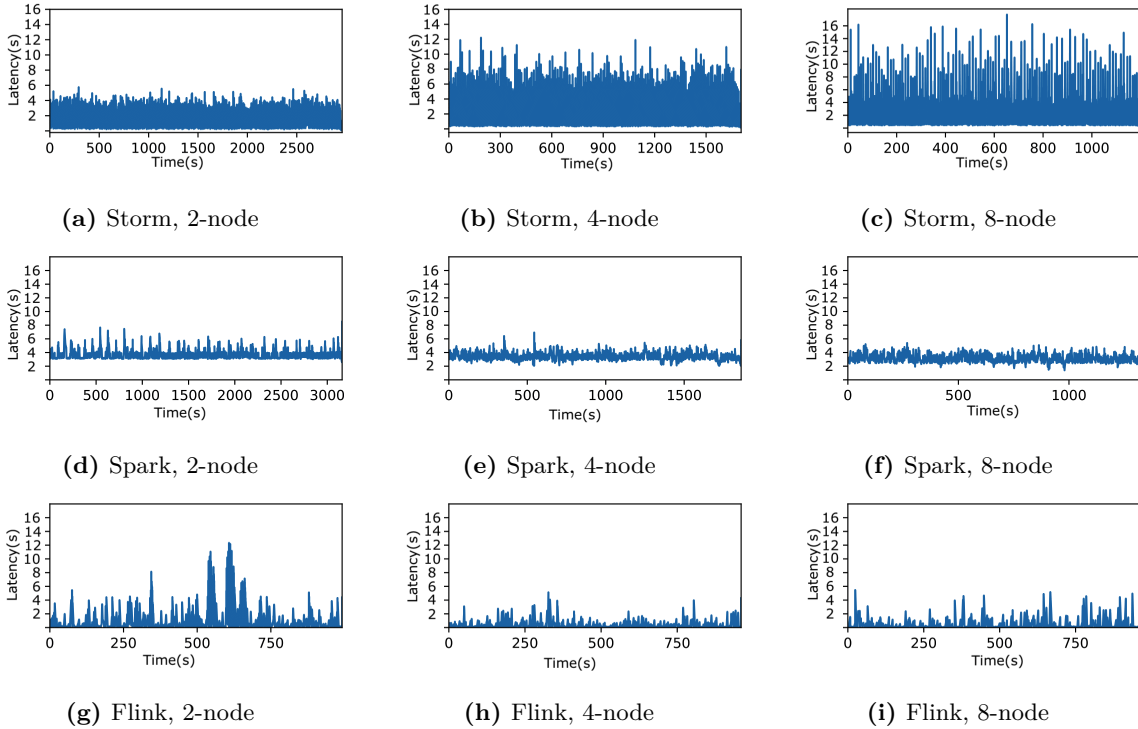
As we see from Table 3.2 Spark has a higher latency than Storm and Flink but it exhibits less variation in *avg*, *min*, and *max* latency measurements. Because Spark processes tuples in mini-batches, the tuples within the same batch have similar latencies and, therefore, there is little difference among the measurements. Moreover, transferring data from Spark’s block manager to DStream by creating RDDs adds additional overhead that results in higher *avg* latencies for Spark compared to Flink and Storm.

The *avg* and *max* latency values increase in Storm with large cluster size, while in Spark we see the opposite behavior, which means Spark can partition the data (RDDs) better in bigger distributed environments. However, from the quantile values we can conclude that the *max* latencies of Storm can be considered as outliers.

Figure 3.9 shows the windowed aggregation latency distribution over time. In all cases, we can see that the fluctuations are lowered when decreasing the throughput by 10% in Figure 3.10. While in Storm and Flink it is hard to detect the lower bounds of latency as they are close to zero, in Spark the upper and lower boundaries are more stable and clearly noticeable. The reason is that a Spark job’s characteristics are highly dependent on the batch size and this determines the clear upper and lower boundaries for the latency. The smaller the batch size, the lower the latency and throughput. To have a stable and efficient configuration in Spark, the mini-batch processing time should be less than the batch interval. We determine the most fluctuating system to be Flink in 2-node setup and Storm in 8-node setup as shown in Figures 3.9g and 3.10b. Those fluctuations show the behavior of backpressure.

Spark splits the input query into multiple sub-queries and executes them in separate jobs. So, the number of batch jobs for each batch-time interval is at least one. For example, we use *reduceByKey()* to parallelize the stages of the mini-batch within a window. It is transformed into two subsequent RDDs: first a ShuffledRDD and then a MapPartitionsRDD. The coordination and pipelining mini-batch jobs and

### 3. Benchmarking Distributed Stream Data Processing Engines



**Figure 3.9:** Windowed aggregation latency distributions in time series with maximum sustainable throughput

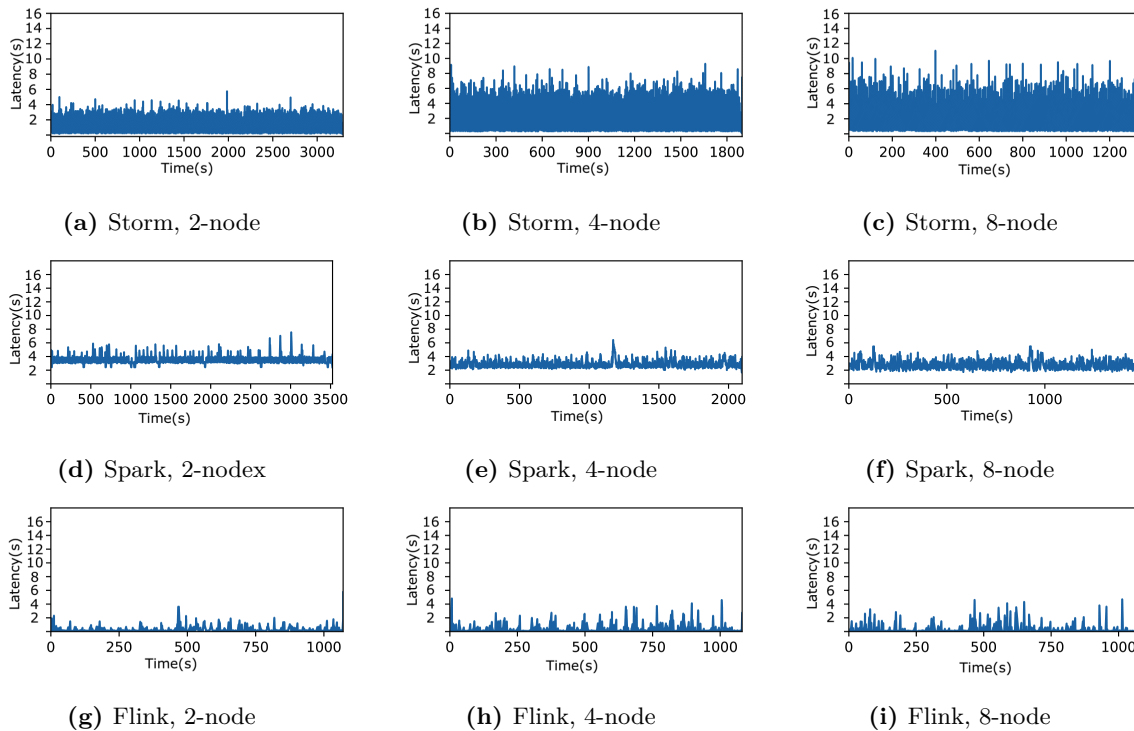
their stages create extra overhead for Spark. In Flink and Storm, on the other hand, this is a single step both in the logical and physical query plan.

#### 3.6.2.2 Windowed Joins

In this section, we use the windowed join query from Figure 3.8 to benchmark Spark and Flink. Storm provides a windowing capability but there is no built-in windowed join operator. Initially, we tried Storm’s Trident v2.0 abstraction, which has built-in windowed join features. However, Trident computed incorrect results as we increased the batch size. Moreover, there is a lack of support for Trident in the Storm community. As an alternative, we implemented a simple version of a windowed join in Storm. Comparing it with Spark and Flink, which have advanced memory and state management features, leads to unfair comparisons. We implemented a naïve join in Storm and examined the sustainable throughput to be 0.14 million events per second and measured an average latency to be 2.3 seconds on a 2-node cluster. However, we encountered memory issues and topology stalls on larger clusters. As a result, we focus on Flink and Spark for the windowed join benchmarks.

Depending on the selectivity of the join operator, a vast amount of join output results can be produced. Sink operators can be a bottleneck in this case. Also, the vast amount of results of the join operator can cause the network to be a bottleneck. To address these issues, we generate the input streams such that the join operator exhibits low selectivity. In general, the experimental results for windowed joins are similar to the experiments with windowed aggregations.

Table 3.3 shows the sustainable throughput of the SUTs. Flink’s throughput for an 8-node cluster configuration is bounded by the network bandwidth. Table 3.4 shows the latency statistics for windowed joins. We can see that in all cases Flink outperforms Spark in all parameters. To ensure the stability of Spark, the runtime of each mini-batch should be less than batch size in Spark. Otherwise, the size of the queued mini-batch jobs will increase over time, and the system will not be able to sustain the throughput. However, we see from Table 3.3 that the latency values for Spark are higher than mini-batch duration (4



**Figure 3.10:** Windowed aggregation latency distributions in time series with 90% sustainable throughput

|       | 2-node   | 4-node   | 8-node   |
|-------|----------|----------|----------|
| Spark | 0.36 M/s | 0.63 M/s | 0.94 M/s |
| Flink | 0.85 M/s | 1.12 M/s | 1.19 M/s |

**Table 3.3:** Sustainable throughput for windowed joins

sec). The reason is that we are measuring the event-time latency. So, the additional latency is due to tuples' waiting in the driver queues.

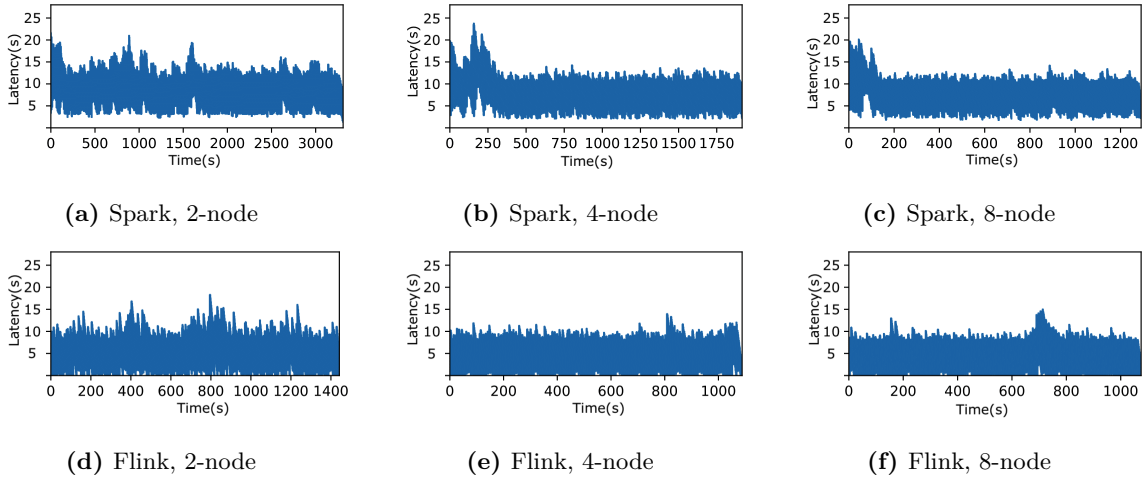
Figure 3.11 shows the windowed join latency distributions as time-series. In contrast to windowed aggregations, we observe substantial fluctuations in Spark. Also, we notice a significant latency increase in Flink compared to its windowed aggregation latency values. The reason is that windowed join is more expensive than windowed aggregation. We also notice that spikes in latency values are significantly reduced with 90% workload in Figure 3.12.

Similar to windowed aggregations, in windowed joins Spark's major disadvantage is having blocking operators. Another limitation is coordination and scheduling overhead across different RDDs. For example, in our windowed join query Spark produces CoGroupedRDD, MappedValuesRDD, and FlatMappedValuesRDD in different stages of the job. Each of these RDDs has to wait for the parent RDDs to be ready before their initialization. Flink, on the other hand, performs operator chaining during the query optimization phase and avoids blocking operations. For example, the reduce operation is a non-blocking operator in Flink. As a result, the system sacrifices some use-cases, which require blocking reduce, to achieve a better performance. Internally, Storm also has a similar architecture; however, the semantics of its operators is highly dependent on their implementation. For example, one implementation of the windowed reduce operator can output the results continuously, while another can chose to perform so in bulk.

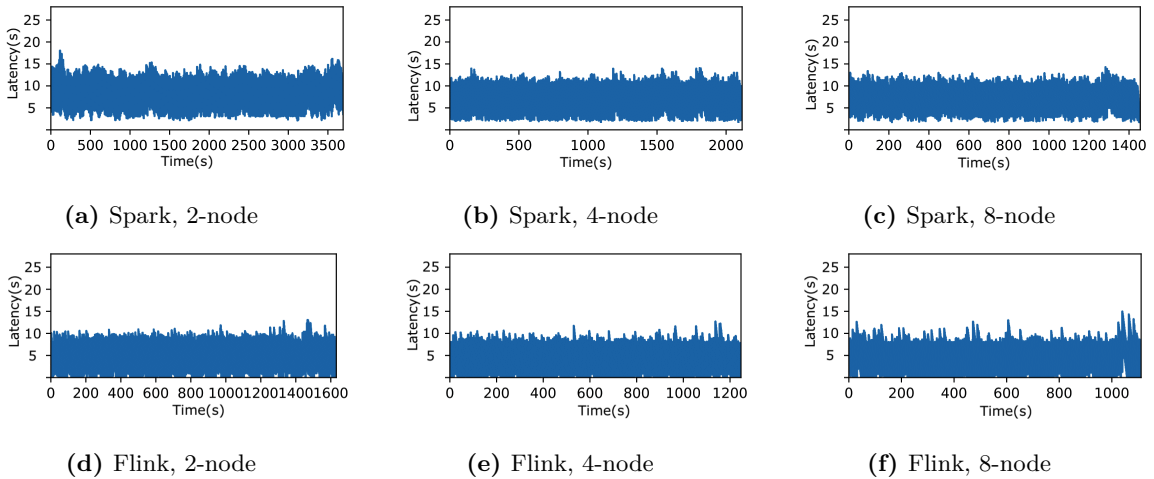
### 3. Benchmarking Distributed Stream Data Processing Engines

|            | 2-node |      |      |                      |
|------------|--------|------|------|----------------------|
|            | avg    | min  | max  | quantiles (90,95,99) |
| Spark      | 7.7    | 1.3  | 21.6 | (11.2, 12.4, 14.7)   |
| Spark(90%) | 7.1    | 2.1  | 17.9 | (10.3, 11.1, 12.7)   |
| Flink      | 4.3    | 0.01 | 18.2 | (7.6, 8.5, 10.5)     |
| Flink(90%) | 3.8    | 0.02 | 13   | (6.7, 7.5, 8.7)      |
|            | 4-node |      |      |                      |
|            | avg    | min  | max  | quantiles (90,95,99) |
| Spark      | 6.7    | 2.1  | 23.6 | (10.2, 11.7, 15.4)   |
| Spark(90%) | 5.8    | 1.8  | 13.9 | (8.7, 9.5, 10.7)     |
| Flink      | 3.6    | 0.02 | 13.8 | (6.7, 7.5, 8.6)      |
| Flink(90%) | 3.2    | 0.02 | 12.7 | (6.1, 6.9, 8)        |
|            | 8-node |      |      |                      |
|            | avg    | min  | max  | quantiles (90,95,99) |
| Spark      | 6.2    | 1.8  | 19.9 | (9.4, 10.4, 13.2)    |
| Spark(90%) | 5.7    | 1.7  | 14.1 | (8.6, 9.4, 10.6)     |
| Flink      | 3.2    | 0.02 | 14.9 | (6.2, 7, 8.4)        |
| Flink(90%) | 3.2    | 0.02 | 14.9 | (6.2, 6.9, 8.3)      |

**Table 3.4:** Latency statistics, avg, min, max and quantiles (90, 95, 99) in seconds for windowed joins. For each system experiments are executed with maximum and 90% sustainable throughput.



**Figure 3.11:** Windowed join latency distributions in time series with maximum sustainable throughput



**Figure 3.12:** Windowed join latency distributions in time series with 90% sustainable throughput

### 3.6.2.3 Unsustainable Throughput

Benchmarking SPEs with the sustainable throughput is essential as otherwise, the performance of SPEs can degrade. We experienced on average 3-10 % performance decrease when we provided the SUTs with 20% more throughput than they can sustain. In general, the decrease in overall throughput is dominated by the time interval where the system is unstable, meaning the system cannot find the best input data rate. This is directly related to the backpressure implementation of the systems. For example, when we execute the windowed aggregation query on 2 nodes with 20% over-saturated workload, the overall throughput of Storm, Spark, and Flink decreased by 6%, 5%, 3% respectively. With the same parameters and with the windowed join query, we examine approximately 1.5 times more performance decrease. We notice that with over-saturated workloads, as the window size gets large and window slide length and buffer size gets smaller, Storm drops the input sockets more frequently.

### 3.6.2.4 Queries with Large Windows

Window size and window slide have a significant impact on the SUT's performance. One interesting observation is that with the same batch size, as the size of the window increases, Spark's throughput decreases significantly. For example, for the aggregation query, with window length and slide 60 seconds, Spark's throughput decreases by 2 times. In the meantime, the *avg* latency increases by 10 times. We find that the main reason for Spark's decreasing performance is caching. Especially with windowed join queries, the cache operation consumes the memory aggressively. Internally when a task receives an input tuple for processing, it checks if the tuple is marked for caching. If yes, all the following tuples of the particular RDD will be sent to the memory store of the block manager. Thus, Spark spills the memory store to disk once it is full. When we disable the caching, we experience a performance decrease due to the repeated computation.

Storm, on the other hand, can handle the large window operations if the user utilizes advanced data structures that can spill to disk when needed. Otherwise, we encountered memory exceptions. Flink (as well as Spark) has built-in data structures that can spill to disk when needed. However, this does not apply for the operations inside the User Defined Functions (UDFs), as Flink and Spark treat UDFs as blackbox. The windowed aggregation and windowed join implementations of Flink, Storm, and Spark are unable to share intermediate aggregate and join results among different sliding windows.

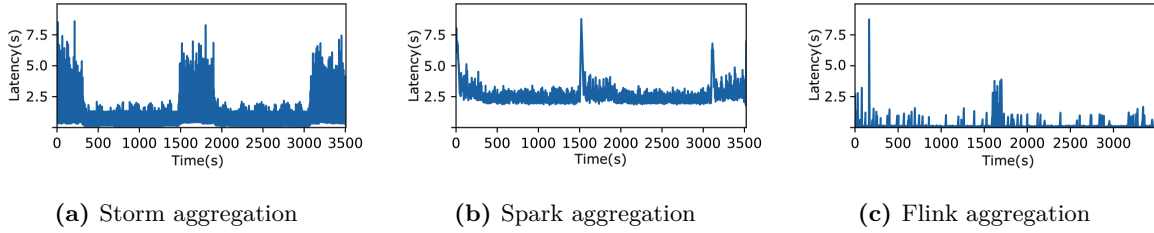
### 3.6.2.5 Data Skew

Data skew is yet another concern to be addressed by SPEs, as in a production environment data distribution can be unpredictable. We analyzed the SUTs with extreme skew, namely their ability to handle data of a single key. In Flink and Storm, the performance of the system is bounded by the performance of a single slot of a machine, meaning it does not scale. For the aggregation query, we measured the throughput 0.48 M tuples/s for Flink and 0.2 M tuples/s for Storm. These measurements do not improve when the SUTs scale out.

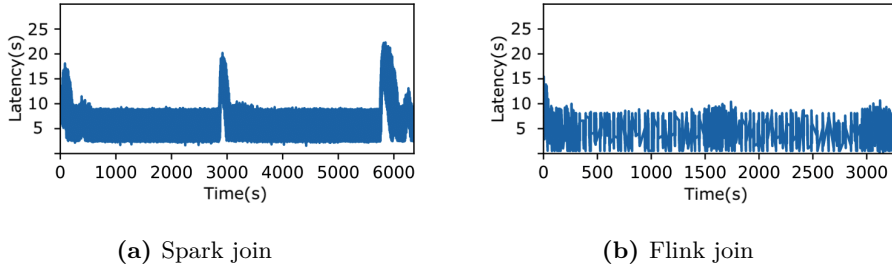
Spark, on the other hand, can handle skewed data efficiently. We experienced 0.53 M tuples/s sustainable throughput for Spark in a 4-node cluster for the aggregation query. For the join query, on the other hand, both Spark and Flink cannot handle skewed data well. That is, Flink often becomes unresponsive in this test. Spark, on the other hand, exhibits very high latencies. The main reason is that the memory is consumed quite fast and the backpressure mechanism fails to perform efficiently.

One reason for the performance difference between Spark and Flink with skewed data lies in how the systems compute aggregations. Flink and Storm use one slot per operator instance. So, if the input data is skewed, this architecture can cause performance issues. Spark has a slightly different architecture. In Spark, forcing all partitions to send their reduced values to a specific computing slot can easily cause

### 3. Benchmarking Distributed Stream Data Processing Engines



**Figure 3.13:** Event-time latency on stream aggregation workloads with fluctuating data arrival rate



**Figure 3.14:** Event-time latency on stream join workloads with fluctuating data arrival rate

the network to become a bottleneck when partition size is big. Therefore, Spark adopts tree-reduce and tree-aggregate communication pattern to minimize the communication and data shuffling. This is the main reason that makes Spark perform better with skewed input data.

#### 3.6.2.6 Fluctuating Workloads

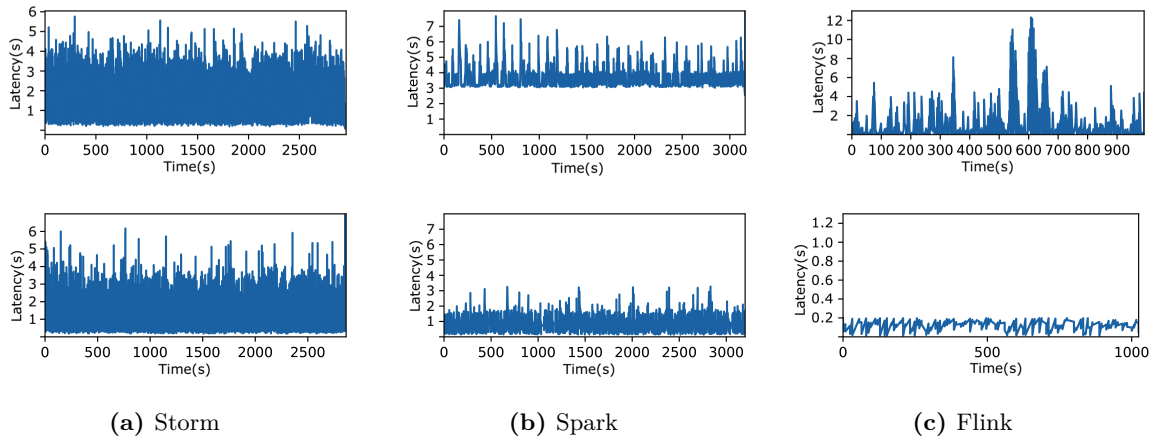
We also analyze the SUTs with fluctuating workloads by simulating workload spikes. We start the benchmark with a workload of 0.84 M tuples/s, then decrease it to 0.28 M tuples/s and increase again after a while. As we can see from Figures 3.13 and 3.14 Storm is the system most susceptible to fluctuating workloads. Spark and Flink have comparable behavior with windowed aggregations. However, for windowed joins, Flink can handle spikes better. One reason behind this behavior is the difference between the systems’ backpressure mechanism. As mentioned above, Spark can be thought of as a chain of jobs with multiple stages. Once the stage is overloaded, passing this information to upstream stages works in the order of the execution time of job stages; however, this time is in the order of the execution time of tuples in Flink. We conduct experiments with different cluster and buffer sizes as well. As we increase the buffer or cluster size the spikes get smoother; however, the overall *avg* latency increases.

#### 3.6.2.7 Event-time vs. Processing-time Latency

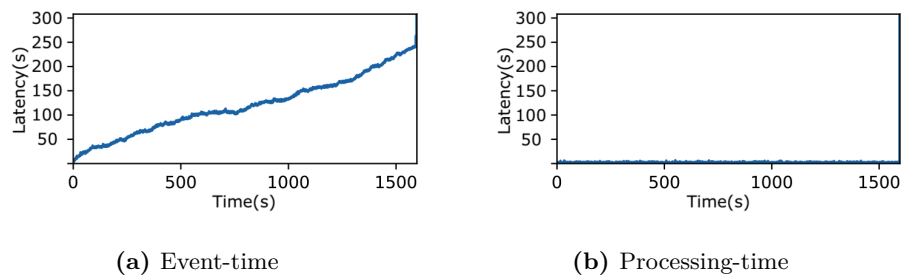
Figure 3.15 shows the comparison between the processing-time and event-time latency. We conduct experiments with the aggregation query (8 seconds window length, 4 seconds window slide) on a 2-node cluster. Even with a small cluster size (Figure 3.15), there is a significant difference between event- and processing-time latencies. The main reason behind this difference is due to the time duration input tuples wait in the data queues. We did not observe any significant changes in results with different cluster configurations and with the join query.

To emphasize the necessity of our definition of latency, we draw the reader’s attention to Figure 3.16, which shows event-time and processing-time latencies for Spark when the system is extremely overloaded. As we can see from the figures, the processing-time latency is significantly lower than event-time latency. The reason is that when the SUT gets overloaded, it starts backpressure and lowers the data ingestion rate to stabilize the end-to-end system latency. We can see that the SUT accomplished this goal as the





**Figure 3.15:** Comparison between event (top row) and processing-time (bottom row) latency



**Figure 3.16:** Comparison between event- and processing-time latency of Spark with unsustainable throughput

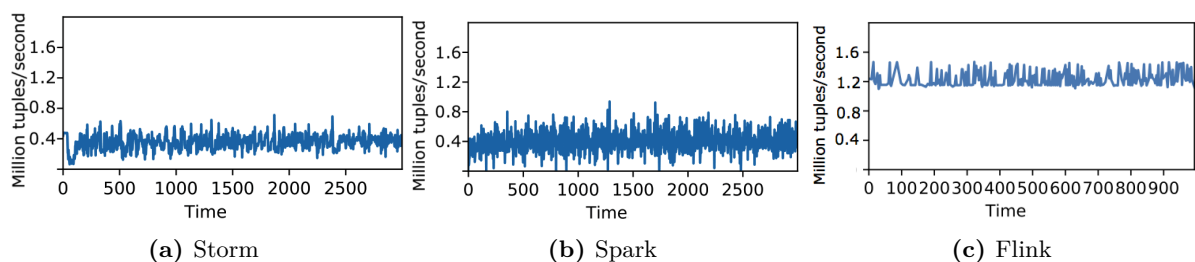
latency stays stable. However, the event-time latency keeps increasing as the input tuples wait in the data queues inside the driver. This is just one scenario where we can draw *unrealistic* or *incorrect* conclusions when using processing-time latency for evaluating SPEs. This is not a specific behavior for Spark, as we observed similar behavior for all SUTs.

### 3.6.2.8 Observing Backpressure

Backpressure is shown in Figures [3.11a](#), [3.11b](#), [3.11c](#), [3.11d](#), [3.11e](#), [3.11f](#) and [3.9g](#). Moreover, our driver can also observe short-term spikes (Figures [3.10b](#), [3.14a](#)) and continuous fluctuations (Figure [3.12d](#)). Furthermore, we can observe a wide range of sustainable *avg* latencies from 0.2 to 6.2 seconds and from 0.003 seconds *min* latency to 19.9 seconds *max* latency.

### 3.6.2.9 Throughput Graphs

As we separate the throughput calculation clearly from the SUT, we retrieve this metric from the driver. Figure [3.17](#) shows the sustainable throughput graphs for the aggregation query with 8 seconds window



**Figure 3.17:** Throughput graphs of systems under test

### 3. Benchmarking Distributed Stream Data Processing Engines

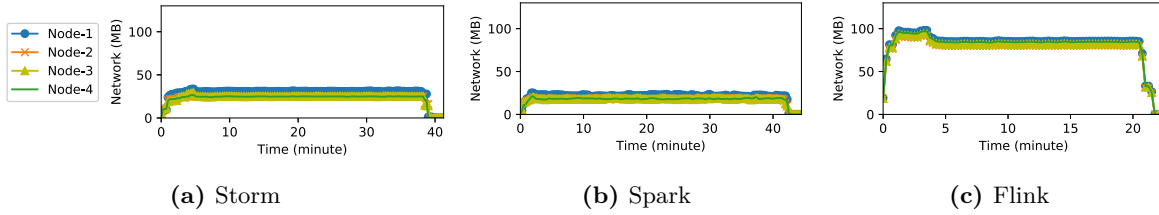


Figure 3.18: Network usages of the SUTs in a 4-node cluster

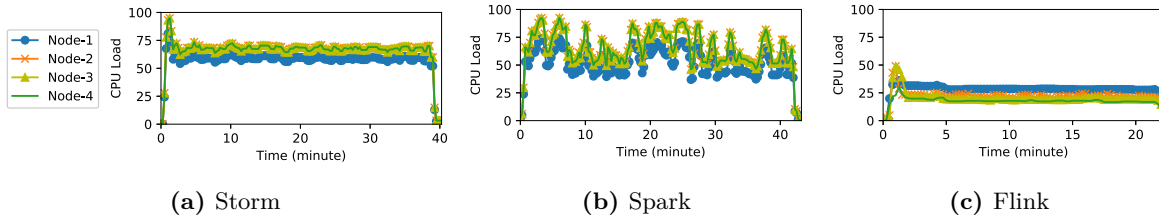


Figure 3.19: CPU usages of the SUTs in a 4-node cluster

length and 4 seconds window slide. We examined the similar behavior in other window settings and for the join query. As we can see from the figure, data pull rates of Spark and Storm are more fluctuating than Flink. Despite having a high data pull rate or throughput, Flink has fewer fluctuations. When we lower the workload, both Flink and Spark have stable data pull rates; however, Storm still exhibits significant fluctuations.

The reason for the highly fluctuating throughput for Storm is that the system lacks an efficient backpressure mechanism to find a near-constant data ingestion rate. The main reason for fluctuation in Spark is the deployment of several jobs at the same batch interval. Each job retrieves the data into its input buffers and fires. Until a job is finished, its input rate is limited. As a result, we can see a highly fluctuating throughput for Spark. Flink, on the other hand, benefits from its internally incremental computation mechanism (like Spark), tuple at a time semantics and efficient backpressure mechanism.

#### 3.6.2.10 Resource Usage Statistics

Figures 3.18 and 3.19 show the network and CPU usages of the SUTs, respectively. Because Flink’s performance is bounded by the network, we can see that CPU load is least. Storm and Spark, on the other hand, use approximately 50% more CPU clock cycles than Flink. As we can see from Figure 3.20, the scheduler overhead is one bottleneck for Spark’s performance. Initially, Spark ingests more tuples than

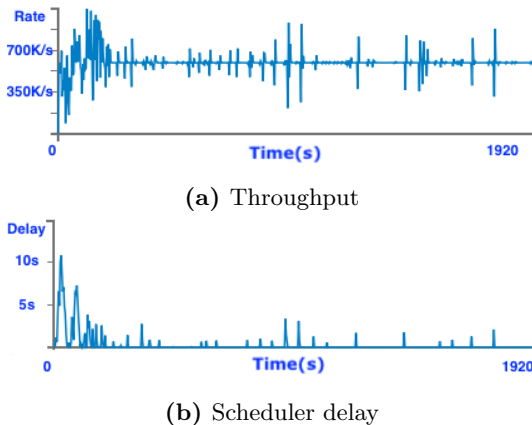


Figure 3.20: Scheduler delay (top row) vs. throughput (bottom row) in Spark.

```

# Streams
PURCHASES(userID, gemPack, price, time)
ADS(userID, gemPack, time)

# Windowed Aggregation Query
SELECT SUM(price)
FROM PURCHASES [Range r, Slide s]
GROUP BY gemPack
WHERE PURCHASES.gemPack > PARAM_VAL1

# Windowed Join Query
SELECT p.userID, p.gemPack, p.price
FROM PURCHASES [Range r, Slide s] as p,
ADS [Range r, Slide s] as a,
WHERE p.userID = a.userID AND
      p.gemPack = a.gemPack AND
      p.gemPack > PARAM_VAL2 AND
      a.gemPack > PARAM_VAL3

```

**Figure 3.21:** Query templates used for multiple stream query workloads. `PARAM_VAL $n$`  is a parameter value given by the user.

it can sustain. Because of the scheduler delay, backpressure fires and limits the input rate. Whenever there is a spike in the input rate, we can observe a similar behavior in the scheduler delay. One reason behind Spark’s efficient CPU usage is its automation, transparent resource usages, and many internal optimizations [48]. For example, Spark handles incremental state management, optimizations with code generation, and dynamic memory management efficiently and transparent to a user.

### 3.6.2.11 Multiple Stream Query Execution

We also evaluate the SUTs with multiple stream queries. We modify the query templates shown in Figure 3.8 with a selection predicate. Figure 3.21 shows the query templates used to generate multiple queries. `PARAM_VAL` is a selection predicate parameter to filter the stream source. We submit generated stream queries to the SUTs at compile-time. Although the aggregation function (`SUM`) and partitioning key (`PURCHASES.gemPack`) is the same among all generated stream queries, none of the SUTs can benefit from data and computation sharing. The similar limitation also appears in stream queries generated with the join query template. In both cases, the data and computation redundancy results in a linear decrease in performance. When we submit the generated stream queries at runtime in an ad-hoc manner, none of the SUTs were able to handle the requests.

## 3.6.3 Discussion

If a stream contains skewed data, then Spark is the best choice (Section 3.6.2.5). Both Flink and Spark are very robust to fluctuations in the data arrival rate in aggregation workloads (Section 3.6.2.6). For fluctuations in the data arrival rate on join queries, on the other hand, Flink behaves better (Section 3.6.2.6). In general, if the average latency is a priority, then Flink is the best choice (Sections 3.6.2.1 and 3.6.2.2). On the other hand, even with higher average latency, Spark manages to bound latency better than others (Sections 3.6.2.1 and 3.6.2.2). If a use-case contains large windows, Flink can have higher throughput with a low latency (Section 3.6.2.4). Overall, we observe that Flink achieves a better overall throughput both for aggregation and join queries. We define event- and processing-time latency and show the significant difference between them (Section 3.6.2.7). Our analysis shows that the SUTs are not able

to leverage the sharing opportunities among multiple stream queries and cannot execute ad-hoc stream queries (Section [3.6.2.11](#)).

## 3.7 Conclusion

Responding to an increasing need for real-time data processing in industry, we have built a novel framework for benchmarking SPEs with online video game scenarios. We have identified current challenges in this area and have built our benchmark to evaluate them. First, we gave the definition of latency of a stateful operator and a methodology to measure it. The solution is lightweight and does not require the use of additional systems. Second, we completely separated the SUTs from the driver. Third, we introduced a simple and novel technique to conduct experiments with the highest sustainable workloads. We conducted extensive experiments with the three major distributed, open-source stream processing engines - Apache Storm, Apache Spark, and Apache Flink. In the experiments, we observed that each system has specific advantages and challenges. We provided a set of rules in our discussion part that can be used as a guideline to determine the SPE choice based on requirements for a use-case. Based on our experiences throughout this work, we will explore one of the main limitations of modern SPEs - ad-hoc query processing and sharing - in the next two chapters.

# 4

## AStream: Ad-hoc Shared Stream Processing

This Chapter contains:

|         |   |    |
|---------|---|----|
| 4.1     | Introduction . . . . .  | 43 |
| 4.1.1   | Motivating Example . . . . .  | 43 |
| 4.1.2   | Ad-hoc Stream Requirements . . . . .                                      | 44 |
| 4.1.2.1 | Integration . . . . .   | 44 |
| 4.1.2.2 | Consistency . . . . .   | 44 |
| 4.1.2.3 | Performance . . . . .   | 44 |
| 4.1.3   | AStream . . . . .   | 44 |
| 4.1.4   | Sharing Limitations in State-of-the-Art Data Processing Systems . . . . . | 45 |
| 4.1.5   | Contributions and Chapter Organization . . . . .                          | 45 |
| 4.2     | System Overview . . . . .   | 46 |
| 4.2.1   | Data Model . . . . .  | 46 |
| 4.2.1.1 | Query-set . . . . .   | 46 |
| 4.2.1.2 | Changelog . . . . .   | 47 |
| 4.3     | Implementation Details . . . . .  | 48 |
| 4.3.1   | Ad-hoc Operators . . . . .  | 48 |
| 4.3.1.1 | Shared Session . . . . .  | 48 |
| 4.3.1.2 | Shared Selection . . . . .  | 48 |
| 4.3.1.3 | Window Slicing . . . . .  | 50 |
| 4.3.1.4 | Shared Join . . . . .   | 50 |
| 4.3.1.5 | Shared Aggregation . . . . .  | 50 |
| 4.3.1.6 | Router . . . . .  | 51 |
| 4.3.2   | Optimizations . . . . .   | 51 |
| 4.3.2.1 | Incremental Query Processing . . . . .                                    | 51 |
| 4.3.2.2 | Data Copy and Shuffling . . . . .   | 51 |
| 4.3.2.3 | Memory Efficient Dynamic Slice Data Structure . . . . .                   | 51 |
| 4.3.2.4 | Changelog-set Size . . . . .  | 51 |

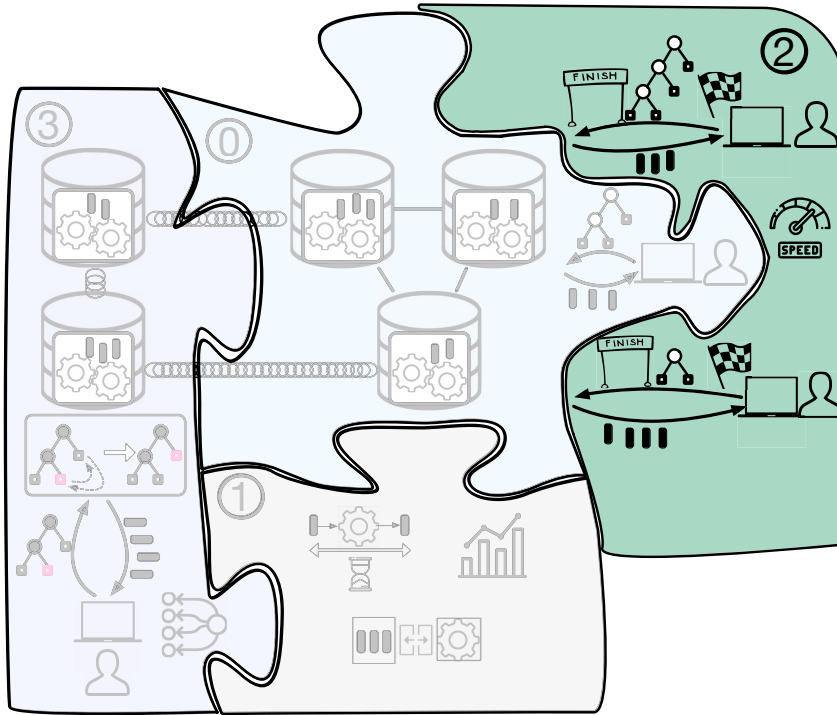


Figure 4.1: Scope of Chapter 4: Ad-hoc Shared Stream Processing

|         |   |    |
|---------|---|----|
| 4.3.3   | Exactly-once Semantics . . . . .                | 51 |
| 4.3.4   | QoS . . . . .                                   | 52 |
| 4.4     | Experiments . . . . .                           | 52 |
| 4.4.1   | Experimental Design . . . . .                   | 52 |
| 4.4.2   | Generators . . . . .                            | 53 |
| 4.4.2.1 | Data Generation . . . . .                       | 53 |
| 4.4.2.2 | Selection Predicate Generation . . . . .        | 53 |
| 4.4.2.3 | Join and Aggregation Query Generation . . . . . | 53 |
| 4.4.3   | Metrics . . . . .                               | 53 |
| 4.4.4   | Setup . . . . .                                 | 54 |
| 4.4.4.1 | Workloads . . . . .                             | 54 |
| 4.4.5   | Workload Scenario 1 . . . . .                   | 54 |
| 4.4.6   | Workload Scenario 2 . . . . .                   | 56 |
| 4.4.7   | Complex Queries . . . . .                       | 57 |
| 4.4.8   | Sharing Overhead . . . . .                      | 58 |
| 4.4.9   | Discussion . . . . .                            | 60 |
| 4.5     | Integration . . . . .                           | 60 |
| 4.6     | Related Work . . . . .                          | 61 |
| 4.6.1   | Query-at-a-time Processing . . . . .            | 61 |
| 4.6.2   | Stream Multi-query Optimization . . . . .       | 61 |
| 4.6.3   | Adaptive Query Optimization . . . . .           | 62 |
| 4.6.4   | Batch Ad-hoc Query Processing Systems . . . . . | 62 |
| 4.6.5   | Stream Query Sharing . . . . .                  | 63 |
| 4.7     | Conclusion . . . . .                            | 63 |

In the last decade, many SPEs were developed to perform continuous queries on massive online data. The central design principle of these engines is to handle queries that potentially run continuously on data streams with a query-at-a-time model, i.e., each query is optimized and executed separately. In many real applications, streams are not only processed with long-running queries, but also thousands of short-running ad-hoc queries. To support this efficiently, it is essential to share resources and computation for ad-hoc stream queries in a multi-user environment.

The goal of this chapter is to bridge the gap between stream processing and ad-hoc queries in SPEs by sharing computation and resources. We define three main requirements for ad-hoc shared stream processing: (1) *Integration*: Ad-hoc query processing should be a composable layer which can extend stream operators, such as join, aggregation, and window operators; (2) *Consistency*: Ad-hoc query creation and deletion must be performed in a consistent manner (i.e., ensure exactly-once semantics and correctness); (3) *Performance*: In contrast to state-of-the-art SPEs, ad-hoc SPEs should not only maximize data throughput but also query throughput via incremental computation and resource sharing.

Based on these requirements, we have developed AStream, an ad-hoc, shared computation stream processing framework. To the best of our knowledge, AStream is the first system that supports distributed ad-hoc stream processing. AStream is built on top of Apache Flink. Our experiments show that AStream shows comparable results to Flink for single query deployments and outperforms it by orders of magnitude with multiple queries.

## 4.1 Introduction

Several open source distributed SPEs, such as Apache Spark Streaming [3], Apache Storm [2], Apache Flink [5], and Apache Apex [49], were developed to cope with high-speed data streams from IoT, social media, and Web applications. Large companies with hundreds of developers use SPEs in their production environment. Developers in the production environment create long-running queries for continuous monitoring or reporting and short-lived stream queries for testing on live streams. The best practice today is to fork the input stream using a message bus like Apache Kafka [50] while adding additional resources for performing new queries [44]. Hundreds of developers, creating thousands of ad-hoc queries, make this a challenging and inefficient setup.

### 4.1.1 Motivating Example

Typical examples for stream processing setups are online services such as games. Online gaming today is often cloud-based to satisfy varying user demands. Gaming companies have to provide a flawless gaming experience to ensure customer satisfaction for millions of concurrent users. According to Tencent [51], the company which owns the most played online game - PUBG, more than half of the company's employees, around 23 thousand, work in research and development departments. These researchers create many ad-hoc stream queries to analyze the most relevant streams in the company.

Figure 4.2 shows a sample use-case of ad-hoc stream queries. In this example, there are two input streams: 1) a stream of advertisements, presented to players during the game, and 2) a purchases stream, which contains purchases of game packs. There are three queries in the figure, the marketing team in Europe submits a short-lived query, Q1, and after getting enough information, the query is shut down. The user experience team initiates a long-living query Q2 to monitor the behavior of users under 18. Query Q3 is a session-based query created and deleted by the system to monitor the loyalty of the pro-level users. It is common to hire pro-level players to a tester position, as they can reveal bugs in a game (e.g., missing or wrong sound effects, crashes, and corruption of graphics).

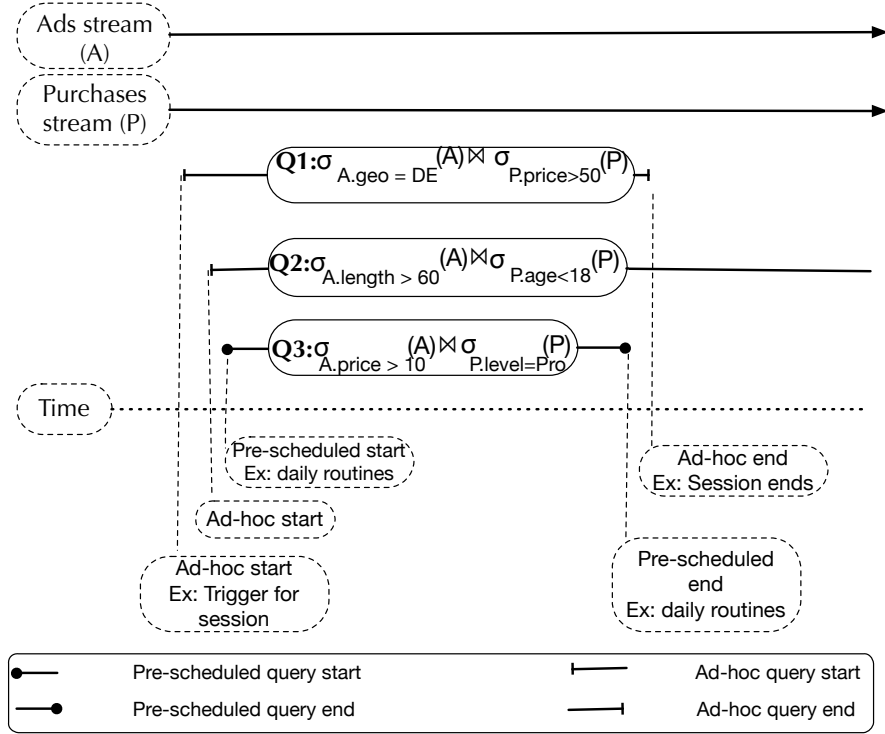


Figure 4.2: Ad-hoc stream queries in online gaming scenarios

### 4.1.2 Ad-hoc Stream Requirements

We identify three main requirements for ad-hoc stream query processing.

#### 4.1.2.1 Integration

SPEs should integrate ad-hoc query support by extending stateful operators, such as window operators with different types and configurations, aggregation, join, and stateless operators, such as filters. This enables users to issue ad-hoc queries while profiting from built-in features of SPEs, such as out-of-order stream processing, event-time processing, and fault tolerance.

#### 4.1.2.2 Consistency

An ad-hoc SPE executes multiple queries and serves multiple users or tenants. When removing existing queries and adding new queries to the system workload, an ad-hoc SPE must handle old and new queries in a consistent way, ensuring exactly-once semantics and the correctness of the results.

#### 4.1.2.3 Performance

State-of-the-art distributed SPEs focus on maximizing the data throughput and minimizing the latency. Several well-known stream benchmarks, such as the Yahoo streaming benchmark [26], StreamBench [41], and Nexmark [52] test systems based on these metrics. Ad-hoc SPEs, in addition to the performance metrics above, need to sustain a high query throughput. The performance of such systems is boosted not only by incremental computation and resource sharing, but also by avoiding redundant computation.

### 4.1.3 AStream

We propose AStream, an ad-hoc shared-computation stream processing framework, which can handle hundreds of ad-hoc stream queries. We design AStream based on the requirements mentioned above:



(1) AStream extends a wide set of components of an existing SPE, Apache Flink, but it is not tightly coupled with it. AStream supports a wide set of use-cases, windowed joins, windowed aggregations, selections, with ad-hoc query support. (2) AStream provides consistent query deletion and creation, and ensures the correctness for all running queries in the presence of ad-hoc queries; (3) Our experiments show that AStream achieves a throughput in the order of hundreds of query creations per second and is able to execute in the order of thousands of concurrently running queries. AStream achieves this level of performance through a set of incremental computations and optimizations. AStream features a rule-based optimizer to trade-off sharing benefits and disadvantages. Cost-based multi-query optimization for batch query processing environments relies on existing data statistics and targets static compile-time optimizations [53]. In streaming environments, there is typically no prior information about data statistics and workloads. Therefore, we propose a simple, robust, and dynamic rule-based optimizer.

#### 4.1.4 Sharing Limitations in State-of-the-Art Data Processing Systems

Distributed stream engines are mostly designed for a query-at-a-time model and focus on optimizing each query separately. To the best of our knowledge, there is no work on ad-hoc query processing for distributed streaming systems where new queries can join the stream processing system, and others leave the system. Forking the input stream for every new query results in significant overhead. Additional resource reservations, the starting and stopping of the new query (which might be negligible for long-running stream queries but significant for ad-hoc short queries), and running new instances of the streaming engine contribute to this overhead.

Workload sharing is a well-studied topic in the context of batch data processing systems. SharedDB [54] is one representative example for such systems. SharedDB batches user queries, creates a global query plan, and shares computation across them. We adopt some ideas from SharedDB, such as tagging tuples with query IDs to identify different subsets of (possibly computed) relations. If all stream queries are created when the system is deployed and run infinitely, meaning no ad-hocness, then this approach perfectly fits for streaming scenarios. In the presence of ad-hoc queries, however, query sharing happens among queries running on fundamentally different subsets of the data sets, determined by the creation and deletion times of each query.

Also, AStream is able to handle out-of-order stream data and to exploit and share windows of different types and configuration. AStream extends ideas from window panes [55], dynamically divides segments of time into discrete partitions at runtime, and shares overlapping parts among different queries.

We also take into consideration that aggressive work sharing among concurrent queries does not always lead to performance improvements [56]. Therefore, we compute overlapping parts of a window via dynamic programming and share if possible. Lastly, AStream is fault tolerant, all changes to query sets are deterministically replayable, which requires that metadata modifications are deterministically woven into the streams.

In real-world setups, the performance of SPEs depends not only on its throughput and latency for individual queries but also on the overall query throughput. The design of AStream reduces the overheads mentioned above by sharing the execution of queries, avoiding computational duplication, and achieving high query throughput.

#### 4.1.5 Contributions and Chapter Organization

The main contributions of this chapter are as follows:

- We present AStream, the first distributed ad-hoc stream processing framework. AStream is fully functional and supports a wide range of ad-hoc stream queries on shared data streams.

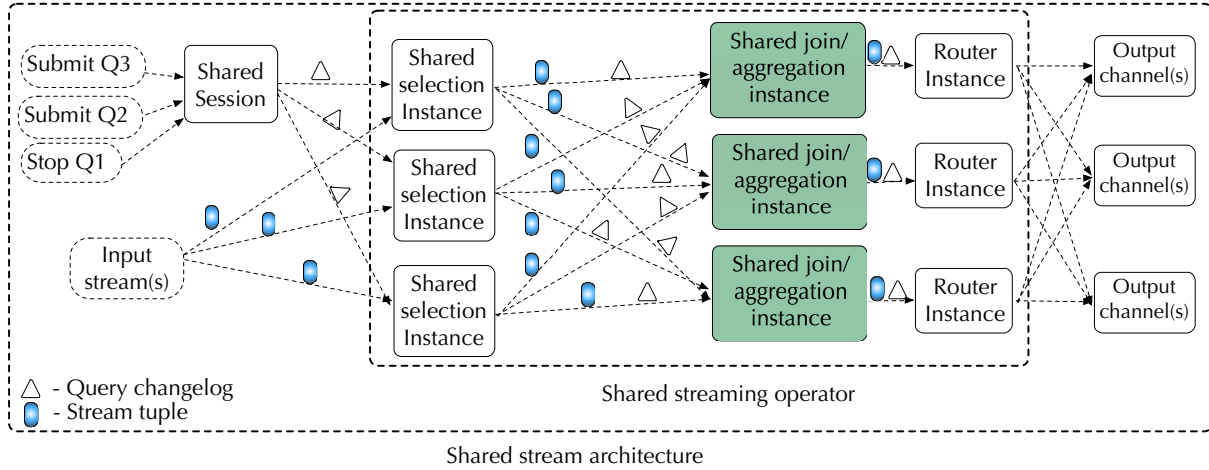


Figure 4.3: AStream architecture

- We provide exactly-once semantics and consistent query creation and deletion for ad-hoc queries.
- We conduct an extensive experimental analysis. AStream shows comparable results to Flink in a single-query deployment and outperforms Flink by orders of magnitude in multi-query deployments.

The rest of this chapter is organized as follows. Section 4.2 describes the system overview. We introduce implementation details in Section 4.3. Section 4.4 shows experimental evaluation. In Section 4.5 we discuss possible integration of AStream components to other SPEs. We discuss related work in Section 4.6 and conclude in Section 4.7.

## 4.2 System Overview

In this section, we describe the architecture of AStream and elaborate on our data models. Figure 4.3 shows the general architecture of AStream. There are four main components. The *shared session* accepts queries from users and submits them to the job manager. The *shared selection*, *aggregation*, and *join operators* process queries in a shared manner. Finally, the *router* sends tuples to their associated query sinks. Our solution supports query sharing for *i*) selection, *ii*) windowed join, *iii*) windowed aggregation, and *iv*) their combination. The main assumption in this work is that operators can be shared as long as they have common upstream operators and common partitioning keys.

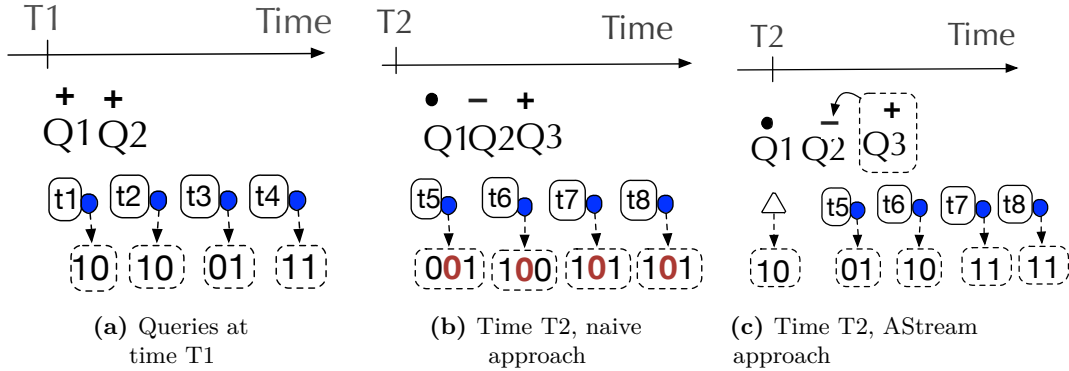
AStream is a framework, which can be integrated to existing SPEs as a separate layer. AStream exploits all the necessary components from an underlying SPE, such as optimizer, scheduler, network layer, and code generators. The main rationale is that we reuse already developed components of distributed data processing systems, which otherwise would require significant engineering work to build from scratch.

### 4.2.1 Data Model

In the following subsection, we describe the data model of AStream.

#### 4.2.1.1 Query-set

AStream extends SharedDB’s data model. To each tuple, we add the set of query IDs, that are potentially interested in a tuple, as an additional column. We call this column **query-set**. Unlike SharedDB we adopt a bitset data representation for the query-set to be able to perform bitset operations. We assume a total numbering of queries in a query-set. We encode queries in a query-set with a bitset data structure.



**Figure 4.4:** AStream and naive data model. At time T1 two new queries are submitted ( $Q1+$ ,  $Q2+$ ). At time T2,  $Q1$  remains running ( $Q1\cdot$ ),  $Q2$  is deleted ( $Q2-$ ),  $Q3$  is created ( $Q3+$ ), and related changelog ( $\Delta$ ) is generated.

For example, a query-set 0010 means that the tuple is relevant only for the query with index 3 ( $Q3$ ). If a tuple is not matching any predicates, meaning all bits of its query-set are zero, then we discard the tuple. In a query-set, each query has a unique index. If a tuple is relevant to  $i$ th query, meaning the tuple matches the selection predicate of the  $i$ th query, then the  $i$ th bit of tuple’s query-set is set.

We compute the intersection of two query-sets through a bitwise AND operation. For any two tuples, we perform a join or aggregation if the tuples share at least one query. This way, we avoid redundant computation. Consider tuples  $t1$ ,  $t2$ ,  $t3$ , and  $t4$  in Figure 4.4a. The bitwise AND of the query-sets of  $t2$  and  $t3$  returns zero, i.e., they do not share any query. However,  $t4$  shares  $Q1$  with  $t2$  and  $t1$ , and  $Q2$  with  $t3$ .

#### 4.2.1.2 Changelog

The above data model works well if stream queries are defined at compile-time and run forever. However, for ad-hoc scenarios, this data model is not enough. For example, when the workload change occurs at time T2 in Figure 4.4, we observe that queries and query-sets before T2 and after T2 are different. In order to perform bitwise operations, we need a consistent query index in all query-sets so that any bitwise operations of tuples, created at different times, is correct. One way to fulfill this requirement is to assign a new index to each new query. We demonstrate this append-only approach in Figure 4.4b. Because  $Q2$  is deleted, its position is permanently zero. So, the new position, 3rd position in the query-set, is assigned to the new query,  $Q3$ . The problem of this approach is that it leads to big and sparse query-sets.

AStream reuses bits of deleted queries for newly created queries in order to keep the changelog-set as compact as possible. If there is no deleted query, we allocate a new position for a new query. We use a **changelog**, a special data structure consisting of *i*) query deletion and creations meta-data and *ii*) a **changelog-set**, a bitset encoding the associated query deletions and creations.

A bit in a changelog-set is set if a query in the respective position remains unchanged. A bit in the changelog-set is unset if a query is deleted or a new query is placed in the respective position. For example, in Figure 4.4c  $Q2$  is deleted and  $Q3$  is created. Because the index of  $Q2$  is empty,  $Q3$  is placed in this position. The associated changelog-set, 10, indicates that the first position in the query-set remains unchanged, but the second position is replaced with another query. Also, Figure 4.5b demonstrates the changelog-sets of the workload shown in Figure 4.5a. At time T5 in Figure 4.5a, there are two new queries ( $Q6$  and  $Q7$ ) and one deleted query ( $Q3$ ). AStream allocates the index of the deleted query to  $Q6$  and provide a new position for  $Q7$ .

By default, we use a changelog-set to indicate query changelogs between two adjacent time slots. For example, changelog-set 100 at time T2 in Figure 4.5b indicates the query changelog with respect to time

slot T1. However, for some operations, we need to perform computations between non-adjacent time slots, such as T3 and T1.

Let  $CL\text{-set}(T_i)$  be the changelog arrived at time slot  $T_i$ ,  $cl\text{-set}(T_i, T_j)$  be the changelog between  $T_i$  and  $T_j$ , and  $f$  be a function that combines multiple changelogs. Equation 4.1 shows the dynamic programming technique to calculate  $cl\text{-set}(T_i, T_j)$ . If  $T_i$  is the same as  $T_j$ , then there is no changelog ( $\emptyset$ ). If  $T_j$  is greater than  $T_i$  by one unit time, then the changelog is already encoded in  $CL\text{-set}(T_j)$ . Otherwise, the function  $f$  is called recursively. The function  $f$  gets two arguments and returns the bitwise AND of them (Equation 4.2). Figure 4.5c shows changelog-sets for each time slot with respect to all previous time slots.

$$cl\text{-set}(T_i, T_j) = \begin{cases} \emptyset & \text{if } T_i=T_j \\ CL\text{-set}(T_j) & \text{if } T_j= T_i+1 \\ f(CL\text{-set}(T_j), cl\text{-set}(T_i, T_j-1)) & \text{otherwise} \end{cases} \quad (4.1)$$

$$f(A, B) = A \& B \quad (4.2)$$

We use changelog-sets to ensure consistency and correctness in any operation among tuples. Also, we avoid redundant computation by finding only overlapping queries among different time slots. If two time slots share queries, meaning changelog-set is non-zero, then we filter tuples by performing bitwise AND between tuples' query-sets and the changelog-set. For example, assume that we perform a join operation between the tuples created before T2 and after T2, as shown in Figure 4.4.  $t_5$  is filtered, because the bitwise AND of the  $t_5$  query-set, 01, and the changelog-set, 10, is zero. As another example, joining  $t_7$  and  $t_4$  would result in the tuple with query-set 10 ( $10 \& 11 \& 11$ ), meaning the resulting tuple matches Q1.

### 4.3 Implementation Details

In this section, we first explain the implementation of ad-hoc operators (Section 4.3.1) and optimization techniques adopted by AStream (Section 4.3.2). We elaborate on fault tolerance in Section 4.3.3 and QoS features in Section 4.3.4

#### 4.3.1 Ad-hoc Operators

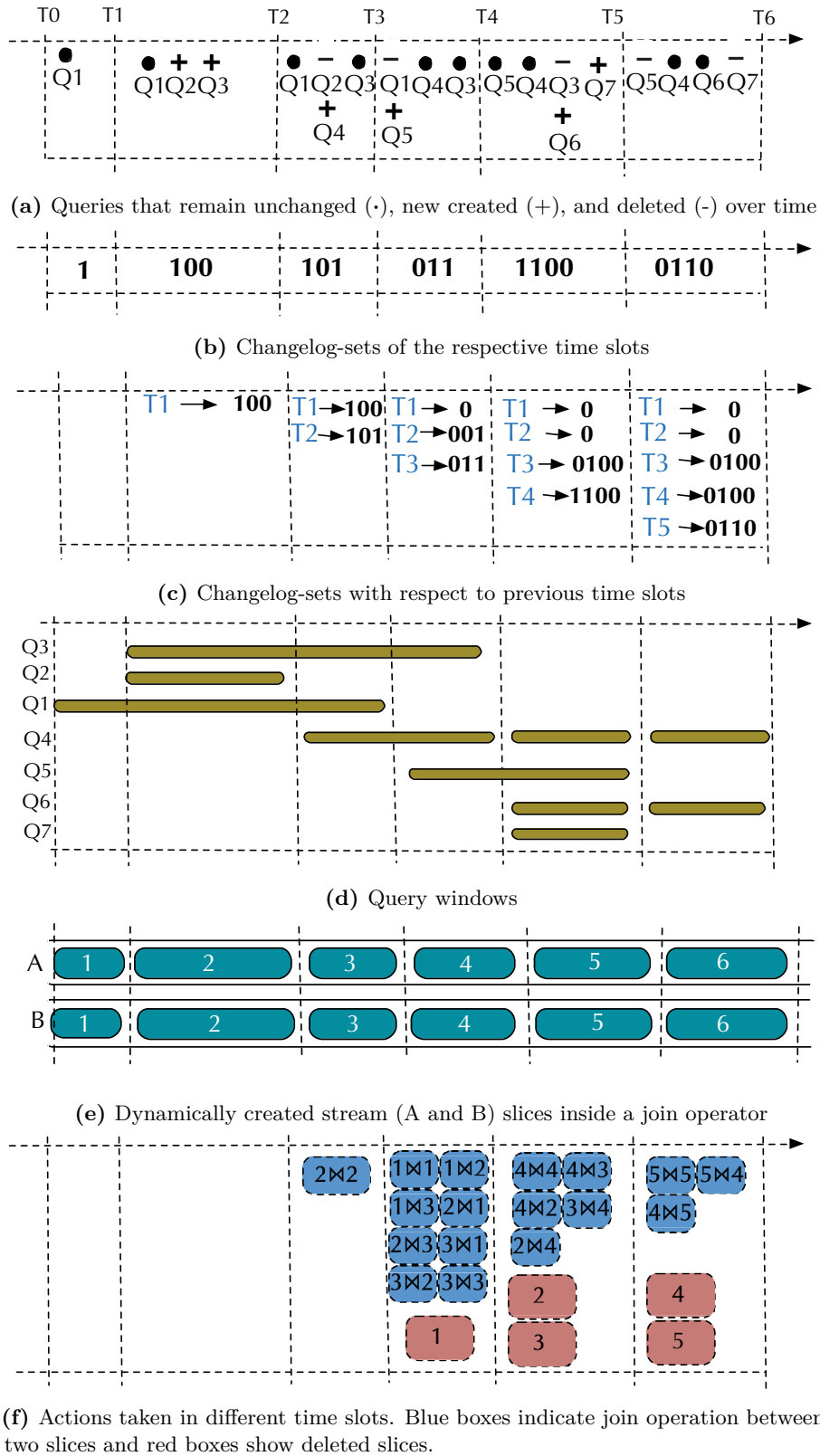
Each operator in AStream keeps a list of active queries. Once active queries are updated with the changelog, operators change their computation logic accordingly.

##### 4.3.1.1 Shared Session

The *shared session* is a client module of AStream. The shared session batches user query requests and generates a changelog. A changelog is generated for every batch-size (number of user requests) or once the maximum timeout is reached. If there is no user request, no changelog is generated.

##### 4.3.1.2 Shared Selection

The shared selection operator computes the query-set for each tuple and appends the resulting query-set to the tuple as a separate column. The shared selection maintains the set of active queries. It updates the set once it receives a changelog.



**Figure 4.5:** End-to-end ad-hoc query example. Ad-hoc queries (Figure 4.5a) with various window configurations (Figure 4.5d) are submitted. Their related changelog-sets are generated in Figure 4.5b. All figures share the same x-axis.

### 4.3.1.3 Window Slicing

AStream supports time- and session-based windows with different characteristics (e.g., length, slide, gap). For queries involving window operators, such as windowed aggregation and windowed join, AStream divides overlapping windows into disjoint **slices**. It performs operations among overlapping slices once and reuses the result for multiple queries. The core of this idea is from window ID representation of events and panes [57], and sharing computation among panes [55]. The core difference between panes and slices is that, the former computes panes in compile-time, while the latter computes slices in runtime based on ad-hoc queries and their corresponding windows. The lengths of slices in Figure 4.5e are determined at runtime based on the created and deleted queries shown in Figure 4.5d. Once a query changelog arrives, its changelog-set is assigned to the corresponding window slice. Also, the set of running queries inside the shared join operator gets updated.

### 4.3.1.4 Shared Join

AStream executes join operations incrementally by joining slices and combining intermediate results. It joins overlapping slices once and reuses the intermediate results. For each slice, AStream keeps a computation history. Based on this information, it avoids unnecessary computation among slices and performs delta query processing. Consider the join operation in Figure 4.5f. At time T2, evaluation of Q2 triggers, and join results are emitted. At time T4, Q1 is evaluated. Note that AStream avoids joining already joined slices (slice-2  $\times$  slice-2). Also, the first slice is deleted, as it is no longer needed. Similarly, at time T5, AStream joins slices once and reuses them for multiple overlapping query windows (Q4, Q5, Q6, Q7).

We join two slices as follows: We group tuples in each slice by their query-sets. First, we check the query-set groups, e.g., G1 in slice 1 and G2 in slice 2. We join tuples residing in G1 and G2, if the tuples residing in these groups share at least one query. For example, if G1=010 and G2=\*0\*, then tuples residing in these groups are never joined.

Grouping tuples inside slices enables sharing tuples on-the-fly. The disadvantage of this method is that the number of possible tuple groups increases exponentially with the number of queries. In early experiments, we noticed that for more than ten concurrent queries, storing tuples as a list is more efficient than storing them inside groups. The number of tuples in tuple groups decreases sharply as the number of tuple groups increases. Therefore, retrieving a tuple group via an index lookup is less beneficial than performing a sequential scan.

For switching between a group and a list data structure, we use the following heuristic. As the number of queries increases, we monitor the average size of tuple groups inside slices. If the average is less than two, meaning most of the tuple groups contain only a single tuple, then we switch to a list data structure.

### 4.3.1.5 Shared Aggregation

The shared aggregation works similar to the shared join. One difference is that the shared join is a binary stream operator (has two input streams), but the shared aggregation is a unary stream operator.

In the shared aggregation, each window slice keeps intermediate aggregation results for all active queries. Instead of materializing input tuples, we update the query intermediate aggregation results for each new tuple. Then, we discard the tuple. For example, a tuple with the query-set 101 is aggregated with intermediate aggregation results of Q1 and Q3 and discarded afterwards. Aggregation between two different slices is also performed in a similar way.

#### 4.3.1.6 Router

The router is another component of AStream. The routing information for each tuple is encoded in its query-set. The router sends each tuple to either query output channels or to downstream operators.

### 4.3.2 Optimizations

AStream uses several optimizations to speed up query processing.

#### 4.3.2.1 Incremental Query Processing

Incremental query processing is a core feature of AStream. As shown in Sections 4.3.1.4 and 4.3.1.5, AStream computes both ad-hoc stream aggregations and joins in an incremental manner.

#### 4.3.2.2 Data Copy and Shuffling

AStream avoids data copy in all its components except for the router. The router avoids data copy if the downstream operator is a shared join or aggregation operator. The query-set attribute in each tuple enables us to avoid data copy. The router performs data copy only if the downstream operator is a sink operator, in which the router has to ship results to different query channels.

AStream also avoids redundant data shuffling by encoding a query-set for each tuple. When running a single query, this has some performance overhead, but for multiple queries, the overhead is outweighed by the performance improvements. The shared aggregation and join operators avoid data copy inside slices. Each tuple is saved only once inside a slice.

#### 4.3.2.3 Memory Efficient Dynamic Slice Data Structure

The shared join operator adapts the data structure based on the workload. If the number of active queries exceeds a threshold, the shared session sends a marker to downstream operators. Once the marker is received, the shared join operator changes the data structure of all slices and resumes its computation.

#### 4.3.2.4 Changelog-set Size

After a query is deleted, AStream reserves its position for a future query. This query position becomes zero after the query is deleted. If no new queries are submitted, then each tuple would carry unnecessary bits in their query-sets. For example, in Figure 4.5b if no new queries are submitted after T6, then each tuple would carry two unnecessary bits, two zeros. We handle this issue via changelog-set compression. If we detect this behavior for some time, then the shared session sends a marker to downstream operators, informing them about the changelog-set compression.

### 4.3.3 Exactly-once Semantics

Exactly-once semantics for SPEs ensure that every input tuple is only processed once, even under failures. Operators in AStream are exactly-once, as long as the underlying distributed streaming architecture supports exactly-once semantics, as systems like Kafka-streams [58], Spark Structured Streaming [4], and Apache Flink [5] do. A Stream requires that both tuples and changelog markers and the state of shared operators are deterministically reproducible by logging the input stream and checkpointing [59].

AStream is deterministic because all its distributed components are deterministic and they are based on event-time semantics. Event-time is the time at which an event was produced; e.g., the time an ad is clicked (for tuples) or the time a query is deleted (for changelogs). Event-time semantics ensure the correctness of out-of-order events because the notion of time depends on the data, not on the system clock.

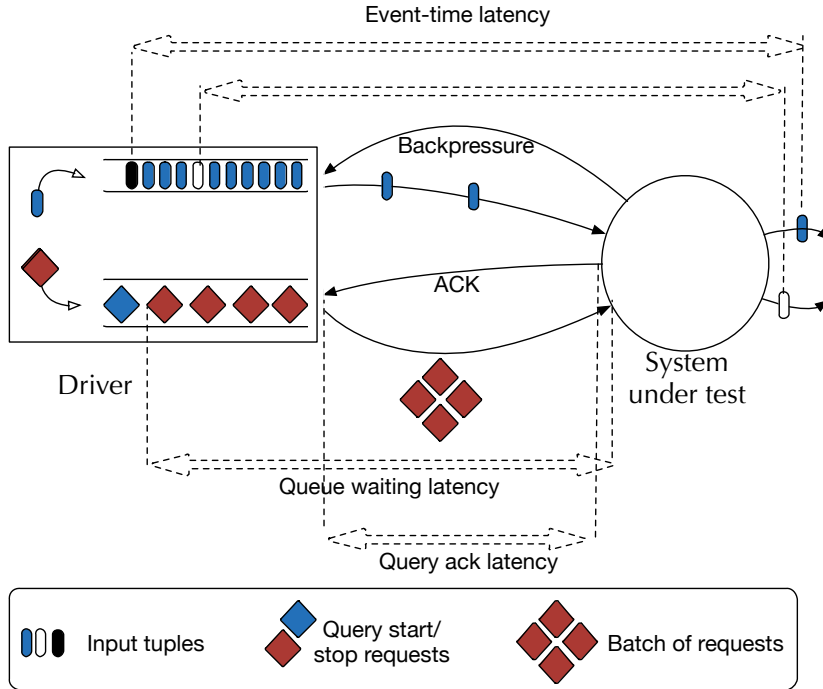


Figure 4.6: Design of the driver for the experimental analysis

In event-time stream processing, tuples are assigned to windows based on their event-time [57, 17]. In the case of a failure, a replayed event is assigned to the same window ID, as the window ID computation is also deterministic [57, 17]. Our slicing technique (Figure 4.5e) is also deterministic. The length of slices depends on changelogs. The changelogs also use event-time, which is the time at which query changes were performed by users.

#### 4.3.4 QoS

Controlling the performance impact of a new query on existing queries is essential to ensure the quality of service in a multi-query environment. In ad-hoc stream workloads, QoS should be ensured in many ways, such as individual query throughput, overall query throughput, data throughput, data latency, and query deployment latency. For example, for data latency, we extend the latency metric implementation of Flink [60]. To be more specific, in the sink operator of every query, we periodically select a random tuple and measure the end-to-end latency. The latency results are collected in the job manager. Also, we show in our experiments (Section 4.4.8) the impact of newly created or deleted queries on existing queries. AStream is capable of providing the above-mentioned metrics to an external component. If measurements for a particular metric are beyond acceptable boundaries, new resources can be added. We discuss elastic scaling in Chapter 5.

## 4.4 Experiments

### 4.4.1 Experimental Design

To evaluate AStream, we simulate a multi-tenant environment with ad-hoc queries. We use a parallel and distributed driver and conduct experiments with two SUTs: Apache Flink [5] (v 1.5.2) and AStream, which we implement on top of Flink.



```

1 SELECT *
2 FROM A, B [RANGE [VAL1]] [SLICE [VAL2]]
3 WHERE A.KEY = B.KEY AND
4     A.[VAL5] [=|>|<|>=|<=] [VAL3] AND
5     B.[VAL6] [=|>|<|>=|<=] [VAL4]

```

**Figure 4.7:** Join query template.  $VAL_n$  is a random number,  $VAL_5$  and  $VAL_6$  are less than  $|fields|=5$

```

1 SELECT SUM(A.FIELD1)
2 FROM A [RANGE [VAL1]] [SLICE [VAL2]]
3 WHERE A.[VAL4] [=|>|<|>=|<=] [VAL3]
4 GROUPBY A.KEY

```

**Figure 4.8:** Aggregation query template.  $VAL_n$  is a random number,  $VAL_4$  is less than  $|fields|=5$

We extend the benchmarking suite described in Chapter 3 to generate queries in addition to tuples. As shown in Figure 4.6, our driver maintains two FIFO queues: user requests, i.e., query creation or query deletions and input tuples. Periodically, the driver pops user requests from the FIFO queue, sends them to a SUT, and waits for the acknowledgement message (ACK) from the SUT. The driver submits the next set of user requests to the SUT if the SUT ACKs the previous batch. This way, we implement a backpressure mechanism for user query requests. The longer the user request stays in the queue, the higher is its deployment latency.

## 4.4.2 Generators

### 4.4.2.1 Data Generation

Each generated input tuple has 6 fields: a *key* field and an array of size 5, named *fields*. Each subsequent tuple is generated with key in the form  $key \leftarrow (key+1) \bmod \text{MAX\_KEY}$ . This way, we balance the data distribution among different partitions. The other fields are generated in a random manner,  $fields[i] \leftarrow \text{random}(0, fields_{max})$ .

### 4.4.2.2 Selection Predicate Generation

To generate a selection predicate, we select a random *field* of a tuple ( $field[i]$ ), generate a random number ( $VAL$ ), select a random binary operator:  $<$ ,  $>$ ,  $=$ ,  $\leq$ , or  $\geq$  ( $o_i$ ), and combine them to a selection predicate ( $o_i(field[i], VAL)$ ).

### 4.4.2.3 Join and Aggregation Query Generation

The join and aggregation query generation consists of two parts: selection predicate generation (see above) and window generation. We generate window length as  $\text{random}(1, window_{max})$  and slide as  $\text{random}(1, length)$ . For session windows, window length and slide are not needed. Figures 5.17 and 4.8 show the query templates for join and aggregation queries. Line 4-5 in Figure 5.17 and Line 3 in Figure 4.8 show selection predicates. For join queries, both input streams have different selection predicates.

## 4.4.3 Metrics

Basic metrics to evaluate SPEs are event-time latency and sustainable throughput [61]. In addition to these, we propose several metrics for ad-hoc streaming environments. **Query deployment latency** is the time duration between a user request to create or delete a query and the actual query start time. For

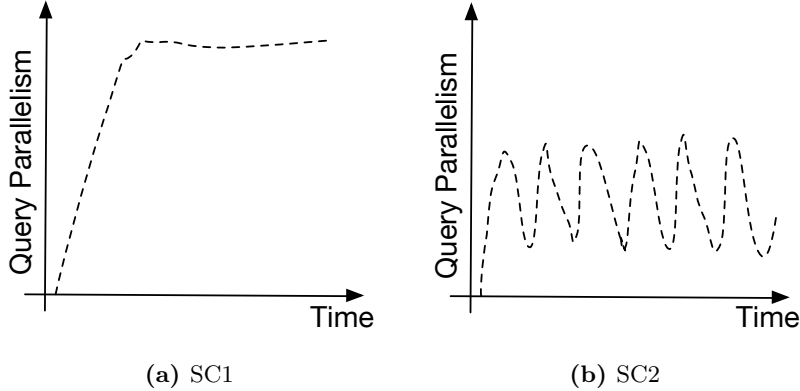


Figure 4.9: Two scenarios for ad-hoc query processing environments

data throughput evaluations, there are two main metrics to consider. **Slowest data throughput** is the minimum sustainable throughput among active stream queries in an ad-hoc environment. This metric is useful for a service or cloud owner, to ensure minimum QoS requirements. **Overall data throughput** is the sum of throughputs of all active queries. **Query throughput** is the highest load of query traffic (query deletion and creation) a system can handle with sustainable query deployment latency and input throughput.

#### 4.4.4 Setup

We conduct experiments in 4- and 8-node cluster configurations. Each node has 16-core Intel Xeon CPU (E5620 2.40GHz) and 48 GB main memory. The data generator produces data with 1000 distinct keys (uniform distribution). If a SUT throws an exception or error while stopping or starting a streaming job or processing submitted queries in an ad-hoc manner (possibly with high frequency), then we consider this as a failure, meaning the SUT cannot sustain the given workload. We repeat our experiments three times and let it run for thousand seconds. For a changelog generation, we tried several combinations of batch-size and maximum timeout configurations. We configure the batch-size to be one hundred and maximum timeout to be one second, as these configurations are the most suitable for our workloads.

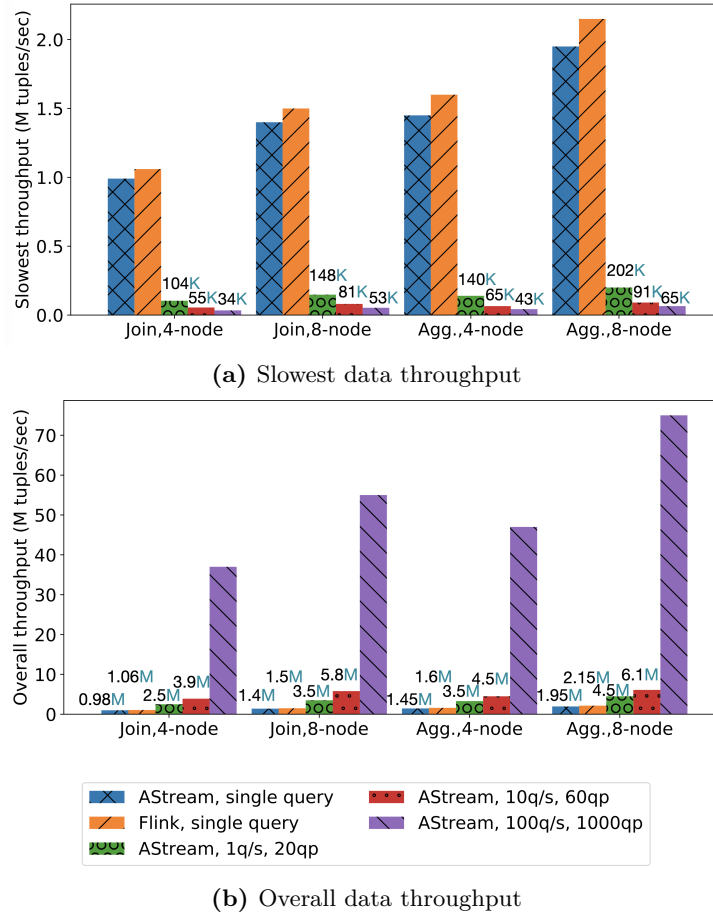
##### 4.4.4.1 Workloads

In Figure 4.9, we show two workload scenarios to evaluate AStream. The main characteristics of the first workload scenario (SC1) are *i*) many users, which leads to many parallel queries, *ii*) few queries that are stopped or changed, resulting in mostly long-running streaming jobs, and *iii*) no new ad-hoc queries after some time. The main characteristics of the second workload scenario (SC2) are *i*) high query throughput, i.e., many queries are created or deleted *ii*) low query parallelism, and *iii*) short-running queries.

#### 4.4.5 Workload Scenario 1

Figure 4.10 shows data throughput for SC1, 4- and 8-node cluster configurations.  $n q/s m qp$  indicates  $n$  queries per second until  $m$  active queries. For a single-query deployment in Figure 4.10a, Flink outperforms AStream. Although query-set generation and bitset operations come with a cost, AStream’s single-query deployment still exhibits a comparable performance to Flink. Flink cannot sustain ad-hoc workloads in Figure 4.10. In each run, it either throws an exception or exhibits very high latency.

In Figure 4.10b there is a sharp increase in the overall throughput of served queries. AStream achieves a better throughput with more ad-hoc queries. However, this performance increase comes with a cost.



**Figure 4.10:** Slowest and overall data throughputs for SC1, 4- and 8-node cluster configurations.  $n$   $q/s$   $m$   $qp$  indicates  $n$  queries per second until  $m$  query parallelism

In Figure 4.10a we see that there is a decrease in the slowest throughput because the number of served queries increases from one query to thousand queries.

In Figure 4.10a we observe a sharp decrease in the throughput from the single query workload to the 1 q/s 20 qp workload. As the query parallelism increases (10 q/s 60 qp and 100 q/s 1000 qp), the decrease in the throughput remains steady. The main reason is that, as the number of queries increases, the probability of sharing a tuple among different queries also increases. As a result, the slowest data throughput decreases less with more queries.

We observe several differences between join and aggregation query performances in Figure 4.10. First, data throughput for join queries is less than for aggregation queries, because joins are computationally more expensive than aggregation in our setup. Second, the performance gap between Flink and AStream is larger for aggregation queries than for join queries. The main reason is that Flink has a built-in support for on-the-fly and incremental aggregation. In contrast, windowed join queries in Flink lack those features.

Figure 4.11a shows the query deployment latency for SC1. The changelog batch-size also has a contribution to the overall latency. For example, 1 q/s, 20 qp has more query deployment latency than 100q/s, 1000qp, as the former has 20 ( $\frac{20}{1}$ ) different query changelog generations, while the latter contains 10 ( $\frac{1000}{100}$ ) different query changelog generations.

Figure 4.12 shows query deployment latency for SC1 (1 q/s, 20 qp). Because Flink cannot sustain this workload, query deployment latency keeps increasing, which is why we do not show this case in Figure 4.10. The longer the query stays in the queue waiting for ACK, the higher is its deployment latency.

#### 4. AStream: Ad-hoc Shared Stream Processing

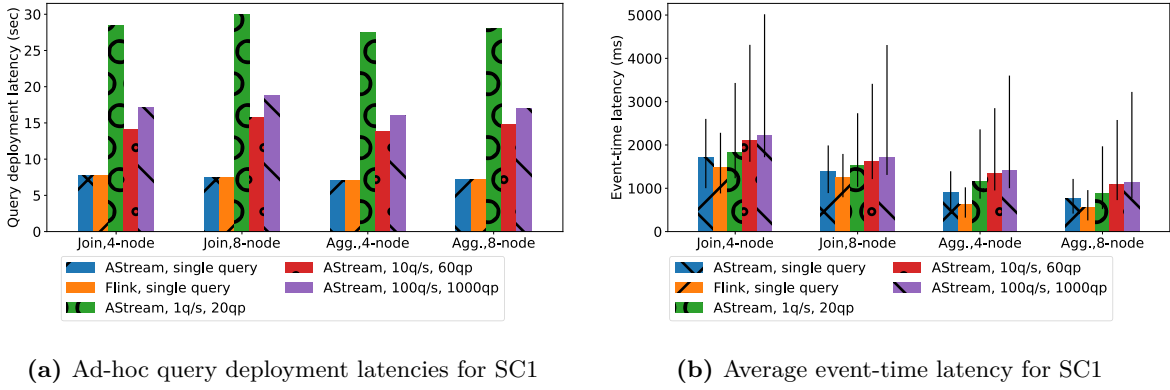


Figure 4.11: AStream performance for SC1

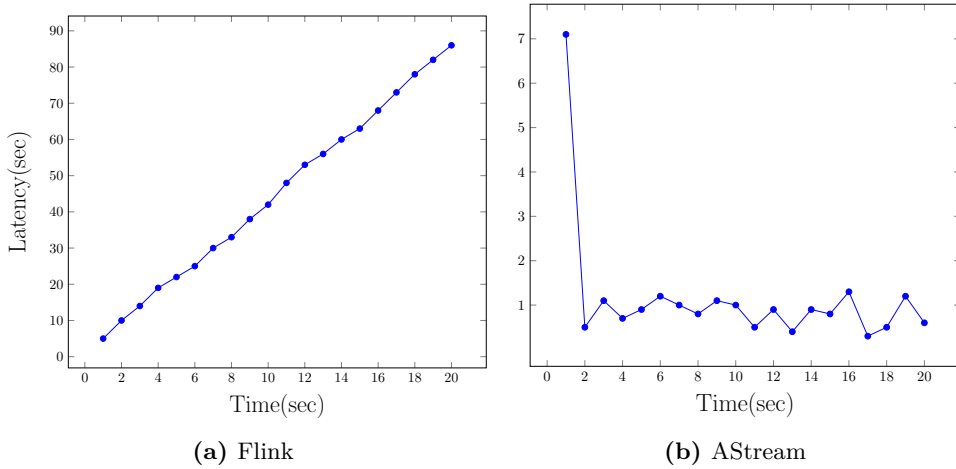


Figure 4.12: Query deployment latency, one query per second, up to 20 queries

For example, the sum of all query deployment latencies for Flink is 910 seconds. In general, the query deployment latency is already high and will be a bottleneck in a multi-tenant environment.

In Figure 4.12 AStream initially exhibits high query deployment latency, because the first query deployment also involves the physical deployment of operators to the cluster nodes, which is time-consuming. Even for batch ad-hoc data processing systems with a dedicated scheduler and optimizer, such as DataPath [62], the first deployment of physical operators is time-consuming. AStream avoids deploying a new streaming topology for each query. Instead, it creates and deletes user queries on-the-fly without affecting the running topology.

Figure 4.11b presents the average event-time latency for streaming tuples. We note that event-time latency for shared aggregation queries is lower than shared join queries because joins are computationally more expensive than aggregations. Throughout our experiments, we observed Flink’s event-time latency for ad-hoc workloads to be higher than eight seconds. As experiments continued, the latency kept increasing, which means the system cannot sustain the given workload.

In Figure 4.11b we notice that event-time latency increases for higher query parallelism for AStream. However, the given latency measurements for AStream are sustainable. Also, the measurements do not exhibit continuous backpressure.

#### 4.4.6 Workload Scenario 2

As explained above, SC2 features a more fluctuating workload than SC1. In this case, an efficient and incremental query sharing is needed to sustain possible churn in the workload.

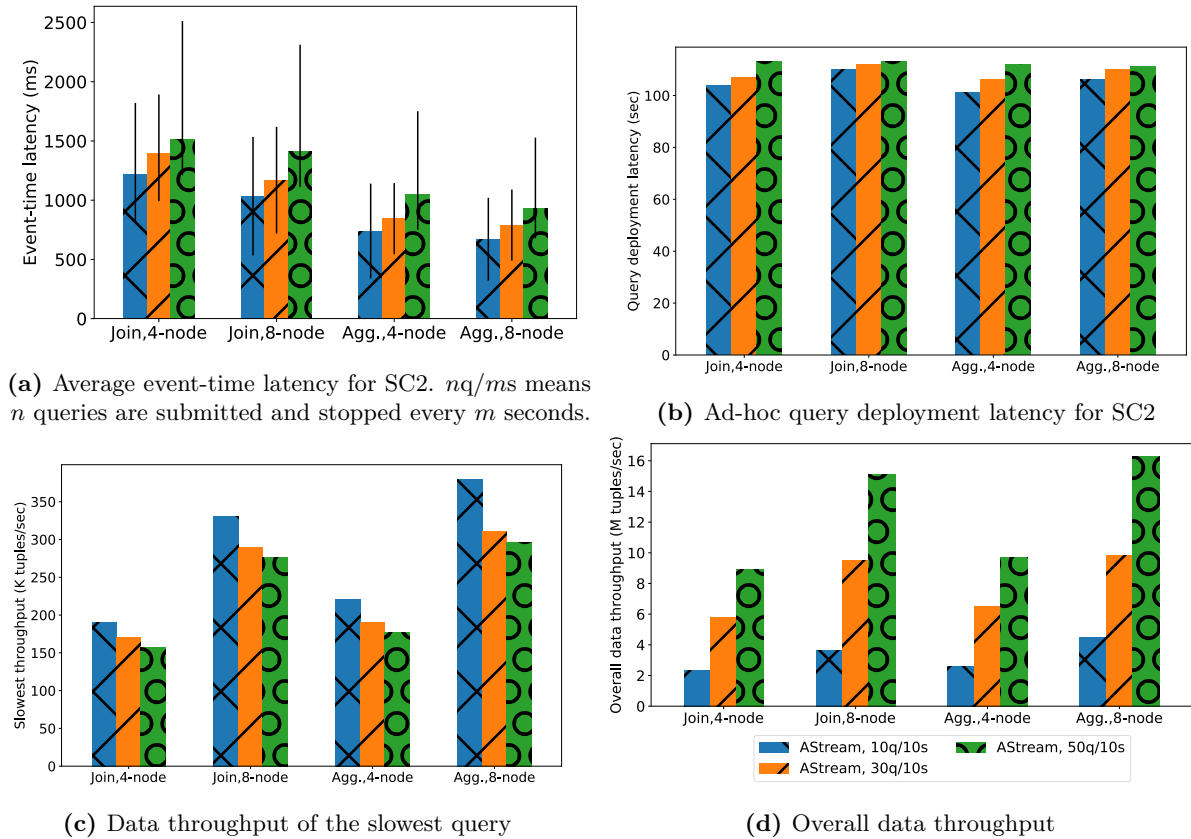


Figure 4.13: AStream performance for SC2

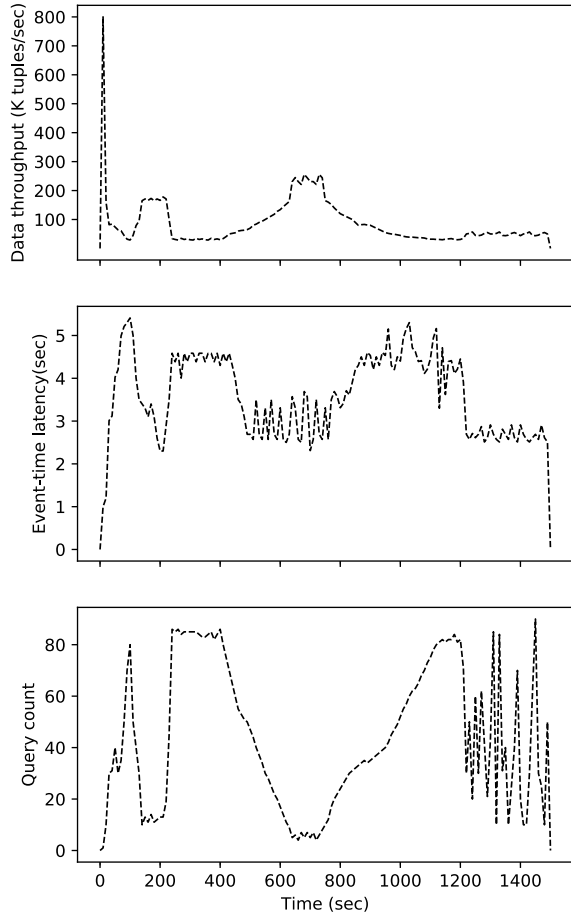
Figure 4.13a shows the average event-time latency for SC2. We notice that event-time latency in SC2 is lower than SC1 (Figure 4.11b). The reason is that in SC2 the query workload is highly changing, but does not increase continuously. So, the majority of the queries running in SC2 are short-running queries.

Figures 4.13c and 4.13d show data throughput for SC2.  $nq/ms$  indicates  $n$  queries are submitted and stopped every  $m$  seconds. Although SC2 exhibits high query fluctuations, the slowest data throughput in SC2 is higher than the one in SC1 (Figure 4.10), which means that AStream works better in more fluctuating workloads. The main reason is that the workload in SC2 is more fluctuating, queries are short-running, and constantly changing; as a result, *i*) the overall number of active queries is less than in SC1 and *ii*) bitset size is less than in SC1. In our experiments, we observe that Flink cannot sustain ad-hoc workloads. For example, for the setup 10q/10s, the input data throughput of AStream was at least  $10\times$  higher than Flink's, before we stopped the experiment.

Figure 4.13b shows the ad-hoc query deployment latency for SC2. We run this experiment for thousand seconds. When we compare the query deployment latency of SC1 and SC2, the latter is significantly higher. The reason is that in SC2, we continuously create and delete queries, while in SC1 we create queries up to predefined query parallelism. Continuously creating and deleting queries results in continuous query changelog generation.

#### 4.4.7 Complex Queries

In this section, we conduct experiments with complex queries, consisting of multiple joins and an aggregation. We generate complex queries by randomly pipelining a selection predicate,  $n$ -ary windowed joins, where  $1 \leq n \leq 5$ , and a windowed aggregation operator. Any complex query involves at least one selection predicate, one windowed join query, and one windowed aggregation query.



**Figure 4.14:** Slowest data throughput (upper), event-time latency (middle), and query count graphs (bottom) for complex ad-hoc queries, with the same  $x$  axis values

Figure 4.14 shows the input data throughput (upper), input latency (middle), and query count graphs (bottom) for complex concurrent queries. We test three cases in this experiment. First, we perform a sharp query throughput increase at timestamps 50 and 200. Second, we gradually decrease query throughput and gradually increase, from time 410 to 1140. Third, we fluctuate query throughput after time 1200.

When we increase query throughput sharply, we notice that the input data latency stays relatively stable. The reason is that we adopt shared streaming operators and do not change the query execution plan, which would cause high latencies. The slowest data throughput drops as we increase query throughput. Also, we notice that in case of fluctuations in query throughput, both slowest data throughput and event-time latency remains stable.

#### 4.4.8 Sharing Overhead

Figure 4.15 shows slowest data throughput for different query parallelism. Similar to Figure 4.10 we note that slowest throughput decreases as query parallelism increases. As the number of queries increases, sharing a tuple among different queries is more probable; as a result, the slope of the figure decreases slowly with increasing query parallelism.

Adding ad-hoc support to an SPE incurs an overhead. We measure this overhead by comparing AStream with Flink. In our experiments, we see that Flink cannot sustain ad-hoc workloads. Conducting ad-hoc experiments with Flink resulted either in an exception or in ever-increasing latency. The main reason is that Flink is not designed for ad-hoc workloads. Therefore, we can only see the overhead of

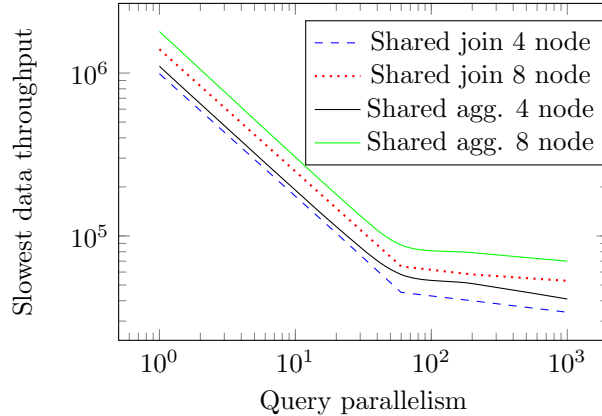
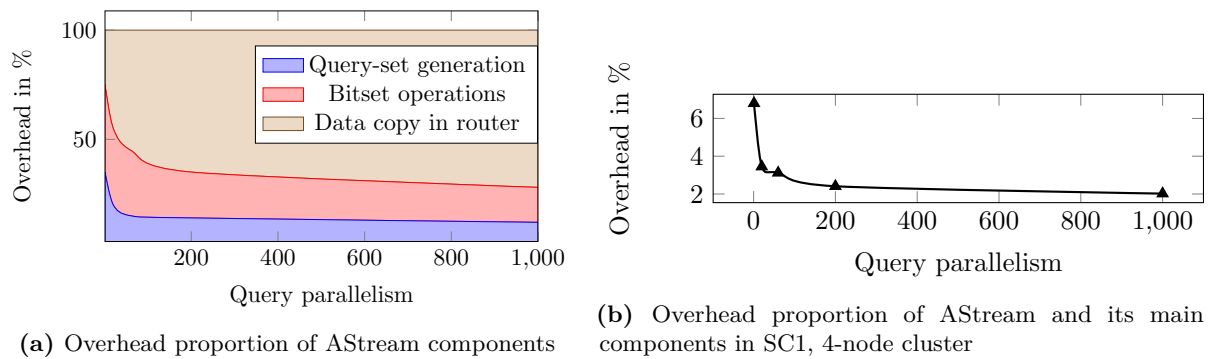


Figure 4.15: Input data throughput for different levels of query parallelism in SC1



(a) Overhead proportion of AStream components

(b) Overhead proportion of AStream and its main components in SC1, 4-node cluster

Figure 4.16: Overhead of AStream

sharing between AStream and original Flink in a single query setup. As shown in Figure 4.10 AStream throughput is 9 % less than Flink’s throughput in the worst case (from 2.15M/sec to 1.95M/sec, windowed aggregation, 8 nodes) because of the sharing overhead.

We also measure the individual cost of AStream’s components. The cost mainly involves generating query-sets, bitset operations, and data copy in the router to ship resulting tuples to different query channels. Figure 4.16a shows an overhead proportion of AStream components in SC1. With low query-parallelism, the proportion is roughly equal. As the number of concurrent queries increases, data copy becomes a dominant overhead. Data copy in the router operator is inevitable as AStream has to send resulting tuples to physically different query channels. Figure 4.16b shows the overhead of AStream (sum of its components). We can see that with more queries, the overhead of AStream is below 2%. The main reason is that with more queries the probability of sharing increases.

Figure 4.17 shows the effect of executing of ad-hoc join queries to the performance of existing ones. We perform experiments in a 4-node cluster. We observe that with many running queries, adding ad-hoc queries does not affect their performance much in both scenarios (SC1 and SC2). Also, with a small number of running queries, SC1 is more susceptible to a performance decrease than SC2. The main reason is that in SC1 long-running queries are created. In SC2, on the other hand, queries are created and deleted periodically. As a result, the overall number of queries and the size of query-sets is less in SC2.

Figure 4.18 shows the scalability of AStream queries with different cluster configurations. In this experiment, we keep the data throughput constant for all cluster configurations. We can see that the number of ad-hoc queries scales with more nodes. We also observe that SC2 scales better than SC1. The main reason is, as mentioned above, in SC2 queries are periodically created and removed, which results in less number of active queries and less bitwise operations.

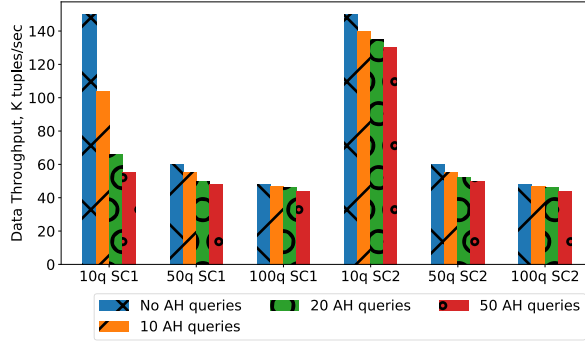


Figure 4.17: Effect of new ad-hoc join queries on existing long-running queries. x-axis shows the number of long-running queries and the workload scenario

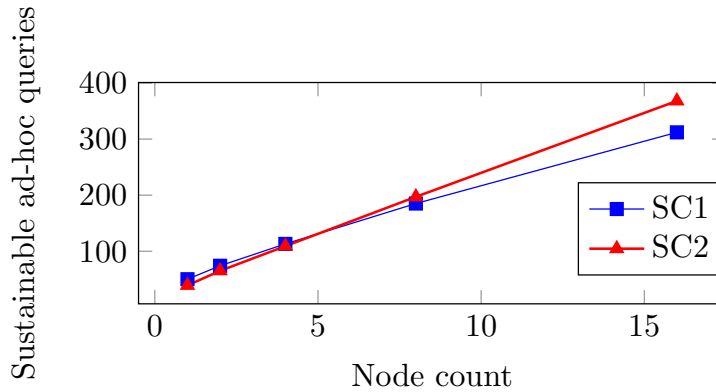


Figure 4.18: Scalability with the number of queries

#### 4.4.9 Discussion

AStream supports high data and query throughput within regular event-time and deployment latency boundaries. With thousand concurrent queries, AStream achieves more than 70 millions tuples per second data throughput (Figure 4.10). Our baseline, Apache Flink is not able to run twenty concurrent queries.

AStream also supports high query throughput. In SC1 AStream is able to start hundred queries in a single changelog and in SC2 it is able to start 50 queries and delete 50 queries in a single changelog.

AStream processes 70 millions tuples per second (Figure 4.10, 100q/s 1000qp) with 1.2 second average event-time latency (Figure 4.11b). For SC2, it handles fluctuating ad-hoc queries (creating and deleting 50 queries per 10 seconds) in less than one second event-time latency.

In our experiments, we see that the deployment latency is a major bottleneck for Flink (Figure 4.11a). AStream, however, has a very low deployment latency, in the order of milliseconds per query.

Integrating AStream has some overhead, which is already outweighed by the efficiency improvement with two concurrent queries. The overhead for a single query is in the order of 10% in the worst case.

### 4.5 Integration

All AStream components can be integrated into an SPE by extending its components. The shared session is an extension of a remote client module, which accepts user requests and executes the requests in a remote cluster. We modify this module to generate the changelog data structure for handling user requests. The shared selection and router are mapper operators with a state. The state is updated for every changelog. We implement window operators, windowed join and aggregation operators, by customizing



triggers, evictors, and window functions [63] to be dynamic and updatable at runtime. Any SPE providing low-level window APIs can integrate shared windowed join and aggregation operators.

We build our prototype of AStream on top of Apache Flink; however, our design is not tightly coupled with the underlying SPE. Below, we briefly sketch possible integrations with Trill [64] and Spark Structured Streaming [4]. Unlike previous work, such as DataPath [62], which is a complete system with dedicated query optimizer and scheduler, AStream is a framework that can be used with different SPEs to enable ad-hoc query processing.

Because AStream is a pluggable component of an underlying SPE, it also supports optimizations for data representation and code generation. For example, Trill uses streaming batched-columnar data representations with a novel dynamic compilation-based system architecture [64]. To support hybrid-columnar processing with AStream, one can integrate the query-set attribute as a separate column in Trill’s hybrid columnar data representation. Trill’s stateful operators can integrate AStream. AStream ensures consistency and correctness; however, the underlying processing semantics, e.g., grouping or join algorithms, can be performed by another component of an underlying SPE. Trill’s optimizations, such as exploiting sort order and skew, are orthogonal to AStream’s processing, as AStream operates on the upper layer. To integrate changelogs, Trill’s punctuation data structure can be extended to include additional query meta-data.

Spark Structured Streaming [4] is an example of a distributed SPE with highly optimized code generation using the catalyst optimizer [65]. In the continuous processing mode, one can adopt a very similar implementation to the one that we used in Flink. In the mini-batch processing mode, Spark Structured Streaming adopts bulk-synchronous-processing semantics, handling batch-size dynamically at runtime. In this setup, all workers can be informed about changelogs at the synchronization phase to use AStream components.

## 4.6 Related Work

Below, we explore several further research directions in database management and stream processing related to our work and discuss the similarities and differences.

### 4.6.1 Query-at-a-time Processing

Query-at-a-time SPEs feature mature and widely accepted optimizers [66, 65]. These systems inherit methods adopted in traditional relational query optimization [67, 68]. However, traditional query optimizers lack optimizing ad-hoc queries submitted in real time, as the solution space is non-convex and the complexity of query optimization in many cases is exponential [53]. AStream adopts shared operators which can handle multiple user queries and share them if necessary. Thus, we avoid the optimization and deployment cost of queries created and deleted in runtime.

### 4.6.2 Stream Multi-query Optimization

Multi-query optimization is one of the fundamental methods to share computation among queries [69]. One drawback of this method is its worst-case complexity (NP-hard) [70]. As the number of stream queries increases, finding shared parts among queries becomes costly. Another drawback of traditional multi-query optimization is that all queries should be known at compile-time. Also, multi-query optimization has limited ability to share queries with blackbox selection predicates, such as with user-defined functions.

Seshadri et al. show the potential limitations of streaming multi-query optimization in a distributed streaming environment [71]. Hong et al. propose rule-based streaming multi-query optimization [72]. Dobra

et al. adopt sketch-based techniques to find approximate results for streaming multi-query optimization [70].

The above work assumes prior knowledge (at compile-time) of stream queries and adopts optimizers that are not able to react to ad-hoc queries at runtime. AStream, on the other hand, has no prior knowledge about a workload and can react to ad-hoc queries.

### 4.6.3 Adaptive Query Optimization

Adaptive query optimization is another method to handle efficient execution of multiple stream queries. Although this methodology might work for a small number of input queries, with high concurrent workloads re-optimization is a limiting factor.

Madden et al. develop an adaptive stream query sharing system [73]. The system works on a single node and is built on top of Eddies [74]. Ives et al. propose an adaptive query processing framework which adjusts processing based on I/O delays and data flow rates, and shares the data across multiple queries [75]. Raman et al. propose STeM, a shared materialization point for join queries [76].

Unlike the work above, AStream is not limited to binary joins but also supports n-way joins, aggregation, selection predicates, and their combination. Moreover, AStream is designed to be executed in a distributed environment.

Drizzle is a distributed, fast, and adaptable stream processing framework [77]. It adopts the bulk-synchronous processing model. Chi is flexible stream processing framework built for tuple-at-a-time systems [78]. By design, these systems assume that workload and cluster properties change rarely, as changing the query execution plan is costly. AStream, however, supports highly fluctuating workloads and performs query creation and deletion on-the-fly, without stopping the topology.

### 4.6.4 Batch Ad-hoc Query Processing Systems

SharedDB [54] proposes query sharing for OLTP, OLAP, and mixed workloads via shared operators. QPipe, adopts on-demand simultaneous pipelining, dynamically sharing an operator's output simultaneously to parent nodes [11]. AStream is conceptually similar to SharedDB, as it also uses shared operators. Psaroudakis et al. compare the two main query sharing approaches: simultaneous pipelining, such as QPipe, and global query plan, such as SharedDB [79].

CJoin [80] and DataPath [62] propose a join operator that supports concurrent queries in data warehouses. MQJoin efficiently uses main memory bandwidth and multi-core architectures and minimizes redundant work across concurrent join queries [81, 82]. Tell has a shared-data architecture, which decouples transactional query processing and data storage into two layers to enable elasticity and workload flexibility [83].

To support multiple queries, scan sharing is a common technique. For example, Blink [84] and Crescendo [9] share disk and memory bandwidth. Similarly, CoScan performs cooperative scan sharing in the cloud merging pig programs from multiple users at runtime [85]. Also, MonetDB [86] and DB2 UDB [87] perform cooperative scans and maximize buffer-pool utilization across queries.

BatchDB adopts similar batching ideas with Crescendo and SharedDB [88]. However, BatchDB isolates batching of OLAP queries from the updates propagated by the primary OLTP replica. OLTPShare specializes in sharing concurrent OLTP workloads [89].

Although there are some similarities between AStream and batch ad-hoc query processing systems, such as bitsets (CJoin, DataPath, SharedDB, MQJoin), common upstream operators and common partitioning key (SharedDB and DataPath), query batching (SharedDB, BatchDB, OLTPShare), redundant computation filtering (MQJoin), scan sharing (Crescendo, Blink, MonetDB, DB2 UDB), supporting complex queries with high consistency (Tell), and high throughput concurrent query processing (MQJoin),

there are also conceptual differences. One difference between ad-hoc streaming and ad-hoc batch query processing is that the former features windows with different configurations. Also, ad-hoc batch data processing systems feature only ad-hoc query creation. AStream supports ad-hoc query creation and deletion in a consistent manner. Finally, AStream also adopts techniques to avoid redundant computation among queries; however, efficiently using hardware resources, such as CPU and memory bandwidth, adopted by MQJoin, is *out of the scope of our thesis*.

#### 4.6.5 Stream Query Sharing

Wang et. al propose sharing windowed join operators for CPU intensive and memory-intensive workloads [90]. The approach assumes that all input queries are known at compile-time. Our approach, on the other hand, supports query creation and deletion in an ad-hoc manner. Hammad et al. propose shared join operator for multiple stream queries [91]. Similar to the previous work, in this work, the main assumption is that input queries are known at compile-time. Another limitation is that this work adopts selection pull-up approach, which might result in *i*) high bookkeeping cost of resulting joined tuples and *ii*) intensive consumption of CPU and memory. Besides the limitations mentioned above, the above works ([90, 91, 92]) adopt an Eddies-like approach [74] to route tuples dynamically, which is hard to scale in distributed environments. Our approach, on the other hand, is designed for distributed stream environments.

Krishnamurthy et al. propose on-the-fly query sharing technique for windowed aggregation queries [92]. The authors partition tuples into fragments and perform incremental aggregation. Traub et.al propose a general stream aggregation technique that automatically adapts to workload characteristics [93]. Although we also adopt a similar techniques to compute results incrementally, our solution is not limited to windowed aggregations. AStream supports windowed queries consisting of selection, aggregation, join, and their combinations.

Li et. al propose window ID representation of events and panes [57], and sharing computation among panes [55]. The core difference between panes and the window sharing technique of AStream is that the former computes overlapping parts of a window at compile-time, while the latter computes them at runtime.

## 4.7 Conclusion

In this chapter, we presented AStream, the first distributed SPE for ad-hoc stream workloads. We showed that current state-of-the-art SPEs were not able to process ad-hoc stream workloads. We observed in our experiments that not only data latency and throughput, but also query deployment latency and throughput were bottlenecks.

AStream is a layer on top of Flink, which extends existing SPE components and supports the majority of streaming use-cases. AStream ensures easy integration, correctness, consistency, and high performance (query and data throughput) in ad-hoc query workloads.

In the next chapter, we will extend AStream with a cost-based optimizer and adaptive query processing techniques. We calculate a better optimized query plan by grouping similar queries based on sharing statistics.



# 5

## AJoin: Ad-hoc Stream Joins at Scale

This Chapter contains:

|         |   |    |
|---------|---|----|
| 5.1     | Introduction . . . . .                            | 67 |
| 5.1.1   | Motivation . . . . .                              | 67 |
| 5.1.2   | Sharing Limitations in Ad-hoc SPEs . . . . .      | 68 |
| 5.1.2.1 | Missed Optimization Potential . . . . .           | 68 |
| 5.1.2.2 | Dynamicity . . . . .                              | 69 |
| 5.1.3   | AJoin . . . . .                                   | 69 |
| 5.1.3.1 | Efficient Distributed Join Architecture . . . . . | 69 |
| 5.1.3.2 | Dynamic Query Processing . . . . .                | 69 |
| 5.1.3.3 | AJoin and AStream: Complete Ad-hoc SPE . . . . .  | 69 |
| 5.1.4   | Contributions and Chapter Organization . . . . .  | 70 |
| 5.2     | Related Work . . . . .                            | 70 |
| 5.2.1   | Shared Query Processing . . . . .                 | 70 |
| 5.2.2   | Adaptive Query Processing . . . . .               | 71 |
| 5.2.3   | Query Optimization . . . . .                      | 71 |
| 5.2.4   | Mini-batch Query Processing . . . . .             | 71 |
| 5.3     | System Overview and Example . . . . .             | 71 |
| 5.3.1   | Data Model . . . . .                              | 73 |
| 5.3.1.1 | Bucket . . . . .                                  | 73 |
| 5.3.1.2 | Changelog . . . . .                               | 73 |
| 5.3.2   | Join Operation . . . . .                          | 73 |
| 5.4     | Optimizer . . . . .                               | 74 |
| 5.4.1   | Query Grouping . . . . .                          | 74 |
| 5.4.2   | Join Reordering . . . . .                         | 76 |
| 5.4.3   | Vertical and Horizontal Scaling . . . . .         | 78 |
| 5.5     | Implementation Details . . . . .                  | 78 |
| 5.5.1   | Join Phases . . . . .                             | 79 |
| 5.5.1.1 | Bucketing . . . . .                               | 79 |
| 5.5.1.2 | Partitioning . . . . .                            | 79 |
| 5.5.1.3 | Join . . . . .                                    | 79 |

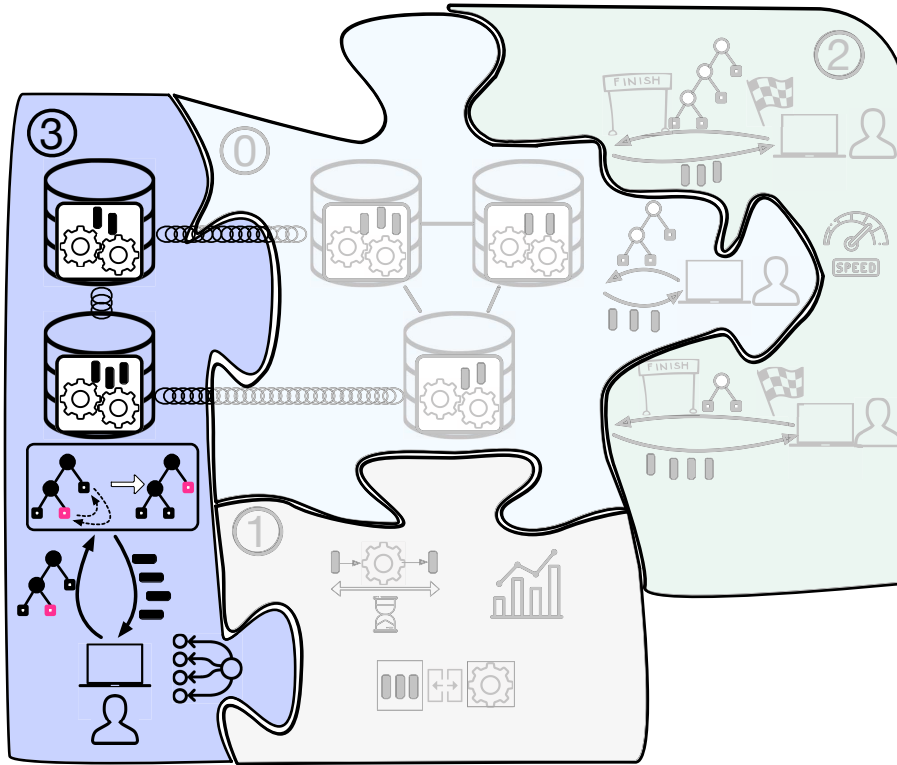


Figure 5.1: Scope of Chapter 5

|         |  |    |
|---------|--|----|
| 5.5.1.4 | Materialization . . . . .                                    | 80 |
| 5.5.2   | Exactly-once Semantics . . . . .                             | 80 |
| 5.5.3   | Optimizer . . . . .  | 80 |
| 5.6     | Runtime QEP changes . . . . .                                | 81 |
| 5.6.1   | Consistency Protocols . . . . .                              | 81 |
| 5.6.2   | Vertical Scaling . . . . .                                   | 82 |
| 5.6.3   | Horizontal Scaling . . . . .                                 | 82 |
| 5.6.4   | Join Reordering . . . . .                                    | 84 |
| 5.7     | Experiments . . . . .  | 85 |
| 5.7.1   | Experimental Design . . . . .                                | 85 |
| 5.7.2   | Metrics and Data Generation . . . . .                        | 85 |
| 5.7.3   | Workload . . . . .   | 86 |
| 5.7.4   | Setup . . . . .  | 87 |
| 5.7.5   | Scalability . . . . .  | 87 |
| 5.7.6   | Distinct Keys . . . . .                                      | 89 |
| 5.7.7   | Dynamicity . . . . .   | 89 |
| 5.7.7.1 | Latency . . . . .  | 89 |
| 5.7.7.2 | Breakdown . . . . .  | 90 |
| 5.7.7.3 | Throughput . . . . .   | 91 |
| 5.7.7.4 | Impact of Each Component . . . . .                           | 92 |
| 5.7.7.5 | Cost of Sharing . . . . .                                    | 93 |
| 5.7.7.6 | Impact of the Latency Threshold Value . . . . .              | 94 |
| 5.7.7.7 | Impact of the Query Reoptimization Threshold Value . . . . . | 94 |
| 5.8     | Conclusion . . . . .   | 94 |

In the last decade, many stream processing engines were developed to overcome the high latency of batch data processing for real-time scenarios. The processing model of these systems is designed to execute long-running queries one at a time. However, with the advance of cloud technologies and multi-tenant systems, multiple users share the same cloud for stream query processing. This results in many ad-hoc stream queries sharing common stream sources and resources. Many of these queries include joins.

There are two main limitations that hinder performing ad-hoc stream join processing. The first limitation is missed optimization potential both in the stream data processing and query optimization layers. The second limitation is the lack of dynamicity in query execution plans.

In this chapter, we present AJoin, a dynamic and incremental ad-hoc stream join framework. AJoin consists of an optimization layer and a stream data processing layer. The optimization layer periodically reoptimizes the query execution plan, performing join reordering and vertical and horizontal scaling at runtime without stopping the execution. The data processing layer implements a pipeline-parallel join architecture. This layer enables incremental and consistent query processing supporting all the actions triggered by the optimizer. We implement AJoin on top of Apache Flink, an open-source data processing framework. AJoin outperforms Flink not only for ad-hoc multi-query workloads but also for single-query workloads.

## 5.1 Introduction

SPEs process continuous queries on real-time data, which are series of events over time. Examples of such data are sensor events, user activity on a website, and financial trades. There are several open-source streaming engines, such as Apache Spark Streaming [4, 45], Apache Storm [2], and Apache Flink [5], backed by big communities.

With the advance of cloud computing [94], such as the Software as a Service model [95], multiple users share public or private clouds for stream query processing. Many of these queries include joins. Stream joins continuously combine rows from two or more bounded streaming sources. In particular, executing multiple ad-hoc queries on common streaming relations needs careful consideration to avoid redundant computation and data copy.

### 5.1.1 Motivation

Stream join services are used in many companies, e.g., Facebook [96]. Clients subscribed to such a service create and delete stream join queries in an ad-hoc manner. In order to execute the queries efficiently, a service owner needs to periodically reoptimize the query execution plan (QEP).

Let  $V=\{vID, length, geo, lang, time\}$  be a stream of videos (videos displayed at user’s profile),  $W=\{usrID, vID, duration, geo, time\}$  a video view stream of a user,  $C=\{usrID, comment, length, photo, emojis, time\}$  a stream of user comments, and  $R=\{usrID, reaction, time\}$  a steam of user reactions, such as like, love, and angry. Figure 5.2 shows an example use-case scenario for ad-hoc stream join queries. The machine learning module initiates Q1 to feed the model with video preferences of users. The module targets people living in Germany ( $\sigma_{W.geo=GER}$ ) and watching videos in English ( $\sigma_{V.lang=ENG}$ ). The editorial team initiates Q2 to discover web brigades or troll armies [97, 98]. The query detects users that comment ( $\sigma_{C.length>5}$ ) on videos published in US ( $\sigma_{V.geo=US}$ ) just a few seconds after watching them ( $\sigma_{W.duration<10}$ ). Usually, these people do not watch videos fully before commenting on videos. The quality assurance team initiates Q3 to analyze the users’ reactions to promoted videos. Specifically, the team analyzes videos that are watched in Europe ( $\sigma_{W.geo=EU}$ ), receive **angry** reactions ( $\sigma_{R.reaction=angry}$ ), and at least one emoji in comments ( $\sigma_{C.emojis>0}$ ). We use the queries, shown in Figure 5.2 throughout this chapter.





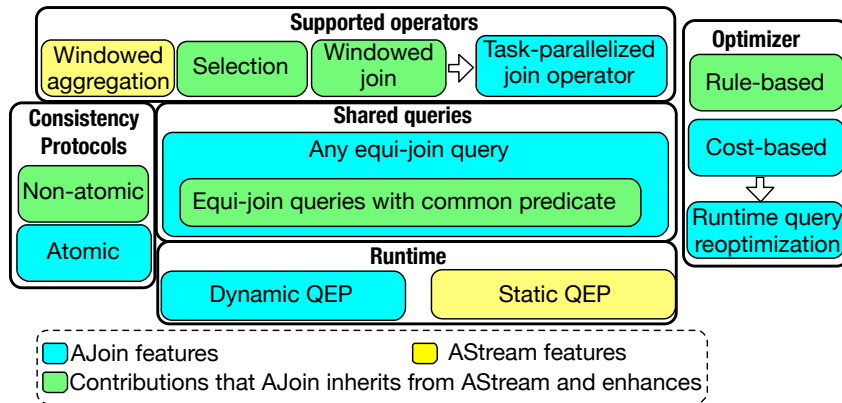


Figure 5.3: AJoin and AStream: Complete Ad-hoc SPE

### 5.1.2.2 Dynamicity

Modern ad-hoc SPEs consider ad-hoc query processing only with a static QEP and with queries with common join predicates. In stream workloads with fluctuating data and query throughput, this is inefficient.

### 5.1.3 AJoin

We propose AJoin, a scalable SPE that supports ad-hoc equijoin query processing. We overcome the limitations stated in Section 5.1.2 by combining incremental and dynamic ad-hoc stream query processing in our solution.

#### 5.1.3.1 Efficient Distributed Join Architecture

Because the join operator in modern SPEs is computationally expensive, AJoin shares the workload of the join operator with a source and sink operator. The join architecture is not only data-parallel but also pipeline-parallel. Tuples are indexed in the source operator. The join operator utilizes indexes for an efficient join operation. AJoin incrementally computes multiple join queries. It performs a scan, data, and computation sharing among multiple join queries with different predicates. Our solution adopts late materialization for intermediate join results. This technique enables the system to compress the intermediate results and passes them to downstream operators efficiently. Also, the AJoin optimizer features incremental and iterative optimization with dynamic programming.

#### 5.1.3.2 Dynamic Query Processing

AJoin supports dynamicity at the optimization and data processing layer: dynamicity at the optimization layer means that the optimization layer performs regular reoptimization, such as join reordering and horizontal and vertical scaling; dynamicity at the data processing layer means that the layer is able to perform all the actions triggered by the optimizer at runtime, without stopping the QEP.

#### 5.1.3.3 AJoin and AStream: Complete Ad-hoc SPE

Together AStream and AJoin form a complete ad-hoc SPE. Figure 5.3 shows the architecture of the resulting system and the main contributions of AStream and AJoin to the resulting system. For example, AStream proposes query-sets and changelogs. AJoin arranges queries with similar selection predicates into the same groups. AJoin features a cost-based query optimizer that performs progressive query optimization periodically at runtime. AStream provides sharing for windowed aggregation queries. AJoin,

contributes the optimizer, which enables sharing data and computation if the sharing is beneficial. Also, AJoin contributes an efficient and pipeline-parallelized join architecture. AStream and AJoin contribute non-atomic and atomic consistency protocols, respectively.

### 5.1.4 Contributions and Chapter Organization

The main contributions of the chapter are as follows.

- We present the first optimizer to process ad-hoc stream queries in an incremental manner.
- We develop a distributed and pipeline-parallel stream join architecture. This architecture also supports dynamicity (modify QEP on-the-fly in a consistent way).
- We perform an extensive experimental evaluation with state-of-the-art SPEs.

The rest of the chapter is organized as follows. We present related work in Section 5.2. Section 5.3 gives the system overview. Section 5.4 presents the AJoin optimizer. We provide implementation details in Section 5.5 and runtime operations in Section 5.6. Experimental results are shown in Section 5.7. We conclude in Section 5.8.

## 5.2 Related Work

### 5.2.1 Shared Query Processing

Shared query processing is a paradigm that shifts from query-at-a-time approaches towards shared work. Shared operators batch concurrent queries and share possible computations among them. SharedDB is based on the batch data processing model and handles OLTP, OLAP, and mixed workloads [101, 102]. Giceva et al. adopt SharedDB ideas and implement shared query processing on multicores [103]. CJoin [80, 104] and DataPath [62] focus on ad-hoc query processing in data warehouses. Braun et al. propose a hybrid (OLTP and OLAP) computation system, which integrates key-value-based event processing and SQL-based analytical processing on the same distributed store [105]. BatchDB implements hybrid workload sharing for interactive applications [88]. SharedHive is a shared query processing solution built on top of the MapReduce framework [106]. The works mentioned above are designed for batch data processing environments. Although we also embrace some ideas from shared join operators, we focus on stream data processing environments with ad-hoc queries.

To increase data throughput, MJoin proposes a multi-way join operator that operates over more than two inputs [107]. While the bucket data structure in AJoin also mimics the behavior of multi-way joins, the join operator of AJoin supports binary input streams. To increase data throughput, AJoin reoptimizes the QEP periodically. FluxQuery is a centralized main-memory execution engine based on the idea of continual circular clock scans and adjusted for interactive query execution [108]. Similarly, MQJoin supports efficient ad-hoc execution of main-memory joins [81]. Hammad et al. propose streaming ad-hoc joins [91]. The solution adopts a centralized router, extended from Eddies [74]. Also, the work adopts a selection pull-up approach, which might result in high bookkeeping cost of resulting joined tuples and intensive CPU and memory consumption. The above works are designed for a single-node environment. However, AJoin is designed for distributed environments. AJoin does not utilize any centralized computing structure. Dynamicity and progressive optimization, are more essential in distributed environments. Also, AJoin exploits pipeline-parallelism. In a single-node environment, however, task-fusion is more beneficial [14].

### 5.2.2 Adaptive Query Processing

Adaptive query processing utilizes runtime feedback to modify a QEP and maximize the given objective function, such as better response time and more efficient CPU utilization [109]. Progressive query optimization, POP, uses cardinality boundaries in which a selected plan is optimal [110]. Our optimizer uses a similar idea, cost sharing, but we target streaming scenarios. Li et al. propose adaptive join ordering during query execution [111]. The solution adds an extra operator, a local predicate on the driving table to exclude the already processed rows if the driving table is changed. We perform join reordering without extra operators.

Gedik et al. propose an elastic scaling framework for data streams [112]. Cardellini et al. propose a similar idea on top of Apache Storm [113]. Both works use state migration as a separate phase to redistribute the state among nodes. AJoin, on the other hand, features a smooth repartitioning scheme, without stopping the topology. Heinze et al. propose an operator placement technique for elastic stream processing [114][115]. AJoin does not perform operator placement optimization for all streaming operators but only for join and selection operators (e.g., grouping queries and executing them in specific operators).

Drizzle is an elastic layer built on top of Spark Streaming [77]. Although it partly eliminates limitations of the bulk-synchronous parallel model of Spark Streaming via group scheduling, the solution still underperforms compared to tuple-at-a-time systems [78]. AJoin, on the other hand, avoids blocking operations.

### 5.2.3 Query Optimization

Query optimization for query-at-a-time systems is a challenging problem. Optimizing queries in the presence of ad-hoc queries is an even harder problem. Trummer et al. solve the join ordering problem via a mixed integer programming model [116]. The authors perform several nonlinear to linear conversions to express equations in a linear format. Although this approach is acceptable in a single query environment, with ad-hoc queries we need an optimization framework that can optimize incrementally. Unlike dynamic programming approaches [117][118], current numerical optimization frameworks lack this feature.

The IK/KBZ family of algorithms can construct the optimal join plan in polynomial time [119][120]. Although this is a very attractive feature, these algorithms perform poorly with large queries (thousands of joins) [121]. In AJoin our target is not large queries, as there are limited use-cases with large stream queries. However, the iterative dynamic programming approach also works well with large queries and combines benefits both dynamic programming and greedy algorithms [25]. We adopt ideas from this technique and enhance them for our scenario.

### 5.2.4 Mini-batch Query Processing

Dividing the data into mini-batches and processing the live stream as a set of batch data computations is also a common technique [4][3]. Adaptive mini-batch SPEs, such as Drizzle [77], modify a QEP at mini-batch boundaries. Similarly, AJoin modifies QEP at bucket boundaries. Mini-batch stream processing utilizes a bulk synchronous processing model, in which all task managers synchronize at every batch interval. However, AJoin adopts a non-blocking continuous operator model [17].

## 5.3 System Overview and Example

In this section, we provide a high-level overview of AJoin. Figure 5.4 shows the architecture of AJoin. The remote client listens to users requests, such as query creation or deletion requests. It batches user requests in a `query batch` and sends this batch to the optimizer. Apart from query batches, the optimizer

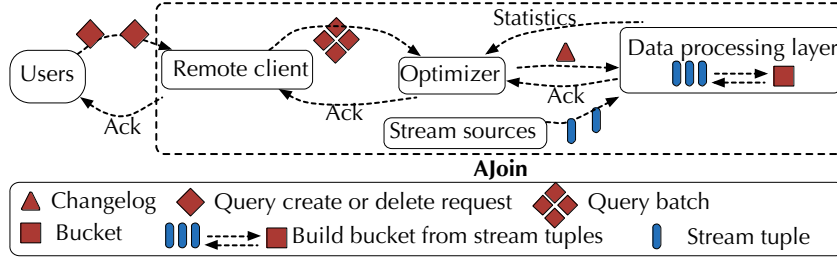
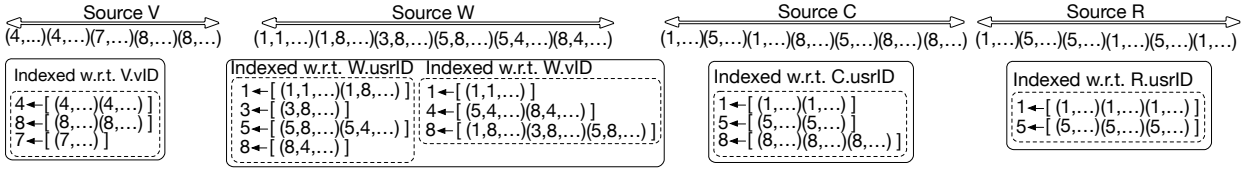
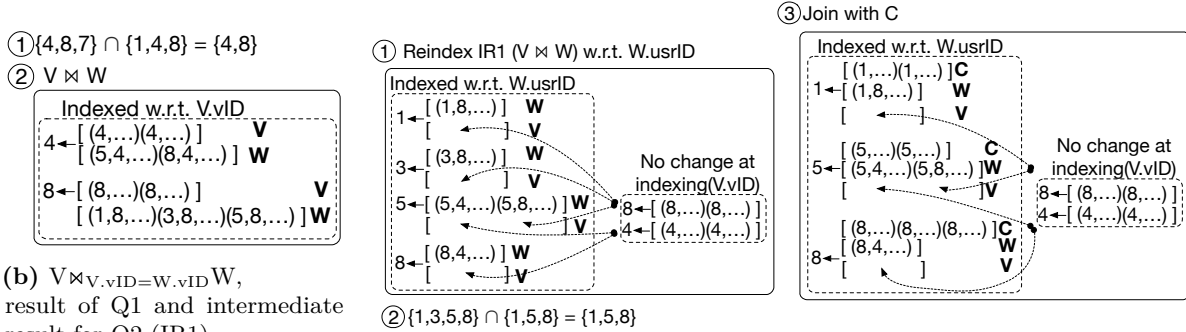


Figure 5.4: AJoin architecture

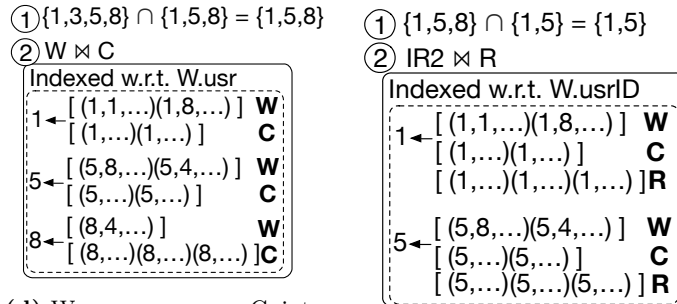


(a) Tuples (after applying filter) from stream sources V, W, C, and R (top row) and constructed buckets from the respective sources (bottom row)



(b)  $V \bowtie_{V.vID=W.vID} W$ , result of Q1 and intermediate result for Q2 (IR1)

(c) Result of Q2



(d)  $W \bowtie_{W.usrID=C.usrID} C$ , intermediate result for Q3 (IR2)

(e) Result of Q3

Figure 5.5: Executing Q1, Q2, and Q3 in AJoin between time  $T_{4C}$  and  $T_{1D}$ . For simplicity, the attributes that are not used by the queries are indicated as '...'.



Figure 5.6: Optimization process

periodically receives statistics from the data processing layer. It periodically reoptimizes the QEP based on statistics and received query batches. As part of the reoptimization, the optimizer triggers actions, such as scale up and down, scale out and in, query pipelining, and join reordering. Similarly, the data processing layer performs all the actions at runtime, without stopping the QEP. AJoin supports equi-joins with event-time windows and selection operators. Below, we elaborate more on the data model (Section 5.3.1) and join operator structure (Section 5.3.2) of AJoin.

### 5.3.1 Data Model

There are three main data structures in AJoin: a stream tuple, a bucket, and a changelog. Source operators of AJoin pull stream tuples from external sources. Then, the tuples are transformed into the internal data structure of AJoin, which is a **bucket**. Below, we discuss the data structures bucket and changelog in detail.

#### 5.3.1.1 Bucket

A **bucket** is the main data structure throughout the QEP. It contains a set of index entries and stream tuples corresponding to the index entries. Each bucket includes a bucket ID. Stream tuples in a bucket can be indexed w.r.t. different attributes. Buckets are read-only. All AJoin operators, except for the source operator, receive buckets from upstream operators and output new read-only buckets. This way, the buckets can be easily shared among multiple concurrent stream queries.

Figure 5.5a shows stream tuples generated from sources V, W, C, and R and generated buckets from the respective stream sources. The bucket generated from the stream source V is indexed w.r.t. the V.vID attribute because the downstream join operator uses the predicate  $V.vID=W.vID$ . However, the bucket generated from the stream source W is indexed w.r.t. two attributes: W.vID and W.usrID. The reason is that *i*) Q1 and Q2 requires indexing w.r.t. the attribute W.vID and *ii*) Q3 requires indexing w.r.t. the attribute W.usrID. Unless stated otherwise, we assume that the join ordering of Q2 is  $(V \bowtie_{V.vID=W.vID} W) \bowtie_{W.usrID=C.usrID} C$ .

#### 5.3.1.2 Changelog

A **changelog** is a special marker dispatched from the optimizer. It contains metadata about QEP changes, such as horizontal or vertical scaling, query deletion, and query creation. A changelog propagates through the QEP. Operators receiving the changelog update their execution accordingly.

### 5.3.2 Join Operation

In modern SPEs, such as Spark [4], Flink [5], and Storm [2], the computation distribution of a join operation is rather skewed among different stream operators: source, join, and sink operators. For example, the source operator is responsible for pulling stream tuples from external stream sources. The join operator buffers stream tuples in a window, finds matching tuples, and builds resulting tuples by assembling the matching tuples. The join operator also implements all the functionalities of a windowing operator. The sink operator pushes the resulting tuples to external output channels. Because most of the computation is performed in the join operator, it can easily become a bottleneck. With more concurrent  $n$ -way join queries ( $n \geq 3$ ), the join operator is more likely to be a limiting factor.

To overcome this issue, we perform two main optimizations. First, we perform **pipeline parallelization** sharing the load of the join operator between the source and sink operators. The source operator combines the input data acquired in the last  $t$  time slots and builds a bucket (Section 5.3.1). With this, we transmit the windowing operation from the join operator to the source operator. Also, buckets contain indexed tuples, which are used at the downstream join operator to perform the join efficiently. Afterwards, the partitioner distributes buckets based on a given partitioning function. Then, the join operator performs a set intersection between the index entries of input buckets. Note that for all downstream operators of the source operator, the unit of data is a bucket instead of a stream tuple. Finally, the sink operator performs full materialization, i.e., it converts buckets into stream tuples, and outputs join results.

Second, we perform **late materialization** of intermediate join results. After computing the matching tuples (via intersecting index entries), the join operator avoids performing the cross-product among them. Figure 5.5b shows the join operation for Q1. Index entries from the two input buckets are joined (①). Then, tuples with the matched indexes are retained in the resulting bucket (②). The late materialization technique can also be used for  $n$ -way joins. For example, Figure 5.5e shows the resulting bucket of Q3. The bucket keeps indexes of matched tuples from stream sources W, C, and R.

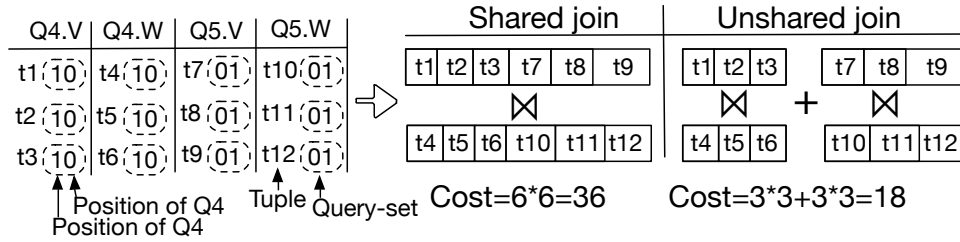
All join predicates in Q3 use the same join attribute (usrID). In this case, the late materialization can be easily leveraged with built-in indexes (Figures 5.5d and 5.5e). However, if join attributes are different (e.g. in Q2), then repartitioning is required after the first join. AJoin benefits from late materialization also in this scenario. To compute Q3, AJoin computes the result of the upstream join operator (Figure 5.5b). Then, the resulting bucket ( $V \bowtie_{V.usrID=W.usrID} W$ ) is reindexed w.r.t.  $W.usrID$  (Figure 5.5c ①). Note that reindexing is related to the tuples belonging to  $W$  because only these tuples contain attribute  $usrID$ . Instead of materializing the intermediate result fully and iterating through it ( $V \bowtie_{V.usrID=W.usrID} W$ ) and reindexing, AJoin avoids full materialization and only iterates over the tuples belonging to  $W$ : (1) every tuple  $tp \in W$  is reindexed w.r.t.  $W.usrID$ ; (2) a list of its matched tuples from  $V$  is retrieved (get list with index  $ID=tp.usrID$ ); (3) the pointer of the resulting list is appended to  $tp$ . When  $tp$  is eliminated in the downstream join operator, all its matched tuples from  $V$  are also automatically eliminated. For example, tuples with  $usrID=3$  in Figure 5.5c ①, are eliminated when joining with  $C$  (Figure 5.5d). In this case, the pointers are also eliminated without iterating through them.

## 5.4 Optimizer

In this section, we discuss the query optimization process in AJoin. Figure 5.6 shows the optimization phases of AJoin. After a changelog ingestion, the optimizer eagerly shares the newly created query with the running queries (①). For example, Q2 is deployed at time  $T_{2C}$  (Figure 5.2). Then, the optimizer searches common subqueries among running queries (Q1 in this case), without considering the selection predicate and the cost. In this case, the optimizer deploys Q2 as  $(V \bowtie W) \bowtie C$  to reuse the existing stream sources and the join operator. In the following phases, the optimizer performs a cost-based analysis and reoptimizes the QEP, when necessary. If the optimizer cannot find common subqueries, it will check for common sources to benefit from scan sharing. The optimizer restarts the optimization process, if a new changelog has arrived. Below, we explain each phase of the optimization separately and describe when the optimizer decides to trigger each of them.

### 5.4.1 Query Grouping

Consider Q4 and Q5 in Figure 5.8. These queries do not share data because of their selection predicates. Figure 5.7 shows an example scenario for performing shared (Q4 and Q5 together) and separate join (Q4



**Figure 5.7:** Cost of shared and separate join execution for Q4 and Q5. Q.S means the stream S of the query Q.

and Q5 separately). Previous solutions, such as AStream [100], share data and computation aggressively. However, this might lead to a suboptimal QEP. For example, in Figure 5.7 the cost of shared query execution is higher than executing queries separately. The reason is that both Q4.V, Q5.V and Q4.W, Q5.W do not share enough data tuples to benefit from shared execution. Throughout this chapter, we denote the stream source S of query Q as Q.S and stream partition  $p_i$  of query Q as Q. $p_i$ .

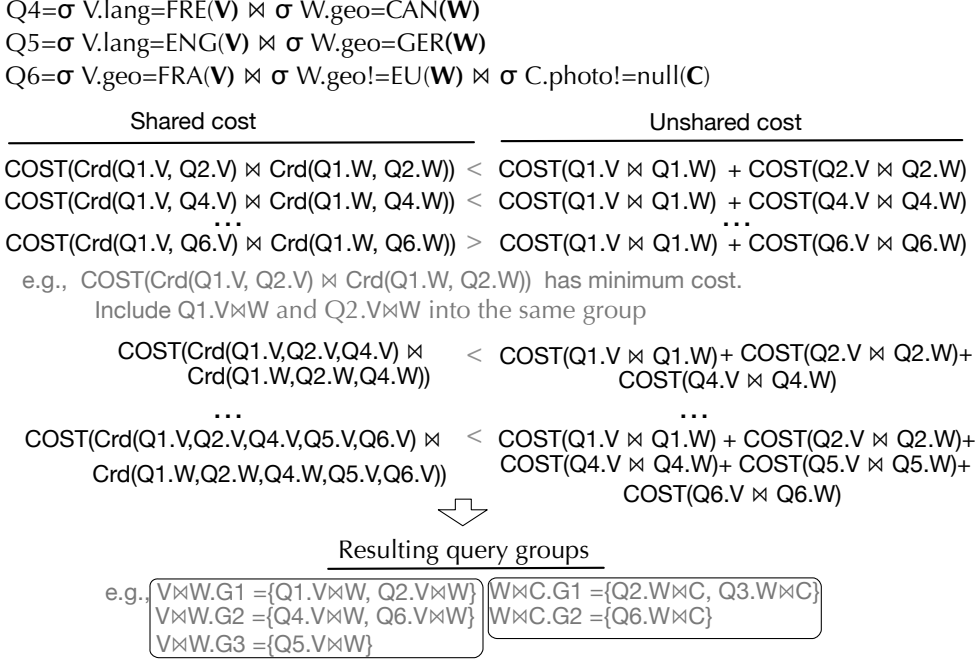
To avoid the drawback of aggressive sharing, we arrange queries in groups. Queries that are likely to filter (or not filter) a given set of stream tuples are arranged in one **query group**. For example, after successful grouping, Q4 and Q5 in Figure 5.7 would reside in different groups. Let t1, t2, t3, and t4 be tuples with query-sets (100100), (101100), (100100), and (100000), respectively, and Q1-Q6 be queries with selection operators. Q1 and Q4 share 3 tuples (t1, t2, t3) out of 4. Also, Q2, Q3, Q5, and Q6 do not share 3 tuples (t1, t3, t4) out of 4. Finding the optimal query groups is an NP-Hard problem, as it can be reduced to the euclidean sum-of-squares clustering problem [122].

Crd is a function that calculates the cardinality of possibly intersecting sets. We use set union operation to calculate the cardinality. For example, for 3 sets (A, B, C) the Crd function is shown in Equation 5.1, which is a specific case of the inclusion-exclusion principle in combinatorial mathematics. Equation 5.2 shows the cost function.  $b_{i1}$  and  $b_{i2}$  are Boolean variables showing if indexing is required on stream S1 and S2, respectively. AJoin performs indexing when stream S is the leaf node of the QEP (source operator) or when repartitioning is performed.  $b_m$  is also a Boolean variable indicating if full materialization is required. AJoin performs full materialization only at the sink operator.

Figure 5.8 shows our approach to calculate query groups. First, we compare the cost of sharing stream sources between two queries and executing them separately. If the cost of the former is less than the latter, we place the two queries into the same query group. Once we find query groups consisting of two queries, we eagerly check other queries, which are not part of any group, to include into the group. The only condition (to be accepted to the group) is that the cost of executing the new query and the queries inside the group in a shared manner must be less than executing them separately (e.g., Figure 5.7). Query grouping is performed periodically during the query execution. When join reordering is triggered, it utilizes recent query groups.

$$|A|+|B|+|C|-|A\cap B|-|B\cap C|-|A\cap C|+|A\cap B\cap C| \quad (5.1)$$

$$\begin{aligned} \text{COST}(S1 \bowtie S2) &= \underbrace{b_{i1} * \text{Crd}(S1)}_{\text{Indexing S1}} + \underbrace{b_{i2} * \text{Crd}(S2)}_{\text{Indexing S2}} + \\ &\underbrace{\text{Min}(\text{DistKey}_{S1}, \text{DistKey}_{S2})}_{\text{Index set intersection}} + \underbrace{b_m * \text{Crd}(S1 \bowtie S2)}_{\text{Full materialization}} \end{aligned} \quad (5.2)$$



**Figure 5.8:** Calculation of query groups. The optimization is performed between time  $T_{3C}$  and  $T_{1D}$ . Assume that Q4-Q6 are also being executed at the time of optimization. In the figure,  $Crd$  refers to the cardinality function, and  $COST$  refers to the cost function in Equation 5.2

### 5.4.2 Join Reordering

After discovering query groups, the optimizer performs iterative QEP optimization. We enhance an iterative dynamic programming technique [25] and adapt it to ad-hoc stream query workloads. Our approach combines dynamic programming with iterative heuristics. In each iteration, the optimizer *i*) calculates the shared cost of subqueries and *ii*) selects a subplan based on the cost. The shared cost is the cardinality of a particular subquery divided by the number of QEPs, sharing the subquery.

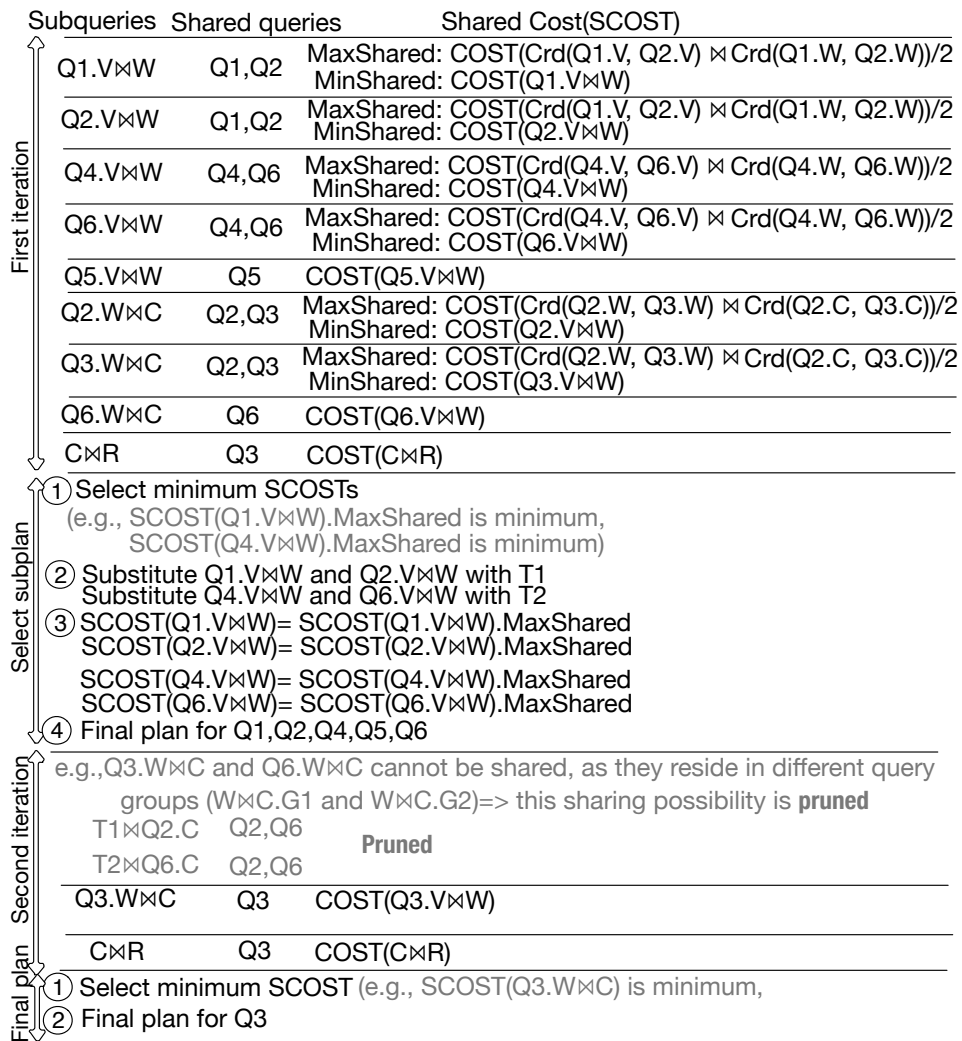
Figure 5.9 shows an example scenario for iterative QEP optimization. Assume that Q4-Q6, which are shown in Figure 5.8, are also added to the existing queries (Q1-Q3). In the first iteration, the optimizer calculates the shared cost of 2-way joins. For example,  $Q1.V \bowtie W$  can be shared between Q1 and Q2 because Q1 and Q2 are in the same group (Figure 5.8). Also, the cost of  $Q1.V \bowtie W$  differs when exploiting all sharing opportunities (MaxShared) and executing the queries separately (MinShared). After the first iteration, the optimizer selects subplans with minimum costs. Then, the optimizer substitutes the selected subqueries with T1 and T2. If the cost is shared with other QEPs (e.g.,  $Q1.V \bowtie W$  is shared between Q1 and Q2), then the optimizer assigns the shared cost to all other related queries.

The second iteration is similar to the first one. Note that  $T1 \bowtie Q2.C$  cannot be shared with Q6 because  $Q6.V \bowtie W$  and  $Q2.V \bowtie W$  reside in different query groups. So, the optimizer prunes this possibility. Also,  $Q3.W \bowtie C$  is no longer shared with Q2 because in the first iteration the optimizer assigned  $(V \bowtie W) \bowtie C$  to Q2.

Computing the optimal QEP for multiple queries is an NP-Hard problem [53, 120]. For ad-hoc queries, this is particularly challenging, since queries are created and deleted in an ad-hoc manner. The optimizer must therefore support incremental computation. Assume that Q4 in Figures 5.8 and 5.9 is deleted, and  $Q7 = \sigma_{sp1}(W) \bowtie \sigma_{sp2}(C)$  is created, where  $sp1$  and  $sp2$  are selection predicates. At compile-time, the optimizer shares Q7 aggressively (without considering the selection predicates) with existing queries. In this case, the optimizer shares Q7 with  $Q3.W \bowtie C$ . After collecting statistics, the optimizer tries to locate Q7 in one of  $W \bowtie C$  groups (e.g., Figure 5.8). If including Q7 is not beneficial to any query group (if



$Q4 = \sigma V.lang = FRE(V) \bowtie \sigma W.geo = CAN(W)$   
 $Q5 = \sigma V.lang = ENG(V) \bowtie \sigma W.geo = GER(W)$   
 $Q6 = \sigma V.geo = FRA(V) \bowtie \sigma W.geo \neq EU(W) \bowtie \sigma C.photo \neq null(C)$



**Figure 5.9:** Join reordering. The optimization is performed between time  $T_{3C}$  and  $T_{1D}$ . Assume that Q4-Q6 are also being executed at the time of optimization. In the figure,  $Crd$  refers to the cardinality function, and  $COST$  refers to the cost function in Equation [5.2](#)

shared execution is more costly than executing queries in the group and the added query separately), the optimizer creates a new group for Q7. Assume that Q7 is placed in  $W \times C.G2$  (Figure 5.8). In this case, only the execution of Q4 and Q6 might be affected. In other words, the optimizer does not need to recompute the whole plan, but only part of the QEP. Also, the optimizer does not recompute query groups from scratch but reuses existing ones.

The cost of incremental computation is high and may result in an suboptimal plan. Therefore, we use a threshold when to trigger a full optimization. If the number of created and deleted queries exceeds 50% of all queries in the system, the optimizer computes a new plan (including the query groups) holistically instead of incrementally. We have determined this threshold experimentally (Section 5.7.7.7), as it gives a good compromise between dynamicity and optimization cost. Computing the threshold deterministically, on the other hand, is out of the scope of this thesis. The decision to reorder joins (② in Figure 5.6) is triggered by the cost-based optimizer using techniques explained above.

There are two main requirements behind our cost computation. The first requirement is that the cost function should include the computation semantics of our pipeline-parallelized join operator. As we can see from Equation 5.2,  $COST$  consists of the cost of the source operator (indexing S1 and S2), the cost of join operator (index set intersection), and the cost of sink operator (full materialization). The second requirement is that the cost computation should include sharing information. We achieve this requirement by dividing  $COST$  by the number of shared queries (Figure 5.9  $MaxShared$ ). We select this cost computation semantics because it complies with our requirements, and it is simple.

### 5.4.3 Vertical and Horizontal Scaling

AJoin uses consistent hashing for assigning tuples to partitions. The partitioning function  $PF$  maps each tuple with key  $k$  to a circular hash space of key-groups:  $PF(k) = (\text{Hash}(k) \bmod |P|)$ , where  $|P|$  is the number of parallel partitions. At compile-time, partitions are distributed evenly among nodes.

The optimizer performs vertical scaling (③ in Figure 5.6), if the latency of tuples residing in specific partitions is high, and there are resources available on nodes, in which overloaded partitions are located. The optimizer checks for scaling up first, because scaling up is less costly than scaling out. Note that when scaling up, the partitioning function and the partitioning range assigned to each node remain the same. Instead, the number of threads operating on specific partitions are increased. When new operators are deployed, and existing operators exhibit low resource-utilization, the optimizer decides to scale down the existing operators.

The optimizer checks for horizontal scaling (④ in Figure 5.6) when new and potentially non-shared queries are created. Also, the optimizer decides to scale out if CPU or memory is a bottleneck. When the optimizer detects a latency skew, and there are no available resources to scale up, it triggers scaling out. In this case, the optimizer distributes the partition range, which is overloaded, among new nodes added to the cluster. Therefore, at runtime, the partition range might not be distributed evenly among all nodes.

## 5.5 Implementation Details

In this section, we discuss implementation details of AJoin. Specifically, we elaborate on operator-specific implementation details (Section 5.5.1) and exactly-once semantics (Section 5.5.2).

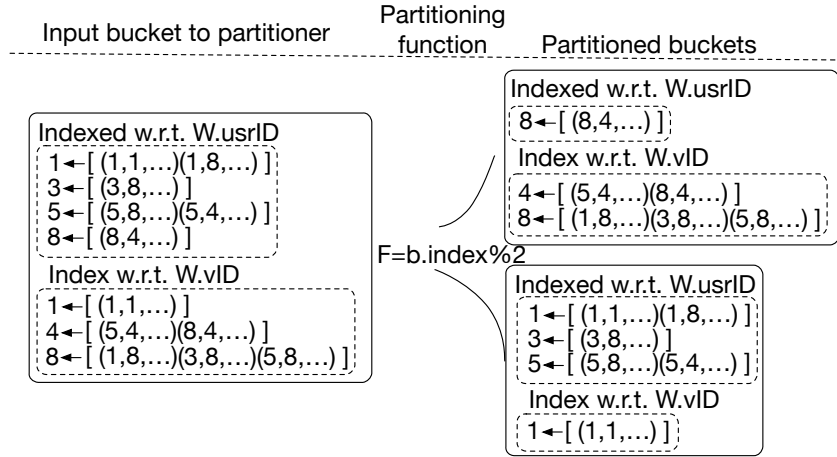


Figure 5.10: Example partitioning of the bucket described in Figure 5.5e

## 5.5.1 Join Phases

### 5.5.1.1 Bucketing

Bucketing is performed in the source operator. The source operator is the first operator in the AJoin QEP. Each index, inside a bucket, points to a list of tuples with the common key. If there are multiple indexes, pointers are used to reference stream tuples. The main intuition is that buckets are read-only; so, sharing the stream tuples between multiple concurrent queries (with different indexes) is safe.

Each source operator instance assigns a unique ID to the generated bucket; however, bucket IDs are not unique across different partitions. The bucket ID is an integer indicating the generation time of the bucket.

### 5.5.1.2 Partitioning

The partitioner is an operator that partitions buckets among downstream operator instances. This operator accepts and outputs buckets. Given an input bucket, the partitioner traverses over existing indexes of the bucket. It maps each index entry and corresponding stream tuples to one output bucket. In this way, the partitioner traverses only indexes instead of all stream tuples.

The partitioning strategy of AJoin with multiple queries is similar to one with a single query. If queries have the same join predicate, the partitioner avoids copying data completely. That is, each index entry and its corresponding tuples are mapped to only one downstream operator instance. If queries possess different join predicates, AJoin is able to avoid data copy partially. For example, in Figure 5.10 the input bucket, is partitioned into two downstream operator instances. Note that tuples that are partitioned to the same node w.r.t. both partitioning attributes (e.g.  $(1,1,\dots), (8,4,\dots)$ ) are serialized and deserialized only once, without data copy.

### 5.5.1.3 Join

Let  $L_{in}$  and  $L_{out}$  be lists inside a join operator storing buckets from inner and outer stream sources, respectively. When the join operator receives buckets,  $b_{in}$  from the inner and  $b_{out}$  from the outer stream source, it *i*) joins all the buckets inside  $L_{out}$  with  $b_{in}$ , all the buckets inside  $L_{in}$  with  $b_{out}$ , and combines the two results in one output bucket, *ii*) emits the output bucket, and *iii*) removes unnecessary buckets from  $L_{in}$  and  $L_{out}$ .

The join operator handles join queries with different join predicates and window constraints. The operator receives query changelogs from upstream operators and updates its query meta-data. Figure 5.11



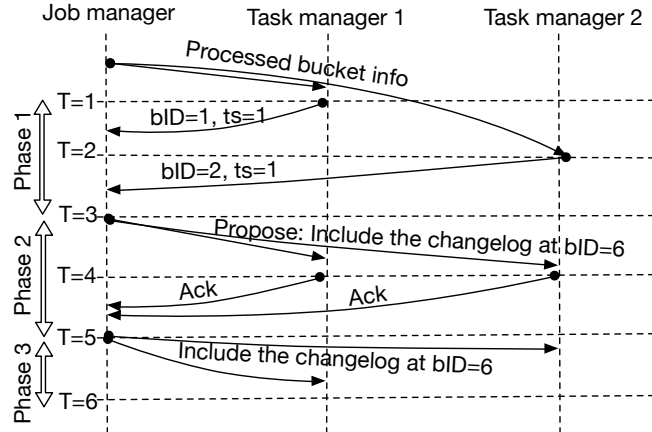


Figure 5.12: 3-phase atomic protocol

and the downstream join operators  $((...) \bowtie C)$  in each node. Also, the optimizer checks if all join operator instances are evenly distributed among the cluster nodes. It is acceptable if some nodes have free (idle) task slots. The free task slots provide flexibility for scaling up during the runtime. If there are join operators that share the same join partitioning attribute, the optimizer schedules them in the same task slot and notifies Flink to share the task slot between the two join operator instances. For example, in a query like  $(A \bowtie_{A.a=B.b} B) \bowtie_{B.b=C.c} C$ , the instances of the upstream join operator  $(A \bowtie_{A.a=B.b} B)$  share the same task slot with the instances of the downstream join operator  $((...) \bowtie_{B.b=C.c} C)$ . The reason is to ensure data locality, as the resulting stream of the upstream join operator is already partitioned w.r.t. the attribute  $B.b$ . The optimizer performs the necessary changes in the physical QEP generated by Flink (second phase) to perform the optimizations listed above.

## 5.6 Runtime QEP changes

In this section, we discuss the dynamicity of AJoin in the data processing layer. It is widely acknowledged that streaming workloads are unpredictable [124]. Supporting ad-hoc queries for streaming scenarios leads to more dynamic workloads. Therefore, AJoin supports several runtime operations updating the QEP on-the-fly. These operations are triggered by the optimizer and are submitted to task managers through the job manager. Below, we discuss consistency protocols of AJoin in Section 5.6.1. Then, we explain runtime QEP changes, such as vertical scaling (Section 5.6.2), horizontal scaling (Section 5.6.3), and join reordering (Section 5.6.4).

### 5.6.1 Consistency Protocols

AJoin features two consistency protocols: atomic and non-atomic. The atomic protocol is a three-phase protocol. Figure 5.12 shows an example scenario for this protocol. In the first phase, the job manager requests `bID`, the current bucket ID, and `ts`, the current time in the task manager, from all task managers. In the second phase, the job manager proposes the task managers to ingest the changelog after the bucket with `bID=6`. If the job manager receives `ack` from all task managers, it sends a confirmation message to the task managers to ingest the changelog. In the non-atomic protocol, on the other hand, the job manager sends the changelog without any coordination with task managers. Also, the task managers ingest the changelog without any coordination.

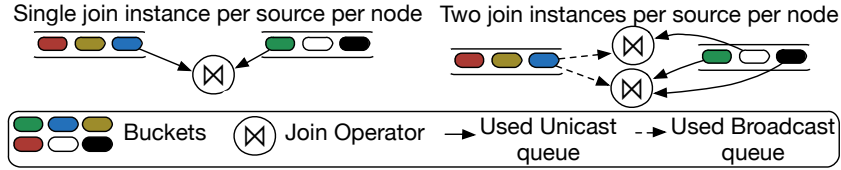


Figure 5.13: Scale up operation

### 5.6.2 Vertical Scaling

AJoin features two buffering queues between operators: a **broadcast queue** and a **unicast queue**. Let  $S$  be a set of subscribers to a queue. In the broadcast queue, the head element of the queue is removed if all subscribers in  $S$  pull the element. Any subscriber  $s_i \in S$  can pull elements up to the last element inside the queue. Afterwards, the subscriber thread is put to sleep mode and awakened once a new element is pushed into the broadcast queue. In a unicast queue, on the other hand, the head element of the queue is removed if one subscriber pulls it. The consequent subscriber pulls the next element in the queue.

The join operation is distributive over union ( $A \bowtie (B \cup C) = A \bowtie B \cup A \bowtie C$ ). We use this feature and the two queues to scale up and down efficiently. Each join operator subscribes to two upstream queues: one broadcast and one unicast queue. When a new join operator is initiated in the same worker node (scale up), it also subscribes to the same input channels. For example, in Figure 5.13 there are two queues. If we increase the number of join instances, then both instances would get the same buckets from the broadcast queue but different buckets from the unicast queue. As a result, the same bucket is joined with different buckets in parallel.

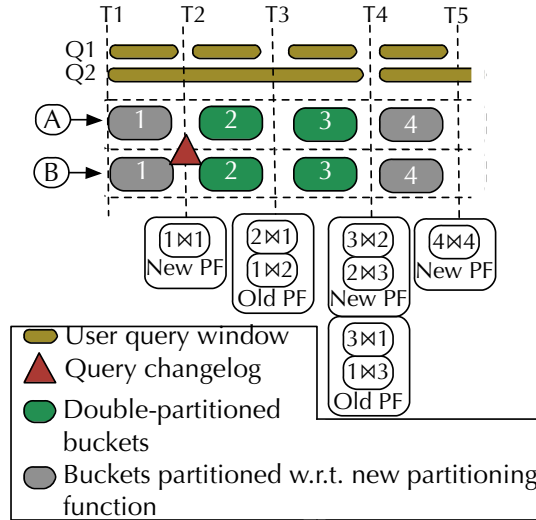
We use the non-atomic protocol for the vertical scaling. Let  $S1$  and  $S2$  be the two joined streams ( $S1 \bowtie S2$ ) and  $P = \{p_1, p_2, \dots, p_n\}$  be parallel partitions in which the join operation is performed. Vertical scaling in AJoin is performed on a partition of a stream (i.e., a vertical scaling affects only one partition). So, we show that the scaled partition produces correct results. Assume that  $k$  new task managers are created at partition  $p_i$ , which output join results to  $p_i^1, p_i^2, \dots, p_i^k$ . Since  $p_i^1 \cup p_i^2 \cup \dots \cup p_i^k = S1.p_i \bowtie S2.p_i$  (distributivity over union), the result of vertical scaling is correct. Since there is no synchronization among partitions, and since each vertically scaled partition is guaranteed to produce correct results, vertical scaling is performed in an asynchronous manner.

### 5.6.3 Horizontal Scaling

AJoin scales horizontally in two cases: when a new query is created (or deleted), and when an existing set of queries needs to scale out (or scale in). We refer to the first case as query pipelining. We assume that created or deleted queries share a subquery with running queries. Otherwise, the scaling is straightforward - adding new resources and starting a new job.

Query pipelining consists of three main steps. Let the existing query topology be  $E$  and the pipelined query topology be  $P$ . In the first step, the job manager sends a changelog to the task managers of  $E$ . Upon receiving the changelog, the task managers switch sink operators of  $E$  to the pause state and ack to the job manager. In the second step, the job manager arranges the input and output channels of the operators deployed inside the task managers, such that the input channels of  $P$  are piped to the output channels of  $E$ . In the third step, the job manager resumes the paused operators. If the changelog contains deleted queries, the deletion of the queries is performed similarly. The job manager pauses upstream operators of deleted stream topologies. Then, the job manager pipelines a sink operator to the paused operators. Lastly, the job manager resumes the paused operators.

Query pipelining is performed via the non-atomic protocol (Section 5.6.1). Thus, all the partitions of the pipelined query are not guaranteed to start (or stop) processing at the same time. However, modern



**Figure 5.14:** Partition function change operation. PF refers to the partitioning function

SPEs [3], [2], [5] also connect to data sources, such as Apache Kafka [125], in an asynchronous manner. Also, when a stream query in the modern SPEs is stopped, there is no guarantee that all sink operators stop at the same time.

Scaling out and in can be generalized to changing the partitioning function and computation resources. We explained the partitioning strategy in Section 5.4.3. Assume that AJoin scales out by  $N$  new nodes, and each node is assigned to execute  $P'$  partitions. Then, the new partitioning function becomes  $PF'(k) = (\text{Hash}(k) \bmod (|P| + |P' \cdot N|))$ . Also, each new node is assigned a partition range. The partition range is determined via further splitting the overloaded partitions. For example, if a partition with hashed key range  $[0, 10]$  is overloaded, and one new partition is initiated in the new node, then the hashed key ranges of the two partitions become  $[0, 5]$  and  $(5, 10]$ . The similar approach applies for scaling in.

The change of the partitioning function is completed in three steps. Assume that the partitioning function of a join operator is modified. There are multiple queries using the join operator with different window configurations. In the first step, the job manager retrieves the biggest window size, say  $BW$ . In the second step, the job manager sends a partition-change changelog via the atomic protocol. Once the partitioner receives this marker, it starts double partitioning, meaning partitioned buckets contain data both w.r.t the old and new partitioning function. The partitioner performs double-partitioning at most  $BW$  time, where  $BW$  is the length of biggest window. Then partitions only w.r.t. the recent partitioning function. In the third step, new task managers are launched (scale out) or stopped (scale in).

Figure 5.14 shows an example scenario for a partitioning function change. First buckets from the stream A and B have single partitioning info (i.e., they contain tuples partitioned w.r.t. a single partitioning function). At time T2 the partition-change changelog arrives at the join operator. So, the tuples arriving before T2 no longer have the new or latest partitioning schema. At time T3, the second and first buckets are joined w.r.t. the old partitioned data. At time T4, the third and second buckets are joined w.r.t. the new partitioned data; however, the third and first buckets are joined w.r.t. the old partitioned data. Starting from T4, the partitioner stops double-partitioning and switches to the new partitioning function.

We use the atomic protocol when changing the partitioning function. Changing the partitioning function possibly affects all partitions. In order to guarantee the correctness of results, there are two main requirements: *i*) all partition operators must change the partitioning function at the same time and *ii*) downstream operators must ensure the consistency between the data partitioned w.r.t. new and old partitioning functions. To achieve the first requirement, we use the atomic 3-phase protocol. To achieve the second requirement, we use a custom join strategy in which we avoid to join old-partitioned and

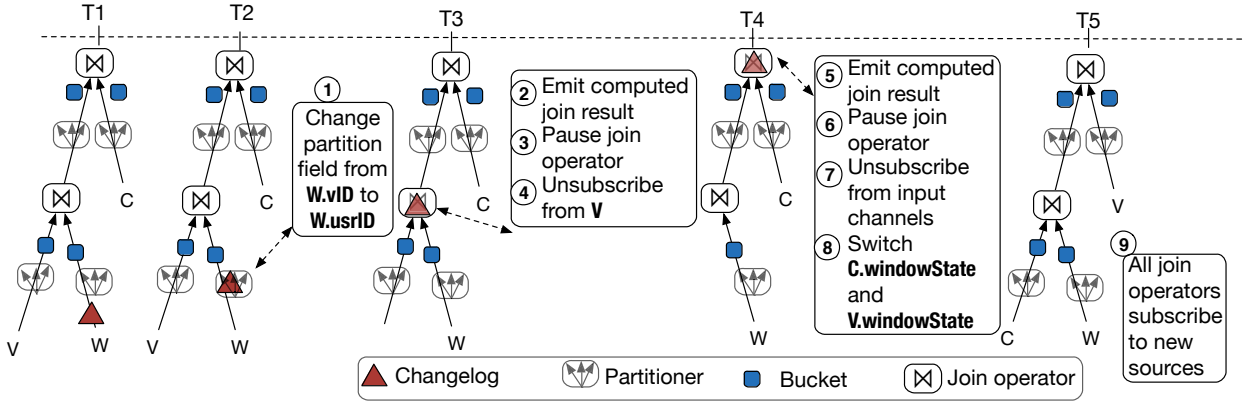


Figure 5.15: Join reordering

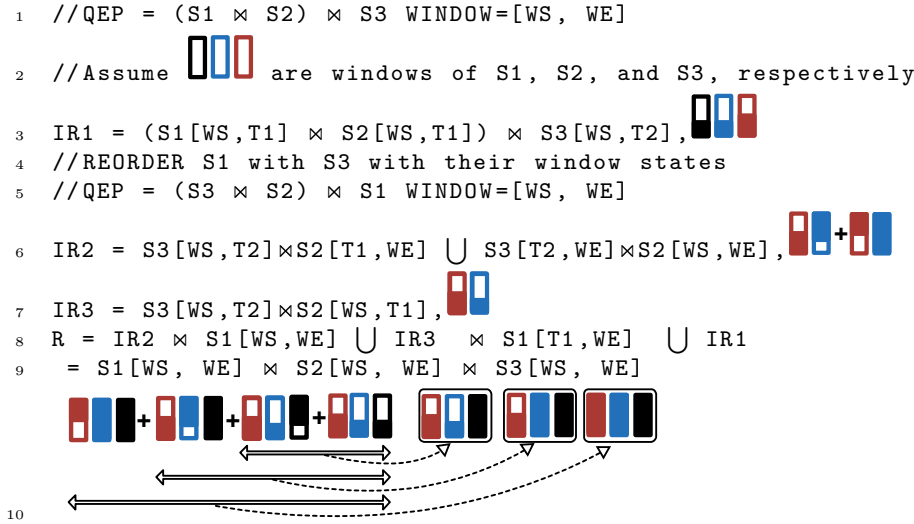
new-partitioned data. Instead, we perform double-partitioning and ensure that any joined two tuples are partitioned w.r.t. the same partitioning function. We apply the similar technique, mentioned above, when query groups are changed.

### 5.6.4 Join Reordering

Suppose at time  $T_{1D}$ , the optimizer triggers to change the QEP of Q2 from  $(V \bowtie_{V.vID=W.vID} W) \bowtie_{W.usrID=C.usrID} C$  to  $V \bowtie_{V.vID=W.vID} (W \bowtie_{W.usrID=C.usrID} C)$ . Figure 5.15 shows the main idea behind reordering joins. At time T1, the job manager pushes the changelog marker via the non-atomic protocol. The marker passes through the partitioner at time T2. The marker informs the partitioner to partition based on  $W.usrID$ , instead of  $W.vID$ . At time T3, the changelog marker arrives at the first join operator. Having received the changelog, the join operator emits the join result, if any, and acks to the job manager. The job manager then *i*) pauses the join operator and *ii*) unsubscribes it from stream V. At time T4, the marker arrives at the second join operator. Similarly, the second join operator emits the join result, if any. It informs the job manager about the successful emission of results. The job manager pauses the operator and unsubscribes it from input channels. Afterwards, the second join operator switches its state with the upstream join operator. Finally, the job manager subscribes both join operators to the modified input channels and resumes computation.

We use the non-atomic protocol for join reordering. The reason is that join reordering is performed in all partitions, independently. Assume that S1, S2, and S3 are streams, W denotes window length, WS and WE are window start and end timestamps, and T1 and T2 are timestamps in which the changelog arrives at the first and the second window. Figure 5.16 shows the formal definition of the join reordering. When the changelog arrives at the first join operator, the intermediate join result (IR1 in Figure 5.16) is computed and emitted. At this point, AJoin switches the window states of S1 and S3. Then, unjoined parts of S3 and S2 are joined (IR2 in Figure 5.16). Although IR3 is included in IR1, IR3 is joined with S1[T1,WE] in the final phase; therefore, the result is not a duplication. Finally, AJoin combines all intermediate results to the final output (R in Figure 5.16), which is correct and does not include any duplicated data.





**Figure 5.16:** Formal definition of join reordering. The black, blue, and red boxes represent the windows of S1, S2, and S3. Filled boxes mean that the respective portion of the boxes are joined.

## 5.7 Experiments

### 5.7.1 Experimental Design

Our benchmarking framework consists of a distributed driver and two SUTs: Apache Flink v1.7.2 and AJoin. The driver maintains two queues: one for stream tuples and one for user requests (query creation or deletion). The tuple queue receives data from tuple generators inside the driver. The driver generates tuples at maximum sustainable throughput [61]. Tuple creation time is appended to each stream tuple as event-time information. A SUT pulls tuples from the data queue with the highest throughput it can process. So, the longer the tuple stays in the queue, the higher its event-time latency. The working principle of the user request queue is similar to the tuple queue.

The input tuples inside the driver are pulled by the SUT. If the SUT exhibits backpressure, it automatically reduces the pull rate. Contrary to data tuples, user requests are periodically pushed to the client module of the SUT. The SUT acks to the driver, after receiving the user request. If the *ack timeout* is exceeded or every subsequent ack duration keeps increasing, then the SUT cannot sustain the given query throughput. Similarly, if there is an infinite backpressure, then the SUT cannot sustain the given workload. In these cases, the driver terminates the experiment and tests the SUT again with a lower query and data throughput. We adopt the workloads from Chapter 4.

### 5.7.2 Metrics and Data Generation

**Query similarity** shows the similarity between the generated query and the pattern query. Equation 5.3 shows the calculation of the query similarity. To evaluate the similarity between a query Q (e.g.  $A \bowtie_{A.a=B.a} B$ ) and the pattern query PQ (e.g.  $A \bowtie_{A.a=B.b} B$ ), we *i*) find the number of the common sources (ComS) between Q and PQ (A and B), *ii*) find the number of the common sources with common join attributes (ComSJA) between Q and PQ (only A.a), and *iii*) divide the multiplication of the two ( $2 \times 1$ ) with square of all sources (AllS) in PQ ( $2^2$ ).

$$\text{Similarity}(\text{ComS}, \text{ComSJA}, \text{AllS}) = \frac{\text{ComS} * \text{ComSJA}}{(\text{AllS})^2} \quad (5.3)$$

```

1 SELECT *
2 FROM S1, S2, ..., Sn WINDOW=[1|2|3|] sec
3 WHERE S1.[JA1] = S2.[JA2] AND
4         S2.[JA3] = S3.[JA4] AND
5         ...
6         Sn-1.[JAj-1] = Sn.[JAj]
7 AND
8 S1.[SA1] [=|>|<|>=|<=] [FV1] AND
9 S2.[SA2] [=|>|<|>=|<=] [FV2] AND
10 ...
11 Sn.[SAn] [=|>|<|>=|<=] [FVn] AND
    
```

**Figure 5.17:** Query template used in our experiments.  $S_n[i]$  means  $i^{th}$  attribute of stream  $n$ .  $JA_i$  (join attribute) and  $SA_i$  (selection attribute) ( $0 \leq JA_i, SA_i < |6|$ ) are random variables (e.g.,  $S_n[SA_i]$  is  $SA_i^{th}$  attribute of stream  $S_n$ ).  $FV_i$  (filtering value) is a randomly assigned value used to filter streams.

To generate query  $Q$  with  $n\%$  similarity with  $PQ$ , we apply the following approach. Assume that  $n$  is 75% and  $PQ$  is  $A \bowtie_{A.a=B.b1} B \bowtie_{B.b2=C.c} C$ .

1. We randomly select  $ComS$ , which is between 1 and  $AllS$  (e.g.,  $ComS=2$ ).
2. Given  $ComS=2$  and  $AllS=3$ , we calculate  $ComSJA$  from Equation 5.3. If a stream is a source stream, it is partitioned by the join attribute of the downstream join operator. If a stream is an intermediate result, two join operators affect the sharing possibility of this stream: the upstream join operators (how the stream was partitioned) and the downstream join operator (how the stream will be partitioned). Therefore, each stream can be affected by maximum of two partitioning attributes. If  $ComSJA$  number of join attributes cannot be used with  $ComS$  number of sources (e.g., 1 stream source can be affected by maximum of 2 join attributes), then we increase  $ComS$  by one and repeat this step.
3. We select  $ComS$  number of random stream sources from  $PQ$ , such that these stream sources are joined with each other with a join predicate and not via cross-product (e.g.,  $A \times C$  is not acceptable). Similarly, we select random  $ComSJA$  number of join attributes from the selected sources.

Figure 5.17 shows a query template for join query generation. For each stream source, we add a selection predicate. After filtering, we join stream sources based on randomly selected join attributes. All attributes of the data tuples can be used as a join attribute. The window length of the generated query is either 1, 2, or 3 seconds. We perform window duration, selection predicate, and join predicate assignment in a uniform manner.

Each data tuple features 6 attributes. Each attribute of a tuple is generated as a random uniform variable between 0 and  $ATR\_MAX$ . We set different seed values for data generation per each stream source. We use the Java `Random` class for uniform data generation. Throughout our experiments, we set  $ATR\_MAX$  to be 500. The data generation speed for all stream sources is equal.

### 5.7.3 Workload

Figure 5.18 shows the two workload scenarios we investigate in our experiments. The first workload scenario (SC1), shown in Figure 5.18a, is applicable when a user activity is higher on specific time periods. Also, in this workload scenario, users execute long-running queries. Figure 5.18b shows the second workload scenario (SC2). The second workload scenario is relevant for fluctuating workloads. Modern SPEs cannot execute ad-hoc stream queries. Therefore, there is no industrial workload for ad-hoc stream query processing. Therefore, we use the workload used in Chapter 4, which is similar to cloud

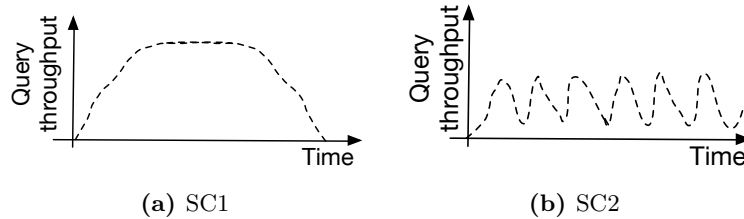


Figure 5.18: Two scenarios for ad-hoc query processing environments

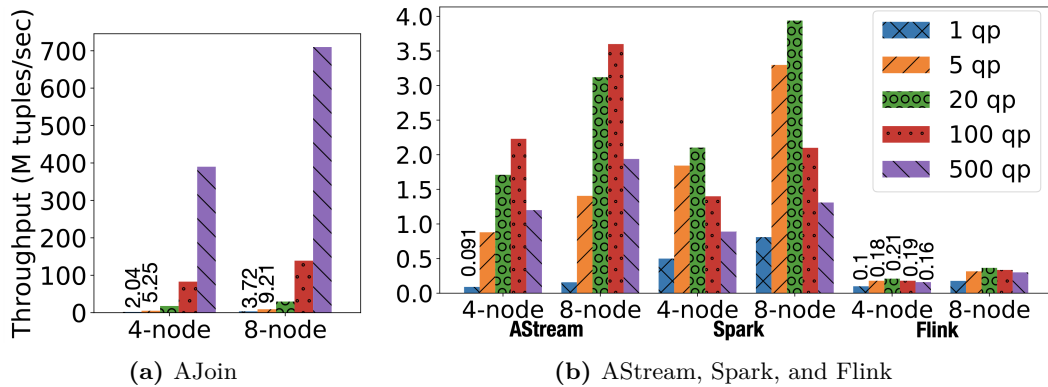


Figure 5.19: Overall data throughput of AJoin, AStream, Spark, and Flink with 1, 5, 20, 100, and 500 parallel queries on 4- and 8-node cluster configurations. qp at the legend means query parallelism.

workloads [126, 127, 128, 129, 130, 131]. Nevertheless, the design of AJoin is generic and not specific to the workloads shown in Figure 5.18.

#### 5.7.4 Setup

We conduct experiments in 4- and 8-node cluster configurations. Each node features 16-core Intel Xeon CPU (E5620 2.40GHz) and 48 GB main memory. We configure the batch interval of queries (in the client module) to be 1 second and ack timeout is 15 seconds, as these configurations are the most suitable for our workloads. The latency threshold for scaling up and out is 5 seconds. The threshold is derived from the latency-aware elastic scaling strategy for SPEs [132]. We measure the sustainable performance of the SUTs [61] to detect if the latency spike is due to backpressure or unsustainable workload. If the latency value is higher than a given threshold because the system cannot sustain the workload, then AJoin scales up or out. Each created query in AJoin features this threshold value. For simplicity, we set the same threshold value for all queries. However, the overall methodology remains the same with different threshold values for each query.

#### 5.7.5 Scalability

Figure 5.19 shows the impact of scalability on the performance of the SUTs. All the queries are submitted to the SUTs at compile-time. The queries are 2-way joins and have 50% query similarity. For this experiment, we remove selection predicates from input queries to measure the performance of pure join operation. We can observe that the performance of all SUTs increases with more resources. Also, with more parallel queries, the overall data throughput of AJoin increases dramatically. The reason is that sharing opportunities increase with more parallel queries.

The throughput of AStream is significantly lower than AJoin. The reason is that AStream performs scan, data, and computation sharing if the input queries have common join predicate. Queries with different join predicates are deployed as separate stream jobs. The computation sharing in AStream is

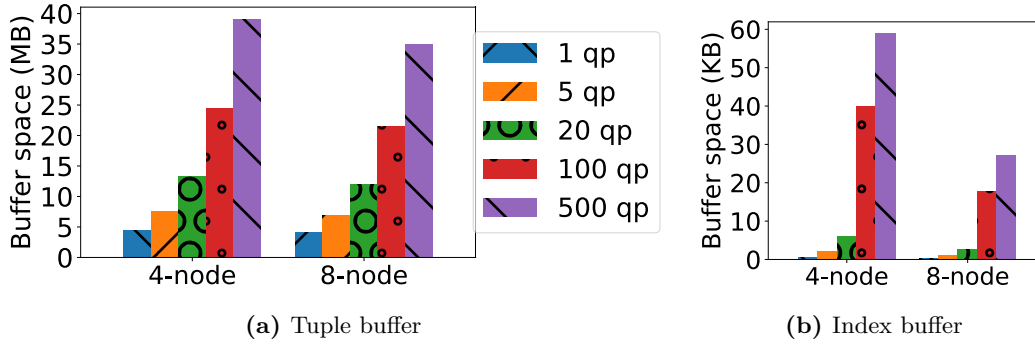


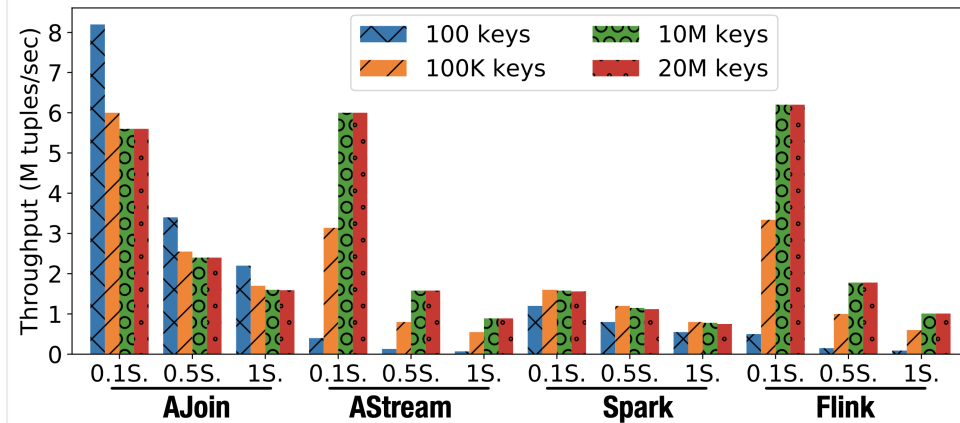
Figure 5.20: Buffer space used for tuples and indexes inside a 1-second bucket

not always beneficial (e.g., Figure 5.7). Because AJoin supports cost-based optimization, in addition to rule-based optimization, it groups queries in query groups and shares the data and computation if the sharing is beneficial. AStream supports static QEP. Each query eagerly utilizes all available resources. Also, AStream utilizes nested-loop joins.

We execute Spark with hash join implementation with the Catalyst optimizer [65]. With multiple queries, submitted at compile-time, the optimizer shares common subqueries, such as joins with the same join predicate. The sharing is possible because there is no selection predicate. For queries with selection predicates, Spark cannot share the computation and data. For joins with different join predicates, Spark deploys a new QEP. Also, Spark does not utilize late materialization. The hashing phase in Spark is blocking. It uses a blocking stage-oriented architecture.

AJoin performs better than AStream, Spark, and Flink even with single query setups. The reason is the join implementation of AJoin. AJoin uses not only data parallelism (like AStream, Spark, and Flink) but also pipeline parallelism for the join operation. The join operator in AStream, Spark, and Flink remains idle and buffers input tuples until the window is triggered. AStream and Flink perform nested-loop joins after the window is triggered. AJoin performs windowing in the source operator. While the tuples are buffered, they are indexed on-the-fly. Therefore the load of join operator is lower in AJoin, as it performs the set-intersection operation. After the join is performed, AStream, Spark, and Flink create many new data objects. These new objects cause extensive heap memory usage. AJoin reuses existing objects, keeps them un-joined (late materialization), and performs full materialization at the sink operator. Because the data tuples are indexed, AJoin avoids to iterate all the data elements while joining them, but only indexes. Also, at the partitioning phase, AJoin iterates over indexes to partition a set of tuples with the same index at once, rather than iterating over each data tuple. Different from Flink, AJoin performs incremental join computation. Quantifying the impact of each component (e.g., indexing, grace join usage, late materialization, object reuse, task-parallelism, etc) stated above, is nontrivial because these components function as an atomic unit. If we detach one component (e.g., indexing), then the whole join implementation would fail to execute. However, there is a significant improvement in throughput from 0.1 M t/s in Flink to 2.04 M t/s in AJoin.

Figure 5.20 shows the space used to buffer tuples and indexes in AJoin. With more queries, AJoin buffers more tuples and indexes. However, AJoin shares tuples among different queries and avoids new object creation and copy. The buffer size increases more for indexes than for tuples. The reason is that each tuple might be reused by different indexes. In this figure, the key space is between 0 and 500. When we increase the key space in the orders of millions, the index buffer space also increases significantly. Although this increase did not cause significant overhead in our setup (48GB memory per node), with low-memory setups and with very large key space, index usage causes significant overhead for AJoin.



**Figure 5.21:** The effect of the number of distinct keys in stream sources and the selectivity of selection operators on the performance of AJoin, AStream, Spark, and Flink. Values on the x-axis show the selectivity of selection operators.

### 5.7.6 Distinct Keys

Figure 5.21 shows the effect of distinct keys and the selectivity of selection predicates on the performance of the SUTs. Given that the data throughput is constant, with less distinct keys Flink and AStream output more tuples as a result of the cross product. This leads to an increase in data, computation, copy, serialization, and deserialization cost. With more distinct keys the performance of AJoin decreases, because AJoin cannot benefit from the late materialization. At the same time, the performance of Flink and AStream increases, because it performs fewer cross products and data copy. As the number of distinct keys increases, the throughput of Spark first increases then decreases slowly. The reason is that Spark utilizes hash join. With more keys, maintaining the hash table in memory becomes costly.

As the selectivity of the selection operator increases, the performance of all SUTs decreases. The decrease is steep in Flink and AStream. The reason is that the performance of the low-selective selection operator dominates the overall throughput. When the selectivity increases, data copy and inefficient join implementation become the bottleneck for the whole QEP.

The effect of the selectivity on Spark is more stable than other systems. In other words, as the selection operator filters more tuples, the overall performance of Spark does not exhibit an abrupt increase. Although Spark utilizes a hash join implementation, it adopts a stage-oriented mini-batch processing model. For example, hashing and filtering are separate stages of the job, which operate on the whole RDD. The subsequent stage cannot be started if all the parent stages are not finished. AJoin, AStream, and Flink, however, perform a tuple-at-a-time processing model. Therefore, the throughput performance of these systems is mainly dominated by the performance of filtering operators, especially with low-selectivity selection operators. Also, Spark’s hash join implementation includes a blocking phase (hashing). Flink and AStream, on the other hand, perform a nested-loop join, which performs better with less data (after the filtering phase).

### 5.7.7 Dynamicity

#### 5.7.7.1 Latency

In this section, we create and delete queries in an ad-hoc manner. Figure 5.22 shows the event-time latency of stream tuples for SC1. Since Flink cannot sustain ad-hoc query workloads, we show its event-time with a single query. During our experiments, we choose the selectivity of filter operators to be approximately 0.5. Although event-time latency of Flink is comparable with AJoin, the data throughput is significantly

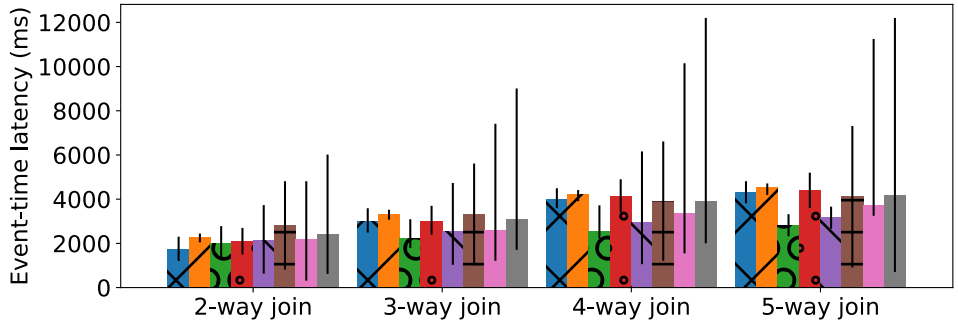


Figure 5.22: Average event-time latency of stream tuples with min and max boundaries for SC1

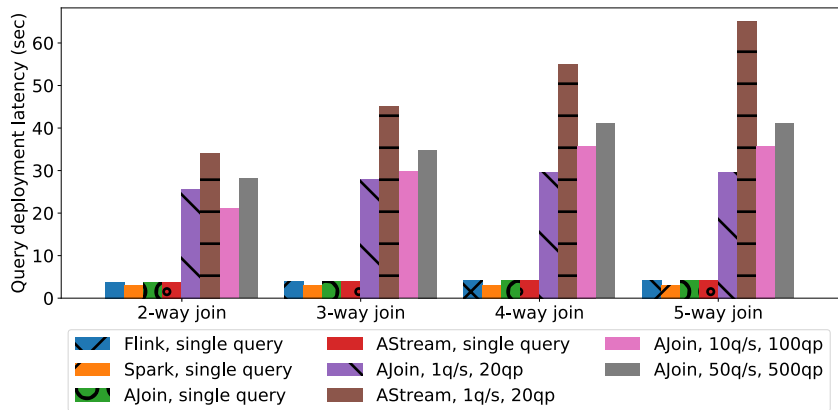


Figure 5.23: Deployment latency for SC1.  $i$ q/s  $j$ qp indicates that  $i$  queries per second were created until the query parallelism is  $j$ .

less than AJoin (Figure 5.19). The error bars in the figure denote the maximum and minimum latency of tuples during the experiment. In SPEs the latency of tuples might fluctuate due to backpressure, buffer size, garbage collection, etc. [61]. Therefore, we measure the average latency of the tuples.

The event-time latency increases with 3-, 4-, and 5-way join queries. The reason is that a streaming tuple traverses through more operators in the QEP. As the query throughput increases, so does the gap between latency boundaries. The reason is that AJoin performs runtime optimizations, which result in high latencies for some tuples. However, these high latencies can be regarded as outliers, because of much lower average latency.

The overall picture for event-time latency is similar for SC2. The only difference is that the average latency is lower and latency fluctuations are wider than SC1. The reason is that in SC2, the average number of running queries are less than SC1, which results in lower average event-time latency. The query throughput is higher in SC2, which results in more fluctuations in event-time latency.

Figure 5.23 shows the deployment latency for SC1 in AJoin. The experiment is executed in a 4-node cluster. The query similarity again is set to 50%. The query deployment latency for 1qs 20qp (create one query per second until there are 20 parallel queries) is higher than 10qs 100qp with 2-way joins. The reason is that query batch time is one second, meaning user requests submitted in the last second are batched and sent to the SUT. However, with 3- and 4-way joins, the overhead of on-the-fly QEP changes also contributes to query deployment latency.

### 5.7.7.2 Breakdown

Figure 5.24 shows a breakdown of the overhead by the AJoin components. We initialize AJoin with a 2-node cluster configuration and enable it to utilize up to 25 nodes. The overhead is based on the

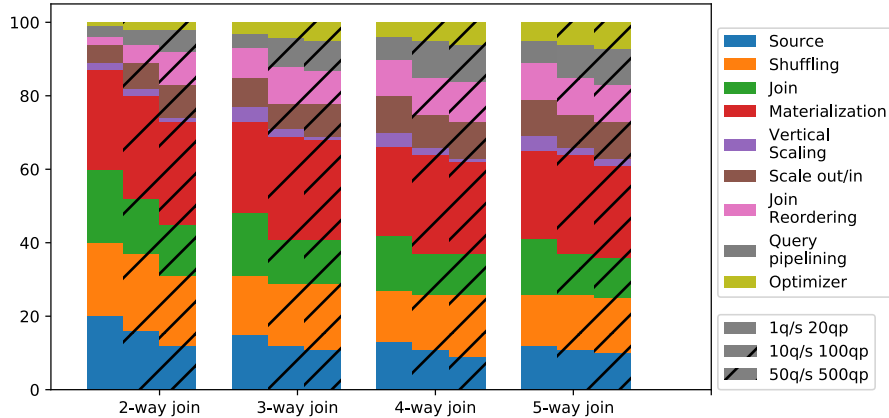


Figure 5.24: Breakdown of AJoin components in terms of percentage for SC1

event-time latency of stream tuples. In this experiment, we ingest a special tuple to the QEP every second. Every component shown in Figure 5.24 logs its latency contribution to the tuple.

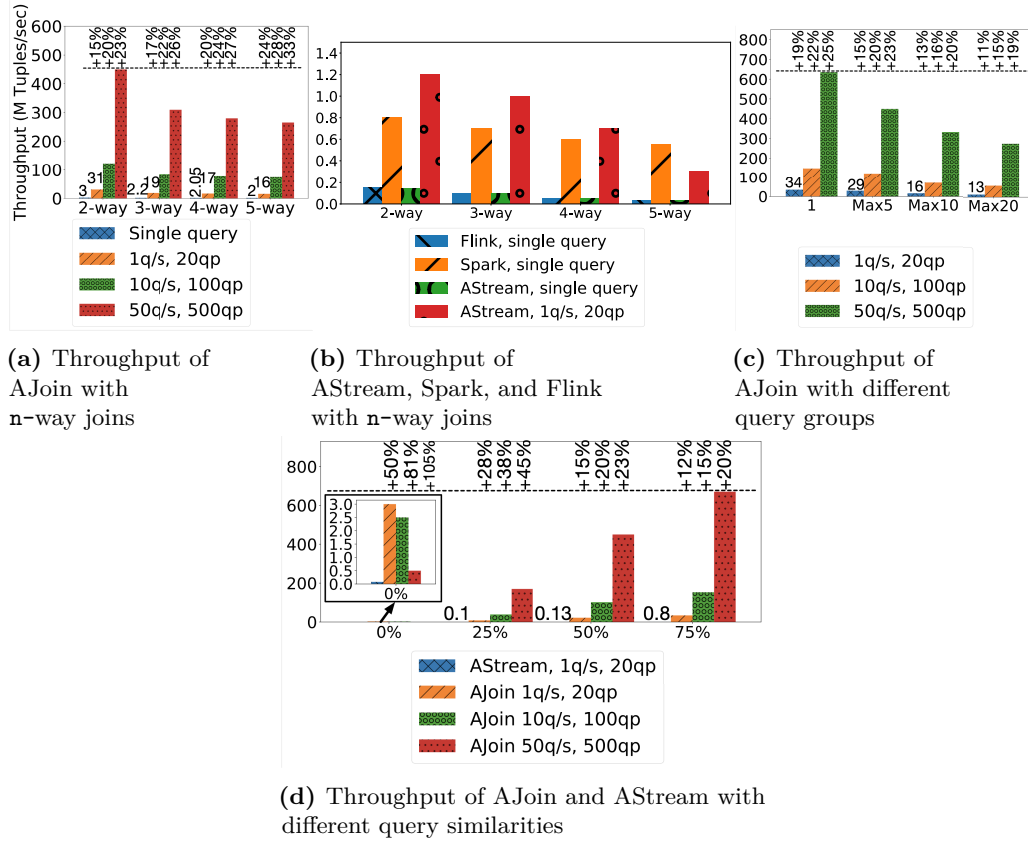
Note that the overhead of source, join, and materialization components are similar. This leads to a higher data throughput in the QEP. As the query throughput increases, the proportional overhead of horizontal scaling increases. The reason is that the optimizer eagerly shares the biggest subquery of a created query and eagerly deploys the remaining part of the query. Although the 3-phase protocol avoids stopping the QEP, it also has an impact on the overall latency. With 3-way and 4-way joins, the cost of query pipelining and join reordering also increases. With more join operators in a query, subquery sharing opportunities are high. So, the optimizer frequently pipelines the part of the newly created query to the existing query. Also, we can see that materialization is one of the major components causing latency. The reason is that tuples have to be fully materialized, copied, serialized, and sent to different physical output channels. We notice that similar overhead of source, join, and materialization leads to a higher data throughput (e.g., the throughput of 2-way is higher than others). The reason is that when  $n$  ( $n$ -way join) increases, new stream sources, join operators, and sink operators are deployed. Therefore, the overall overhead for these operators remains stable. The overhead of the optimizer also increases as  $n$  ( $n$ -way join) gets higher and as query throughput increases. The reason is that the sharing opportunities increase with more queries and with 3- and more  $n$ -way joins.

### 5.7.7.3 Throughput

Figure 5.25 shows the effect of  $n$ -way joins, query groups, and query similarity to the performance of the SUTs. We show the performance improvement of AJoin when submitting queries at compile-time above the dashed lines in the figure. As  $n$  increases in  $n$ -way joins, the throughput of AJoin drops (Figure 5.25a). The performance drop is sharp from 2-way join to 3-way join. The reason is that 3- and more way joins benefit from the late materialization more. Also, the performance difference between ad-hoc and compile-time query processing increases as the query throughput and  $n$  increase.

Figure 5.25b shows the throughput of AStream, Spark, and Flink with  $n$ -way join queries. Because of the efficient join implementation, Spark performs better than other SUTs with single query execution. The performance of Flink and AStream decreases with more join operators. In some 4- and 5-way join experiments, Flink and AStream were stuck and remained unresponsive. The reason is that each join operator creates new objects in memory, which leads to intensive, CPU, network usage and garbage collection stalls. While Spark also performs data copy, its Catalyst optimizer efficiently utilizes on-heap and off-heap memory to reduce the effect of data copy on the performance.

## 5. AJoin: Ad-hoc Stream Joins at Scale



**Figure 5.25:** Throughput measurements for AJoin, AStream, Spark, and Flink. +P% above the dashed lines denote that the throughput increases by P% when queries are submitted at compile-time.

Figure 5.25c shows the effect of the number of query groups on the performance of AJoin. With more query groups the throughput of AJoin decreases. However, the decreasing speed slows down gradually. Although there are less sharing opportunities with more query groups, updating the QEP becomes cheaper (as a result of incremental computation). The incremental computation also leads to a decrease in the overhead of executing queries ad-hoc.

Figure 5.25d shows the effect of query similarity on the performance of the SUTs. Both AStream and AJoin perform better with more similar queries. However, the performance increase is higher in AJoin. AStream lacks all the runtime optimization techniques AJoin features. As a result, AStream shares queries only with the same structure (e.g., 2-way joins can be shared only with 2-way joins) and the same join predicates. The effect of executing queries in an ad-hoc manner decreases as the query similarity increases. The overall picture in SC2 is similar with SC1.

### 5.7.7.4 Impact of Each Component

Figure 5.26 shows the impact of AJoin’s optimization components on the performance. In this experiment, we disable one optimization component (e.g., join reordering) and measure the performance drop. When the number of join operations in a query increases, the impact of join reordering and query pipelining also increase. Also, with more query throughput, the optimizer shares input queries aggressively. Therefore, the impact of the query pipelining increases with higher query throughput. As the number of query groups increases, the impact of the join reordering optimization decreases because of the drop in sharing opportunities. This also leads to the extensive use of scaling out and in. When all queries are dissimilar, the join reordering and query pipelining have zero impact on overall execution. With more similar queries, the effect of other components, especially the join reordering component, increases.



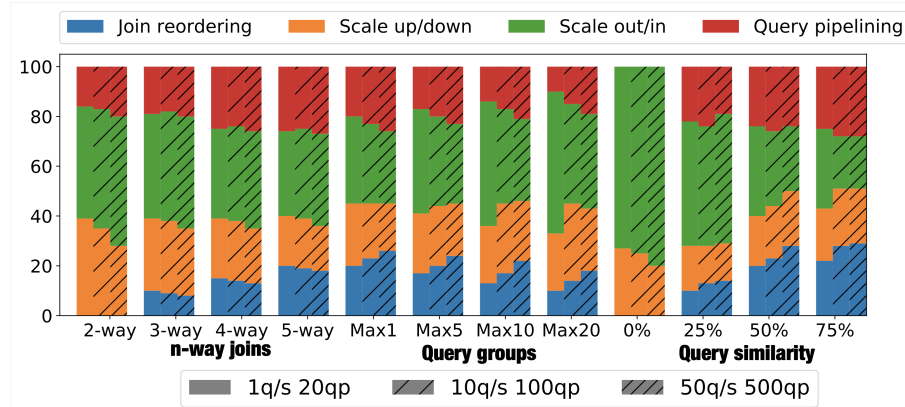
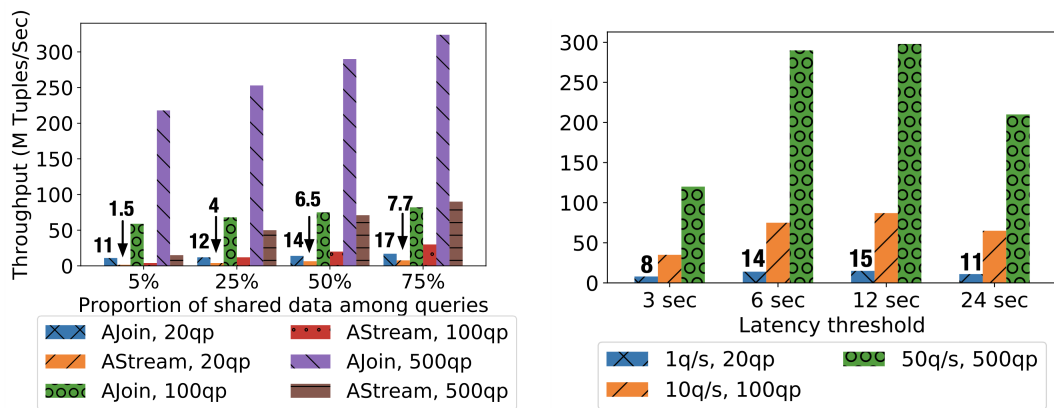


Figure 5.26: Impact of AJoin components in terms of percentage



(a) Impact of data sharing and query-set payload on the throughput of AJoin and AStream

(b) Impact of the latency threshold value on the throughput of AJoin

Figure 5.27: Cost of data sharing and the impact of the latency threshold value with 3-way join queries

The overall picture is similar in SC2. The most noticeable difference is that the impact of scaling out and in is less, and the impact of join reordering is more. The execution time and the query throughput in SC1 are higher than SC2. In SC2, queries are not only created but also deleted with lower throughput. This leads to a higher impact on join reordering.

### 5.7.7.5 Cost of Sharing

Figure 5.27a shows the performance of AStream and AJoin with four input streams: 5%, 25%, 50%, and 75% shared. For example, 50% shared data source means that tuples are shared among 50% of all queries. We omit experiments with 0% shared data source, as in this scenario all the data tuples are filtered and no join operation is performed. We perform this experiment with a workload suitable for AStream (i.e., all join queries have the same join predicate and the same number of join operators) and disable the dynamicity property (except query grouping) of AJoin. This setup enables us to measure the cost of sharing and query-set payload of AStream and AJoin. As the proportion of shared data decreases, the performance gap between AStream and AJoin increases. The reason is that AJoin performs query grouping that leads to an improved performance (Figure 5.7). The impact of the query grouping is more evident when the proportion of shared data is small.

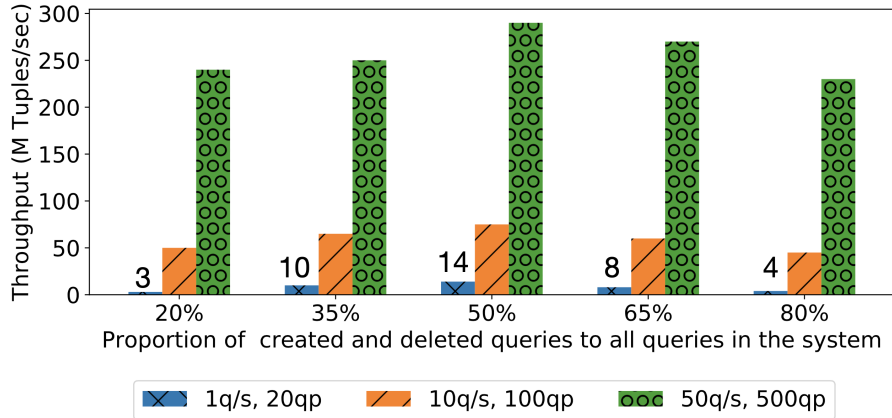


Figure 5.28: Impact of the threshold value of query reoptimization on the performance of AJoin

### 5.7.7.6 Impact of the Latency Threshold Value

Figure 5.27b shows the throughput of AJoin with different latency threshold values. The latency threshold value, which is 5 seconds in our experiments, needs to be configured carefully. When it is too low (3 seconds in Figure 5.27b), we experience an overhead for frequent optimizations. When it is too high (24 seconds in Figure 5.27b), there is a loss in optimization potential.

### 5.7.7.7 Impact of the Query Reoptimization Threshold Value

If the number of created and deleted queries exceeds the threshold value of query reoptimization, the optimizer computes a new plan (including the query groups) holistically instead of incrementally. Figure 5.28 shows the impact of the threshold value on the performance of AJoin. When the threshold value is low (20% and 35%), we experience an overhead for frequent optimizations. When it is high (65% and 80%), there is a loss in optimization potential.

## 5.8 Conclusion

In this chapter we presented AJoin, an ad-hoc stream join processing engine. We developed AJoin based on two main concepts: (1) Efficient distributed join architecture: AJoin features pipeline-parallel join architecture. This architecture utilizes late materialization, which significantly reduces the amount of intermediate results between subsequent join operators; (2) Dynamic query processing: AJoin features an optimizer, which reoptimizes ad-hoc stream queries periodically at runtime, without stopping the QEP. Also, the data processing layer supports dynamicity, such as vertical and horizontal scaling and join reordering;

We benchmarked AJoin, AStream, Spark, and Flink. When all the queries were submitted at compile-time, AJoin outperformed Flink by orders of magnitude. With single query workloads, AJoin also outperformed AStream, Spark, and Flink. With more join operators in a query (3-, 4-, 5-way joins) the performance gap between AJoin and the other systems even increased. With ad-hoc stream query workloads, Flink and Spark could not sustain the workload, and AStream’s performance was less than AJoin’s.

# 6

## Additional Contributions

This chapter outlines additional research contributions which have been made by the author while working on this thesis. Although these additional contributions, which are listed below, are not part of the thesis contents, they are closely related to the thesis topic.

- Bonaventura Del Monte, **Jeyhun Karimov**, Alireza Rezaei Mahdiraji, Tilmann Rabl, Volker Markl, Harry Xuegang Huang, Christian Thomsen.  
*PROTEUS: Scalable online machine learning for predictive analytics and real-time interactive visualization.*  
In Proceedings of the 1st International Workshop on Big Data Management in European Projects (EuroPro) 2017.
- **Jeyhun Karimov**, Tilmann Rabl, Volker Markl.  
*PolyBench: The First Benchmark for Polystores.*  
In Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC) 2018.
- Steffen Zeuch, Bonaventura Del Monte, **Jeyhun Karimov**, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl.  
*Analyzing Efficient Stream Processing on Modern Hardware.*  
In Proceedings of the International Conference on Very Large Data Bases (PVLDB), 2019.

In the paper *PROTEUS: Scalable online machine learning for predictive analytics and real-time interactive visualization*, we design, develop, and provide an open-source and ready-to-use big data software architecture. The architecture is able to handle extremely large historical data and data streams and supports online machine learning predictive analytics and real-time interactive visualization. The overall evaluation of PROTEUS is carried out using a real industrial scenario.

Modern business intelligence requires data processing not only across a huge variety of domains but also across different paradigms, such as relational, stream, and graph models. This variety is a challenge for existing systems that typically only support a single or few different data models. Polystores were proposed as a solution for this challenge and received wide attention both in academia and in industry. These are systems that integrate different specialized data processing engines to enable fast processing of a large variety of data models. Yet, there is no standard to assess the performance of polystores. In the paper *PolyBench: The First Benchmark for Polystores* we develop the first benchmark for polystores. To

capture the flexibility of polystores, we focus on high level features in order to enable an execution of our benchmark suite on a large set of polystore solutions.

In the paper *Analyzing Efficient Stream Processing on Modern Hardware* [14], we conduct an extensive experimental analysis of current SPEs and SPE design alternatives optimized for modern hardware. We reveal potential bottlenecks of modern SPEs and show that they do not exploit the full power of current and emerging hardware trends, such as multi-core processors and high-speed networks. We propose a set of design changes to the common architecture of SPEs to scale-up on modern hardware. Our experimental results show that the single-node throughput can be increased by up to two orders of magnitude compared to state-of-the-art SPEs by applying specialized code generation, fusing operators, batch-style parallelization strategies, and optimized windowing. This speedup allows for deploying typical streaming applications on a single or a few nodes instead of large clusters.

# 7

## Conclusion and Future Research

This thesis establishes fundamentals for ad-hoc stream query processing. Also, it lays the groundwork for objectively evaluating SPEs. Objective and realistic evaluation of SPEs is essential not only for ad-hoc stream query processing but also for any system analysis procedure. The major challenges and contributions in this thesis follow a general-to-specific pattern. First, we analyze current challenges in benchmarking SPEs. We propose the first benchmarking framework design that *i)* is able to compute the latency and throughput for stateful streaming operators, *ii)* separates the SUT and the test driver completely, and *iii)* measures the sustainable performance of SPEs. Second, we analyze modern SPEs with a new workload, i.e., with ad-hoc stream queries. Realising that the modern SPEs are not capable of executing ad-hoc stream queries, we propose the first ad-hoc SPE that *iv)* can be implemented as a composable layer on top of any SPE, *v)* is consistent, and *vi)* is highly performant. Third, we further explore ad-hoc stream join query processing and discover the two main limitations: missed optimization potential and dynamicity. Our solution overcomes the limitations above by adopting *vii)* new join operator structure that enables not only data parallelism but also task parallelism and *viii)* dynamic query processing techniques. Our solution exhibits comparable performance with single-query workloads when compared with baselines. With ad-hoc stream queries, our solution always outperforms baselines.

### Future Research

This thesis lays the foundation for future research in several directions. In Chapter 3 we brought a new perspective to benchmarking SPEs. Also, we showed that existing SPE evaluation techniques might lead to unrealistic results. A future research goal is to extend our benchmarking framework along the lines of TPC database benchmarks. The main intuition is to define both a workload of queries that should be concurrently executed and then base the benchmark on a small number of operators that are part of that workload.

Chapter 5 focuses on optimization and dynamicity ad-hoc join stream queries. A future research goal is to extend AJoin to support not only stream join queries but also stream queries consisting of arbitrary stream operators. Also, there are many use-cases which unify stream and batch data computation, such as enriching stream tuples with lookups from historical data. Supporting ad-hoc queries for these use-cases is yet another future work.

## 7. Conclusion and Future Research

---

Our contributions in this thesis are based on a shared-nothing distributed architecture. However, with the advance of Internet of Things, the computation environment is becoming rather heterogenous. Fog computing, which is an architecture that uses edge devices to carry out a substantial amount of computation, storage, communication locally and routed over the internet backbone, is one example architecture that supports Internet of Things. A future research goal is to support ad-hoc queries on IoT databases, which enables diverse new opportunities for novel query optimization techniques.

# References

- [1] Sam Lucero et al. “IoT platforms: enabling the Internet of Things”. In: *White paper* (2016).
- [2] Ankit Toshniwal et al. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.
- [3] Matei Zaharia et al. “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters”. In: *Presented as part of the*. 2012.
- [4] Michael Armbrust et al. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 601–613.
- [5] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).
- [6] *Lime Electric Bike Sharing*. <https://www.li.me/electric-assist-bike/>. [Online; accessed 4-June-2019]. 2019.
- [7] *On-demand electric bikes and scooters*. <https://de.jump.com/fr/en/>. [Online; accessed 4-June-2019]. 2019.
- [8] Dibyendu Bhattacharya and Manidipa Mitra. *Analytics on big fast data using real time stream data processing architecture*. EMC Corporation, 2013.
- [9] Philipp Unterbrunner et al. “Predictable performance for unpredictable workloads”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 706–717.
- [10] Mitch Cherniack et al. “Scalable Distributed Stream Processing.” In: *CIDR*. Vol. 3. 2003, pp. 257–268.
- [11] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. “QPipe: a simultaneously pipelined relational query engine”. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 383–394.
- [12] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. “Exploiting the power of relational databases for efficient stream processing”. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM. 2009, pp. 323–334.
- [13] Erietta Liarou et al. “Enhanced stream processing in a DBMS kernel”. In: *Proceedings of the 16th International Conference on Extending Database Technology*. ACM. 2013, pp. 501–512.
- [14] Steffen Zeuch et al. “EfficiAnalyzing Efficient Stream Processing on Modern Hardware”. In: *Proceedings of the VLDB Endowment* (2019).
- [15] *International Workshop on Performance Analysis of Big data Systems (PABS)*. <https://web.rniapps.net/pabs/>. [Online; accessed 4-June-2019]. 2019.
- [16] *FlinkForward Conference*. <https://berlin-2019.flink-forward.org/>. [Online; accessed 4-June-2019]. 2019.

## REFERENCES

---

- [17] Tyler Akidau et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.
- [18] Tyler Akidau et al. “MillWheel: fault-tolerant stream processing at internet scale”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.
- [19] *Apache Storm*. <http://storm.apache.org> Accessed: 2017-01-28.
- [20] *Storm’s Trident abstraction*. <http://storm.apache.org/releases/2.0.0-SNAPSHOT/Trident-state.html> Accessed: 2017-01-28.
- [21] *Apache Storm issue: Disable Backpressure by default*. <https://issues.apache.org/jira/browse/STORM-1956> Accessed: 2017-01-17.
- [22] *Apache Spark*. <http://spark.apache.org> Accessed: 2017-01-28.
- [23] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.
- [24] Alexander Alexandrov et al. “The Stratosphere platform for big data analytics”. In: *The VLDB Journal* 23.6 (2014), pp. 939–964.
- [25] Donald Kossmann and Konrad Stocker. “Iterative dynamic programming: a new class of query optimization algorithms”. In: *ACM Transactions on Database Systems (TODS)* 25.1 (2000), pp. 43–82.
- [26] Sanket Chintapalli et al. “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2016, pp. 1789–1792.
- [27] M Andreoni Lopez, A Lobato, and OCMB Duarte. “A performance comparison of Open-Source stream processing platforms”. In: *IEEE Globecom*. 2016.
- [28] Anshu Shukla and Yogesh Simmhan. “Benchmarking distributed stream processing platforms for iot applications”. In: *arXiv preprint arXiv:1606.07621* (2016).
- [29] Shengsheng Huang et al. “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis”. In: *New Frontiers in Information and Software as Services*. Springer, 2011, pp. 209–228.
- [30] Tom White. *Hadoop: The definitive guide*. " O’Reilly Media, Inc.", 2012.
- [31] Min Li et al. “Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark”. In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015.
- [32] Ahmad Ghazal et al. “BigBench: towards an industry standard benchmark for big data analytics”. In: *Proceedings of the 2013 ACM SIGMOD*. ACM, 2013, pp. 1197–1208.
- [33] Lei Wang et al. “Bigdatabench: A big data benchmark suite from internet services”. In: *2014 IEEE HPCA*. IEEE, 2014, pp. 488–499.
- [34] Ovidiu-Cristian Marcu et al. “Spark versus flink: Understanding performance in big data analytics frameworks”. In: *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*. 2016.
- [35] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7.



- [36] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.
- [37] *Karamel, Orchestrating Chef Solo*. <http://storm.apache.org> Accessed: 2017-01-28.
- [38] Shelan Perera, Ashansa Perera, and Kamal Hakimzadeh. “Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds”. In: *arXiv preprint arXiv:1610.04493* (2016).
- [39] DataArtisans. *Extending the Yahoo! Streaming Benchmark*. <http://data-artisans.com/extending-the-yahoo-streaming-benchmark/> [Online; accessed 19-Nov-2016]. 2016.
- [40] Arvind Arasu et al. “Linear road: a stream data management benchmark”. In: *Proceedings of the VLDB- Volume 30*. VLDB Endowment. 2004, pp. 480–491.
- [41] Ruirui Lu et al. “Streambench: Towards benchmarking modern distributed stream computing frameworks”. In: *IEEE/ACM UCC*. IEEE. 2014.
- [42] Leonardo Neumeyer et al. “S4: Distributed stream computing platform”. In: *2010 IEEE International Conference on Data Mining Workshops*. IEEE. 2010, pp. 170–177.
- [43] Zhengping Qian et al. “Timestream: Reliable stream computation in the cloud”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 1–14.
- [44] Rajiv Ranjan. “Streaming big data processing in datacenter clouds”. In: *IEEE Cloud Computing* 1.1 (2014), pp. 78–83.
- [45] Matei Zaharia et al. “Discretized streams: Fault-tolerant streaming computation at scale”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 423–438.
- [46] *How NOT to Measure Latency*. <https://www.infoq.com/presentations/latency-response-time>. Accessed: 2017-07-11.
- [47] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. “Coordinated omission in nosql database benchmarking”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017), pp. 215–225.
- [48] *Spark Code Generation*. <https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html/>. Accessed: 2017-05-21.
- [49] *Apache Apex*. <https://apex.apache.org/>. [Online; accessed 19-July-2018]. 2018.
- [50] Guozhang Wang et al. “Building a replicated logging system with Apache Kafka”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1654–1655.
- [51] *Tencent Multinational conglomerate company*. <https://www.tencent.com/>. [Online; accessed 19-July-2018]. 2018.
- [52] Pete Tucker et al. *NEXMark—A Benchmark for Queries over Data Streams (DRAFT)*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.
- [53] Georgios Giannikis et al. “Shared workload optimization”. In: *Proceedings of the VLDB Endowment* 7.6 (2014), pp. 429–440.
- [54] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “SharedDB: killing one thousand queries with one stone”. In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 526–537.
- [55] Jin Li et al. “No pane, no gain: efficient evaluation of sliding-window aggregates over data streams”. In: *Acm Sigmod Record* 34.1 (2005), pp. 39–44.
- [56] Ryan Johnson et al. “To share or not to share?” In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 351–362.

## REFERENCES

---

- [57] Jin Li et al. “Semantics and evaluation techniques for window aggregates in data streams”. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 311–322.
- [58] Matthias J Sax et al. “Streams and Tables: Two Sides of the Same Coin”. In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. ACM. 2018, p. 1.
- [59] Paris Carbone et al. “State management in Apache Flink®: consistent stateful distributed stream processing”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.
- [60] *Apache Flink Latency*. <https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/metrics.html/>. [Online; accessed 4-Feb-2019]. 2019.
- [61] Jeyhun Karimov et al. “Benchmarking Distributed Stream Processing Engines”. In: *Data Engineering (ICDE), 2018 IEEE 34th International Conference on*. IEEE. 2018.
- [62] Subi Arumugam et al. “The DataPath system: a data-centric analytic processing engine for large data warehouses”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 519–530.
- [63] Buğra Gedik. “Generic windowing support for extensible stream processing systems”. In: *Software: Practice and Experience* 44.9 (2014), pp. 1105–1128.
- [64] Badrish Chandramouli et al. “Trill: A high-performance incremental query processor for diverse analytics”. In: *Proceedings of the VLDB Endowment* 8.4 (2014), pp. 401–412.
- [65] Michael Armbrust et al. “Spark sql: Relational data processing in spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.
- [66] Martin Hirzel et al. “A catalog of stream processing optimizations”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014), p. 46.
- [67] F. Palermo. “A Database Search Problem”. In: *Proc. of the 4th Symposium on Computer and Information Science*. ACM. 1974, pp. 67–101.
- [68] P Griffiths Selinger et al. “Access path selection in a relational database management system”. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.
- [69] Timos K Sellis. “Multiple-query optimization”. In: *ACM Transactions on Database Systems (TODS)* 13.1 (1988), pp. 23–52.
- [70] Alin Dobra et al. “Sketch-based multi-query processing over data streams”. In: *International Conference on Extending Database Technology*. Springer. 2004, pp. 551–568.
- [71] Sangeetha Seshadri, Vibhore Kumar, and Brian F Cooper. “Optimizing multiple queries in distributed data stream systems”. In: *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE. 2006, pp. 25–25.
- [72] Mingsheng Hong et al. “Rule-based multi-query optimization”. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM. 2009, pp. 120–131.
- [73] Samuel Madden et al. “Continuously adaptive continuous queries over streams”. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM. 2002, pp. 49–60.
- [74] Ron Avnur and Joseph M Hellerstein. “Eddies: Continuously adaptive query processing”. In: *ACM sigmod record*. Vol. 29. 2. ACM. 2000, pp. 261–272.
- [75] Zachary G Ives et al. “Adaptive query processing for internet applications”. In: (2000).

- 
- [76] Vijayshankar Raman, Amol Deshpande, and Joseph M Hellerstein. *Using state modules for adaptive query processing*. IEEE, 2003.
- [77] Shivaram Venkataraman et al. “Drizzle: Fast and adaptable stream processing at scale”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 374–389.
- [78] Luo Mai et al. “Chi: a scalable and programmable control plane for distributed stream processing systems”. In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1303–1316.
- [79] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. “Sharing data and work across concurrent analytical queries”. In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 637–648.
- [80] George Candea, Neoklis Polyzotis, and Radek Vingralek. “A scalable, predictable join operator for highly concurrent data warehouses”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 277–288.
- [81] Darko Makreshanski et al. “MQJoin: efficient shared execution of main-memory joins”. In: *Proceedings of the VLDB Endowment* 9.6 (2016), pp. 480–491.
- [82] Darko Makreshanski et al. “Many-query join: efficient shared execution of relational joins on modern hardware”. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 27.5 (2018), pp. 669–692.
- [83] Simon Loesing et al. “On the design and scalability of distributed shared-data databases”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 663–676.
- [84] Vijayshankar Raman et al. “Constant-time query processing”. In: (2008).
- [85] Xiaodan Wang et al. “CoScan: cooperative scan sharing in the cloud”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 11.
- [86] Marcin Zukowski et al. “Cooperative scans: dynamic bandwidth sharing in a DBMS”. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 723–734.
- [87] Christian A Lang et al. “Increasing buffer-locality for multiple relational table scans through grouping and throttling”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 1136–1145.
- [88] Darko Makreshanski et al. “BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 37–50.
- [89] Robin Rehrmann et al. “OLTPshare: the case for sharing in OLTP workloads”. In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1769–1780.
- [90] Song Wang et al. “State-slice: New paradigm of multi-query optimization of window-based stream queries”. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 619–630.
- [91] Moustafa A Hammad et al. “Scheduling for shared window joins over data streams”. In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 297–308.
- [92] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. “On-the-fly sharing for streamed aggregation”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 623–634.
- [93] Jonas Traub et al. “Efficient Window Aggregation with General Stream Slicing”. In: *22th International Conference on Extending Database Technology (EDBT)*. 2019.

## REFERENCES

---

- [94] Armando Fox et al. “Above the clouds: A berkeley view of cloud computing”. In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28.13* (2009), p. 2009.
- [95] Mark Turner, David Budgen, and Pearl Brereton. “Turning software into a service”. In: *Computer* 36.10 (2003), pp. 38–44.
- [96] Gabriela Jacques-Silva et al. “Providing streaming joins as a service at facebook”. In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1809–1821.
- [97] Samantha Bradshaw and Philip Howard. “Troops, trolls and troublemakers: A global inventory of organized social media manipulation”. In: (2017).
- [98] Whitney Phillips. “Meet the trolls”. In: *Index on Censorship* 40.2 (2011), pp. 68–76.
- [99] Yanlei Diao et al. “Path sharing and predicate evaluation for high-performance XML filtering”. In: *ACM Transactions on Database Systems (TODS)* 28.4 (2003), pp. 467–516.
- [100] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. “AStream: Ad-hoc Shared Stream Processing”. In: *SIGMOD 2019*. ACM. 2019.
- [101] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. “SharedDB: killing one thousand queries with one stone”. In: *Proceedings of the VLDB Endowment* 5.6 (2012), pp. 526–537.
- [102] Georgios Giannikis et al. “Workload optimization using shareddb”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 1045–1048.
- [103] Jana Giceva et al. “Deployment of query plans on multicores”. In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 233–244.
- [104] George Candea, Neoklis Polyzotis, and Radek Vingralek. “Predictable performance and high query concurrency for data analytics”. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 20.2 (2011), pp. 227–248.
- [105] Lucas Braun et al. “Analytics in motion: High performance event-processing and real-time analytics in the same database”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 251–264.
- [106] Tansel Dokeroglu et al. “Improving the performance of Hadoop Hive by sharing scan and computation tasks”. In: *Journal of Cloud Computing* 3.1 (2014), p. 12.
- [107] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. “Maximizing the output rate of multi-way join queries over streaming information sources”. In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 285–296.
- [108] RoeE Ebenstein, Niranjan Kamat, and Arnab Nandi. “FluxQuery: An execution framework for highly interactive query workloads”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1333–1345.
- [109] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. “Adaptive query processing”. In: *Foundations and Trends® in Databases* 1.1 (2007), pp. 1–140.
- [110] Volker Markl et al. “Robust query processing through progressive optimization”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 659–670.
- [111] Quanzhong Li et al. “Adaptively reordering joins during query execution”. In: *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE. 2007, pp. 26–35.
- [112] Bugra Gedik et al. “Elastic scaling for data stream processing”. In: *IEEE Transactions on Parallel & Distributed Systems* 1 (2014), pp. 1–1.

- [113] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. “Elastic stateful stream processing in storm”. In: *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, 2016, pp. 583–590.
- [114] Thomas Heinze et al. “FUGU: Elastic Data Stream Processing with Latency Constraints.” In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 73–81.
- [115] Thomas Heinze et al. “Online parameter optimization for elastic data stream processing”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 276–287.
- [116] Immanuel Trummer and Christoph Koch. “Solving the Join Ordering Problem via Mixed Integer Linear Programming”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1025–1040.
- [117] Guido Moerkotte and Thomas Neumann. “Dynamic programming strikes back”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 539–552.
- [118] P Griffiths Selinger et al. “Access path selection in a relational database management system”. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 1979, pp. 23–34.
- [119] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. “Optimization of Nonrecursive Queries.” In: *VLDB*. Vol. 86. 1986, pp. 128–137.
- [120] Toshihide Ibaraki and Tiko Kameda. “On the optimal nesting order for computing n-relational joins”. In: *ACM Transactions on Database Systems (TODS)* 9.3 (1984), pp. 482–502.
- [121] Thomas Neumann and Bernhard Radke. “Adaptive Optimization of Very Large Join Queries”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 677–692.
- [122] Daniel Aloise et al. “NP-hardness of Euclidean sum-of-squares clustering”. In: *Machine learning* 75.2 (2009), pp. 245–248.
- [123] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. “Application of hash to data base machine and its architecture”. In: *New Generation Computing* 1.1 (1983), pp. 63–74.
- [124] Brian Babcock et al. “Models and issues in data stream systems”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.
- [125] *Apache Kafka*. <https://kafka.apache.org/>. [Online; accessed 17-August-2019]. 2019.
- [126] Aaron Beitch et al. “Rain: A workload generation toolkit for cloud computing applications”. In: *University of California, Tech. Rep. UCB/EECS-2010-14* (2010).
- [127] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. “FC2Q: exploiting fuzzy control in server consolidation for cloud applications with SLA constraints”. In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 4491–4514.
- [128] Andrew Turner et al. “C-mart: Benchmarking the cloud”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2012), pp. 1256–1266.
- [129] Basem Suleiman et al. “On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure”. In: *Journal of Internet Services and Applications* 3.2 (2012), pp. 173–193.
- [130] Lei Lu et al. “Application-driven dynamic vertical scaling of virtual machines in resource pools”. In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.
- [131] Nikolas Roman Herbst, Samuel Kounev, et al. “Modeling variations in load intensity over time”. In: *Proceedings of the third international workshop on Large scale testing*. ACM, 2014, pp. 1–4.

## REFERENCES

---

- [132] Thomas Heinze et al. “Latency-aware elastic scaling for distributed data stream processing systems”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM. 2014, pp. 13–22.