Stream Processing of Financial Tick Data with In-Order Guarantees

Stefanos Kalogerakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Professor *Kostas Magoutis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS). The research in this thesis was also supported by the Hellenic Foundation for Research and Innovation (HFRI/ΕΛΙΔΕΚ) through the STREAM-STORE faculty grant (Grant ID HFRI-FM17-1998).

University of Crete
Computer Science Department

Stream Processing of Financial Tick Data with In-Order Guarantees

Thesis submitted by
Stefanos Kalogerakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Stefanos Kalogerakis

Committee approvals: _____
Kostas Magoutis
Associate Professor, Thesis Supervisor

_____
Angelos Bilas
Professor, Committee Member

_____
Dimitris Plexousakis
Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, June 2023

Stream Processing of Financial Tick Data with In-Order Guarantees

Abstract

Data related to all types of societal activity are nowadays being produced and available in high volumes and velocity, in the form of data streams. This thesis focuses on financial tick data generated by stock exchanges and the necessity for stream processing analytics to assist traders in identifying trading opportunities. We design and implement the Tick Analysis Platform (TAP), a streaming analytics application that performs event aggregation and complex event processing to compute trend indicators and detect patterns, enabling the identification of buy/sell opportunities for traders. Based on the need to process streaming data as rapidly as possible, we investigate techniques for scalable stream processing, with an additional guarantee, namely the in-order processing of such data based on sequencing information available in batches of incoming data. The solutions designed and implemented in this thesis, S-TAP (Single-source TAP) and P-TAP (Parallel-source TAP), progressively enhance the scalability of TAP to achieve high performance on a cluster of multi-core servers while ensuring the accuracy of results via the in-order guarantees. An additional challenge investigated by this thesis is efficient fault-tolerance mechanisms to achieve low down-times during recovery of data analysis jobs. This is achieved by aligning the deployment of recovery tasks with the location of externally-stored checkpoint replicas, taking advantage of data locality where possible. The solutions implemented and demonstrated in this thesis advance the state of the art in scalable streaming analytics of financial tick data that are also rapidly recoverable in the face of failures.

Επεξεργασία Ροών Οικονομικών Δεδομένων με Εγγυήσεις Διάταξης

Περίληψη

Στις μέρες μας, δεδομένα που σχετίζονται με όλες τις μορφές κοινωνικής δραστηριότητας παράγονται και γίνονται διαθέσιμα σε μεγάλο όγκο και ταχύτητα με τη μορφή ροών (χρονοσειρών) δεδομένων. Η συγκεκριμένη μεταπτυχιακή διατριβή επικεντρώνεται σε ροές οικονομικών δεδομένων (τιμών μετοχών) που παράγονται από χρηματιστηριακές αγορές και στην ανάγκη για ανάπτυξη τεχνικών επεξεργασίας τους για την ανίχνευση ευκαιριών συναλλαγών. Στην διατριβή αυτή σχεδιάζουμε και υλοποιούμε την Πλατφόρμα Ανάλυσης Δεδομένων Μετοχών (TAP), μια εφαρμογή ανάλυσης ροών που χρησιμοποιεί τεχνικές συγχώνευσης και επεξεργασίας γεγονότων για τον υπολογισμό δεικτών τάσης και την ανίχνευση προτύπων, δίνοντας τη δυνατότητα αναγνώρισης ευκαιριών αγοράς/πώλησης. Βασιζόμενοι στην ανάγκη για ταχεία επεξεργασία των ροών δεδομένων, ερευνούμε τεχνικές για την κλιμακωσιμότητα τους, με μια επιπλέον εγγύηση, την επεξεργασία των δεδομένων αυτών με βάση τις πληροφορίες διάταξης που είναι διαθέσιμες στα εισερχόμενα δεδομένα. Οι λύσεις που σχεδιάστηκαν και υλοποιήθηκαν σε αυτήν τη διατριβή, S-TAP (TAP απο μια πηγή δεδομένων) και P-TAP (TAP πολλαπλών πηγών δεδομένων), βελτιώνουν την κλιμακωσιμότητα του TAP για την επίτευξη υψηλής επίδοσης σε κατανεμημένο περιβάλλον πολυπύρηνων διακομιστών, διασφαλίζοντας την ακρίβεια των αποτελεσμάτων μέσω των εγγυήσεων επεξεργασίας σε διάταξη. Μια επιπλέον πρόκληση που μελετά αυτή η διατριβή είναι οι αποδοτικοί μηχανισμοί ανοχής σφαλμάτων για να επιτευχθεί ταχεία ανάκαμψη εργασιών μετά από αστοχίες. Αυτό επιτυγχάνεται ευθυγραμμίζοντας την εκτέλεση των εργασιών ανάκαμψης με την δικτυακή θέση των αντιγράφων ασφαλείας, εκμεταλλευόμενοι την τοπικότητα των δεδομένων όπου αυτό είναι εφικτό. Οι λύσεις που υλοποιήθηκαν και αξιολογήθηκαν σε αυτήν τη διατριβή προάγουν την τεχνολογία αιχμής στην κλιμακώσιμη και ταχέως ανακάμψιμη ανάλυση ροών χρηματιστηριακών δεδομένων.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Kostas Magoutis, for his continuous guidance and encouragement throughout my entire postgraduate studies. His expertise and insightful feedback have been instrumental in shaping the direction and quality of this work.

Special thanks to Antonis Papaioannou for his invaluable assistance and for presenting our work during the participation in the DEBS 2022 Grand Challenge. His contributions have significantly enriched the outcomes of this research.

I would also like to thank Prof. Dimitris Plexousakis and Prof. Angelos Bilas for agreeing to serve as members of my thesis committee and for dedicating their time to evaluating my thesis.

Furthermore, I would like to acknowledge the University of Crete and FORTH Institute for providing the necessary resources and facilities that greatly facilitated the completion of this thesis. Their support has been instrumental in enabling me to conduct comprehensive research and achieve my academic goals.

On a personal note, I would like to express my deepest gratitude to my family and friends. Their encouragement, love, and belief in my abilities have been a constant source of inspiration throughout this journey. I am immensely thankful for their patience and support, which have motivated me to strive for excellence throughout this journey.

to my family

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's era of rapid technological advancements, the exponential growth of data has transformed the way we approach information processing and analysis. The term *Big Data* has emerged to describe the vast and complex datasets that are generated from various sources, including social media, Internet of Things (IoT) devices, scientific experiments, and financial transactions. Big Data is characterized by its volume, velocity, variety, and veracity, posing significant challenges for traditional data processing techniques. However, this paradigm shift also brings new possibilities for organizations across various industries, including the financial sector.

The financial industry, in particular, has experienced an astounding increase in the overall volume of events published by different exchanges and handled by technical solution providers like Infront Financial Technology (IFT). For instance, between 2019 and 2021, the daily average of event notifications processed by IFT escalated from 18 billion to an astonishing 24 billion [22]. This exponential growth in data underscores the significance of efficient and advanced data processing techniques to extract valuable insights and drive informed decision-making in the financial domain.

The unprecedented amount of data generated by financial markets, has created a demand for efficient stream processing solutions. Stream processing involves the continuous analysis of data as it is generated, enabling organizations to gain insights and make informed decisions in real time. Unlike traditional batch processing, which handles data in large chunks, streaming analytics operates on continuous data streams, allowing for faster response to changing conditions. In the financial domain, where timely decisions are crucial for investment strategies and risk management, real-time processing capability is of utmost importance.

Frameworks like Apache Flink have emerged as powerful tools for stream processing and analytics on the fly. Apache Flink is an open-source framework that provides high-throughput, low-latency processing of streaming data. It offers a unified programming model that combines batch processing and stream processing,

enabling seamless analysis of both historical and real-time data. With its fault-tolerant and scalable architecture, Apache Flink has gained popularity in various industries, including finance, for its ability to handle large-scale data streams efficiently.

However, stream processing in the financial domain faces its own set of challenges. Financial tick data, which consists of fine-grained time-stamped updates for financial instruments, plays a crucial role in understanding market dynamics, detecting trading patterns, and optimizing investment strategies . Analyzing tick data in real-time presents unique challenges due to the sheer volume and speed of incoming data streams, requiring the development of efficient and scalable processing techniques. In this context, ensuring in-order guarantees through stream processing is essential in financial applications to accurately capture market trends and maintain the integrity of derived analytics.

As stream processing systems (SPSs) operate in dynamic and often unpredictable environments, they are susceptible to various types of failures that can disrupt the processing pipeline and lead to data loss or inconsistencies. Failures can manifest in various forms, including hardware malfunctions, network disruptions, or even human error. Therefore, ensuring fault tolerance becomes a crucial aspect of SPSs. Fault tolerance encompasses the ability of an SPS to gracefully handle failures and recover from them, minimizing the adverse effects on data processing and maintaining the system's reliability and consistency. Techniques such as checkpointing, replication, and recovery algorithms enable the system to recover the state and progress from a known consistent point in the event of a failure.

In this thesis, our efforts focus on addressing two distinct challenges. The primary focus is on developing an efficient Tick Analysis Platform that utilizes event aggregation and complex event processing to compute trend indicators and detect patterns in real-time tick data (Fig. 1.1), while achieving in-order processing of tick data. The goal is to highlight breakout patterns and consequently identify buy/sell opportunities for real-life traders (Fig. 1.2). We address this challenge through two applications, S-TAP (Chapter 3) and P-TAP (Chapter 4). S-TAP, the initial solution proposed, enables sequential ingest on the data source to achieve the desired semantics, while P-TAP represents the evolution of its predecessor, allowing for parallel ingestion on the data source.

The second challenge addressed in this thesis is rapid recovery of SPSs to ensure uninterrupted analysis in the face of failures. Our approach revolves around achieving fast recovery by aligning recovery tasks with externally stored state. We build upon the state-of-the-art incremental distributed checkpointing capabilities of the Flink SPS and extend them in this direction (Chapter 5).

1.1 Thesis Contributions

The contributions of this thesis are as follows

- A single source (S-TAP) and a fully parallelized solution (P-TAP) to the

Figure 1.1: Breakout Patterns Example: Price for RDS A plotted against EMA(38) (green) and the EMA(100) (orange) showing at least three crossings that trigger buy/sell advice [1]

problem of discovering breakout patterns in financial tick data via parallel stream processing with in-order guarantees

- An evaluation of P-TAP on a 32-core 4-server cluster demonstrating its scalability over the sequential-ingest version (S-TAP) on the same cluster
- An investigation of the benefits possible with finer tuning of parallelism vs. default settings in Flink
- An investigation of techniques to achieve rapid recovery of SPSs by aligning recovery tasks with externally stored state

1.2 Thesis Organization

The thesis outline consists of the following chapters:

Chapter 2 - Background: This chapter offers an introduction to the fundamental theoretical knowledge and key concepts that underpin this thesis. It explores the concept of stream processing, emphasizing its significance in the financial domain. Additionally, it presents an analysis of the financial dataset employed in the subsequent chapters. Furthermore, a detailed examination of the frameworks

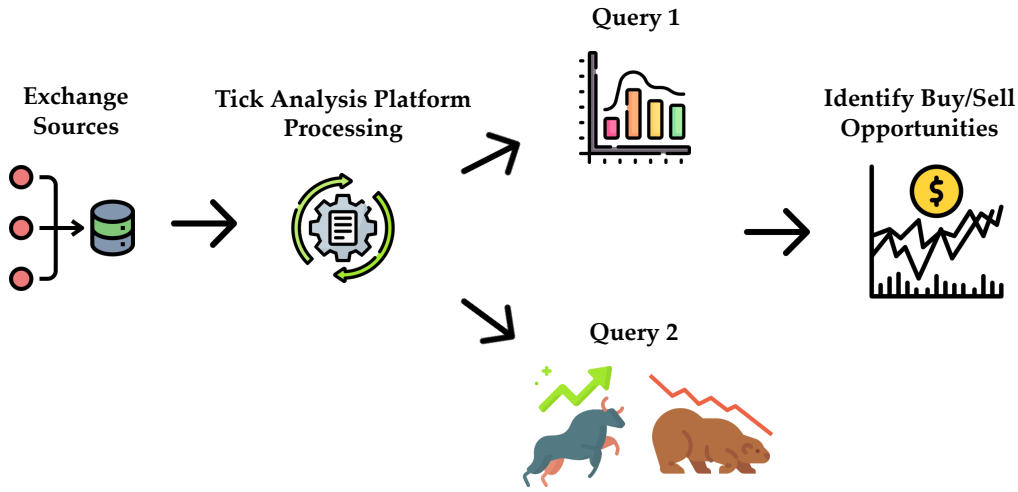

Figure 1.2: High-Level Overview of Tick Analysis Platform [10]

utilized in this thesis, namely Apache Flink, Apache Kafka, and the Hadoop Distributed File System (HDFS), is conducted.

Chapter 3 - Single-source Tick Analysis Platform (S-TAP): Chapter 3 delves into the analysis of S-TAP (Single-source Tick Analysis Platform), a solution developed for the 2022 DEBS Grand Challenge (GC). The primary objective of the challenge was to effectively calculate specific trend indicators and identify patterns resembling those utilized by real-life traders when making decisions regarding buying or selling on financial markets. During our analysis, we identified the handling of late (out-of-order) events and the mapping between batches of events and the corresponding window-closings as significant correctness challenges. To address these challenges, S-TAP offers a solution that maintains a single instance of the source operator and batch-unpack logic, eventually limiting the achievable parallelism.

Chapter 4 - Parallel-source Tick Analysis Platform (P-TAP): This chapter introduces P-TAP (Parallel-source Tick Analysis Platform), a new solution that builds upon our initial S-TAP solution from Chapter 3. P-TAP addresses the challenge of parallel data ingestion while maintaining the same in-order batch-processing guarantees as S-TAP.

Chapter 5 - Rapid Recovery of SPSs: In this chapter, our primary focus is to achieve fast recovery of Stream-processing Systems (SPSs) by effectively aligning recovery tasks with externally stored checkpoint state. Specifically, we devote our efforts to two key aspects: controlling task recovery decisions and task placement within the Flink SPS, as well as extracting information from and influencing HDFS block placement on data nodes.

Chapter 6 - Related Work: This chapter provides a comprehensive review of

the existing literature and research efforts related to in-order stream processing, processing of financial tick data and recovery of SPSs. We discuss the strengths and limitations of the current approaches while emphasizing the contributions made by this thesis.

Chapter 7 - Conclusions & Future Work: Finally, Chapter 7 concludes the thesis by summarizing the key contributions of the research and discussing potential future directions.

Chapter 2

Background

The background section aims to provide a comprehensive understanding of the key components and concepts that form the foundation of this thesis. Section 2.1 starts by exploring the key concept of stream processing, while Sections 2.2, 2.3 focus on the frameworks utilized for our financial analytics applications. Specifically, Section 2.2 delves into Apache Flink a powerful open-source framework utilized for data processing while Section 2.3 highlights Apache Kafka, a distributed messaging system used for ingestion purposes. Additionally, Section 2.4 depicts the domain of financial analytics and emphasizes the identification of breakout patterns. The subsequent section, Section 2.5, provides a detailed analysis of the financial dataset utilized in this study, including its characteristics, sources, and relevance. Finally, Section 2.6 examines the Hadoop Distributed File System (HDFS), a highly scalable and fault-tolerant storage system, which plays a crucial role in the chapter covering the recovery of SPSs (Chapter 5).

2.1 Stream Processing

Streaming describes continuous, never-ending sequences of data that are made available over time. That way, a constant feed of data is provided in applications, allowing for on-the-fly manipulation without the need for downloading. These data streams can originate from diverse sources, each with their own rates and volumes, and can be merged into a unified stream. Examples of streaming data applications include real-time stock exchange updates, sensor monitoring, and website activity tracking. Streaming is particularly crucial in the realm of big data as it facilitates real-time analytics, data ingestion, and data integration. [27]

2.1.1 Bounded and Unbounded Streams

Streaming can be examined from two different paradigms: **bounded** and **unbounded**. Each paradigm represents a different approach to processing and analyzing data in a streaming context (Fig. 2.1).

An **unbounded stream** refers to a stream of data that does not have a predetermined ending point. Due to the infinite nature of unbounded streams, it is not possible to wait for all input data to arrive before processing it. Instead, events in an unbounded stream need to be processed on-the-fly as they are acquired. The distinguishing factor among these events is typically their received time.

In contrast to unbounded streams, **bounded streams** have a well-defined start and end point. Bounded data streams can be processed by fetching all the data before performing any calculations or analysis. Unlike unbounded streams, ordered fetching of events is not typically required for processing bounded streams. Bounded data sets can be sorted and processed in batches, hence the term *batch processing* is often used interchangeably with bounded stream processing.

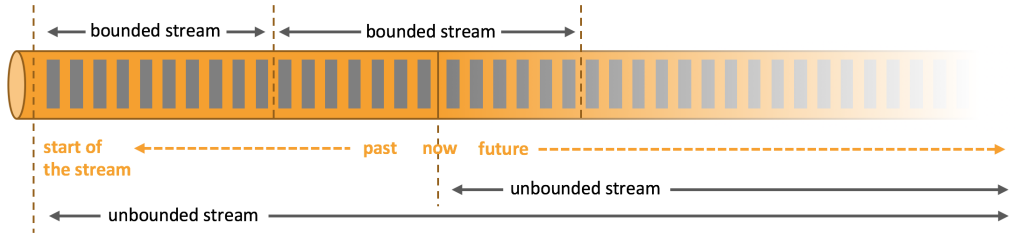

Figure 2.1: Bounded and Unbounded Streaming [16]

2.2 Apache Flink

2.2.1 Overview

Apache Flink is an open-source distributed framework for processing large-scale streaming and batch data. It offers a unified platform for handling both real-time and batch data, making it a valuable tool for use cases in data analytics, machine learning, and data-driven applications. The framework offers a set of programming interfaces and libraries for processing data in real-time, with capabilities for high throughput, low latency, fault tolerance, and scalability. Flink's main goal is to express different classes of processing applications, such as stream processing, batch processing, and iterative algorithms, as pipelined fault-tolerant dataflows. [5]

Flink provides several key features that set it apart from other data processing systems. One of its primary features is its ability to support both real-time data processing through stream processing, as well as offline batch processing for large datasets. Flink's fault tolerance capabilities ensure that it can recover from system failures without losing any data, making it a reliable tool for critical data processing tasks. Additionally, Flink is highly flexible and extensible, with support for a variety of data sources and sinks, including **Hadoop Distributed File System (HDFS)**, **Apache Kafka**, and **Amazon S3**. Flink's low latency and high throughput processing capabilities make it an ideal choice for real-time data

processing use cases, while its support for machine learning algorithms and complex event processing (CEP) provides a versatile platform for building a variety of data processing applications. With its ability to process data quickly and reliably, Apache Flink is a powerful framework that can handle data processing tasks of all types and sizes.

2.2.2 Ecosystem and Integration

Apache Flink provides a rich ecosystem comprising various of components that enhance its functionality. This ecosystem follows a layered structure, with each component building upon the previous layer to raise the abstraction layer of the program representation they accept. Figure 2.2 illustrates the hierarchical arrangement of these layers.

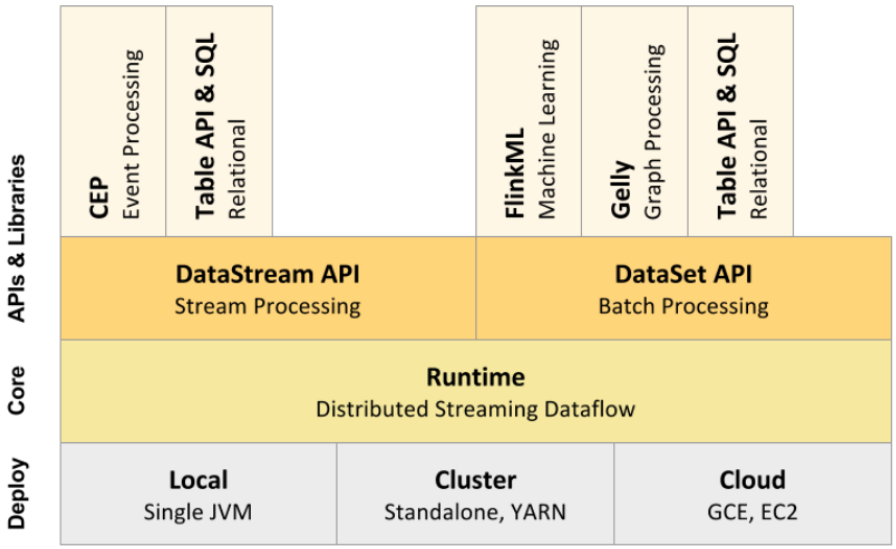

Figure 2.2: Flink Software Component Stack [16]

Runtime Layer: It is the core of Flink, responsible for the distributed execution of dataflows. Flink represents dataflows as Directed Acyclic Graphs (DAGs), known as JobGraphs within the Flink framework. These JobGraphs consist of tasks that can generate and consume data streams.

APIs Layer: Dataset and DataStream are the two core APIs of Flink for handling batch and stream processing, respectively. Both APIs generate JobGraphs, which are executed within the runtime layer. Notably, the compilation process differs between the two APIs; Dataset API utilizes an optimizer to establish the optimal execution plan, while DataStream API relies on a stream builder. This decision to distinguish the APIs was based on achieving optimal performance and ease of use.

Libraries Layer: On top of Flink's basic functionality APIs, several domain-specific libraries and APIs are connected with Flink facilitating the development of Dataset and DataStream programs. Noteworthy libraries include FlinkML, which addresses machine learning tasks, Gelly, designed for graph processing, and Table, enabling SQL-like computations on Flink.

Deployment Layer: Flink offers diverse deployment options for planning, monitoring, and managing resources in the execution of Flink jobs. These options include Local deployment for local testing, Remote deployment for executing jobs on remote clusters, and Yarn deployment for leveraging resource management capabilities, among others.

2.2.3 Architecture of Apache Flink

Apache Flink's architecture (Fig. 2.3) plays a critical role in providing fault-tolerant, scalable, and efficient stream processing capabilities. At the core of this architecture is the master-worker pattern, where the JobManager takes charge of job execution and resource coordination, while multiple TaskManagers perform the actual data processing. In this section, we will investigate the architecture of Apache Flink, exploring the roles and functionalities of its fundamental components. These key components include:

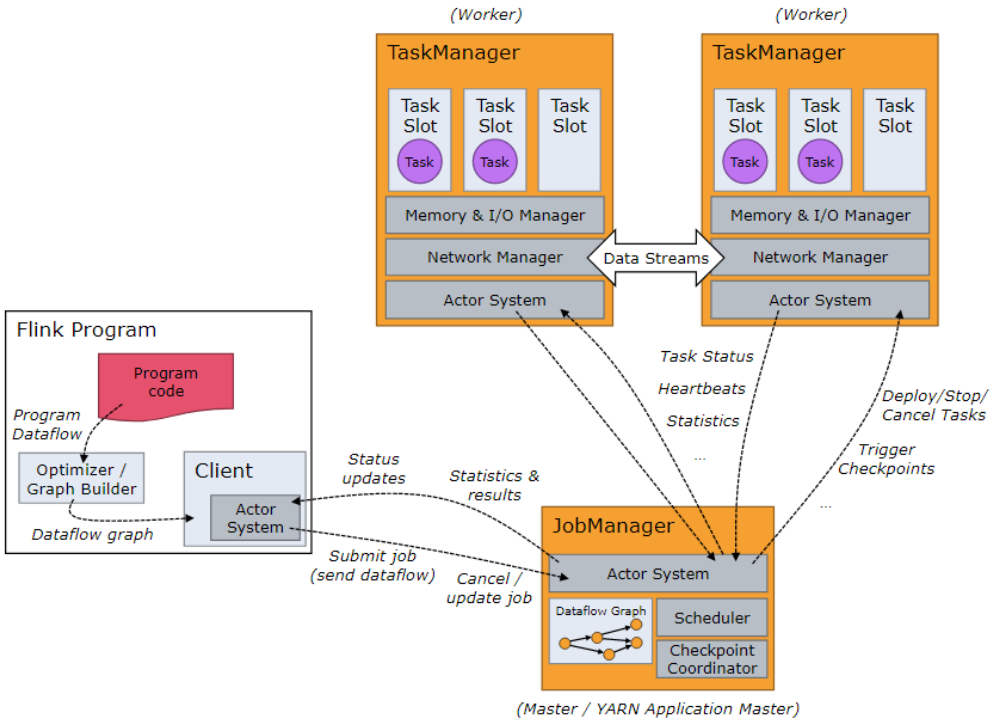

Figure 2.3: Apache Flink Architecture [16]

Client: While it is not part of the runtime environment, the client is Flink's first interaction with the user, serving as the interface between users or applications and the Flink cluster. It enables job submission, configuration management, and dependency handling, allowing users to specify job parameters and manage job dependencies. The client interacts with the Flink cluster to obtain cluster information and provides a user interface or command-line interface (CLI) for job monitoring and management. It further contributes to performance optimization by performing client-side optimizations, such as task chaining and optimizing the job graph. Overall, the client component simplifies the interaction with Apache Flink, streamlining job submission, management, and monitoring processes.

JobManager: Following the master-worker architecture, the JobManager serves as the master node, coordinating the distributed execution of Flink Applications. It acts as the central component responsible for job submission, execution plan creation, and task scheduling. By accepting job submissions, the JobManager receives the job graph, which represents the dataflow of the application, and divides it into a series of tasks. It efficiently schedules these tasks onto available TaskManagers in the cluster, considering resource availability, task dependencies, and desired parallelism to achieve optimal distribution. Additionally, the JobManager ensures fault tolerance by maintaining checkpoints(introduced in Section 2.2.5), periodically capturing snapshots of the application's state. In case of failures, it can recover the application's state and resume processing from the last successful checkpoint. In high-availability setups, multiple JobManagers can exist simultaneously, with each preserving additional metadata information in a persistent storage to facilitate recovery and seamless resumption of execution by alternative JobManagers.

Task Manager: TaskManagers, the worker nodes, have essential responsibilities in executing tasks and processing dataflows. A Flink application requires at least one TaskManager for successful execution. TaskManagers operate by running parallel instances of tasks, known as task slots, assigned by the JobManager. These task slots represent the unit of resource allocation in Flink and can be configured to execute tasks concurrently. TaskManagers maintain communication with the JobManager to receive task assignments, report progress, and exchange data with other TaskManagers. In addition, TaskManagers utilize buffer pools to efficiently handle data streams and enable seamless data exchange between operators over the network. By performing these crucial functions, TaskManagers contribute to the distributed and parallel execution of tasks within Flink, facilitating efficient processing of data in various streaming applications.

2.2.3.1 Task Slots and Resource Management

Resource management plays a crucial role in the efficient execution of distributed computing systems. In Apache Flink, task slots are employed as fundamental units for resource allocation and parallelism control. In this subsection, we explore the

concept of task slots in Flink and their role in managing computing resources.

A task slot represents a fixed-size subset of computing resources owned by a TaskManager, which is a set of independent resources. When configuring a TaskManager, the number of task slots can be set in the cluster configuration file (at least one). For example, if a TaskManager has three slots, it will divide the managed memory equally into three parts, with each slot having an exclusive copy.

The primary purpose of task slots is to provide isolation between subtasks. Each worker, represented by a TaskManager in Flink, is a JVM process that can execute one or more subtasks in separate threads. By controlling the number of task slots, the isolation level between subtasks can be adjusted. When multiple slots are available, subtasks can share the same JVM, leading to benefits such as shared TCP connections, heartbeat messages, and reduced per-task overhead.

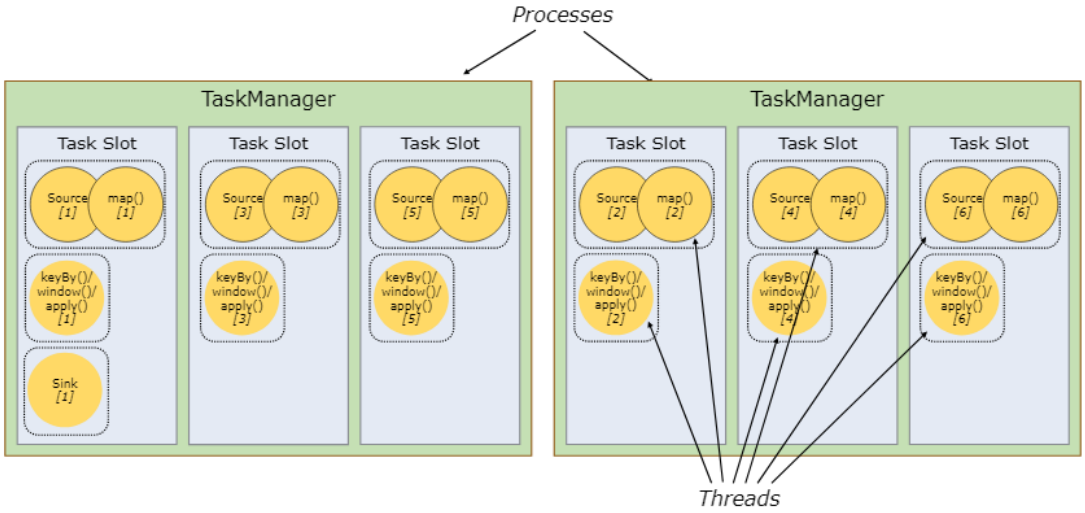

Figure 2.4: Task Slots default behavior: slots are shared between subtasks of different tasks under the same job [16]

By default, Flink allows subtasks from the same job to share slots, even if they are subtasks of different tasks (Fig. 2.4). This means that one slot may hold an entire pipeline of the job. This slot sharing approach offers two main benefits; first, the number of task slots required for a Flink cluster is solely determined by the highest parallelism used in the job, eliminating the need for calculating the total number of tasks in the program with varying parallelism. Second, it facilitates better resource utilization by ensuring that resource-intensive subtasks are fairly distributed among the TaskManagers, preventing resource blocking by non-intensive subtasks.

In cases where users want to control the level of slot sharing, Flink provides the ability to specify *SlotSharingGroup* for operators. This allows manual configuration of whether certain operators should occupy a slot exclusively or share a slot with specific operators. Subtasks belonging to the same slot sharing group

enable slot sharing, while tasks between different groups are completely isolated and must be assigned to different slots. The total number of slots required in this scenario is the sum of the maximum parallelism of each slot sharing group.

2.2.4 State Management

Before going into further details about Flink's features, it is essential to introduce the concept of **state**. In Flink, this term refers to operations that retain and preserve their information across time [17]. It enables Flink to capture snapshots of operators, allowing them to have knowledge of all the events that occurred in the application until a specific point in time. State can encompass various types of mutable information, such as aggregations, counts, user-defined variables, and machine learning models. The ability to maintain state is crucial for rescaling Flink applications by redistributing state across parallel instances and ensuring fault-tolerance.

2.2.4.1 Types of State in Flink

Apache Flink provides different types of state to cover a wide range of stream processing use cases. These types differ in how the state is partitioned and distributed across the processing nodes. The two primary types of state in Flink are **Operator State** and **Keyed State**.

Operator state in Flink is associated with individual operators or functions in the stream processing application. It represents the local state maintained by each operator during the stream processing. This state is partitioned and managed independently by each operator, offering fine-grained control over the state management process.

Keyed state, on the other hand, is state associated with keys present in the stream data. It allows Flink to maintain state for different keys in a scalable and fault-tolerant manner. Keyed state is partitioned based on the keys, ensuring that all events with the same key are processed by the same operator instance.

Both keyed state and operator state have two forms: *Raw* and *Managed*. Raw state is represented as raw bytes and does not have knowledge of the state's data structure. Operators are responsible for managing the state in their own data structures. On the other hand, managed state is represented in data structures controlled by the Flink runtime. It is recommended to use managed state as Flink can automatically redistribute the state when the parallelism is changed, and it offers better memory management. [35]

The managed keyed state in Flink includes the following types: **ValueState**, **ListState**, **ReducingState**, **AggregatingState** and **MapState**

These managed keyed states provide flexible options for storing and managing state within Flink applications.

2.2.4.2 State Backends in Flink

The state backend in Apache Flink determines how the internal state is represented and persisted during checkpoints. Currently, Flink offers two built-in state backends: the **HashMapStateBackend** and the **EmbeddedRocksDBStateBackend**

The **HashMapStateBackend** is the default state backend in Flink. It maintains all the "working" state in memory, resulting in faster operations compared to the RocksDB state backend. Since the accesses occur in memory, there is no need for data serialization/deserialization. However, this state backend is constrained by the available memory in the application. Therefore, its capacity is limited by the memory size within the cluster.

On the other hand, the **EmbeddedRocksDBStateBackend** uses a co-located key-value store to keep the "working" state of Flink application while it spills data into disks. However, this backend requires serialization/deserialization of data, resulting in approximately 10 times slower operations compared to the HashMapStateBackend. The memory size of RocksDB is practically unlimited because it is only bounded by the disk size, which can be adjusted as needed. [45]

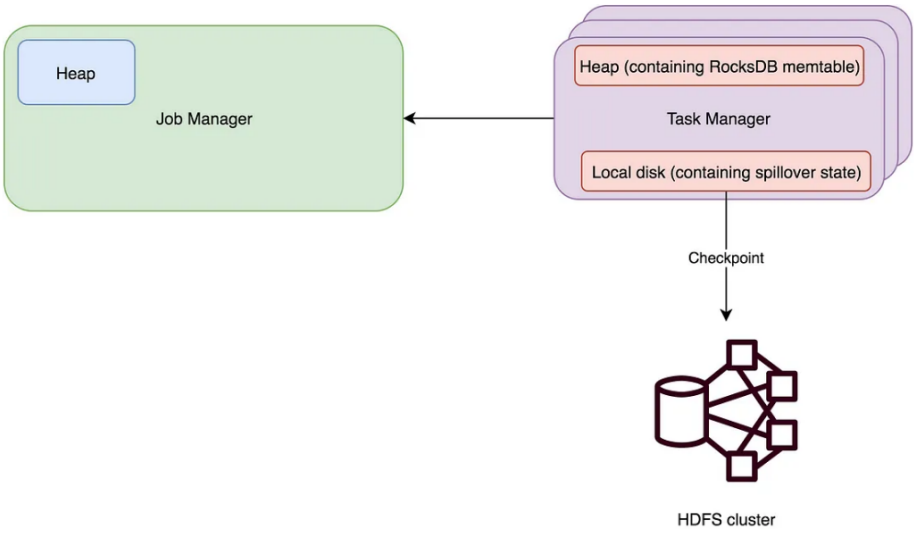

Figure 2.5: Apache Flink: RocksDB Checkpoint [44]

One significant advantage of the EmbeddedRocksDBStateBackend is its support for incremental checkpointing (Fig. 2.5)[12]. Unlike traditional full checkpoints, incremental checkpoints only record the changes made since the last completed checkpoint, resulting in significantly reduced checkpointing times. This makes it a favorable choice where efficiency and scalability are crucial.

The choice between the HashMapStateBackend and the EmbeddedRocksDBStateBackend depends on the specific requirements of the application and the

trade-off between performance and scalability. The HashMapStateBackend is very fast, as it operates entirely in memory. However, its size is limited by the available memory within the cluster. On the other hand, RocksDB scales based on the available disk space and is the only state backend that supports incremental checkpoints. Although the performance of RocksDB is slower due to disk accesses and serialization/deserialization operations, it offers greater scalability. [12]

2.2.5 Checkpoints and Savepoints

Checkpointing is a fundamental mechanism in Apache Flink that ensures fault tolerance and recovery capabilities in stateful stream processing applications. It involves capturing the state of an application at regular intervals and persisting it to durable storage, such as HDFS. This process requires coordination between the JobManager and TaskManagers, with the JobManager triggering the checkpoint and the TaskManagers taking snapshots of their respective tasks' state, as shown in Fig. 2.6.

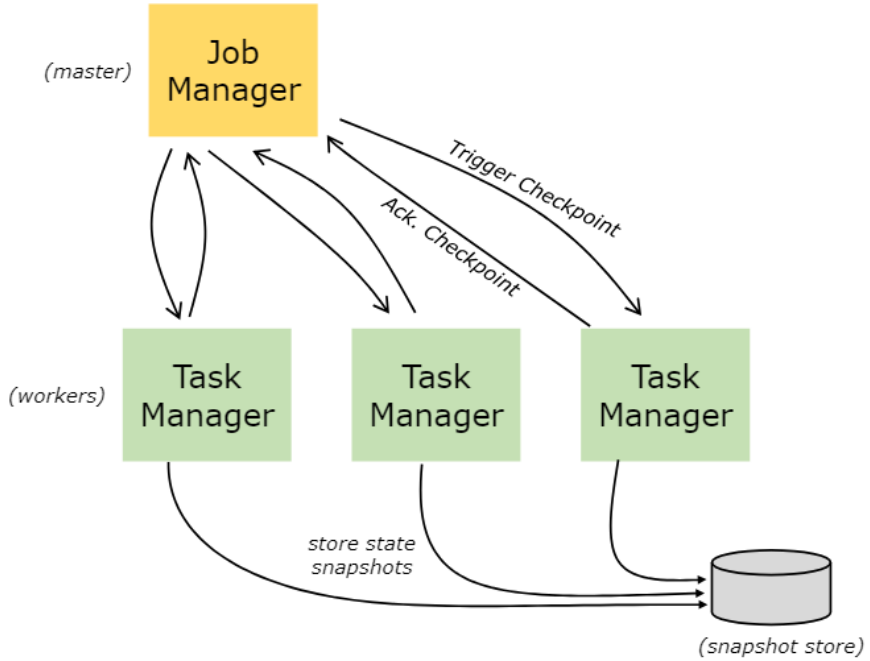

Figure 2.6: Apache Flink: Checkpoint Snapshot [17]

Flink combines stream replay and checkpointing to provide fault tolerance and maintain consistency in streaming dataflows. Checkpoints draw global, asynchronous snapshots [4] of input streams and corresponding operator state, following the Chandy-Lamport algorithm for distributed snapshots (Fig. 2.7). The snapshot includes not only the dataflow, but the state attached to it. Maintaining consistency in streaming dataflows is also rather challenging, but with its checkpointing

mechanism that enables restoring state of the operators and replaying the records from the latest checkpoint, Flink ensures exactly-once semantics.

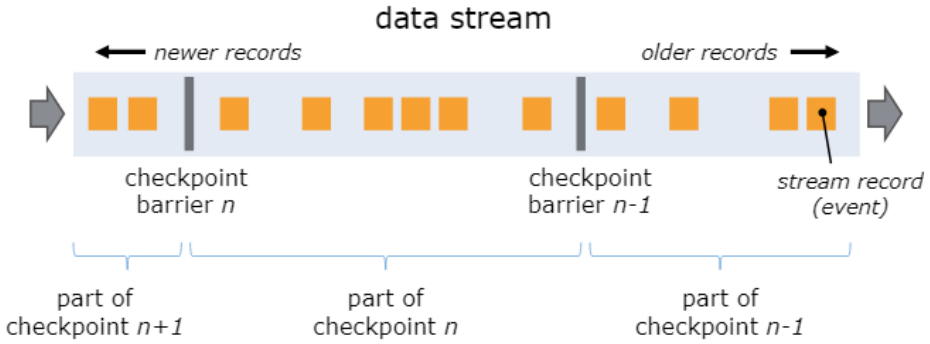

Figure 2.7: Apache Flink Checkpointing mechanism, periodically inserting lightweight barriers into the data stream [17]

In the event of a failure, Flink initially halts the distributed streaming dataflow, then restarts the operators, and finally resets them to the last successful checkpoint. The input stream is also reset to the point of the state snapshot. In the special case of incremental checkpoints, instead of preserving in each checkpoint the full state, simply maintain the differences between each checkpoint and store only the "delta" between the last checkpoint and the current state. Incremental checkpoints can have significant performance impact especially when working with large states, which is the case in our problem.

Savepoints offer advanced state management in Flink. They are manually triggered snapshots of an application's state and provide greater control and flexibility compared to checkpoints. Savepoints can be used for application recovery, migration across different Flink versions or setups, and experimentation.

Enabling checkpointing in Flink involves configuring the checkpoint interval and ensuring the source can replay records for a certain duration. Additionally, the storage for state should be persistent, such as HDFS. Savepoints are triggered by the user and provide a backup mechanism for restoring the application, with or without state, in case of failure or upgrade. By default, checkpoints are not retained and are deleted when a program is cancelled. However, you can configure periodic checkpoints to be retained for resuming jobs from failures.

2.2.6 Data Processing

2.2.6.1 Time Notion, Windows & Watermarks

In stream-processing frameworks like Apache Flink, understanding the notion of time is essential for effectively processing unbounded data in real-time. Apache Flink supports different time concepts, namely *event time*, *ingestion time*, and *processing time*, which determines how events are processed and grouped. One

widely used approach for stream handling is through the utilization of windows. Furthermore, in order to handle out-of-order events and ensure progress in event time, Flink incorporates a mechanism called **watermarks**. This subsection delves into the various time notions employed in Flink, explores the built-in window mechanisms available, and discusses the significance of watermarks in preserving temporal order during stream processing.

Time Notions

The different time notions that Apache Flink supports are [18]:

- **Processing time** refers to the time based on the clock of the machine where the event is being processed. This notion is straightforward to use, as it relies on the local system clock and requires no coordination between streams and machines. However, it can lead to inconsistent results since the processing time may vary across different executions of the job. In distributed systems or scenarios involving network delays, processing time might not be ideal as events can arrive out of order.

- **Event time** represents the time when individual events are generated at their source. It is typically based on a timestamp field included in the event's metadata. When using event time, the same input consistently produces the same result, ensuring result determinism. However, working with event time can be more challenging due to factors like handling out-of-order events.

- **Ingestion time** reflects the timestamp when events reach the stream processing application. It considers the processing delay and assigns a timestamp as soon as the processing system "consumes" the message. Ingestion time provides more predictability compared to processing time but still cannot handle time-of-origin and out-of-order data completely.

Windowing in Flink

Windowing is an essential in stream processing frameworks when dealing with infinite data streams, as they allow aggregations to be executed on a bounded set of data. By dividing the stream into finite blocks called windows, windowing enables the computation of aggregate functions on well-defined segments. [46]

In Flink, a window opens when the first data element arrives and closes when certain criteria are met to indicate the end of the window. Those criteria can be based on time, count of messages, or more complex conditions. When defining a time-based window in Flink, it is necessary to specify the notion of time, as discussed in Section 2.2.6.1.

Flink provides various windowing strategies to effectively handle infinite data streams. Furthermore, users can create their own strategies beyond the predefined ones. The default window strategies in Flink are the following:

- **Tumbling windows** that divide the data stream into fixed-size, non-overlapping windows based on elapsed time. Computation is performed on the data within each window independently.
- **Sliding windows** that are similar to tumbling windows but allow for overlapping windows. Each window slides over the data stream at a specified interval, enabling continuous computations.
- **Session windows** that group data based on periods of activity rather than fixed time intervals. They start with the arrival of data and close when no data is received for a specified duration.
- **Global windows** that cover the entire data stream and execute computations based on specified triggers, such as the arrival of a certain number of elements.

Watermarks and Event Time in Flink

Watermarks are a critical mechanism in Apache Flink for measuring progress in event time. They are part of the data stream and carry a timestamp indicating the event time reached at that point in the stream. Watermarks help handle out-of-order streams and asynchronous operations where events may not arrive in timestamp order. A watermark declaration at timestamp t signifies that no more events with a timestamp t' ¡= t (i.e. events with timestamps older or equal to the watermark)should be expected. Once an operator receives a watermark, it can advance its internal event time clock accordingly. [19]

Watermarks provide a way to track the completeness of incoming data and aid in operations such as windowing. For example, in an hourly-window operation, watermarks allow Flink to determine when the specific hourly window surpasses an hour, triggering the closure of the window (Fig. 2.8).

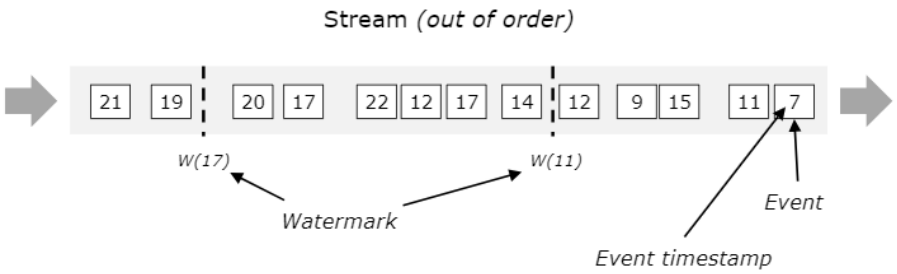

Figure 2.8: Flink watermarks in an out-of-order stream [19]

Late elements pose a common challenge in event-time windowing, where elements arrive after the watermark has passed the end timestamp of a window. By default, Flink drops these late elements, but it provides the flexibility to specify a

maximum allowed lateness for window operators. This parameter determines how long elements can be delayed before they are dropped. If an element arrives within the allowed lateness period, it can still be added to the window, potentially triggering additional computations based on the window's trigger. Flink retains window state until the allowed lateness expires, after which the window is removed, and its state is purged. Effectively managing late elements and setting an appropriate allowed lateness value is crucial for accurate event-time processing and windowing in Flink. [13]

Choosing the Appropriate Time

The choice of time notion depends on the requirements of your streaming application and the characteristics of the system. While processing time is simple and provides low latency, it may not guarantee consistent results in all scenarios. Event time offers result determinism but can be more challenging to work with. Ingestion time strikes a balance between the two, providing more predictability but still facing limitations.

By default, Apache Flink uses processing time as the time characteristic. However, depending on the application's requirements, it may be necessary to switch to event time, especially in scenarios involving asynchronous or distributed systems. Event time is supported by watermarks, allowing for accurate event-based computations. This time characteristic can be customized using Flink's environment variable.

2.2.6.2 Flink Programs & DataStream API

Flink programs are regular programs that manipulate distributed collections by implementing various transformations such as filtering, mapping, state updates, joining, grouping, defining windows, and aggregating. These collections are initially created from different sources like files, Kafka topics, or local in-memory collections. The results are returned via sinks, which can write data to distributed files or standard output.

In Flink, programs can be executed in standalone mode on a local JVM or on clusters of multiple machines. Depending on the type of data sources (bounded or unbounded), users can choose to write either a batch program or a streaming program. The DataSet API is used for batch processing offering a programming model similar to traditional batch processing and is well suited for offline data processing. the DataStream API, on the other hand, is specifically designed for streaming, making it suitable for real-streaming applications. It is important to note that Flink's DataStream API can also combine both stream and batch capabilities. Given that our work focuses on real-time streaming analytics, we have chosen to utilize the DataStream API in the upcoming chapters.

The DataStream API in Flink represents a collection of data and provides operations for working on immutable streams. Similar to Java collections, DataStreams cannot have elements added or removed directly, and their elements can

only be accessed and manipulated through the API's transformations. By adding a source and applying operations like map and filter, new streams can be derived and combined, allowing for data processing within the DataStream.

A characteristic of Flink programs is their lazy evaluation. When the program's main method is executed, the loading of data and transformations do not happen immediately. Instead, operations are added to the program's plan, and the execution is triggered explicitly by invoking the *execute()* method on the execution environment. The lazy evaluation allows for the construction of sophisticated programs executed as one holistically planned unit. [15]

The application structure in Flink follows a similar pattern for both batch and streaming processing, with the main components being:

1. **Get the execution environment:** Obtain the reference to the appropriate execution environment (StreamExecutionEnvironment for streaming and ExecutionEnvironment for batch).

2. **Source:** Read the data from the desired data source, such as files, messaging systems, or databases. Flink provides various source functions for different types of sources, including *readTextFile* for reading text files, *socketTextStream* for reading from a socket, and fromCollection or *fromElements* for creating a stream from given elements or a Java collection.

3. **Transformations:** Apply transformations on the data stream or dataset using operations like *map*, *filter*, *flatMap*, *union*, or *reduce*. These transformations enable applying desired business logic on individual elements or combining elements from different streams.

4. **Sink:** Store or output the results of the computations using sinks provided by Flink. Supported sinks include *print* (outputs to console), *writeAsText* (writes to multiple files based on parallelism), *writeAsCsv* (writes as comma-separated values), and *addSink* (calls a custom sink function or connector, such as Apache Kafka).

Listing 2.1 demonstrates the implementation of a typical WordCount application using the DataStream API.

To configure Flink applications, it is recommended to use command-line parameters or configuration files instead of modifying the runtime code. This approach enhances flexibility, as the same application can be executed in various deployment modes without modifying the code.

```
object WindowWordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter {
        _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(0)
      .timeWindow(Time.seconds(5))
      .sum(1)

    counts.print()

    env.execute("Window Stream WordCount")
  }
}
```

Listing 2.1: DataStream API: example of streaming window word count application, that counts the words coming from a web socket in 5 second windows

2.2.7 Environment Setup

To work with Apache Flink, it is crucial to set up the required environment properly. This section will guide you through the process of setting up an Apache Flink cluster, which is essential for testing and deploying Flink applications in multiple machines.

2.2.7.1 Setup a Multi-Node Flink cluster

The recommended approach for getting started with Apache Flink is to download the binaries from the official website [16]. The standalone cluster version offers a straightforward setup without requiring additional effort. However, it is essential to modify the configuration files to ensure optimal execution, as the default configurations may not suffice for executing even simple scenarios.

A multi-node Flink cluster consists of multiple machines interconnected to form a distributed topology. To set up a cluster, follow these steps:

1. Install the standalone version of Apache Flink on each machine, as mentioned in the previous paragraph.

2. Modify the configuration files on each machine to specify the IPs of the Job Manager and Task Managers. This ensures proper communication and coordination within the cluster.

In distributed clusters and production environments, it is highly recommended to fine-tune Flink configurations based on the specific requirements of your job and workload. Consider modifying the default memory configurations, choosing

an appropriate state backend, and configuring checkpointing options. These optimizations can significantly impact the performance of your Flink applications.

2.2.7.2 Building from source

In certain cases, modifying the source code of Apache Flink becomes necessary to achieve specific project goals and desired functionality. The source code for different versions of Flink can be found in the public repository on GitHub.

When working with the source code, it is crucial to pay attention to the project version. Open-source projects often have multiple variations, even within the same version, which can cause confusion. It is recommended to work with the latest stable edition to minimize errors and unexpected behavior.

Building the Apache Flink project from source can be time-consuming, typically taking more than 30 minutes, excluding the tests. Despite the project being divided into multiple modules, changes in the code only take effect after building the entire project from scratch. This limitation can hinder effective testing. Therefore, exploring alternative methods to build Flink after applying small changes would be beneficial for future development if additional functionality is required.

After successfully building the source code, a new directory named /build-target is generated within the build path. This directory contains all the files typically found in a binary installation, allowing the installation steps to be followed as outlined in the previous section.

2.3 Apache Kafka

Apache Kafka is an open-source, distributed streaming platform known for its high throughput and fault-tolerant design. It serves as a reliable and high-throughput publish/subscribe messaging system, often described as a distributed event log where new records are immutable and appended at the end of the log. Kafka has become a ubiquitous solution for real-time data ingestion, analytics, and processing of streaming data. It is compatible with popular Big Data platforms like Spark, Storm, and Flink, making it a versatile choice for organizations seeking efficient and scalable data pipelines.

Originally developed at LinkedIn in 2010 by Jay Kreps et. al [30], Kafka was created to address the challenges associated with handling massive volumes of real-time data streams. Key features of Kafka include its distributed design, which ensures high throughput, fault tolerance, and horizontal scalability. The platform organizes data into topics and partitions, with multiple brokers forming a Kafka cluster. By leveraging a publish-subscribe messaging model, Kafka allows producers to publish messages to specific topics, while consumers can subscribe to those topics to receive the messages. This flexibility enables a wide range of use cases, from real-time stream processing to batch consumption, and facilitates the integration of data from multiple sources.

2.3.1 Publish/Subscribe messaging

Publish/Subscribe (Pub/Sub) messaging pattern is a communication paradigm extensively utilized in distributed systems (Fig. 2.9). It enables loosely coupled and scalable message exchange between senders (known as the publishers) and receivers (known as subscribers). Publishers generate messages, categorize them into topics, and transmit them to a central message broker without any knowledge of the subscribers interested in those messages. Subscribers express their interest in specific topics and receive relevant messages from the broker, independent of the knowledge of publishers. The broker efficiently distributes messages to subscribed subscribers, enabling decoupled and asynchronous communication. Pub/Sub is widely employed in real-time data streaming, event-driven architectures, message queuing systems, and microservices, offering scalability, flexibility, and resilience to distributed systems.

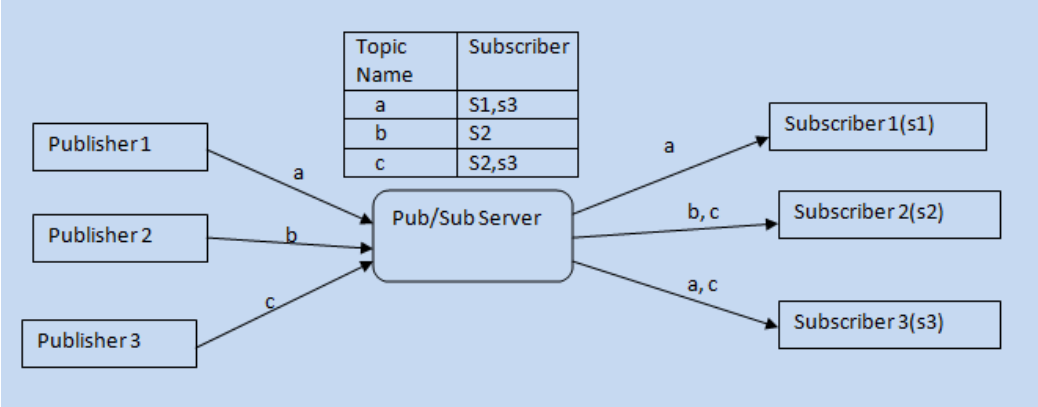

Figure 2.9: Publish/Subscribe messaging pattern

2.3.2 Architecture

Before breaking down the architecture even further, it is essential to introduce some fundamental concepts that form the building blocks of Kafka.

Topics serve as the central organizing unit within Kafka, representing a particular stream of records. Messages published by producers are categorized into these topics and are continuously appended to them. This design ensures a persistent and ordered collection of records associated with each topic.

Producers are the clients responsible for publishing messages on specific topics. They can choose a specific partition within a topic or allow Kafka to handle the partitioning automatically based on the specified topic's partitioning strategy.

Consumers subscribe to topics (one or more) and read messages from the partitions assigned to them. Each consumer maintains its offset, which represents

the position of the last consumed message within each partition. This offset enables consumers to have control over their progress and easily handle failures or reprocessing scenarios.

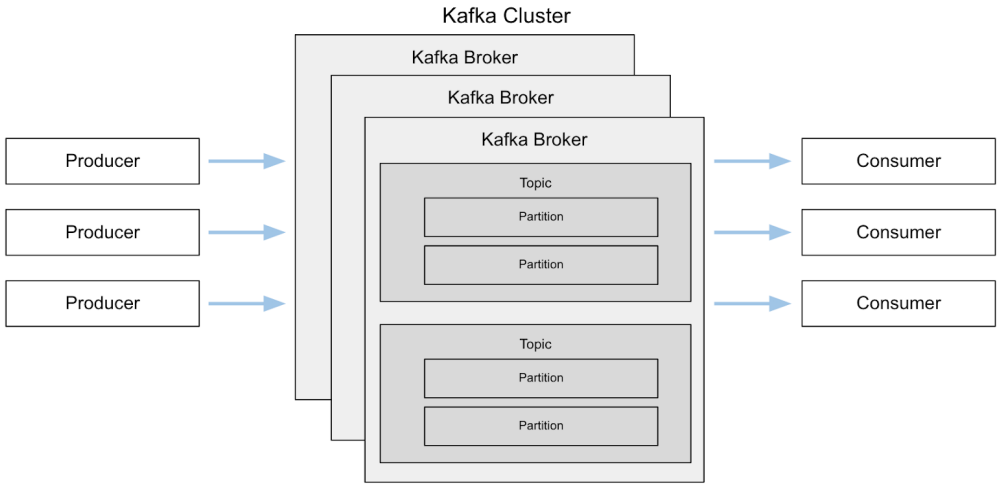

Figure 2.10: Apache Kafka - Architecture

On a high level, Kafka comprises three main components: the **Kafka cluster**, **producers**, and **consumers**. A single Kafka server within the cluster is called a **broker**(usually at least three brokers for redundancy). The broker is responsible for several crucial tasks, including collecting messages from producers, assigning offsets, committing messages to disk, responding to consumers' fetch requests, and sending messages. Within the cluster, one broker acts as the **cluster controller**, responsible for monitoring broker failures and managing various administrative tasks.

Following the publish/subscribe messaging pattern, producers create new messages and send them to specific topics. A topic serves as a category or classification for the data being sent and can be further divided into **partitions**. Each partition maintains separate commit logs, and the order of messages can be guaranteed within the same partition. Partitioning a topic into multiple partitions facilitates easy scaling, as different consumers can read from different partitions. This allows partitions and consumers to be distributed across different servers, leading to higher throughput. Consumers read messages by subscribing to one or more topics. Messages are consumed in the order they were produced, which is achieved by keeping track of the offsets (sequential IDs of specific messages within specific partitions). Consumers are organized into **consumer groups**, and all partitions of a topic are evenly distributed among the members of a consumer group.

Kafka also incorporates a unique retention policy, based on which messages are persistent on disk for a configured amount of time, and after expiration, they are released.

2.4 Financial Analytics: Discovering Breakout Patterns

Technical analysis of financial markets relies on price charts and other technical indicators to identify patterns and predict future price movements. Unlike fundamental analysis, which focuses on the underlying financial and economic factors that influence asset prices, technical analysis is primarily concerned with analyzing patterns in price movements themselves and can contribute to identifying these patterns and can lead traders to make informed decisions about when to buy or sell.

$$EMA^{j}_{s,w_i} = \left[Close_{s,w_i} \cdot \left(\frac{2}{1+j}\right)\right] + \underbrace{EMA^{j}_{s,w_{i-1}}}_{\text{prev. window}} \cdot \left[1 - \left(\frac{2}{1+j}\right)\right] \tag{2.1}$$

with

$$|w| \quad : \quad \text{window duration in minutes} \tag{2.2}$$

$$j \quad : \quad \text{smoothing factor for EMA with } j \in \{38, 100\} \tag{2.3}$$

$$s \quad : \quad \text{symbol } s \in S = \{s_1, \ldots, s_n\} \tag{2.4}$$

$$Close_{s,w_i} \quad : \quad \text{last price event for } s \text{ observed in window } w_i \tag{2.5}$$

$$EMA^{j}_{s,w_0} \quad = \quad 0 \tag{2.6}$$

An essential tool in technical analysis is the exponential moving average (EMA). EMAs are calculated by weighting recent prices more heavily than older prices, which makes them more responsive to short-term price changes (Eq. 2.1). Traders often use two EMAs, calculated over different intervals, to identify *breakout patterns*, which describe meaningful changes in the development of a price that indicate the start of a trend, even if only temporary. Breakout patterns can be either bullish or bearish. A bullish breakout (Fig. 2.11) occurs when the price of an asset starts to rise steadily (crossover from below), while a bearish breakout (Fig. 2.12) occurs when the price starts to decline steadily (crossover from above). These patterns can help traders identify opportunities to buy or sell an asset, potentially maximizing revenue or minimizing losses. While calculating EMAs is a popular method for identifying breakout patterns, they are not the only method used in technical analysis. Traders often use a combination of technical indicators, such as Volume Price Trend Indicator (VPT) or Relative Strength Index (RSI), to confirm or validate their trading decisions. The effectiveness of these indicators depends on a variety of factors, including market conditions, the time frame being analyzed, and the skill and experience of the trader.

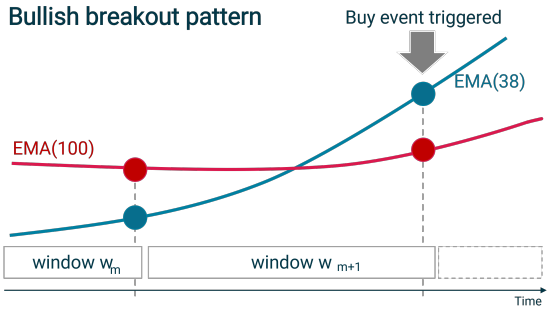

Figure 2.11: Bullish Breakout [21]

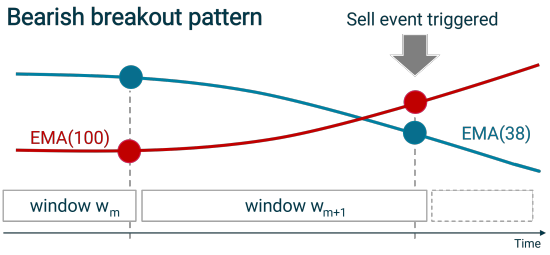

Figure 2.12: Bearish Breakout [21]

The applications developed in our work uses EMAs to detect breakout patterns and provide traders with advice on when to buy or sell. Specifically, we track two EMA intervals for dynamically changing sets of symbols using event aggregation over tumbling windows (Query 1), and provide the latest three breakout/pattern detections (Query 2). To mimic the typical behavior of traders, our application subscribes to multiple symbols and provides advice on opportunities to buy or sell based on the detected breakout patterns. We use batches that continuously update their set of subscriptions to achieve dynamic subscription to symbols. Subscription patterns can be implemented to be unpredictable but reproducible, and subscriptions can change dynamically over an evaluation session (see Section 3.1 for S-TAP and Section 4.2.1 for P-TAP).

2.5 Financial Dataset

The dataset we used consists of high-volume tick data captured the full week of November 8th to 14th, 2021(i.e. five trading days Monday to Friday + Saturday and Sunday), provided by Infront Financial Technology GmbH [20]. The raw dataset includes 289 million tick data events covering 5504 equities and indices that are traded on three European exchanges: Paris (FR), Amsterdam (NL), and Frankfurt/Xetra (ETR). The dataset contains all tick data events for security types equities and indices captured on these days.

The dataset is originally available as a collection of flat comma-separated values (CSV) files, one file per day. Each line in a file represents a single event. The attributes directly relevant to our study are highlighted in Table 2.1. Global CEST timestamps are in the format HH:MM:SS.ssss while dates are stored as DD-MM-YYYY.

It is worth noting that some events in the raw dataset appear to come with no payload. This is because only a certain subset of attributes is required for evaluation, and several attributes have been eliminated to preserve the number of events and their update patterns over time while minimizing the overall size. During the evaluation of S-TAP (§3.4), we utilized the Challenger Platform provided by the

Debs Challenge. In this particular case, the data had already been filtered, including only the essential events required for the application (more details in Sec. 3.1). To evaluate P-TAP after the conclusion of the Challenge, we adopted the approach used in the DEBS'22 GC. We pre-filtered the initial raw data, eliminating events that contained attributes irrelevant to our application, and accomplished this using the Data Ingestion Manager (§4.2.1). This filtering process significantly enhances the evaluation process by removing unnecessary noise.

Attribute	**Description**
ID.[Exchange]	Unique identifier for this symbol with trading exchange: Paris (FR)/Amsterdam (NL)/ Frankfurt (ETR)
SecType	Security type: [E]quity or [I]ndex
Last	Last trade price
Date	System date for last received update
Trading time	Time of last update (bid/ask/trade)

Table 2.1: Useful attributes from dataset with their description

Name of Dataset	**Description**
Raw	289 million events consisting of tick data and housekeeping events
Day 1-5	59 million events of tick data after filtering unnecessary events from raw dataset
Day 1-4	47 million events of tick data, subset of the Day 1-5 Dataset for 4 days of trading
Day 1-3	35 million events of tick data, subset of the Day 1-5 Dataset for 3 days of trading
Day 1-2	23 million events of tick data, subset of the Day 1-5 Dataset for 2 days of trading
Day 1	11 million events of tick data, subset of the Day 1-5 Dataset for 1 day of trading

Table 2.2: Different Datasets description

2.6 HDFS - Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a fundamental element of the Apache Hadoop framework, serving as an open-source, fault-tolerant storage data storage file system that operates on commodity hardware. It was designed to address the limitations of traditional databases, particularly in terms of handling large amounts of data with reliability and speed. Inspired by Google's MapReduce [7] and Google File System (GFS) [23] research papers, HDFS offers a reliable and efficient framework for storing and retrieving distributed data in data-intensive applications. Its scalability and fault-tolerant design make it an essential solution for effectively managing and analyzing the ever-increasing volumes of complex data in the field of big data analytics.

2.6.1 Key Features of HDFS

1. **Handles big data:** HDFS efficiently handles data sets ranging from gigabytes to terabytes in size, providing a solution that traditional file systems cannot match. By dividing data into manageable blocks and leveraging its cluster architecture, HDFS ensures fast processing times, with the capability to deliver more than 2GB of data per second.

2. **Fault-tolerance:** By maintaining multiple replicas of each data block on different machines, HDFS can mitigate the impact of node failures and data corruption. Multiple replicas, configured by the replication factor, are distributed across different machines in the cluster, minimizing the risk of data loss. In the event of a node failure, HDFS automatically replicates the lost blocks to other available machines, ensuring data durability.

3. **Scalability:** HDFS provides mechanisms for managing and scaling resources in each system. It offers vertical and horizontal scalability, allowing it to handle enormous datasets and meet the growing demands of big data applications.

4. **Portability:** HDFS is designed to be portable across multiple heterogeneous hardware platforms and compatible with various underlying operating systems. This flexibility enables organizations to deploy HDFS on diverse infrastructures, from commodity hardware to cloud-based environments. The cross-platform compatibility of HDFS makes it accessible and adaptable for different deployment scenarios.

2.6.2 Architecture

HDFS incorporates a master-slave topology, consisting of a master node called the **NameNode** (typically one or more in high-availability setups per cluster) and multiple slave nodes known as **DataNodes**, as shown in Fig. 2.13.

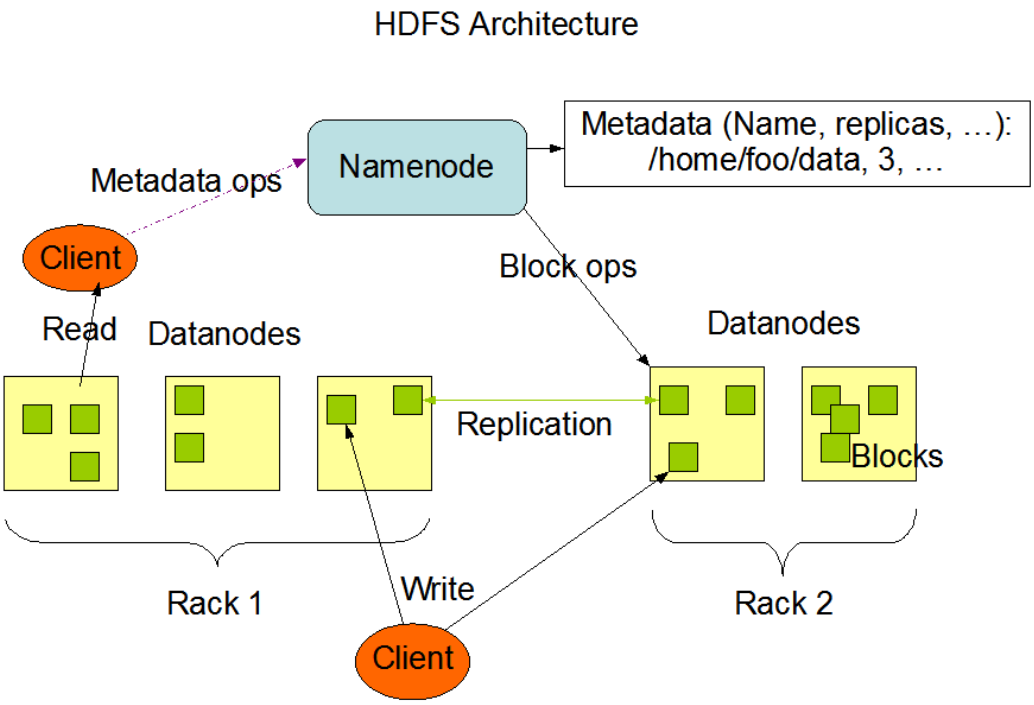

Figure 2.13: HDFS - Architecture [24]

2.6.2.1 NameNode

NameNode serves as the central coordinator in HDFS and is responsible for managing and maintaining DataNodes. NameNode also keeps track of all the metadata in each data block, and its replicas by maintaining two persistent files; **editLog and FSimage**. EditLog records every change that happens within the file system metadata, and FSimage stores the entire file system image since the beginning. Metadata contains information about where data is stored, permissions, number of replicas and directory structure.

The NameNode keeps this metadata in memory for faster access and uses a persistent storage mechanism, typically on disk, to ensure durability. As the single point of failure in the HDFS architecture, the NameNode's high availability and fault tolerance are crucial. Various techniques, such as backup solutions like HDFS Federation or High Availability (HA), can be employed to mitigate the risk of NameNode failure.

2.6.2.2 DataNodes

DataNodes are responsible for storing actual data in blocks assigned by the NameNode. Each DataNode manages the storage of data on its local disk and communicates with the NameNode to perform operations such as reading, writing, and

replicating data blocks. HDFS overcomes the issue of DataNode failure, HDFS creates replicas (copies) of the data. The default replication factor is three, and it is strongly advised not to set it below three.

DataNodes maintain regular communication with the NameNode by sending heartbeats to report their health status and update the NameNode with information about the blocks they store. They also provide block reports to inform the NameNode about the list of blocks present on their local disks. This communication between the NameNode and DataNodes enables the NameNode to maintain an updated view of the cluster's data distribution and availability.

2.6.3 Data Manipulation

Similar other distributed files systems, HDFS stores system files as a sequence of fixed-size blocks on DataNodes. These blocks serve as the fundamental units of storage and processing within HDFS. By default, the block size in Apache Hadoop 2.x and 3.x is 128MB (64MB in Apache Hadoop 1.x), but this size can be modified through configuration. Before the NameNode can store and manipulate data, files are divided into smaller block-sized chunks that are stored independently.

2.6.3.1 Block Division

The number of blocks required to store a file depends on its initial size. Except for the last block, which may be smaller, all other blocks have the same size as the configured block size (128MB by default). For instance, a 320MB file would be split into two blocks of 128MB each, with the remaining 64MB stored in a third block (Figure 2.14)

Finally, each block is replicated into multiple copies based on the replication factor.

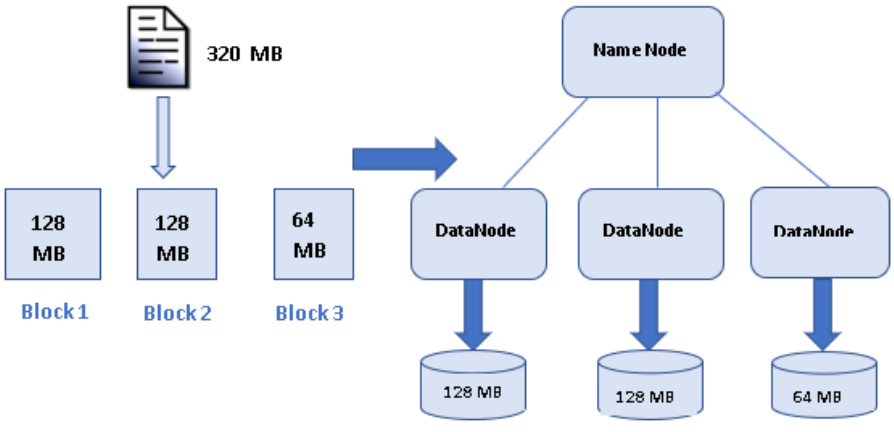

Figure 2.14: HDFS - Block Division [27]

It is important to note that the selection of a default block size of 128MB aims

to strike a balance between overhead and processing time. If the block size is too small, the presence of numerous data blocks and associated metadata can lead to increased overhead. Conversely, if the block size is excessively large, the processing required for each block also increases.

2.6.3.2 Replication

Data replication plays a crucial role in handling unexpected failures and ensuring data reliability within HDFS. The NameNode creates multiple copies, or replicas, of each data block. By default, HDFS maintains three replicas of each block, although this replication factor can be adjusted based on the desired level of redundancy and fault tolerance. These replicas are distributed across different DataNodes in a strategic manner, considering rack awareness policies to maximize fault tolerance and network bandwidth while ensuring high availability.

HDFS employs the following rack awareness policies:

1. One DataNode can store one replica of a data block.
2. A single rack cannot contain more than two replicas of a specific block.
3. The number of racks used inside an HDFS cluster must be smaller than the number of replicas.

The NameNode maintains the mapping between blocks and DataNodes, ensuring that the replication factor is preserved. In the event of a DataNode failure or unavailability, the NameNode schedules replication of the missing replicas to other available DataNodes to maintain the desired replication factor.

2.6.4 Environment Setup

In order to work with HDFS, it is essential to ensure that all systems are properly installed and functioning. Setting up an HDFS cluster, especially in a multi-node environment, requires careful optimization of the installation process to save valuable time and effort.

2.6.4.1 Setup a Multi-Node HDFS Cluster

Deploying an HDFS multi-node cluster can be a challenging task that often involves trial and error. Even setting up the HDFS standalone version can be complex, let alone configuring it across multiple machines. Additionally, the original project involved modifying the HDFS source code, which necessitated building and installing HDFS from source. The following steps summarize the sequential process involved:

1. Installing the standalone version of HDFS (§2.6.4.1)

2. Setting up the multi-node HDFS cluster (§2.6.4.1)
3. Building HDFS from source (§2.6.4.2)
4. Automating the building and testing process (§2.6.4.3)

Installing the standalone version of HDFS

The most common and straightforward approach to installing HDFS is by downloading the binaries from the official Hadoop website [24]. Before setting up a full multi-node cluster, it is recommended to start with a local standalone version. However, even deploying the standalone version of HDFS can be non-trivial, as it involves addressing various issues such as permission errors, environment variable problems, and different configurations. To ensure a smooth installation process, a detailed 3-4 page manual was created, highlighting the installation steps in detail and providing troubleshooting methods for various scenarios.

Alternatively, Docker images can provide an easier way to use HDFS, including pre-existing images for multi-node clusters. However, in this project, the requirement for high customizability ruled out the use of dockerized environments.

Setting up the multi-node HDFS cluster

To create an HDFS cluster, multiple machines were utilized to form a cluster topology. Configuring standalone versions of HDFS on each machine is a crucial step in creating an HDFS cluster. The final stage involves modifying the corresponding configurations to specify the master and slave IPs across the machines. Upon successful installation of the cluster, the status is displayed in Figure 2.15.

Ensuring homogeneity among the different machines is key to a successful multi-node cluster deployment. Creating such a cluster involves a complex, multi-step process, and maintaining consistency across the machines is considered a best practice. Failure to maintain consistency can make monitoring and managing the machines challenging.

2.6.4.2 Building HDFS from source

Building HDFS from source differs from using pre-built binaries and involves downloading the project from GitHub and compiling it. Selecting the appropriate version can be challenging, as there are numerous versions available, and not all of them function properly. Different versions may also have different requirements and may cause errors during the building process. Generally, stable versions from the main branch are preferred.

The initial build process is time-consuming, taking more than 30 minutes excluding tests and pre-built requirements. During this phase, the entire Hadoop project is built, resulting in binaries similar to those found on the official Hadoop website. Once this step is completed, steps 1 and 2 from the setup process are utilized to deploy the HDFS cluster.

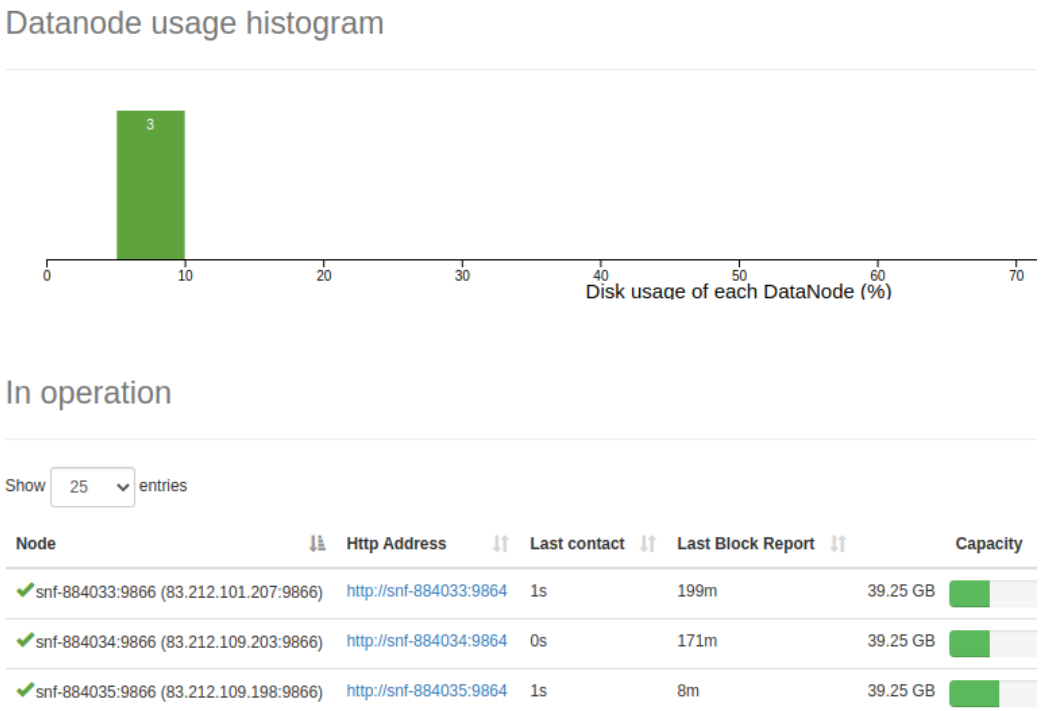

Figure 2.15: HDFS Multi-Node Cluster status

Fortunately, modifying the HDFS code in later stages does not require the lengthy build time. After the initial build, HDFS can be treated and built separately from the rest of the Hadoop project, reducing the execution time to less than a minute. The output of this build process produces a subset of the whole project, and the new files should overwrite the old ones to produce the modified version of the code.

2.6.4.3 Automating the building and testing process

Manually replacing the newly produced files from the previous step can be time-consuming and frustrating. To streamline this process, a bash script was created to automate the building and testing process(Lis. 2.2). This script simplifies the repetitive tasks involved in building and testing HDFS, reducing the potential for human error required and enhancing efficiency.

By automating these processes, developers can focus more on the actual development and experimentation with HDFS, rather than spending excessive time on manual tasks.

```
#!/bin/sh
projectDir="/home/skalogerakis/Projects"

tar -xvf  ${projectDir}/hadoop/hadoop-hdfs-project/hadoop-hdfs/target/hadoop
    -hdfs-3.2.2.tar.gz -C $HOME

echo  "Extracting new version completed.\n\n"

rsync --update -raz --progress $HOME/hadoop-hdfs-3.2.2/. $HADOOP_HOME

echo  "Copy new version to previous config completed\n\n"

rsync --update -raz --progress etc/ $HADOOP_HOME

echo  "Completed"

# Remove temporary files
rm -r $HOME/hadoop-hdfs-3.2.2/
```

Listing 2.2: Shell Script to automate building/testing procedure from source

Chapter 3

Single-source Tick Analysis Platform(S-TAP)

3.1 Introduction & DEBS Grand Challenge 2022

The 2022 DEBS Grand Challenge (GC) [21] supported by Infront Financial Technology[1] focuses on real-time complex event processing of high-volume tick data. In the real-world dataset provided (Section 2.5), about 5000+ financial instruments are being traded on three major exchanges over the course of a week. The goal of the challenge is to efficiently compute specific trend indicators and detect patterns that resemble those used by real-life traders to decide on buying or selling on financial markets. The 2022 DEBS GC requires developers to implement a basic trading strategy aiming at (a) identifying trends in price movements for individual equities using event aggregation over tumbling windows (Query 1) and (b) triggering buy/sell advises using complex event processing upon detecting specific patterns (Query 2). The first query implements the exponential moving average (EMA) [29], an indicator per symbol used in technical analysis to identify trends. Q2 uses the quantitative indicators of query 1, tracking two EMAs (with different smoothing factors) per symbol computed over different intervals to identify breakout (indication of market turning to bullish or bearish) patterns.

The evaluation dataset (Section 2.5) is provided by the GC platform via a gRPC-based API in a continuous stream of event batches, B_i, $i = 0, 1, 2, \ldots$ Each batch B_i includes a list of events, each event comprising a symbol (identified by unique ID and exchange), type (equity or index), last trade price, date of last trade, and time of last update (bid/ask/trade). Each batch also specifies *lookup symbols* that the evaluation platform *subscribes to* for this batch. The analytics pipeline must report answers to Query 1 and 2 for the subscribed symbols for each batch B_i back to the GC platform. Performance is evaluated based on average throughput and mean (for the two queries) of the 90th-percentile latency for each batch. The reporting mechanism is also based on the supported gRPC API.

[1] `https://www.infrontfinance.com/`

We decided to use the Apache Flink [5] framework as our data analysis platform to leverage the scalability and operational reliability afforded by the base Flink platform, customizing the application logic to solve the DEBS 2022 GC in an accurate manner and avoiding loss of information. Apart from Flink, our complete software stack includes a data ingestion and reporting service, fetching data from the GC platform via the gRPC-based API [42], and Apache Kafka [30] as a messaging and persistence service (Fig. 3.1) that decouples ingestion from data analysis, simplifying their integration.

In designing our solution to the DEBS 2022 GC, we identified the handling of late (out-of-order) events and the mapping between batches of events and window-closings that contribute to them as major correctness challenges. To address them we designed a custom window operator that leverages event semantics to correctly order events and to map event-batches to window-closings. While tuning the performance of our solution, we identified the need to rate-control the data ingestion process (which pulls event-batches off of the GC platform) to ensure a suitable latency-throughput operating point. Addressing this as well, we achieved a solid response to the 2022 GC objectives. While our code parallelizes most operators (including the custom window logic), it maintains a single instance of the source operator and batch-unpack logic (as parallelizing this introduces further correctness considerations), eventually limiting the achievable parallelism. Consequently, we refer to the system described in this chapter as **S-TAP** (Single-source Tick Analysis Platform). In chapter 4, we analyze how we have tackled these challenges with P-TAP, and in Section 4.4, we present our experience and insights gained from transitioning from S-TAP to P-TAP.

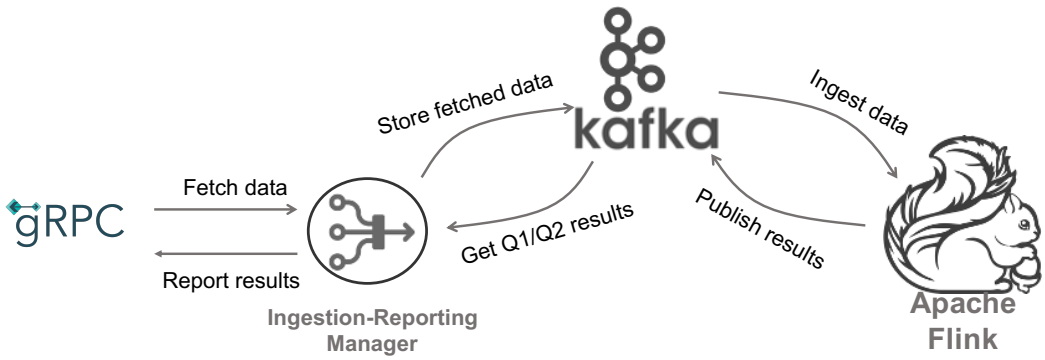

Figure 3.1: Data analysis pipeline

3.2 Design and Implementation

The proposed architecture of our data analysis pipeline comprises three major components (Fig. 3.1): (1) the Data Ingestion-Reporting Manager component, a tailor-made Java process that acts as an interface to ingest data from and report query results to the evaluation platform; (2) the Apache Kafka [30] component that is used to decouple the data ingestion/reporting phase from data analysis, and; (3) the stream analytics engine built on top of Flink. Here we describe the design and implementation of each component.

3.2.1 Data Ingestion-Reporting Manager (DIRM)

The Data Ingestion-Reporting Manager (DIRM) component is a Java process specifically designed to act as an interface with the DEBS'22 GC evaluation platform. It can ingest data and report the query results using the GC-supported gRPC-based API. It is also responsible to report query results back to the GC platform. The GC platform makes data available in batches, identified by an ID assigned by the gRPC service.

The GC platform evaluates the latency of our solution by monitoring the response time of each batch by the time we fetch it through the gRPC API until we report query results for the batch back to the platform. Having data go through an additional process, the DIRM, can increase latency. However, we opt for this design, as decoupling the data ingestion/reporting phase from the data analysis improves portability and interoperability of the solution. The ingestion/reporting component can be extended to fetch data from different data sources (e.g. files) and/or reporting services without affecting the implementation of the rest of the pipeline.

The ingestion and reporting tasks are implemented as separate threads. The ingestion thread fetches and stores data on a Kafka topic, while the reporting thread subscribes to the topics that the analytics task publishes query results (more in Section 3.2.2).

The data ingestion rate from the GC platform should match the results-reporting rate. Fetching data at a high rate leads to batch queue-up within DIRM, waiting for subsequent analysis. While this could improve throughput as the data analytics job will never be idle waiting for data, an unnecessarily-high fetch rate may overly penalize latency. To handle the latency vs. throughput trade-off, we implemented a rate controller in the data ingestion task of this component. The controller throttles the ingestion rate according to the rate results are generated and reported. We empirically (through repeated measurements and informed parameter settings) achieved a balance between latency and throughput of data analysis in our evaluation runs (more in Section 3.4.1).

The reporter can also be configured to report results for query 1 or query 2 or both (see Section 3.3). Finally, the DIRM component keeps track of the total number of ingested batches and the reported queries. When all queries have

been reported back to the evaluation platform, the reporting component signals benchmark completion using the GC-supported gRPC API.

3.2.2 Use of Kafka for asynchronous messaging

Kafka [30] is used to simplify the integration of the ingestion and data-analysis components. It maintains the ingested data before the subsequent analysis, and the query results before the reporter component reports them back to the GC-evaluation platform. The use of Kafka allows us to leverage an existing Kafka connector that is already well integrated with Flink (data ingestion, checkpointing) rather than having to implement such a connector from scratch.

DIRM uses Kafka producers to publish data to a Kafka topic. We opt for the synchronous Kafka producer API (instead of the more performant asynchronous mode) to reduce the window of uncertainty as to the status of published batches in case of DIRM or Kafka crash. Finally, we built our own custom (de)serializers to transfer data objects from and to Kafka topics in binary format.

3.2.3 Data Processing

The data analytics job comprises a stream-processing graph with multiple operators depicted in Figure 3.2. These operators a) consume the ingested data; b) unpack events included in batches; c) implement custom window logic to determine the last observed price per symbol in 5-minutes time-frames[2], guaranteeing there will not be dropped late events and window-closing will be correctly associated with event batches; d) calculate the EMA; e) discover crossover events; and e) gather and report the query results on all lookup symbols per batch. Next we describe the design choices and implementation details of each operator.

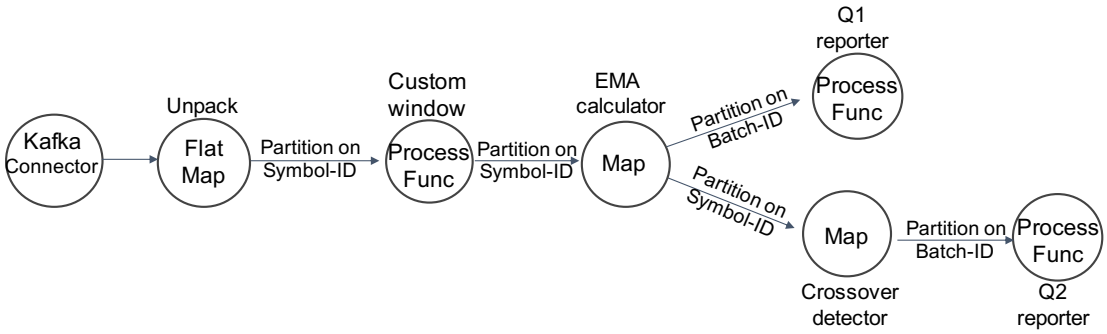

Figure 3.2: Stream-processing job

[2]We use the term *time-frame* rather than *window* to refer to the different time intervals/ranges whose state may be simultaneously maintained by the window operator

Source operator. This operator consumes data from Kafka by subscribing to the corresponding topic. We use the Flink-supported Kafka connector[3] as our data source operator. We assume a single instance of this (and the Unpack) operator, processing batches in batch-ID order. We will discuss the impact of this choice in this section.

Unpack operator. Each batch fetched consists of a list of events (trade actions for symbols) and a list of symbols of interest, *lookup symbols*, that a user subscribes to (requests query results for that symbols). The *Unpack* operator extracts and emits events from a batch and also injects metadata on each output tuple (Listing 3.1). The metadata include the (1) batch ID the event is extracted from; (2) a flag per event symbol that marks if it is included in the lookup list; (3) the number of lookup symbols in the batch; and (4) a flag that indicates if the event is the last occurrence of the symbol in the batch. The metadata are necessary for the subsequent analysis on downstream operators.

```
case class EventUnpackSchema (
    symbol: String,
    securityType: SecurityType,
    Price: Double,
    timestamp: Long,
    batchID: Long,
    isSymbolsLastOccurence: Boolean,
    lookupSymbolBool: Boolean,
    lookupSize: Int,
    isLastBatch: Boolean )
```

Listing 3.1: Emitted output of unpack operator

Window operator. Following the unpack operator is a window operator that emits the last observed price per symbol in 5-minute time-frames. A major challenge is to handle out-of-order late events, i.e., events that arrive after a window has closed. These events are typically dropped and as a results this could result in correctness issues in the subsequent trend analysis. Flink's built-in window operators support *allowedLateness* option that can accept late events for the specified amount of time when a window closes. However, it is still challenging to predict an appropriate *allowedLateness* value; in the general case, it is not possible to achieve a guarantee that there will not be events that arrive later than the specified setting.

Our goal was to design an application that will not sacrifice correctness over performance. We thus decided to build our own custom window operator that closes a time-frame when, based on event semantics, it determines that there are no events left out that belong to the corresponding 5-minute time-frame. For each symbol our window operator maintains a table of 5-minutes time-frames. Upon the arrival of an event, the window operator has to decide in which time-frame the event is to be assigned according to the event time. Our mechanism performs

[3]https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/kafka

event grouping and alignment using Equation 3.1:

$$f(event\_ts) = \lfloor (event\_ts/win\_interval) * win\_interval \rfloor \tag{3.1}$$

The input to Equation 3.1 is the timestamp of each processed event. The window interval is set to 5 minutes. The function returns the starting time of the 5-minute time-frame that the event belongs to. For example, event timestamps 14:00:00.001, 14:00:02.421, 14:00:04.343 all belong to the time-frame starting at 14:00:00.000.

For every 5-minute time-frame, the operator maintains the last-price seen for the symbol along with its timestamp. Our custom operator applies incremental processing logic, i.e., it updates the last-price seen of the symbol upon processing an incoming event by comparing if the new event's timestamp is later than the previous stored last-price. Thus we maintain minimal state per time-frame, avoiding buffering of all events within the same time-frame.

The operator also maintains a list of all batches seen and keeps track of the progress of processing each batch, namely whether the operator has processed all events of a batch according to the metadata emitted by the upstream unpack operator. The operator also identifies the 5-minutes time-frames affected by each batch. Specifically, we *link* each batch with the **last time-frame** *affected by the batch*. To do this we use the timestamp of the last occurence of the symbol within the batch (the metadata flag *isSymbolsLastOccurance* in Listing 3.1). In Figure 3.3, the last timestamp of symbol ABC in batch B_5 is 12:28, affecting *up to* time-frame 12:25-12:30.

As events are aggregated from multiple sources we cannot assume events for different symbols are timestamp-ordered. However, we assume that events for the same symbol (always ingested from a single source) have monotonically increasing timestamps across batches, a fact that we have validated in the GC data set [20]. Thus, for a given symbol the timestamp of its first event in batch B_i is later than the last timestamp of that symbol in batch B_{i-1}. Based on these ordering properties, for a given symbol, the batch B_i cannot affect a time-frame preceding the time-frame linked to B_{i-1}. For instance, in the example of Fig. 3.3 for symbol ABC, B_4 cannot affect a time-frame before the one linked to B_3.

When a window operator fully processes a batch, i.e., all events of the batch have been processed, the operator checks if there are *safe-to-close* time-frames to emit the symbol's last price observed in such time-frames. A time-frame is considered as safe-to-close when all batches linked to it and, *at least the first batch linked to the next time-frame*, are completely processed. In the example of Fig. 3.4, batches B_3 and B_4 are linked with time-frame 12:20-12:25. However, the time-frame is not considered as safe-to-close after the processing of these batches as we are not sure if the next batch contains events that contribute to it. As soon as we have processed the first batch of the next time-frame, i.e., batch B_5, we are sure that the subsequent batches cannot contain events for symbol ABC affecting time-frames preceding the one that B_5 is linked to.

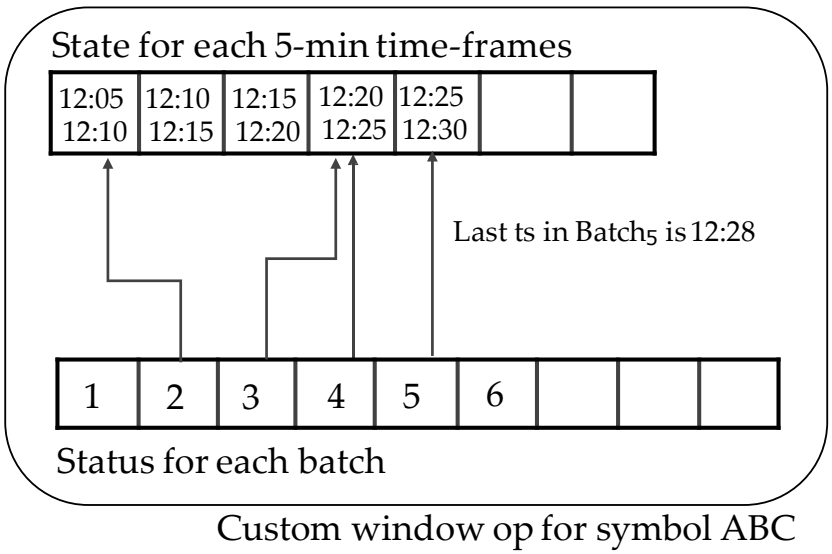

Figure 3.3: Custom window-operator state for symbol ABC. Each batch B_i, $i = 1, 2, \ldots$ points to the time-frame affected by the last occurrence (last ts) of symbol ABC in that batch

Another challenge for our custom window operator is *empty batches*, i.e., batches that do not contain events for a given symbol (e.g., batch B_1 in Fig. 3.3 does not contain events for ABC). As the events of each symbol are timestamp-ordered and we use a single instance of the source and unpack operator forwarding events to the corresponding window operator instance, we derive the following property: a batch B_i for which a window operator for symbol ABC has seen no events from, while having seen events from B_{i+1} or later, means that B_i is empty for ABC (e.g. B_1 in Fig. 3.4).

When a batch is complete, the window operator identifies the time-frame it is linked to and checks if there are pending time-frames that can now be marked as safe-to-close. The example in Fig. 3.4 illustrates the aforementioned scenario for symbol ABC: The last occurrence of ABC in batch B_2 has its timestamp within 12:05-12:10 (the time-frame is still not considered as safe-to-close when B_2 is fully processed). Batch B_3 is linked to the time-frame 12:20-12:25 (i.e., is the last occurrence of events regarding ABC fall into this time-frame). When B_3 is fully processed we can mark time-frames 12:05-12:10, 12:10-12:15 and 12:15-12:20 as safe-to-close. This is because all events for ABC in subsequent batches are expected to have timestamp later than the last occurrence of ABC in B_3.

The operator emits to its output the closing price of the symbol for each safe-to-close time-frame and purges its state. If there are no events for the symbol associated with a time-frame (e.g. time-frames 12:10-12:15, 12:15-12:20 for symbol ABC in Fig. 3.4), the operator ignores the time-frame and purges its state.

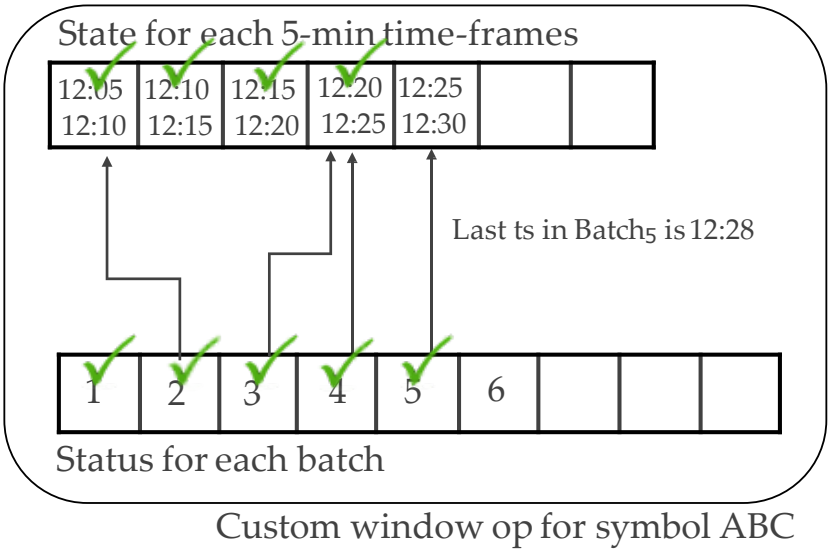

Figure 3.4: Window closing example: The checkmarks indicate that batches B_1-B_5 have been fully processed and four 5-minute time-frames are safely considered fully closed

However, if the symbol is in the lookup-symbols list, the operator emits a special-crafted tuple to indicate to the downstream operators that there is no closing price for the 5-minute time-window and thus, they should report the previous EMA (see EMA calculator described next).

When B_4 is completely processed, there are no new safe-to-close time-frames. In this case the custom window operator emits a specially-crafted output tuple for the lookup symbols that indicate that the processing of the batch has completed. These specially-crafted tuples indicate to the downstream operator (the EMA calculator described below) that it can rely on the last computed EMA corresponding the last closed time-window. In the example of Fig. 3.4 assuming ABC is a lookup symbol, when batch B_4 closes, the time-frame 12:20-12:25 is not safe-to-close, hence it emits a tuple signaling batch completion. That specific time-frame will be marked as safe-to-close only when B_5 is fully processed.

EMA calculator. The last observed price for a 5-minute time-frame emitted by the window operator is necessary for the EMA calculation. There is a separate instance of the EMA calculator per symbol. The EMA calculator operator computes the EMA according to Equation 3.2.

$$EMA^j_{w_i} = [Close_{w_i} * (\frac{2}{1+j})] + EMA^j_{w_{i-1}} * [1 - (\frac{2}{1+j})] \tag{3.2}$$

w_i: the 5-minute time-frame
j: the smoothing factor for EMA with $j \in \{38, 100\}$

$Close_{w_i}$: the last price observed within time-frame w_i

The operator computes EMA for a given time-frame for two different j values specified by the user (see Table 3.1). When a tuple indicates that a batch has completed but no new time-frames have closed, we just fetch the latest EMA for this symbol and pass it to the next operator. Otherwise, on entries indicating that new time-frame(s) have closed, we calculate first the newest EMA(s) and emit the newest results.

Q1 reporter. For the first query of the challenge we have to report the EMAs for all the lookup symbols in a batch. The Q1 reporter operator gathers all computed EMAs for a batch and reports the query result. The output of the EMA calculator is partitioned on the batch ID. The metadata included on each event indicating the total number of lookup symbols in a batch indicated when the Q1 reporter has gathered all the requires EMAs for that batch. When a lookup symbol emits multiple *safe-to-close* time-frames in a batch and therefore multiple results, an indicator points to the latest time-frame result for the reporter to expect.

Crossover calculator. The second query of the GC requires to identify breakout patterns that indicate the start of a trend in the development of a symbol's price. This process is based on the computed EMAs for a symbol over different intervals (i.e., EMA j paramater). The Crossover calculator operator consumes the output of the EMA calculator and discovers crossover events (breakout patterns) as described in the GC. The operator maintains the three most recent breakout events per symbol as required by query 2. Upon detecting a new crossover event, the operator updates its state and discards outdated state.

Q2 reporter. Similar to Q1 reporter, this operator gathers all crossover events for all the lookup symbols in the batch before it reports the query 2 results. The input stream of the operator is partitioned on the batch ID.

Both Q1 and Q2 reporters also act as sink operator, using a Kafka producer to publish the results to the corresponding Kafka topic.

3.3 Automation Script

Our code repository[4] ships with a configurable deployment and execution management script. The script makes the installation and deployment process of the dependent software components easy. A simple command is enough to install the necessary library dependencies and the software stack (Java runtime, Apache Flink and Apache Kafka).

```
$> ./manage.sh install
```

The management script can also be used to build the submitted software components (DIRM, Analytics application) from source with the following command:

```
$> ./manage.sh build
```

[4] `https://github.com/skalogerakis/DEBS_2022_GrandChallenge`

Finally, the execution of the whole analytics pipeline can be invoked using the management script:

```
$> ./manage.sh start
```

However, running the application also supports user defined configuration settings (Table 3.1) including different smoothing factors for the EMA calculation (parameters i and j in Table 3.1) and the reported queries (parameter q). Our analytics application also supports scalable deployments (parameter p). We also support operation reliability using the Flink checkpointing mechanism (using the checkpointing interval option c).

Parameter	Description	Default
p	Parallelism of Flink Application	1
i	Parameter to Calculate EMA	38
j	Parameter to Calculate EMA	100
c	Checkpointing interval (mins)	None
q	Specify the required queries for reporting. 1 for Q1, 2 for Q2	Both

Table 3.1: Configuration Options

3.4 Evaluation on Challenger Platform

In this section we provide a summary of our experience with how our code performs in the GC evaluation platform. The results of the following section are averages over at least 4 runs with negligible standard deviation. While there is a multitude of possible evaluation dimensions, here we showcase key aspects affecting performance of our implementation.

3.4.1 Effect of ingestion rate-control (throttling)

As analyzed in Section 3.2.1 (DIRM) we created a rate-control mechanism as a way to increase throughput via pre-fetching of batches from the gRPC service, and to effectively balance the tradeoff between latency and throughput. Tuning this mechanism demanded extensive evaluation of different parameters. Table 3.2 shows the impact of different degrees of throttling (number of batches the DIRM reads-ahead from the gRPC service) tested with 5GB of memory and 1 slot (i.e., parallelism is set to 1 for all Flink operators). We observe the tradeoff between latency and throughput in the results. One may choose throttling settings based on specific goals (such as rankings in this GC), and during our evaluation we chose 15 as this seemed to provide the best outcome versus competition. Throttle 10 may have been another good choice as it leads to significantly lower latency with a small impact on throughput. Based on our experience during evaluation trials

and a focus on throughput at the time, we have narrowly opted for throttle 15 in our code and use it to conduct the rest of our evaluation tests.

Throttle	Latency (ms)	Throughput (batches/sec)
5	287	27.1
10	328	38.3
15	484	38.8
20	602	39.0

Table 3.2: Varying degrees of throttle (1 slot, 5GB mem)

3.4.2 Effect of memory allocated to Flink

Choosing the most efficient memory to allocate in the Flink component (TaskManager setting) is a challenge when building new applications. Our choice of using 5GB memory in the experiments of Section 3.4.1 was made based on early experience. The choice is supported by the systematic evaluation shown in Table 3.3. Setting Flink memory at 5GB achieves the best performance in both latency and throughput compared to 4GB or 6GB (performance with 6GB is practically indistinguishable from that of 5GB). Note that had cost-effectiveness rather than sheer performance been the key criterion here, 4GB would have been a better choice as it leads to a better performance *per GB* ratio in both latency and throughput.

Mem (GB)	Latency (ms)	Throughput (batches/sec)
4	503	36.8
5	484	38.8
6	485	38.9

Table 3.3: Varying memory size (1 slot, throttle 15)

3.4.3 Effect of parallelism on single TaskManager

After experimenting with the throttling mechanism and different memory configurations, we also tested different parallelism options to obtain the best performance results possible. Our Flink setup currently operates in standalone mode with multiple task slots enabling parallelism of a Flink job within one machine.

For this set of experiments, we utilized the best configuration from the throttling section (throttling 15) and the most effective memory configuration (5GB of memory). Table 3.4 showcases results for different slot options.

# slots	Latency (ms)	Throughput (batches/sec)
1	484	38.8
2	404	47.3
3	401	46.2

Table 3.4: Varying parallelism (throttle 15, 5GB mem)

Our application performed best when we assigned two task slots to it. Increasing the number of slots beyond that does not yield any additional improvement. This is due to the fact that our source operator is serial, so scaling the rest of the application even more does not improve overall throughput/latency results.

Chapter 4

Parallel-source Tick Analysis Platform (P-TAP)

4.1 Introduction & S-TAP Restrictions

In chapter 3 [28], we presented **S-TAP** (Single-source Tick Analysis Platform), a system designed to address the DEBS'22 Grand Challenge [21], supported by Infront Financial Technology. S-TAP leverages the fact that event sources (Exchanges or Brokers) typically group ticks in a monotonically increasing sequence of *batches*. In the example of Fig. 4.1, event e_{t0}^{ABC} in batch B_i refers to a tick for *symbol*[1] ABC with event time $t0$. S-TAP features a specially-designed per-symbol window operator that takes into account all ticks falling within a time interval (T, T+ΔT] by processing batches sequentially, and ordering ticks within batches according to their timestamps (Fig. 4.2). S-TAP however assumes that batches are serialized at the query source and thus, a batch B_{i+1} that does not contain any ticks for symbol ABC can be identified by processing an ABC tick from B_{i+2} after an ABC tick from B_i. This amounts to a special form of *watermarking* [2] through which S-TAP fully accounts for ticks in its per-symbol windows. The single-source design choice however impacts scaling, preventing S-TAP from fully taking advantage of available parallelism (only part of the query is allowed to scale). A remaining challenge was thus how to ingest in parallel from multiple sources while maintaining batch ordering guarantees.

In this chapter, we introduce **P-TAP** (Parallel-source Tick Analysis Platform), a new solution, extending our initial S-TAP solution, that can handle parallel data ingestion and achieve the same in-order batch-processing guarantees as S-TAP. Figure 4.2 highlights the key challenge for P-TAP: Having multiple source (ingest) paths means that batches can progress out-of-order and thus a window for ABC that has seen ABC ticks from B_i and B_{i+2} cannot be certain that B_{i+1} does not have ABC ticks, as these may (or may not) still be on the way. In P-TAP, we utilized several features of the Apache Flink framework, such as side outputs,

[1] A *symbol* refers to the short name of any equity in financial pricing data (tick)

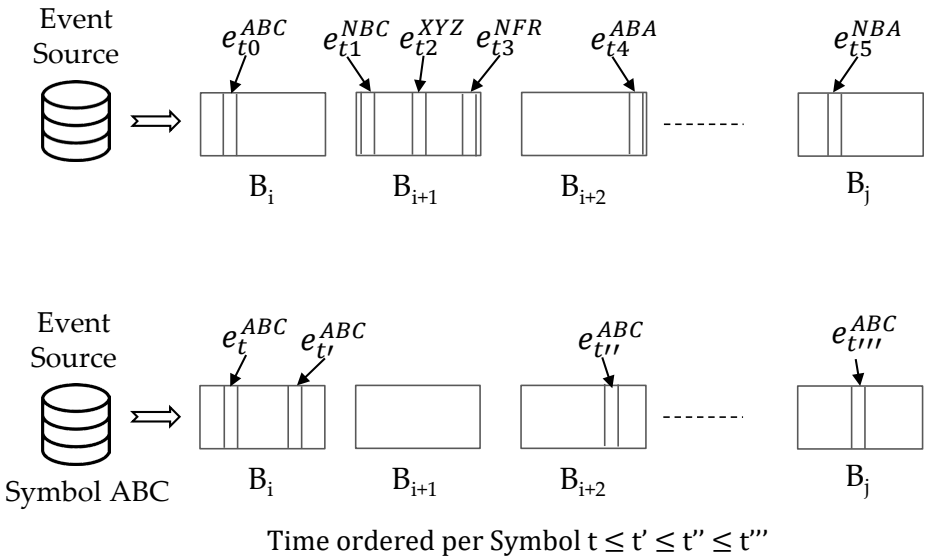

Figure 4.1: Events emitted by a Source (e.g., an Exchange) are grouped in *batches* B_i (upper figure); events for a given symbol ABC within batches are time ordered (lower figure)

timers, and the broadcast state pattern, to enrich the main event stream with batch-order and batch-content metadata, achieving in-order batch processing and thus full event accounting without the need for watermarks.

Our results show that P-TAP achieves better scalability compared to S-TAP. Whereas S-TAP achieved initial scaling up to 8 cores, it eventually levels off beyond that. P-TAP demonstrates increasingly higher throughput (as well as higher latency) for up to 32 cores over 4 servers, with a speedup of up to ∼1.50x versus S-TAP when using the default parallelism settings. Upon further investigation, we were able to fine-tune P-TAP to achieve an overall speedup of up to ∼1.78x vs. S-TAP for 32 cores, while also reducing latency. However, S-TAP is a more efficient solution than P-TAP for less than 16 cores, leading us to suggest a hybrid solution: use of S-TAP for less than 16 cores and P-TAP above that. Overall, our work demonstrates that stream-processing systems that can fully account for late events in their window processing and achieve in-order processing, are also able to scale.

The contributions of this work are as follows

- A fully parallelized solution (P-TAP) to the problem of discovering breakout patterns in financial tick data via parallel stream processing with in-order guarantees

- An evaluation of P-TAP on a 32-core 4-server cluster demonstrating its scalability over the sequential-ingest version (S-TAP) on the same cluster

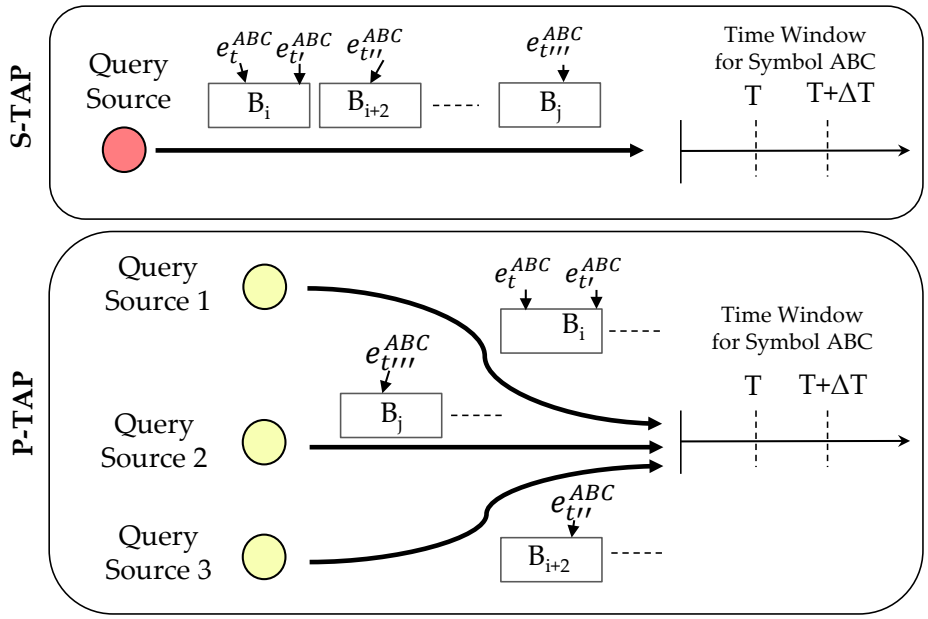

Figure 4.2: S-TAP easily detects batches with no events of symbol ABC by their absence in sequence, whereas P-TAP requires additional info as batches come from multiple paths

- An investigation of the benefits possible with finer tuning of parallelism vs. default settings in Flink

The remainder of this chapter proceeds as follows: In Section 4.2 we present the design and implementation of P-TAP, along with sufficient information on S-TAP (described in detail in Chapter 3) to help the reader understand our core application design, while in Section 4.3 we present our experimental evaluation.

4.2 Design and Implementation

The Parallel-source Tick Analysis Platform (P-TAP) data analysis pipeline comprises three major components (Fig. 4.3): the Data Ingestion Manager, a Flink job that prepares data for ingestion by the analytics application; Apache Kafka [30], used to decouple the data ingestion/reporting phase from data analysis; the stream analytics application (P-TAP) built on top of Flink [5]; and a fourth component used in our evaluation, the Result Validation Manager, a Java process used to validate results between different versions of the applications. The data ingestion phase is decoupled from the core data analysis (P-TAP) to ensure portability and interoperability with a variety of data source types. The Data Ingestion Manager can be extended to fetch data from different data sources (e.g. network sources or other types of files) without affecting the implementation of the rest of the

pipeline. Next, we describe the design and implementation of these components in detail.

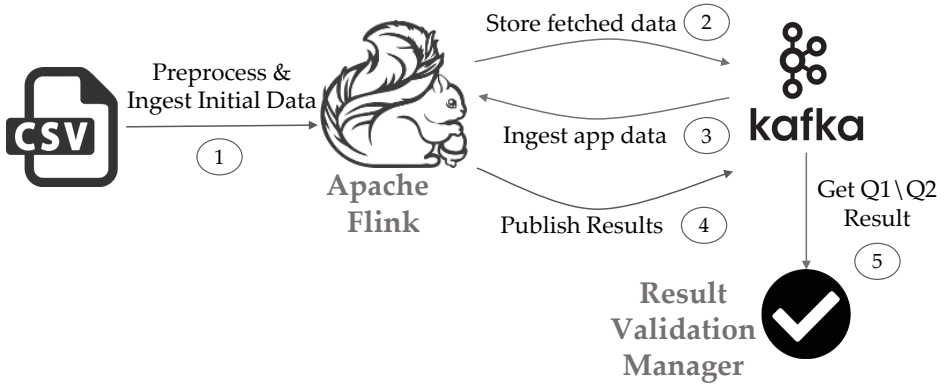

Figure 4.3: Data analysis pipeline

4.2.1 Data Ingestion Manager (DIM)

The Data Ingestion Manager (DIM) is designed to efficiently ingest data from CSV files and make it available to downstream applications in *batches*. DIM performs pre-filtering on the original data to remove all fields that are either irrelevant to application queries or empty, thereby reducing noise. The created batches are published in Kafka topics to which P-TAP subscribes. The topic is configured for multiple partitions to accommodate higher parallelism from the consumer (P-TAP) side. With S-TAP the topic should be configured for just a single partition to assure batch ordering.

Regarding event ordering, we assume that subsequent events of the same symbol (tick) have monotonically increasing timestamps (Fig. 4.1). Events of different symbols may not necessarily be timestamp-ordered, i.e., we do not assume global timestamp order for all events. Our original datasets indeed have this property: we validated this by running through them sequentially and verified that events of the same symbol are timestamp ordered, as anticipated. To ensure that our data feed (batch creation in DIM and insertion to Kafka) preserves this property, we set DIM's parallelism to 1 to ensure in-order batch *creation* (where each batch is identified by a monotonically increasing batch ID) and insertion to Kafka during the ingestion phase. Batch processing in batch-ID order (to satisfy event processing in timestamp order per symbol) is a key requirement affecting the design of P-TAP (Section 4.2.3).

Another important aspect of DIM is the choice of `lookupSymbols` subscriptions to insert into each batch (emulating the query interests and intensity of financial brokers). During the pre-processing of the raw dataset we create a symbol registry with all distinct symbols encountered in the dataset and for each new batch to be created, we pseudo-randomly select a new set of `lookupSymbols` from that

registry, to update batch subscriptions and thus simulate a real-time trader. The number of `lookupSymbols` to insert into each batch is user-defined. This approach provides flexibility and allows the system to adapt to different subscription and user requirements.

DIM proactively loads event batches to Kafka, ensuring that DIM and Kafka have sufficient memory and CPU capacity to ensure that the speed of downstream processing by TAP is never constrained by them. This design choice provides scalability and allows the system to handle large volumes of data efficiently.

4.2.2 Use of Kafka for asynchronous messaging

We rely on Flink's existing Kafka connector that is already well integrated with Flink (data source/sink, checkpointing) to serve the ingested data for subsequent analysis and to receive query results. DIM uses Kafka producers to publish data to a Kafka topic. Finally, we built our own custom (de)serializers to transfer data objects from and to Kafka topics in binary format.

4.2.3 P-TAP data processing

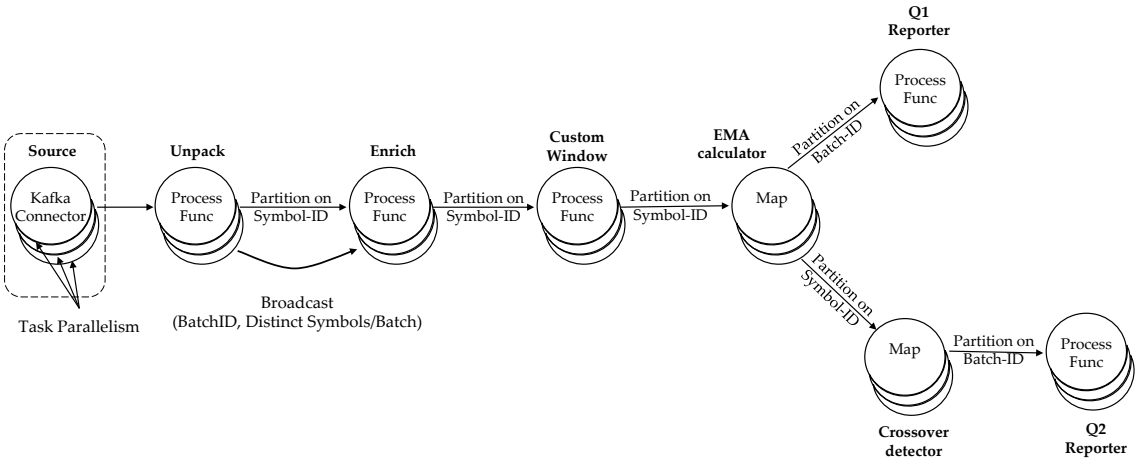

Figure 4.4: P-TAP stream-processing query (overlapping circles denote multiple tasks per operator)

The data analytics job comprises the stream-processing graph depicted in Figure 4.4. These operators a) consume the ingested data; b) unpack events included in batches; c) determine dependency between batches that the downstream operator expects in order to make progress; d) implement custom window logic to determine the last observed price per symbol in 5-minutes time-frames[2], guaranteeing there will not be dropped late events and window-closing will be correctly

[2]We use the term *time-frame* rather than *window* to refer to the different time intervals/ranges whose state may be simultaneously maintained by the window operator

associated with event batches; e) calculate the EMA; e) discover crossover events; and f) gather and report the query results on all lookup symbols per batch. Next we describe the design choices and implementation details of each operator.

Source operator. The source operator consumes data from Kafka by subscribing to the corresponding topic. We use the Flink-supported Kafka connector[3] as our data source operator. To fully exploit parallelism, the subscribed topic must also support multiple consumers. This is achieved by assigning multiple partitions to the topic during its creation. To ensure that all instances are active, the number of partitions in a topic should be at least equal to the number of parallel instances in the source operator.

Unpack operator. Each batch fetched consists of a list of events (trade actions for symbols) and a list of symbols of interest, *lookup symbols*, that a user subscribes to (i.e., queries results for that symbols). The *Unpack* operator is responsible for extracting and emitting events from a batch, while also injecting metadata into each output tuple (Listing 4.1). Such metadata include: (1) batch ID the event is extracted from; (2) a flag per event symbol that marks if it is included in the lookup list; (3) the number of lookup symbols in the batch; and (4) a flag that indicates if the event is the last occurrence of the symbol in the batch. The metadata are needed in the subsequent analysis on downstream operators.

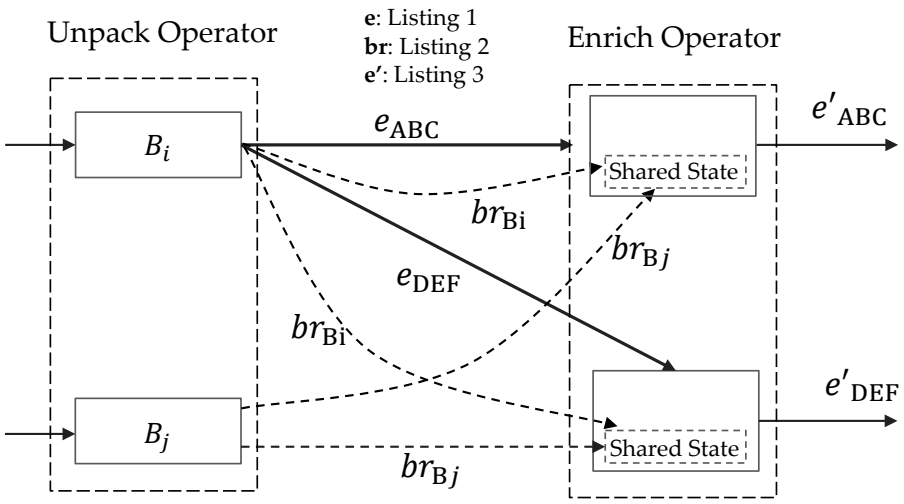

Figure 4.5: Tick events e are emitted on the main stream (solid lines) while metadata br are broadcast through the side stream (dashed line) to all tasks of the Enrich Operator

In order to meet the required ordering constraints, additional metadata is needed before the upcoming Enrich Operator (Figure 4.5). To accomplish this, we

[3]https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/kafka

use Flink's *Side Outputs*, which allow for the creation of multiple output streams of different schemas from a single data stream. In addition to the main data stream, which consists of the output tuples emitted from the previous analysis, we create a *side stream* to provide the necessary metadata. After unpacking each batch and emitting the output for the main stream (e.g., events e_{ABC} and e_{DEF}), we emit a tuple as a side stream that contains the batchID and a list of all the distinct symbols encountered in that batch (e.g., events br_{B_i} and br_{B_j}, see Listing 4.2). This side communication between the Unpack and Enrich operators is an instance of the **broadcast state pattern**, which enables the sharing of state among all parallel instances of a task. In this pattern, a single copy of the shared state is maintained on each task manager and is updated by broadcasts to all parallel instances of the task.

```
final case class EventEnrichmentSchema (
    symbol: String,
    batchID: Long,
    securityType: SecurityType,
    Price: Double,
    timestamp: Long,
    isSymbolsLastOccurence: Boolean,
    lookupSymbolBool: Boolean,
    lookupSize: Int )
```

Listing 4.1: Emitted output of unpack operator

```
final case class BroadcastedSchema (
    batchID: Long,
    distinctSymbols: List[String] )
```

Listing 4.2: Broadcasted side-stream schema

Enrich operator. We aim to enrich our main stream based on logic applied to the side stream, which contains the batch ID and a list of all the distinct symbols encountered in that batch (Listing 4.2). The goal is to create specially crafted tuples that will inform downstream operators about the batch IDs that each symbol depends on in order to make progress. To achieve this, we connect both the main and side streams in a single stream using broadcast state, which enables us to process incoming tuples from both streams. Flink distinguishes between processing incoming tuples from the main-stream with the *processElement* function and processing tuples from the broadcasted stream with the *processBroadcastElement* function. There is no cross-task communication regarding broadcasted state, so all tasks must modify the contents of the broadcast state in the same way to produce consistent results.

In our implementation, the *processBroadcastElement* function, responsible for the shared state across task managers, calculates two essential concepts: the *last-in-sequence* batch and the *dependency-batch-registry*. The *last-in-sequence* batch signifies the last consecutive batch ID received from the start, whereas the

dependency-batch-registry is a hashmap that maintains all symbols and the batch IDs on which each symbol depends to progress. Figure 4.6 illustrates how the *last-in-sequence* state evolves as a sequence of new batches arrive, along with the most recent state of the *dependency-batch-registry* when the final expected batch arrives. Initially, when the *processBroadcastElement* is triggered by batch B_5, all symbols encountered in this batch update the *dependency-batch-registry* (i.e. symbols ABC, CDF). However, we cannot update the value of *last-in-sequence* since there are no consecutive batches from the start. A similar action takes place when the sequence of batches B_3, B_1, and B_2 tuples follow. The *last-in-sequence* batch updates with value 3 when the batch B_0 arrives, as shown in Figure 4.6.

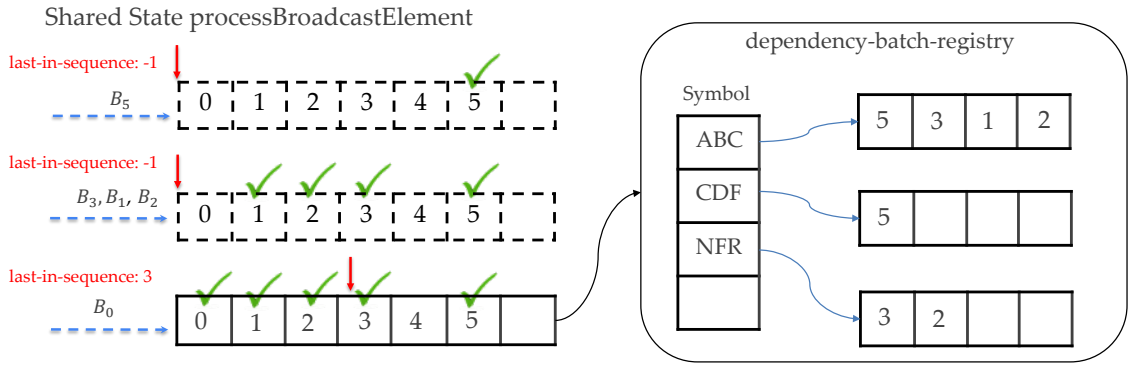

Figure 4.6: Shared State update of last-in-sequence and dependency-batch-registry after the sequence of batches B_5, B_3, B_1, B_2 and B_0 trigger the processBroadcastElement

The *processElement* function forwards the tuples from the unpack operator to downstream operators. Moreover, when the last occurrence of a symbol within a batch occurs, we verify if the *dependency-batch-registry* has any new dependency batches for the corresponding symbol. If this is the case, we emit a specially crafted tuple containing all the dependency batches from the registry along with the *last-in-sequence* batch (Listing 4.3). Once we emit the dependency batches for a symbol we purge its state from the *dependency-batch-registry* to avoid forwarding duplicates to downstream operators. In the example of Figure 4.6, once Batch B_0 arrives, the subsequent triggering tuple for the NFR symbol will emit batches B_3 and B_2 as dependency batches, and the *last-in-sequence* batch is B_3. This information signals downstream operators that for the NFR symbol, we have encountered all the batches until batch B_3 and must await the completion of batchIDs B_3 and B_2.

```
final case class EventUnpackSchema (
    EventEnrichmentSchema(....), //Listing 1
    dependencyBatches: Option[Vector[Long]])
```

Listing 4.3: Emitted output of enrich operator

An important aspect to consider when working with broadcast state is the ordering of events. Broadcasting guarantees that all events will eventually go to all downstream tasks, but elements may arrive in a different order for each task. Therefore, we maintain the maximum batch ID encountered for each symbol and trigger periodic timers to ensure that the broadcast state catches up to the main-stream. Flink's timers, which are associated with keys, schedule actions to occur in the future based on processing time. When a timer fires, it triggers a callback function *onTimer* that implements logic identical to the *processElement* function. Timers are triggered continuously until the *last-in-sequence* batch reported is greater than or equal to the maximum batch encountered, ensuring that all events are processed in the correct order.

Window operator. Following the enrich operator is a window operator that emits the last observed price per symbol in 5-minute time-frames. A major challenge is to handle out-of-order late events, i.e., events that arrive after a window has closed. These events are typically dropped and as a results this could result in correctness issues in the subsequent trend analysis. Flink's built-in window operators support *allowedLateness* option that can accept late events for the specified amount of time when a window closes. However, it is still challenging to predict an appropriate *allowedLateness* value; in the general case, it is not possible to achieve a guarantee that there will not be events that arrive later than the specified setting.

Our goal was to design an application that will not sacrifice correctness (i.e., not accounting for delayed out-of-order tuples) over performance. We thus decided to build our own custom window operator that closes a time-frame when, based on event semantics, it determines that there are no events left out that belong to the corresponding 5-minute time-frame. For each symbol our window operator maintains a table of 5-minutes time-frames. Upon the arrival of an event, the window operator has to decide in which time-frame the event is to be assigned according to the event time. Our mechanism performs event grouping and alignment using Equation 4.1:

$$f(event\_ts) = \lfloor (event\_ts/win\_interval) * win\_interval \rfloor \tag{4.1}$$

The input to Equation 4.1 is the timestamp of each processed event. The window interval is set to 5 minutes. The function returns the starting time of the 5-minute time-frame that the event belongs to. For example, event timestamps 14:00:00.001, 14:00:02.421, 14:00:04.343 all belong to the time-frame starting at 14:00:00.000.

For every 5-minute time-frame, the operator maintains the last-price seen for the symbol along with its timestamp. Our custom operator applies incremental processing logic, i.e., it updates the last-price seen of the symbol upon processing an incoming event by comparing if the new event's timestamp is later than the previous stored last-price. Thus we maintain minimal state per time-frame, avoiding buffering of all events within the same time-frame.

The operator also maintains a list of all batches seen and keeps track of the progress of processing each batch, namely whether the operator has processed all

events of a batch according to the metadata emitted by the upstream unpack operator. The operator also identifies the 5-minutes time-frames affected by each batch. Specifically, we *link* each batch with the **last time-frame** *affected by the batch*. To do this we use the timestamp of the last occurrence of the symbol within the batch (the metadata flag *isSymbolsLastOccurance* in Listing 4.1). In Figure 4.7, the last timestamp of symbol ABC in batch B_5 is 12:28, affecting *up to* time-frame 12:25-12:30.

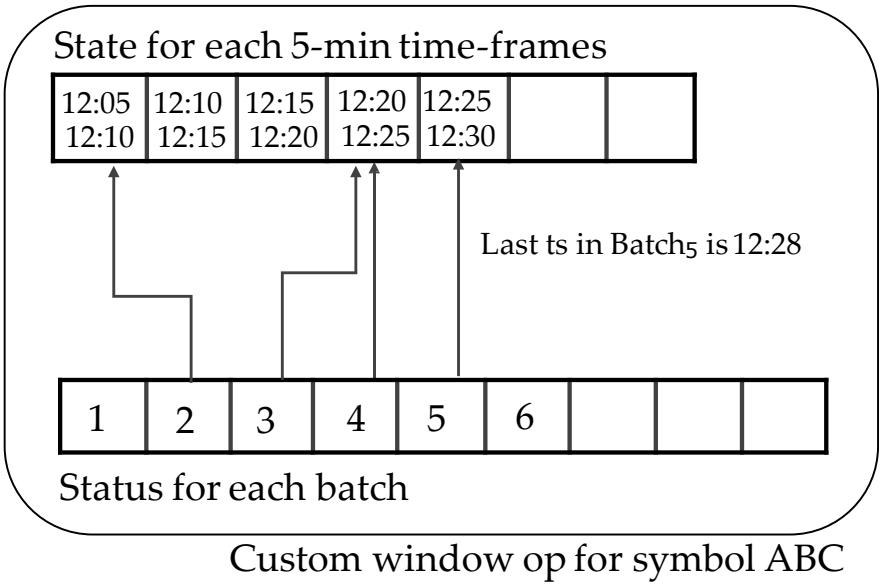

Figure 4.7: Custom window-operator state for symbol ABC. Each batch B_i, $i = 1, 2, \ldots$ points to the time-frame affected by the last occurrence (last ts) of symbol ABC in that batch

As events are aggregated from multiple sources we cannot assume events for different symbols are timestamp-ordered. However, we assume that events for the same symbol have monotonically increasing timestamps across batches, as analyzed in Section 4.2.1.

Thus, for a given symbol the timestamp of its first event in batch B_i is later than the last timestamp of that symbol in batch B_{i-1}. Based on these ordering properties, for a given symbol, the batch B_i cannot affect a time-frame preceding the time-frame linked to B_{i-1}. In the example of Fig. 4.7 for symbol ABC, B_4 cannot affect a time-frame before the one linked to B_3.

Despite the ordering guarantees for the same symbol across batches, due to the parallel nature of the source operator, identifying which batches contribute to each symbol is a non-trivial task. This challenge arises due to the possibility of *empty batches*, i.e., batches that do not contain events for a given symbol (e.g., batch B_1 in Fig. 4.7 does not contain events for ABC). The progress of the batches

is determined by the dependency batches emitted from the upstream operator, followed by the *last-in-sequence* batch ID. Using this information, we can identify a *safe-to-report* batch. A batch is deemed *safe-to-report* when all the dependency batches from the start until this batch, including this batch, have been fully processed (i.e., all their events have been processed). When new dependency batches arrive, we initialize them in the batch list in case it is their first encounter. We then iterate from the latest *safe-to-report* batchID until reaching the maximum *last-in-sequence* batchID and update the *safe-to-report* batchID value as required. We perform this check continuously as each batch finishes processing to allow the *safe-to-report* batches to report in a timely manner.

Every time the window operator updates the *safe-to-report* value, it also checks if there are *safe-to-close* time-frames to emit the symbol's last price observed in such time-frames. A time-frame is considered as safe-to-close when all batches linked to it and, *at least the first batch linked to the next time-frame*, are completely processed. In the example of Fig. 4.8, batches B_3 and B_4 are linked with time-frame 12:20-12:25. However, the time-frame is not considered as safe-to-close after the processing of these batches as we are not sure if the next batch contains events that contribute to it. As soon as we have processed the first batch of the next time-frame, i.e., batch B_5, it is certain that the subsequent batches cannot contain events for symbol ABC affecting time-frames preceding the one that B_5 is linked to.

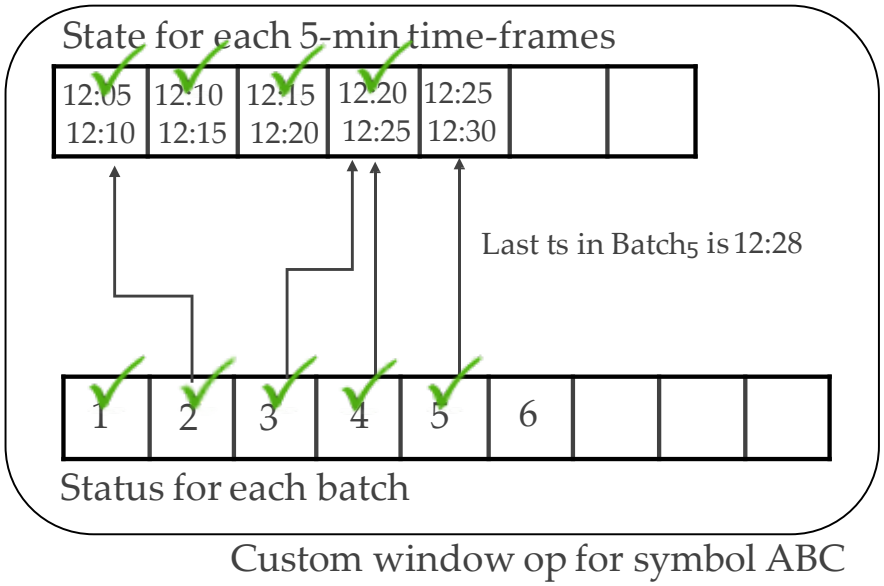

Figure 4.8: Window closing example: The checkmarks indicate that batches B_1-B_5 are considered safe-to-report and four 5-minute time-frames are safely considered fully closed

When a batch is *safe-to-report*, the window operator identifies the time-frame it is linked to and checks if there are pending time-frames that can now be marked as safe-to-close. The example in Fig. 4.8 illustrates the aforementioned scenario for symbol ABC: The last occurrence of ABC in batch B_2 has its timestamp within 12:05-12:10 (the time-frame is still not considered as safe-to-close when B_2 is safe-to-report). Batch B_3 is linked to the time-frame 12:20-12:25 (i.e., is the last occurrence of events regarding ABC fall into this time-frame). When B_3 is safe-to-report we can mark time-frames 12:05-12:10, 12:10-12:15 and 12:15-12:20 as safe-to-close. This is because all events for ABC in subsequent batches are expected to have timestamp later than the last occurrence of ABC in B_3.

The operator emits at its output the closing price of the symbol for each safe-to-close time-frame and purges its state. If there are no events for the symbol associated with a time-frame (e.g. time-frames 12:10-12:15, 12:15-12:20 for symbol ABC in Fig. 4.8), the operator ignores the time-frame and purges its state. However, if the symbol is in the lookup-symbols list, the operator emits a specially-crafted tuple to indicate to the downstream operators that there is no closing price for the 5-minute time-window and thus, they should report the previous EMA (see EMA calculator described next).

When B_4 is safe-to-report, there are no new safe-to-close time-frames. In this case the custom window operator emits a specially-crafted output tuple for the lookup symbols that indicate that the processing of the batch has completed. These specially-crafted tuples indicate to the downstream operator (the EMA calculator described below) that it can rely on the last computed EMA corresponding the last closed time-window. In the example of Fig. 4.8 assuming ABC is a lookup symbol, when batch B_4 is safe-to-report, the time-frame 12:20-12:25 is not safe-to-close, hence it emits a tuple signaling batch completion. That specific time-frame will be marked as safe-to-close only when B_5 is safe-to-report.

EMA calculator. The last observed price for a 5-minute time-frame emitted by the window operator is necessary for the EMA calculation. There is a separate instance of the EMA calculator per symbol, computing the EMA using Equation 4.2:

$$EMA^{j}_{w_i} = [Close_{w_i} * (\frac{2}{1+j})] + EMA^{j}_{w_{i-1}} * [1 - (\frac{2}{1+j})] \tag{4.2}$$

w_i: the 5-minute time-frame
j: the smoothing factor for EMA with $j \in \{38, 100\}$
$Close_{w_i}$: the last price observed within time-frame w_i

The operator computes EMA for a given time-frame for two different user-specified j values (Table 4.1). When a tuple indicates that a batch has completed but no new time-frames have closed, we just fetch the latest EMA for this symbol and pass it to the next operator. On entries indicating that new time-frame(s) have closed, we calculate the newest EMA(s) and emit the newer results.

Q1 reporter. For the first query we have to report the EMAs for all the lookup symbols in a batch. The Q1 reporter operator gathers all computed EMAs for a batch and reports the query result. The output of the EMA calculator is partitioned on the batch ID. The metadata included on each event indicating the total number of lookup symbols in a batch indicated when the Q1 reporter has gathered all the requires EMAs for that batch. When a lookup symbol emits multiple *safe-to-close* time-frames in a batch and therefore multiple results, an indicator points to the latest time-frame result for the reporter to expect.

Crossover calculator. The second query requires identifying breakout patterns that indicate the start of a trend in the development of a symbol's price. This process is based on the computed EMAs for a symbol over different intervals (i.e., EMA j paramater). The Crossover calculator operator consumes the output of the EMA calculator and discovers crossover events (breakout patterns) as described in the Introduction Section. The operator maintains the three most recent breakout events per symbol as required by query 2. Upon detecting a new crossover event, the operator updates its state and discards outdated state.

Q2 reporter. Similar to Q1 reporter, this operator gathers all crossover events for all the lookup symbols in the batch before it reports the query 2 results. The input stream of the operator is partitioned on the batch ID.

Both Q1 and Q2 reporters also act as sink operator, using a Kafka producer to publish the results to the corresponding Kafka topic.

4.2.4 Result Validation Manager (RVM)

The Result Validation Manager (RVM) is a Java process designed to verify that P-TAP results are identical to those of S-TAP, i.e., the parallel implementation does not affect the output of our processing logic. Due to the real-time nature of streaming applications, it is in general challenging to validate query outputs in a timely and efficient manner. In our case, we use the fact that TAP (S-TAP and P-TAP) publishes its query results for each batch; this enables us to directly compare the outputs of P-TAP to those of S-TAP for multiple executions. While this does not offer a formal proof, it helps us build confidence in the functional equivalence of the two implementations (and helped us solve bugs along the way).

RVM operates as follows: After ingesting data in batches via DIM and executing either version of TAP, the output of queries Q1 and Q2 are published on distinct topics. RVM then consumes those outputs for the desired query of both versions and directly compares their results. As both queries generate complex objects with multiple symbols per batch, RVM utilizes functionality similar to the *Unpack Operator* (Section 4.2.3) to unpack the results into distinct events. RVM then compares the results from the two distinct topics; finding identical results over multiple executions is a strong indicator that S-TAP and P-TAP are functionally identical, since both queries run on the same inputs. By default, RVM validates both Q1 and Q2 queries in separate threads. It can also be configured to validate either Q1 or Q1 separately.

4.3 Evaluation

We evaluate P-TAP through a series of experiments that test the impact of different parameters and its overall scalability compared to the sequential-ingest version (S-TAP). Our main experimental testbed consists of five servers, each equipped with a Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DDR4 2666MHz DIMMs, 256GB Intel D3-S4610 SSD and 2TB Ultrastar 7K2 HDD, running Ubuntu Linux 16.04.6 LTS, interconnected via a 10Gb/s Dell N4032 switch. One of the servers is dedicated to hosting DIM and Kafka.

The results presented here are the averages of at least 5 executions. Our main evaluation metric is *elapsed time*, the end-to-end time taken to process the ingested dataset in each case, measured from the timestamp before the first unpack operation to the timestamp after the last reporter operation. We also evaluate the impact of parallelism on *latency*, measured as the time between ingesting a batch of data (timestamp before the unpack operation on a batch) and producing a query response for that batch (timestamp at which the batch reports its results). We do not include the time to interact with Kafka in reporting Q1 and Q2 results, as we consider this an external step independent of the time to produce query results that are the core of our elapsed-time measurements.

P-TAP offers support for user-defined configuration settings (Table 4.1) including different smoothing factors for the EMA calculation (parameters **i** and **j** in Table 4.1) and the reported queries (parameter **q**). Our analytics application also facilitates scalable deployments (parameter **p**) and ensures operational reliability by using the Flink checkpointing mechanism (with the checkpointing interval option **c**). We attempted to evaluate our experiments using default parameters for c and q, which were the same as those used in the Grand Challenge. However, when experimenting with the RVM component (Section 4.2.4), we observed that these parameters resulted in very few responses from the Crossover Detector. Therefore, we selected a different set of smoothing factors, with i=0 and j=1, that led to the Crossover Detector producing more results, resulting in a more intensive workload.

Parameter	Description	Default
p	Parallelism of Flink Application	1
i	Parameter to Calculate EMA	38
j	Parameter to Calculate EMA	100
c	Checkpointing interval (mins)	None
q	Specify the required queries for reporting. 1 for Q1, 2 for Q2	Both

Table 4.1: Configuration Options

One out of our 5-node cluster servers hosts the JobManager of the Flink cluster and a Kafka Broker. The remaining four machines act as TaskManagers of the cluster, each one supporting 8 Task slots, resulting in a maximum of 32 parallel

instances. We built our application stack using Flink 1.14.3, Kafka 3.3.2, and Scala 2.12.

Sections 4.3.1, 4.3.2, 4.3.3 examine the impact of timer setting, batch size and number of lookup symbols on performance, respectively. In Section 4.3.4 we evaluate the benefits of increasing physical parallelism (1-32 cores) on the performance of S-TAP and P-TAP. In Section 4.3.5 we compare the performance of S-TAP to P-TAP with increasing dataset size, and finally, in Section 4.3.6 we evaluate the impact of symmetric parallelism (the default, where the parallelism of all operators is the same and set to the maximum physical parallelism) vs. asymmetric parallelism, where parallelism of each operator is tuned to their expected computational needs.

4.3.1 Impact of timer setting

P-TAP's use of periodic timers to ensure that the broadcast state catches up to the main-stream raise the question of what impact timer settings have on performance. An ideal timer setting should balance the tradeoff between frequency and overhead, as timers should be triggered frequently enough to take into account information as soon as it is available while avoiding excessive overhead.

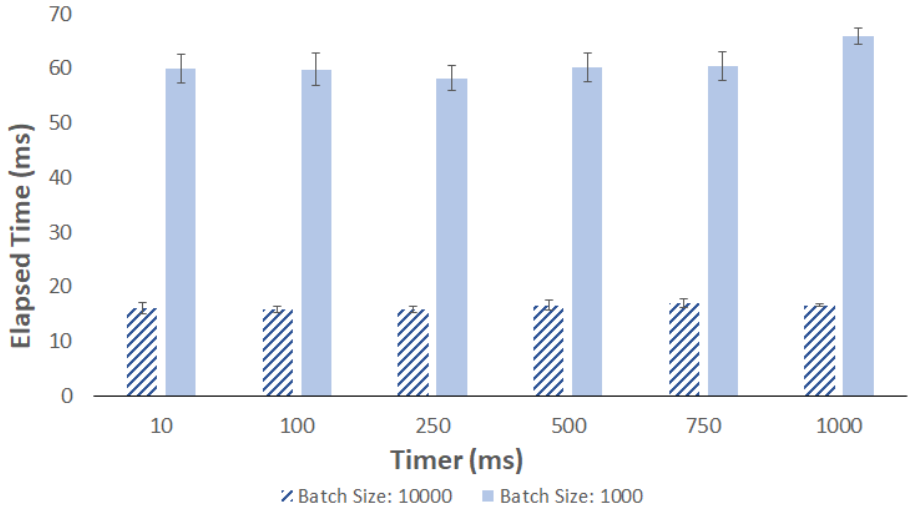

Figure 4.9: Impact of timer configuration (§4.3.1) for batch size 1000 and 10000

We evaluated the impact of different timer settings on the performance of P-TAP against the Day 1 Dataset (described in Table 2.2) with 1000 lookup symbols and 32 parallel instances. To assess the impact of lookup symbols against different batch sizes, we tested with batch-size 1000 and 10000.

Our results(Fig. 4.9) show that performance declines somewhat when the timers are set too high (period $\geq$ 500ms) or too low (period of 10ms). Setting the timer at 100ms or 250ms achieves the best performance. We eventually chose to set it to

250ms for the remaining experiments, as this marginally outperforms the 100ms setting.

4.3.2 Impact of batch size

The batch size is another important parameter as it affects the frequency of lookup symbol subscriptions, which simulates the actions of actual traders. Here we evaluate the impact of batch size on the performance of both P-TAP and S-TAP. We set the lookupSymbols and parellelism parameters to 100 and 32, respectively, and ran tests against the Day 1 Dataset for different batch sizes. Modifying the batch size accordingly affects the total number of batches processed. In Figure 4.10, we observe that for batch sizes above 1000, performance either levels off (S-TAP) or marginally improves (P-TAP). However, smaller batch sizes (e.g., 100 events) are challenging for both systems, as they entail more frequent lookup symbol subscriptions (impacting both S-TAP and P-TAP), and in the case of P-TAP an additional synchronization overhead as a higher number of batches results to more frequent use of broadcast communication. In subsequent experiments we fix the batch size to 10,000.

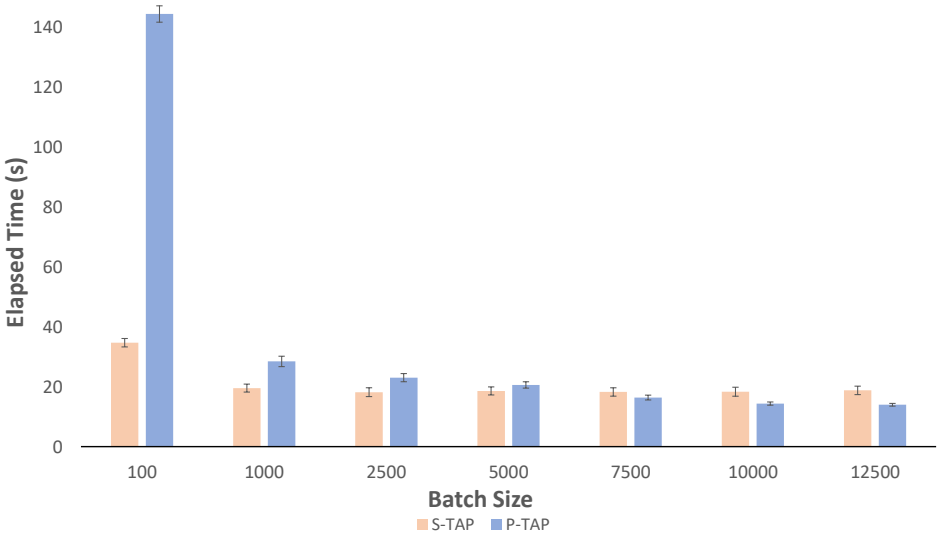

Figure 4.10: Impact of batch size (§4.3.2) with parallelism 32

4.3.3 Impact of number of lookup symbols

The number of lookup symbols is another important parameter impacting performance, as the higher the number of subscribed symbols per batch, the higher the amount of work expected in the course of TAP's continuous query processing. Here we use a batch size of 10,000 and parallelism of 32 and evaluate performance using the Day 1 Dataset.

Our results (Fig. 4.11) indicate that the number of lookup symbols has only a moderate impact on performance in either S-TAP or P-TAP. In both versions we observe that the performance is nearly unchanged (within the standard deviation) when the number of lookup symbols does not exceed 500. We do note a moderate increase in elapsed time when the number of lookup symbols exceeds 500, which is reasonable as the overhead from symbols that need to be reported in each batch increase significantly. Based on these results, we select a lookup-symbols value of 1000 in the remaining experiments to create a demanding, compute-intensive workload.

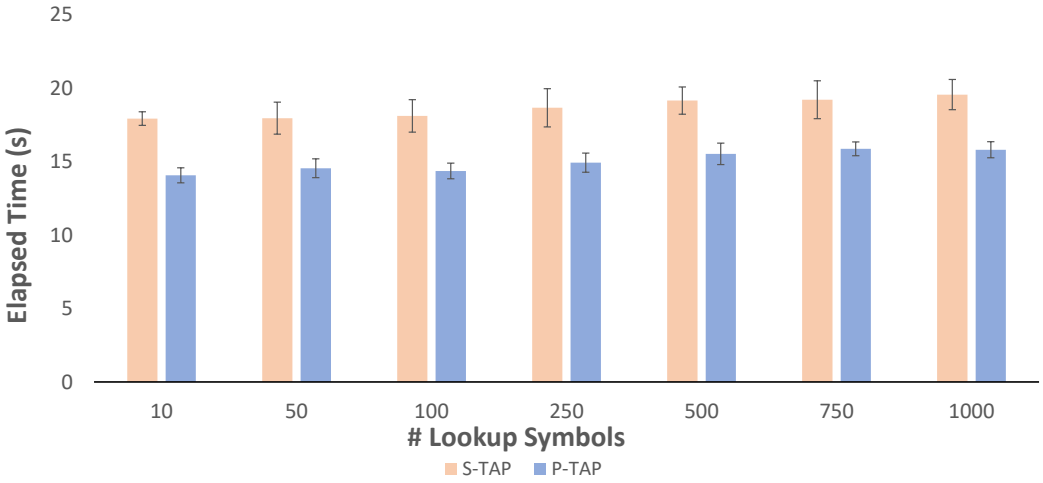

Figure 4.11: Impact of # of lookup symbols (§4.3.3) with parallelism 32

4.3.4 Scalability with increasing parallelism

Here we evaluate the behavior of S-TAP and P-TAP as we add parallel instances using the Flink `parallelism` parameter. We set other parameters of the application as follows: batch size set to 10,000 and lookup-symbols to 1,000. We used the Day 1 Dataset, which consists of one day of trading data, and incrementally add data for each trading day until we reach the Day 1-5 Dataset and evaluate the complete week of trading (Table 2.2 for Datasets description). In this way we evaluate the scalability of S-TAP and P-TAP as both the workload (dataset size) and parallelism increase. The scalability of S-TAP and P-TAP for the Day 1-5 Dataset with an increasing number of cores is depicted in Figure 4.12.

For S-TAP, we observe an improvement by a factor of ∼1.2x when scaling from 1 to 8 cores in all experiments, but no further speedup as we continue increasing to 16, 24, and 32 cores. This is due the source and unpack operators forming a performance choke point preventing further scalability for S-TAP.

For P-TAP, when scaling from 1 to 8 cores we observe a speedup ranging from ∼2.4x to ∼2.8x (Fig. 4.12). Scaling from 8 to 16 cores results in a speedup ranging

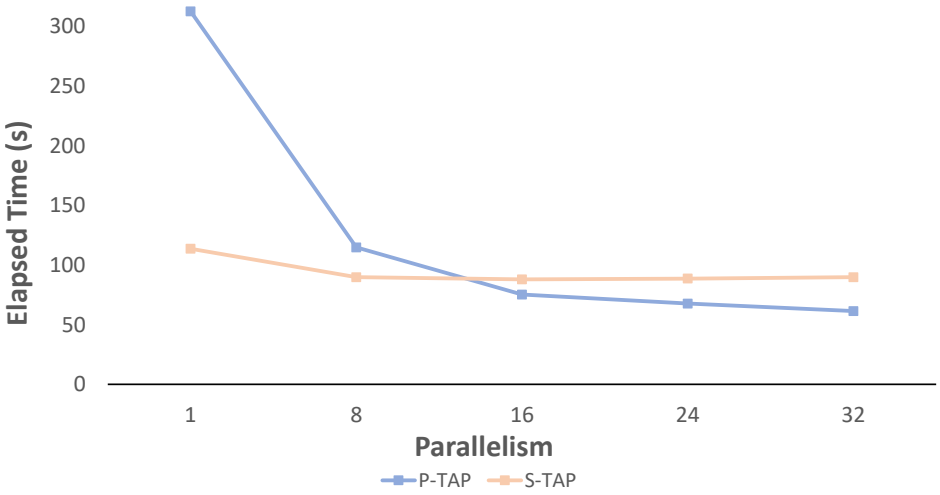

Figure 4.12: Scalability of P-TAP, S-TAP (§4.3.4), Day 1-5 Dataset

from ∼1.3x to ∼1.5x for larger datasets. Increasing parallelism to 24 and 32 cores led to a similar speedup of ∼1.1x in each case for all datasets.

Although P-TAP scales and eventually outperforms S-TAP (Fig. 4.12), at parallelism 1 and 8 it is ∼3x and ∼1.2x slower than S-TAP, respectively. This is due to the additional synchronization cost of P-TAP that makes S-TAP a more efficient design when not saturated at the source. We thus believe that a hybrid scheme where S-TAP is used at low levels of parallelism (less than 16), switching to P-TAP above that level will produce a system delivering all-around best results.

Contrary to the consistent improvement in *elapsed time*, the increase in parallelism leads to a decline in *latency*. Fig. 4.13 depicts *average latency* as well as the *90th percentile latency* derived from all the batches encountered in the corresponding dataset. For S-TAP, latency remains relatively low, from an average of ∼30ms for parallelism 1 to ∼250ms for parallelism 32. Conversely, P-TAP demonstrates more pronounced variations in latency and appears to be sensitive to changes in parallelism, as illustrated in Figure 4.13. For parallelism 1, average latency is ∼225ms (∼350ms at the 90th-percentile), increasing to ∼5.4sec (∼6.7sec at the 90th-percentile) for parallelism 32. As we show in Section 5.6, latency can be drastically improved through appropriate tuning.

We highlight the importance of aligning the number of Kafka partitions to the parallelism set for P-TAP. When the number of partitions in a Kafka topic exceeds the number of parallel source operators, source operators consume multiple partitions. This results in out-of-order batch ingestion that gets worse with more partitions, adding extra complexity and impacting performance, in both elapsed time and latency. We initially assigned 32 partitions for all parallelism parameters and observed significantly worse performance, particularly for experiments with lower parallelism, resulting in performance degradation of up to ∼5.7x for parallelism 1. Our results in this section use a number of partitions in the ingested

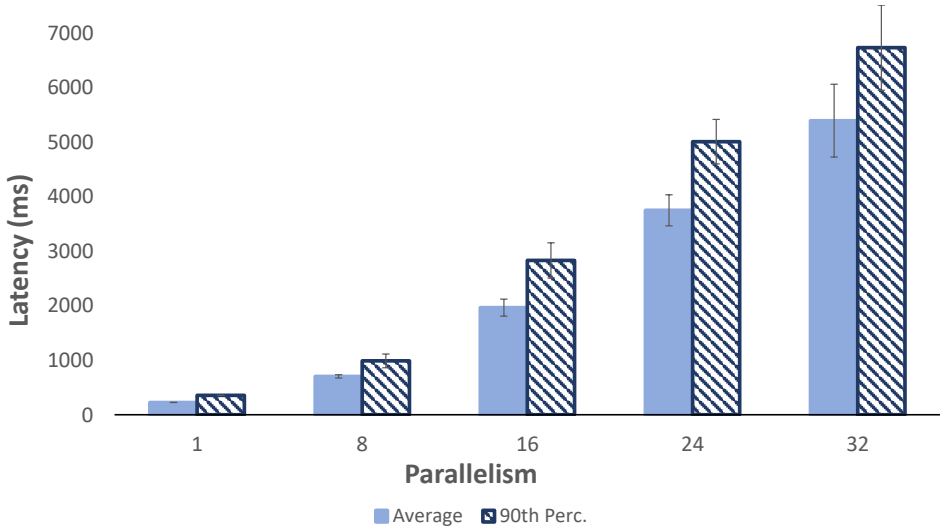

Figure 4.13: Latency of P-TAP vs Parallelism, Day 1-5 Dataset

topic equal to the parallelism parameter at the source, empirically seen to yield best results.

In summary, our experiments highlight the different the scaling behavior of S-TAP and P-TAP. S-TAP failed to scale beyond 8 cores, whereas P-TAP improved overall performance and demonstrated good scalability. However, S-TAP is a more performant solution when not saturated at the source (which happens at low levels of parallelism), not being set back by synchronization costs. We note P-TAP's latency increase when scaling, which may affect deployment decisions in a production environment. In Section 4.3.6 we further explore tuning P-TAP for even better performance.

4.3.5 Performance comparison of S-TAP to P-TAP

Here we compare the performance of S-TAP and P-TAP, and evaluate the speedup achieved by P-TAP as the workload (dataset size) increases with the maximum available level of parallelism (32 cores on 4 servers). We perform the same set of experiments as in the previous section (§4.3.4), with a batch size of 10,000 and 1,000 lookup symbol subscriptions. We start with the Day 1 Data set and gradually increase the dataset size by repeatedly adding another trading day until we reach the Day 1-5 Dataset. This enables us to observe the speedup of P-TAP over S-TAP as the workload increases.

Our results (Fig. 4.14) indicate that P-TAP achieves a processing speedup compared to S-TAP, ranging from approximately 1.2x to 1.5x for larger workloads, translating into an absolute performance difference of about 30 seconds when processing Days 1-5, clearly winning over S-TAP in terms of elapsed time. The latency increase with parallelism seen in the Day 1-5 Dataset (Fig. 4.13) is

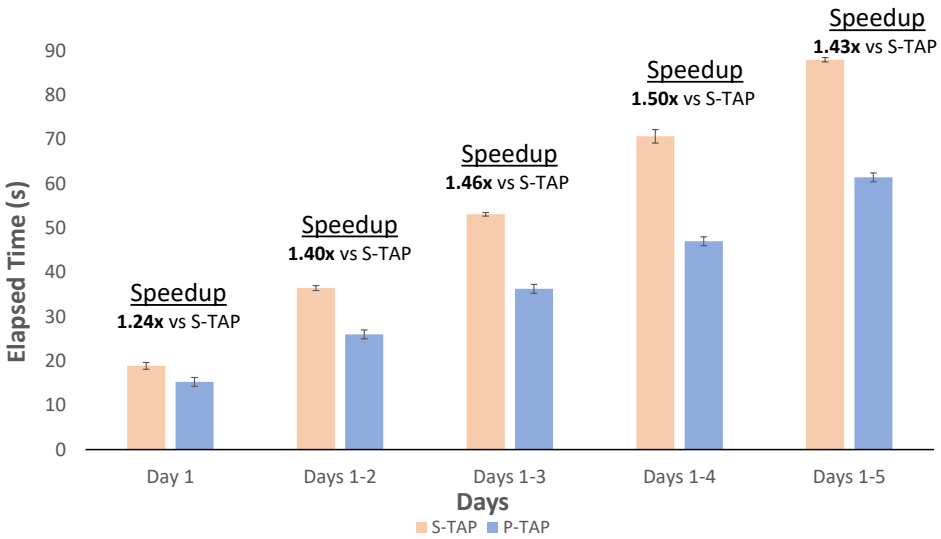

Figure 4.14: Performance comparison of S-TAP to P-TAP, for parallelism 32 (§4.3.5)

representative of latencies seen with other datasets as well.

Overall, these findings demonstrate the efficiency of P-TAP, which can handle larger workloads and achieve faster execution times especially when the workload increases.

4.3.6 Further tuning of P-TAP

Having established P-TAP's better scalability in terms of processing speed over S-TAP, we investigate the impact of tuning the parallel implementation to achieve even better results.

Parallelism of source operator. We initially set our sights on the throughput of the source operator. We isolated it from the rest of the pipeline and found that its throughput did not increase proportionally to increasing parallelism as initially anticipated. This bottleneck was caused by having a single Kafka broker, which could not scale indefinitely. We achieved the maximum throughput at parallelism 8 and did not see any further improvements for parallelism 16, 24, or 32 (Fig. 4.15). We attempted to address this by adding a second Kafka broker, which led to the throughput of the source operator eventually scaling for higher parallelism (up to 32 cores). However, when we executed the complete pipeline with the additional broker, we did not see any performance improvements due to the synchronization cost and computational demanding job as additional Kafka partitions increased the number of out-of-order batches entering the system, working counter-productively for performance. Therefore, we returned to the original configuration with a single Kafka broker, which however was efficient enough to overall drive our experiments

to saturation (all 32 cores on all 4 servers working at full speed, approaching 100% utilization).

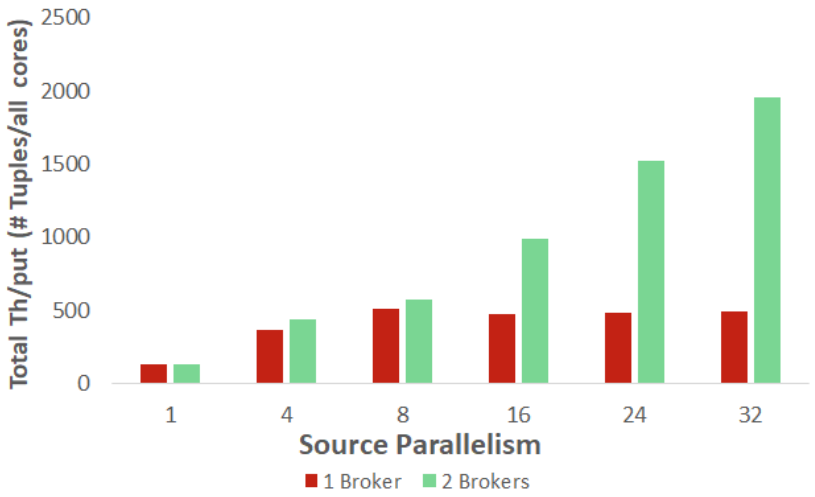

Figure 4.15: Total Throughput of the source operator with 1 and 2 kafka brokers

Leveraging SlotSharingGroups. With the bottleneck at the source operator investigated, we turned our attention to optimizing resource utilization for the rest of P-TAP. Flink uses Task Slots to define a fixed slice of resources for a Task Manager. Each subtask (parallel instance of an operator) requires a slot to be executed. However, not all operators are equally resource intensive, so Flink allows subtasks of different operators to be deployed into the same slot, which is controlled by the SlotSharingGroup. Tasks that share the same SlotSharingGroup can be executed in the same slot and share resources. By default, all operators are assigned to the same SlotSharingGroup.

In our previous experiments with parallelism 32 we used the default settings, with all operators assigned to the same SlotSharingGroup (parallelism 32). To better utilize resources, we experimented with three additional configurations of three SlotSharingGroups each with different degrees of parallelism, in all cases aiming to fully utilize all 32 available Task Slots, as shown in Table 4.2.

Configuration	**SSG1: Source & Unpack**	**SSG2: Q1 & Q2 Reporters**	**SSG3: Remaining Operators**
#1	8	2	22
#2	6	4	22
#3	4	4	24

Table 4.2: Parallelism of different operator groups in different configurations of three SlotSharingGroups (SSGs)

Comparing the results of the 3 configurations to the baseline (default) in Figure 4.16 we observe that Configuration 1 yields an additional ∼1.1x-1.2x speedup (vs. default P-TAP) for larger datasets, leading to an overall speedup of ∼1.73x on *elapsed time* when compared to S-TAP. In Configurations 2 and 3, we observe less speedup in elapsed time and in some cases slightly worse results that the default P-TAP. However, all configurations result to significantly lower latency (Fig. 4.17), with Configuration 3 achieving approximately 4.5x improved latency vs. default P-TAP. Our investigation highlights that optimizing resource utilization through careful selection of SlotSharingGroups can provide significant performance improvements for complex parallel stream-processing queries. Ultimately, the choice of configuration depends on striking a balance between reduced elapsed time and an acceptable latency level.

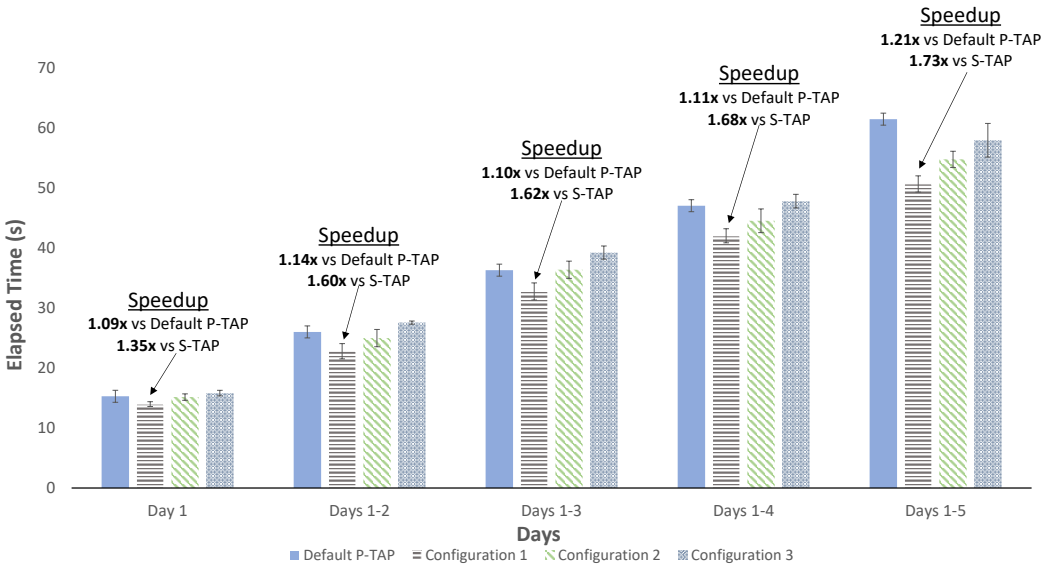

Figure 4.16: P-TAP with different SlotSharingGroups on 32 cores

4.4 Discussion

Transitioning from S-TAP (Chap. 3) to P-TAP (Chap. 4) proved to be a daunting task, with numerous challenges that needed to be addressed. In this section, we discuss the different attempts and insights that eventually led us to the successful solution of P-TAP.

As outlined in a previous section (Sec. 4.1), the primary challenge in transitioning from S-TAP to P-TAP was effectively handling the out-of-orderness caused by parallelizing the source operator of the pipeline. This out-of-orderness introduced a level of uncertainty to our windowing mechanism, making it difficult to determine which instances of symbols had completed and could report their results

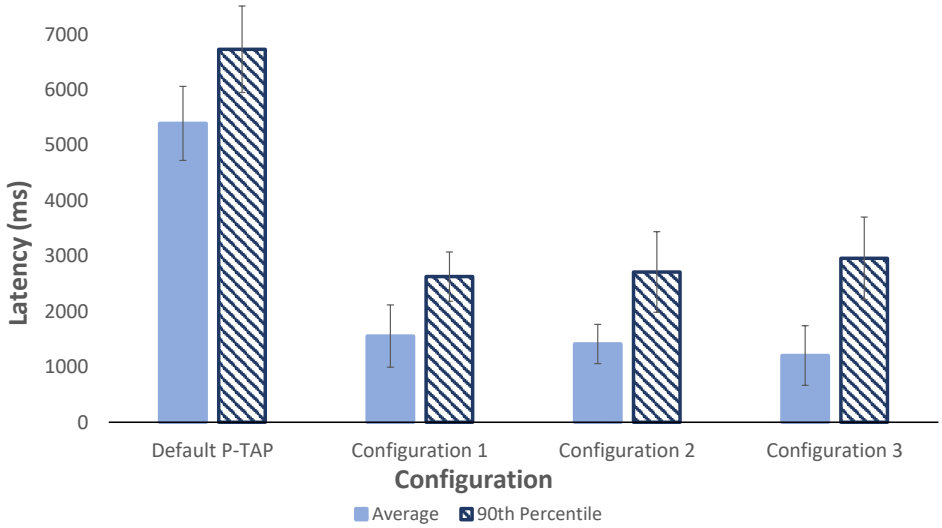

Figure 4.17: P-TAP Latency with different SlotSharingGroups and parallelism 32, Days 1-5 Dataset

while still maintaining batch ordering guarantees. We thus needed to find a way to determine whether we should wait for events from batches or if such events did not exist and we should not expect to account for them within the maintained windows of the operator.

Our first approach was to utilize an efficient set-membership test mechanism such as a Bloom Filter (BF) to let operators ask queries such as "is symbol ABC a member of the set of symbols appearing in batch B_i?". As background, BF[4] is a probabilistic data structure used to evaluate whether an element is member of a set. We implemented such a solution in conjunction with an external Redis Database for maintaining BF state.

Every instance of the Unpack operator created a new BF when processing a batch. The BF contained all the distinct symbols in that batch, and it was stored in Redis under the current batch Id. Careful consideration was given to minimizing the probability of false positives, which could lead to erroneous behavior. Every instance of our custom window operator retrieved the bloom filter for the missing batches and checked whether it contained the symbol processed by that instance of the window operator. If the BF did not contain the symbol, the operator could safely mark the batch as complete since we did not expect to arrive any event regarding that specific symbol extracted from that batch. Otherwise, the window operator marked the batch as pending and awaited to process the late events. In cases where the requested BF did not exist in Redis, indicating that it had not been created or stored yet, the batch was treated as missing and would be checked again in the future i.e., when a next batch is fully processed by the operator.

While this approach initially seemed promising, we encountered a bottleneck

[4] `https://en.wikipedia.org/wiki/Bloom_filter`

caused by the database used as part of the pipeline. The synchronous interaction performed by Flink when accessing the external database was the main contributing factor. Although Flink supports Async I/O [14], the constraints imposed by its API made it impossible for us to solve the problem at hand. Consequently, we quickly abandoned the idea of interacting with an external database as part of the pipeline.

We also eventually discarded the use of BFs to solve this problem. The primary issue with BFs was the possibility of false positives. In our solution, the progress of each symbol was blocked until all previous encounters of that symbol were processed. If a false positive occurred, it would wrongly indicate to a symbol that it should wait for an event that did not exist, causing not only the current batch to be blocked but also all subsequent batches in the future.

In reaching the current version of P-TAP, several optimizations significantly improved the system's performance. Firstly, we minimized the number of control tuples transmitted between operators. Although seemingly insignificant, when dealing with large datasets, excessive transmission of control tuples could cause severe bottlenecks. Working with broadcast state required careful manipulation, and the importance of purging unnecessary state cannot be understated in addressing performance limitations. Additionally, it is advisable to divide and conquer each pipeline as much as possible, as operators with complex logic are difficult to debug and can lead to degraded performance. Lastly, it is crucial to avoid unnecessary re-calculations of variables, loops, and other redundant computations.

Chapter 5

Rapid Recovery of SPSs

5.1 Introduction

This chapter focuses on ways to achieve fast recovery in stream-processing systems (SPSs). The core idea we explore is the alignment of recovery tasks with externally stored checkpoint state, to benefit from locality in accessing state. We build upon the state-of-the-art incremental distributed checkpointing capabilities of the Flink SPS (Section 2.2.5) and extend them towards that direction.

Currently, Flink's incremental checkpointing involves local checkpoints on the RocksDB instances of each node, followed by remote copies to a distributed Hadoop Distributed File System (HDFS) for reliability. However, a potential challenge arises during the recovery process if a Flink task manager crashes and its local state cannot be recovered (for example, if the hosting machine fails to boot up again). In such cases, the recovery task faces time-consuming remote transfers to retrieve the operator state, which significantly prolongs the recovery time compared to instances where the state is stored locally.

To address this issue, we build upon the notion of alignment between processing tasks and external state, an idea that is explored in recent work in a slightly different context [36]. Our objective is to ensure that recovery tasks are placed on the same node as an HDFS replica that stores the state required for the recovery task. By doing so, we expect to significantly enhance the efficiency of the recovery process compared to the original version of Flink, which relies on transferring state from (in general) remote replicas. It is important to note that the resulting architecture and system are designed to achieve cross-layer coordination between Flink and HDFS with minimal modifications to the underlying software. Although we currently utilize HDFS as the backend, the principles discussed here can be applied to other distributed storage backends as well.

In designing the recoverable SPS, we encounter two key challenges. Firstly, we need to control task recovery decisions and task placement within the Flink SPS. Secondly, we seek to extract information from and influence HDFS block placement on data nodes. Throughout the design process, we evaluate various replica and

block placement policies offered by HDFS. Our approach leverages the features of both Flink and HDFS to attain data locality and optimize the performance of state recovery procedures.

The subsequent section delves into the experimentation with baseline software systems to gain confidence in their capabilities and extract the locations of data replicas (Section 5.2). Section 5.2.1 explores the question "what is the performance difference of reading from a local vs. a remote file, when using the standard POSIX file API?". Following that, we will show that it is possible to steer the deployment of a recovery task in Flink (Section 5.3), and then experimentally investigate the question "what is the performance difference in Flink task recovery when performing local vs. remote checkpoint access?" in Section 5.3.1.

5.2 Replica Placement Awareness

In general, both replica and block placement policies aim to offer fault tolerance, high availability, and reliability while also maintaining (or improving if possible) performance. Knowledge of the actual data location has the potential to enhance processing performance by exploiting data locality and reducing network overhead. There are two approaches to achieving this; either instructing HDFS where to locate data blocks (or replicas) via a block-placement policy, or extracting the locations of specific blocks and harness that information in the processing application.

The first approach entails making modifications to the HDFS source code. Upon thorough examination of the source code, it became clear that a simple solution focusing solely on enforcing data placement locations could compromise other valuable features offered by HDFS. The existing placement policies encompass load balancing and geospatial considerations, intricately interconnected, and as such a new placement solution should take these factors into account when created from scratch. We thus decided that the creation of such a policy would exceed the scope of this thesis.

As a result, the second approach was chosen, as it offered a simpler implementation using the powerful Hadoop API. The goal was to identify the different locations where all blocks and replicas of a specific file reside, using the default block placement policy. HDFS provides built-in reporting capabilities that offer detailed information about files and their blocks. Specifically, the following command produces the desired information:

```
hdfs fsck file_path -files -blocks -locations
```

By supplying the desired file path and the above arguments, a detailed report for that file path can be obtained. Since the generated report contains a lot of extra information and noise, a Python script was developed to extract the necessary details (Fig. 5.1).

The user simply inputs the desired file path, and the script outputs the IP addresses where the replicas and blocks are located. In the example shown in

```
user@snf-884035:~/fileGen$ python3 blockTracer.py /user/skaloger/hdfs_sample_1.txt
Total Data Size =  256000000
Sorted by value size
IP: 83.212.109.203      SIZE:  128000000 [50.0 %]
IP: 83.212.101.207      SIZE:  128000000 [50.0 %]
```

Figure 5.1: Python HDFS block Tracer sample output

Fig. 5.1, the file "hdfs_sample_1.txt" had a size of 128MB, fitting into a single block, and a replication factor of 2, resulting in distribution across two machines.

5.2.1 Evaluation

This section simulates a simple execution scenario to examine the impact of data locality on performance. While the example may seem simple initially, it can be generalized to more complex real-world applications, as explained later. Additionally, for the purpose of this experiment, the Python script analyzed in the previous section was not utilized. Although the script provides a portable and user-friendly solution for more complex scenarios, a new implementation based on the same principles was developed to best fit the current experiment.

5.2.1.1 Experiment Description and setup

The experiment involves two applications: the **writer** and the **reader**. The writer application generates random files of specific sizes and writes them to a designated location in the HDFS cluster. The reader application is responsible for reading those files later on. Our main experimental testbed consists of four servers, hosted on Okeanos cloud service VMs each equipped with an 8-core CPU, 8GBs of RAM, running Ubuntu Linux 16.04.6 LTS. The HDFS cluster for the experiment consists of 1 NameNode and 3 DataNodes, with a replication factor of 2 for all files.

The **writer** application has a straightforward implementation and allows HDFS to determine the placement of data blocks and replicas based on its own policies. However, the **reader** application requires modifications to distinguish between local and non-local file reading. One of the optimizations in HDFS is the use of geospatial policies to minimize network overhead. Thus, if a replica of a block is requested locally on a machine, HDFS will prioritize the local replica regardless of the position of the other replicas. To examine this behavior, an indexing step is necessary to track the replica locations for each file. Non-local execution occurs when a file's replicas do not reside on the machine that requested the file. Given the small size of the cluster and the rarity of non-local cases, it is also important to count the total number of non-local cases to obtain fair and comparable results.

5.2.1.2 Results

A pool of files with equal sizes was created using the writer application for the reader function to consume. Each file has a size of *122.07MB* fitting into a single block (Fig. 5.2). Table 5.1 and Figure 5.3 present the collective *elapsed-time* results for reading different numbers of files, both locally and non-locally. *Elapsed Time* refers to the end-to-end time taken to process all the desired files. The first two columns also demonstrate that the current cluster is relatively small for the experiment at hand. In the first case, out of 10 files, only 2 had non-local replicas, and so on. Consequently, only 2 out of the remaining 8 local replicas were read and taken into consideration. Non-local sparsity posed a significant challenge during this experiment.

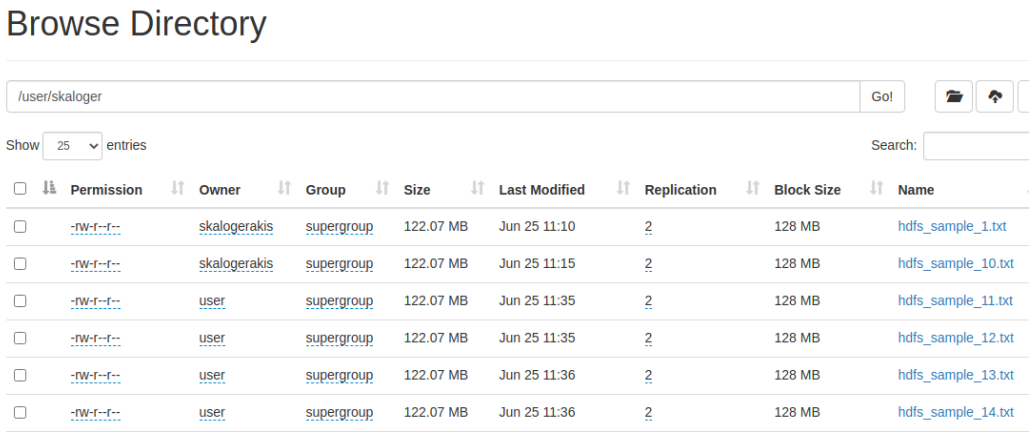

Figure 5.2: HDFS pool of equally large-sized data (122MB files)

Total Number of Files	Total Non-Local Files	Local (s)	Non-Local (s)	Speedup
10	**2**	6.671	6.931	x1.04
20	**6**	9.705	22.498	x2.31
30	**7**	11.598	25.176	x2.17
40	**8**	12.739	29.398	x2.31
50	**12**	15.681	41.938	x2.67
60	**15**	25.040	67.573	x2.70
70	**18**	26.700	68.799	x2.58

Table 5.1: Evaluation of elapsed time during local and non-local data reading in large files (122MB)

A pool of equally-sized small files, each sized at 1MB, was also created to assess the reading behavior in an experiment identical to the previous one (Tab. 5.2). As

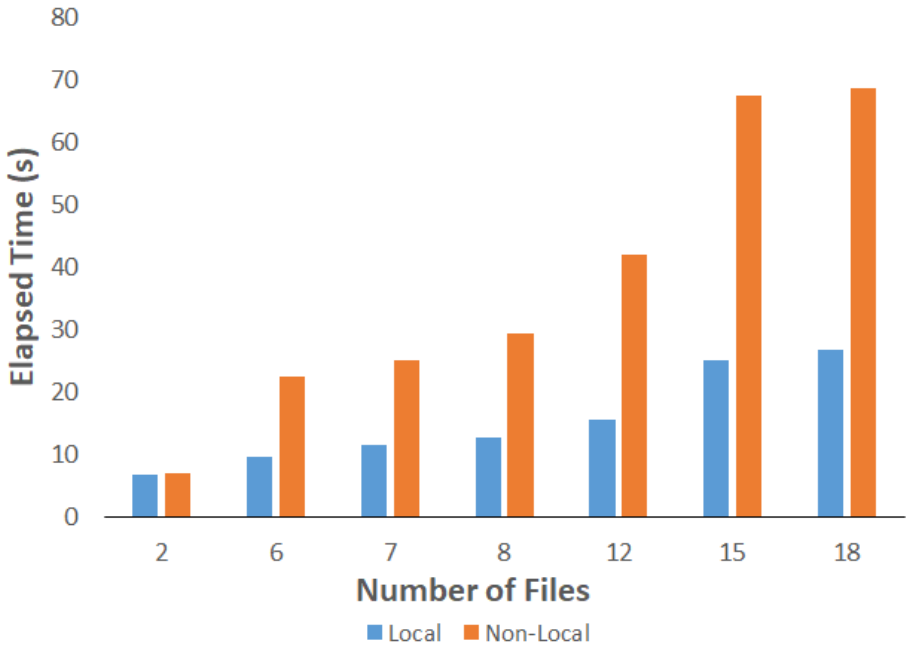

Figure 5.3: Evaluation of elapsed time during local and non-local data reading in large files (122MB), as shown in Tab. 5.1

expected, the network impact was negligible in this case and almost identical to the local case, so further investigation was not pursued.

Total Number of Files	**Total Non-Local Files**	**Local (s)**	**Non-Local (s)**	**Speedup**
50	**17**	1.321	1.373	x1.00

Table 5.2: Evaluation of elapsed time during local and non-local data reading in small files (1MB each)

5.2.1.3 Discussion

The experiments demonstrate the impact of data locality, with local reading achieving a speedup up to ∼2.7x compared to non-local reading. Even better results are anticipated in similar experiments with a higher number of files.

A question arises of how likely is for local access to happen as a probability of finding a replica of a particular file block on a specific machine, without any modification of HDFS replica placement policies. It is important to note that the experimental environment was not tuned or designed to achieve the best possible

results. A cluster consisting of only 3 DataNodes with a replication factor of 2 implies that approximately *66.6%* of the data would have a local replica. However, this is not representative of more typical, large cluster topologies, where there is a lower probability that the data needed resides on a local machine. The probability that a replica of the data exists on a local machine can be calculated using the equation:

$$\frac{RF}{ND} * 100 \tag{5.1}$$

where *RF* represents the *replication factor* and *ND* denotes the *number of DataNodes*. For example, with a 10-DataNode cluster and a replication factor of 2, the probability drops to 20%. In such cases, monitoring the location of each replica and utilizing directed recovery on specific nodes when possible becomes particularly important for enhancing overall performance. In the remainder we will thus focus on control of task placement during recovery actions as a way to achieve local recovery.

5.3 Control Task Recovery with Flink & HDFS

In this section, we further investigate rapid recovery by aligning recovery tasks with externally stored state. We recognize that combining frameworks like Flink with external persistent file systems such as HDFS or S3 is a common use case, as the former do not typically provide built-in data storage capabilities. By persisting state reliably, a recoverable stream processing system can effectively recover from unexpected failures. Thus, the combination of Flink with HDFS, the latter as a checkpointing back-end for Flink, is a good choice to accomplish our target.

An ideal implementation should be fully portable and involve minimal intrusion into existing technologies, apart from extracting and utilizing management information, preferably through standard APIs, from both systems. Figure 5.4 illustrates this core idea. Beyond the preliminary experimentation described in previous sections, exploiting data locality and avoiding network overheads in the general case is the main focus of our efforts.

In more practical terms, we emphasize on the state-of-the-art incremental distributed checkpointing capabilities of the Flink SPS (Section 2.2.5) as a common mechanism for maintaining state in a fault-tolerant manner. As demonstrated in our earlier experiments (Sec. 5.2.1), knowing where the state is stored can impact the performance of the recovery significantly by avoiding network overheads. For this to happen, however, it is a prerequisite that Flink and HDFS are co-located in the same machines (at least the Flink TaskManager and HDFS DataNodes, creating opportunities for local recovery as depicted in Figure 5.5).

In order to tackle this problem, we have identified two alternative approaches that we can pursue. The first approach involves influencing data placement policies on the HDFS side, ensuring that specific data blocks are placed on machines where

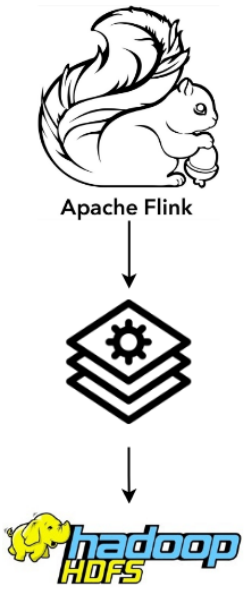

Figure 5.4: Core Idea Overview: Create a portable middle layer to provide cross-layer cooordination of Flink and HDFS during operator state recovery, with minimal changes to the Flink and HDFS systems

the information is needed (e.g., if the locations of recovery TaskManagers are known in advance). The second approach focuses on the Flink side, determining where recovery TMs are placed after a failure, given apriori knowledge about the locations of the data.

After evaluating the advantages and disadvantages of both directions, we have selected the second approach as the preferred option for initial exploration, on the grounds of feasibility and ease of implementation. This approach primarily focuses on modifying Flink to make informed decisions about recovery TaskManager placement while leveraging the powerful HDFS reporting capabilities that provide insights into replica placement status. As described in Section 5.2 on the replica-placement awareness system, the script we created to draw information regarding block locations of specific directories in the HDFS would be used in a continuous process to learn and maintain knowledge about replica locations in this solution. Specifically, the script produces a file with a list of all DataNode IPs that are involved in storing information of files within the specific directory given by the user (the checkpoint directory). The list of IPs is sorted in descending order of the total size required for that directory. One way to use this information is to derive the highest probability of achieving data locality (the more state stored in a machine the more likely it is that a piece of required data may be found in the same machine). However, this probabilistic approach alone may not provide a clear perspective on the comparison between local and non-local recovery.

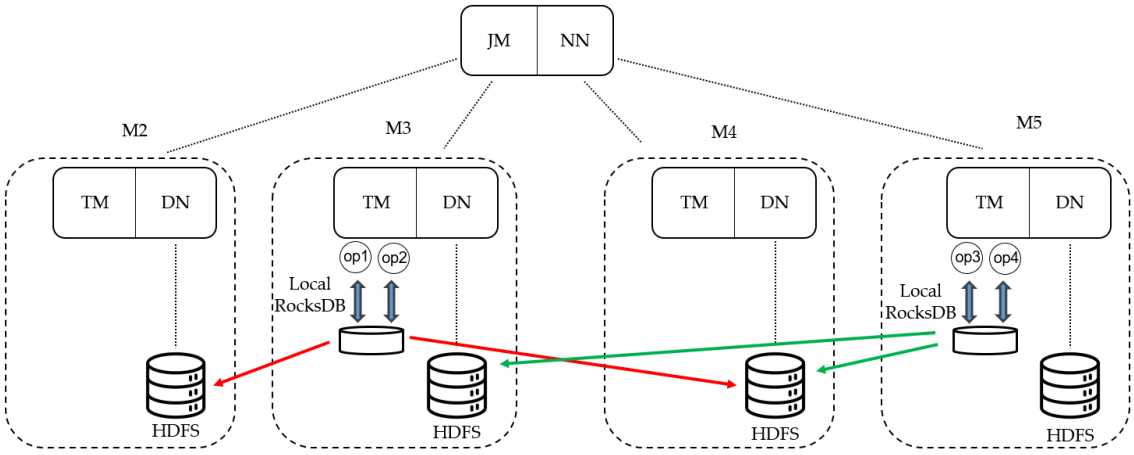

Figure 5.5: Cluster topology example: TaskManagers (TM) and DataNodes (DN) must be co-located on all machines. JobManager (JM) and NameNode (NN) be located on either the same or different machines. M3, M5 machines execute a Flink pipeline of multiple operators and produce local RocksDB state. Arrows point to the locations where checkpointed state persist its replicas

Therefore, we have designed experiments that clearly differentiate between the two methods by using a small cluster and appropriate replication factor in each case (see Section 5.3.1 for details).

Another crucial aspect of implementation is ensuring that Flink can effectively leverage replica-placement information. By default, Flink is TaskManager-agnostic, meaning it does not prioritize which TaskManager executes a given task as long as the resource requirements are met. To address this, we have decided to identify the IP addresses of each TaskManager and, in the event of a failure, restart the TaskManager on the machine that maximizes the probability of achieving data locality while meeting the resource requirements. Our modified version of Flink supports this informed decision-making process.

Let us consider the example illustrated in Figure 5.6. Assuming equal chunk sizes, the script will return the IP of the TM1/DN1 server whose total DN size will be higher in comparison to others. A higher total size indicates a greater probability of achieving data locality. In this simple example, 5 chunks with replication factor 2 are distributed among 3 TM/DNs, and as we see DN1 contains $\frac{4}{5}$ chunks in contrast with DN2, DN3 that contain $\frac{3}{5}$ of the chunks. In a more realistic scenario, the node topology can be much more complex with data getting distributed evenly in more machines, which may result in lower data-locality probabilities. Figure 5.7 depicts a flowchart outlining the steps involved in the probabilistic-centered recovery process.

When probabilities are balanced across the cluster, in which case there is no clear pointer on where to restart a failed task, we need to dig deeper into the

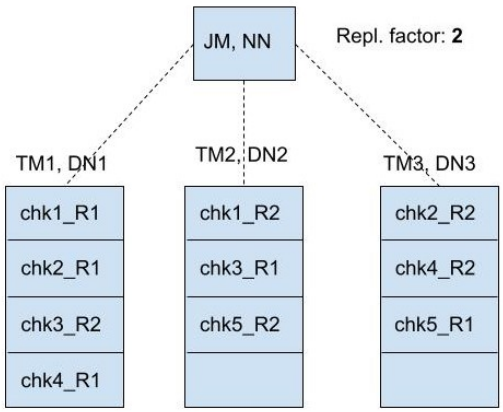

Figure 5.6: Execution example HDFS chunk distribution. In the example, the replication factor is 2 with 5 chunks of data. Naming the chunks follows the pattern chk(#Chunk_number)_(#Chunk_replica). The term chunks is used to describe HDFS data blocks that also represents how checkpoints are stored

replica-placement information and derive exact location information, namely which TM/DN server contains all or part of the required recovery state. Instead of just taking into consideration the complete checkpoint persisted in the HDFS, a more effective approach would be to isolate the information that is actually required for the recovery of each specific TM/DN server.

To provide a more detailed explanation of this approach, it is important to highlight a few additional technical details about how Flink stores and persists state. When utilizing RocksDB as state backend, each TM machine maintains its own local RocksDB instance and snapshots of this state are checkpointed to a durable store. Given a specific Flink job, multiple RocksDB folders are generated with each folder being a database itself. Naming those folders follows a specific pattern as shown in Figure 5.9. Persisted checkpoints also follow a specific directory structure with the following main directories: **shared** a directory for state that is possibly part of multiple checkpoints, **taskowned** is for state that must never be dropped by the JobManager and **chk-(chk_number)** which is for state that belongs to one checkpoint only [11].

Upon examining the mechanism through an example, we discovered that each of the different RocksDB files persisted differentiate their final name on the HDFS side by performing some kind of hashing, as shown in Figure 5.8. Furthermore, we observed that the file names on the HDFS side change from one checkpoint to another, which is expected due to the incremental nature of the checkpoints. Currently, work is in progress to determine the exact mapping between local RocksDB instances and the HDFS checkpointed files. One possible solution involves decrypting a file named **_metadata** found under the **chk-(chk_number)** directory

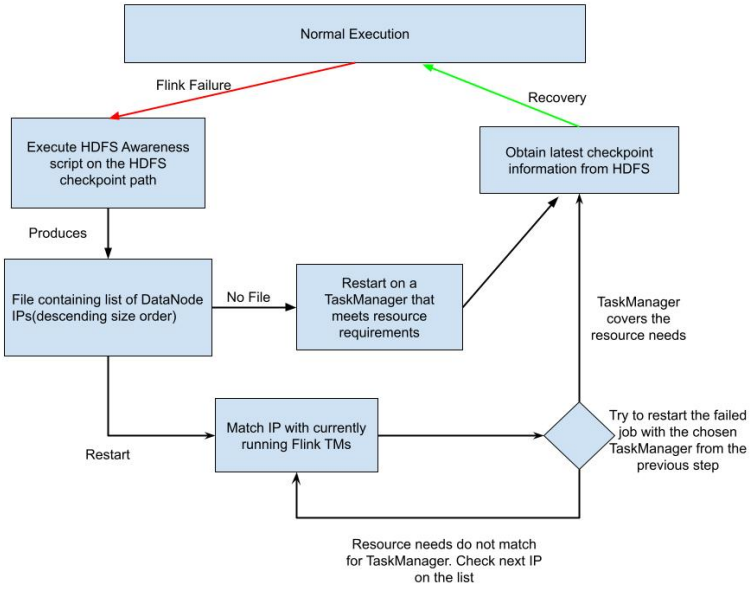

Figure 5.7: Probabilistic approach recovery execution flow of an application from the moment of failure until the recovery

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	user	supergroup	372.36 KB	Oct 21 15:45	1	128 MB	0012fedc-f69b-4b2a-b32b-2b7149c7042f
-rw-r--r--	user	supergroup	375.11 KB	Oct 21 15:45	1	128 MB	18686a9b-599e-42c5-805d-64f02227cc46
-rw-r--r--	user	supergroup	367.03 KB	Oct 21 15:45	1	128 MB	26aeb51b-50ee-406e-8ae9-1e848cf83317

Figure 5.8: HDFS persistent checkpointing name hashing

in HDFS. This file is updated on every checkpoint and contains metadata information and properties of all the RocksDB instances that are persisted as part of the checkpoint. However, some of the information in the file is currently unreadable and cannot be used in its current form. Flink's log files could also be a possible solution to the mapping problem. When restoring state from failure we can observe in the log files a form of mapping between hdfs files and sst table that RocksDB generates. Whether these sst tables are sufficient to reconstruct the entire RocksDB local instance during the recovery process or whether it requires extra information are questions left for future work.

After mapping each TM's Local RocksDB information to its corresponding HDFS files we can look up those HDFS file names required for the recovery of a specific TM using the replica awareness script. This forms the core idea of the Deterministic approach, and after this step, the rest of the recovery steps follow the probabilistic approach outlined in Figure 5.10.

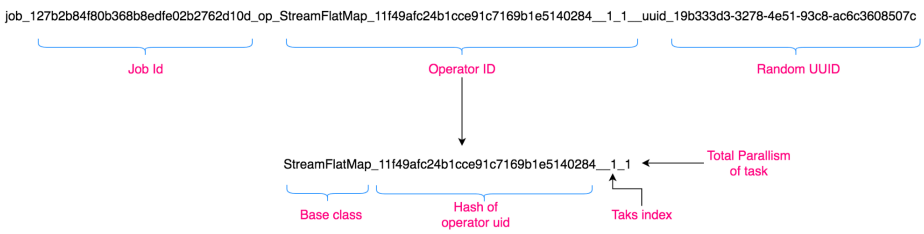

Figure 5.9: Local RocksDB checkpointing naming format

By examining the operator-state naming used in the TM, we can look up HDFS file names into the NN and translate to blocks, then to replicas and their locations. Using the results of this lookup we believe that we can identify the nodes (DNs) that hold replicas of the state involved in the recovery of a TN. We then use the IP to restart the TN on that machine.

To achieve optimal locality results using the deterministic approach, an additional step is required: creating a custom HDFS affinity policy based on the example shown in Fig. 5.5. This policy is crucial for ensuring that all checkpointed state originating from a specific Local RocksDB TM is persisted on the same machines. The same principle applies to the replicas as well. This becomes especially important in complex scenarios where multiple state operators may exist in the local directories of RocksDB. As discussed earlier, due to the generation of multiple files during incremental checkpoints, the default HDFS placement policy can result in the placement of checkpoints from the same local RocksDB on different machines, thereby leading to non-local recovery cases.

5.3.1 Evaluation

This section presents a simulation of a simple execution scenario designed to evaluate the performance impact of data locality during state recovery in streaming applications. For this evaluation, we utilize Apache Flink as the processing engine, RocksDB as the state backend and HDFS for providing a persistent storage solution. While the application used for the evaluation is relatively simplistic, focusing on the recovery aspect allows us to isolate it from the complexities of the application itself. It is worth noting that the findings and techniques presented here can be applied to more complex stateful applications as well.

5.3.1.1 Experiment description and setup

The experiment focuses on a variation of a typical WordCount application, which counts the number of words in a given input. In this case, the application is designed to generate 128-bit UUIDs and count the instances of each UUID while maintaining the timestamp of its last occurrence. Both ValueState and MapState

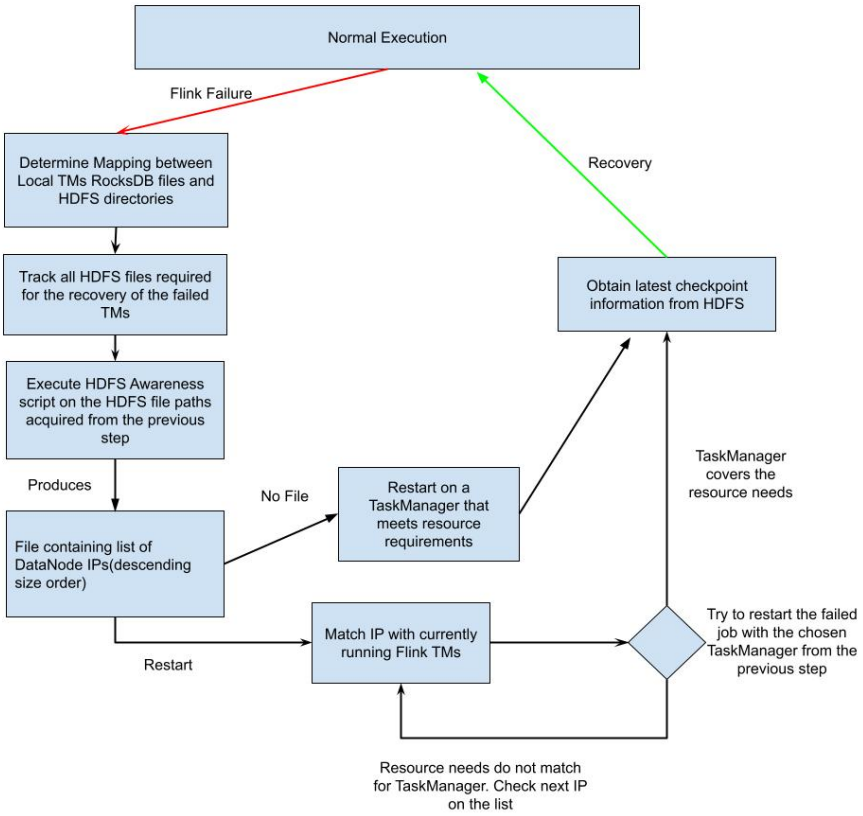

Figure 5.10: Deterministic approach recovery execution flow of an application from the moment of failure until the recovery

state primitives are utilized to generate as much state as possible, resulting in large checkpoints. The checkpoint interval is set to 15 seconds.

While streaming applications are typically unbounded, which can pose challenges for evaluation purposes, the source operator in this experiment is specifically crafted to ensure fair results. The input is generated from a pseudorandom UUID Synthetic Source Generator, and the size of the input (in MBs) is determined by the user. This approach allows for a fair assessment of the performance impact during recovery.

The results presented in this section are the averages of at least 5 executions. Our primary evaluation metric, referred to as the *recovery time*, focuses on the duration it takes for Flink to retrieve the checkpointed state from persistent storage (in our case, HDFS) and reconstruct the local RocksDB instances to resume processing. It is important to highlight that our metric differs significantly from the

built-in Flink metric *recoveryTime*, which measures the time it takes the JobManager to establish new connection with a TaskManager upon after a failure occurs. To evaluate our *recovery time* metric, we subtract the connection timestamp of the recovered TaskManager from the timestamp when the local instance of RocksDB completes the reconstruction of the checkpointed state. To maintain accuracy and clarity, we have excluded the time it took for the JobManager to track the failure and establish a new connection with a TM. Including this time would introduce noise to our measurements due to the internal handling of failures within Flink.

The experiments follows this timeline: First, the application is executed normally in Apache Flink. The progress of the application is monitored through Flink Metrics in the WebUI. Since source generates input of specific size, it is easy to determine when the processing is complete. After the initial execution of each experiment, the number of records for each file size is also tracked (Tab. 5.3), providing previously unknown information. Once all the records are processed, the TaskManager executing the job is intentionally terminated. This triggers Flink's recovery mechanism, and the job is restarted on an available Task Manager.

Two different experiments are conducted to evaluate the performance of local recovery compared to non-local recovery. Our main experimental testbed consists of five servers, each equipped with a Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DDR4 2666MHz DIMMs, 256GB Intel D3-S4610 SSD and 2TB Ultrastar 7K2 HDD, running Ubuntu Linux 16.04.6 LTS, interconnected via a 10Gb/s Dell N4032 switch. In both experiments, the JobManager of Flink and the NameNode of HDFS are colocated on the same machine.

In the local recovery case, four TaskManagers and four DataNodes coexist on a separate servers from the JobManager and NameNode, with a replication factor set to 4. This configuration ensures that a replica of the data exists on each machine and can be found locally during recovery.

In the non-local recovery case, we utilize two TaskManagers and two DataNodes all of which reside on different servers, in addition to the JobManager and NameNode. The replication factor is set to 1, and as the existing DataNodes are detached from the TaskManagers, no local replicas can exist. This scenario enables the evaluation of recovery performance without the benefit of data locality.

5.3.1.2 Results

Table 5.3 and Figure 5.11 showcase the *recovery time* of local compared to non-local state for input of different sizes. The first column of Table 5.3 shows the file size of the processed files in MBs while the next two columns depict the recovery time for state recovery in seconds.

In addition to *recovery time*, the number of records is also a relevant metric. Table 5.3 provides the correspondence between file size and the number of records.

File size (MBs)	Number of Records Sent	Local recov-eryTime AVG (s)	Non-local recovery-Time AVG(s)	Speedup
1	29k	2.0296	2.2454	x1.11
250	7M	2.5724	2.9022	x1.13
500	14M	2.857	3.572	x1.25
750	21M	3.3196	4.0604	x1.22
1000	29M	3.8112	4.964	x1.30
1250	36M	4.9868	6.1786	x1.24
1500	43M	6.0304	7.6368	x1.27
1750	50M	7.0768	8.9166	x1.26
2000	58M	7.862	10.4488	x1.33

Table 5.3: Results of elapsed time during local and remote state recovery

5.3.1.3 Discussion

The results presented in the previous section align with expectations. In general, when data locality exists, faster execution times are expected since processing and data co-locate on the same machine, eliminating the need for additional network IOs.

In the first two cases, we observe a speedup of ∼1.1x between local and non-local execution. This can be attributed to two reasons. Firstly, as mentioned earlier, Flink maintains a minimal checkpoint state for recovery, resulting in a smaller file size required for recovery compared to the processed state. Secondly, the fast network bandwidth between the servers minimizes the impact of transferring data, making the difference between remote and local execution negligible especially when recovering a smaller amount of state.

In the remaining cases, where the state for recovery increases we observe speedup ranging from ∼1.2x to ∼1.3x. While the results demonstrate the performance gain achieved through data locality during recovery, we do not observe linear speedup with file size. The comparison is expected to be much clearer when recovering much larger state, which would require a more complex application or larger input state. Nevertheless, even in this simplified scenario, the performance improvement gained from leveraging data locality during recovery is evident.

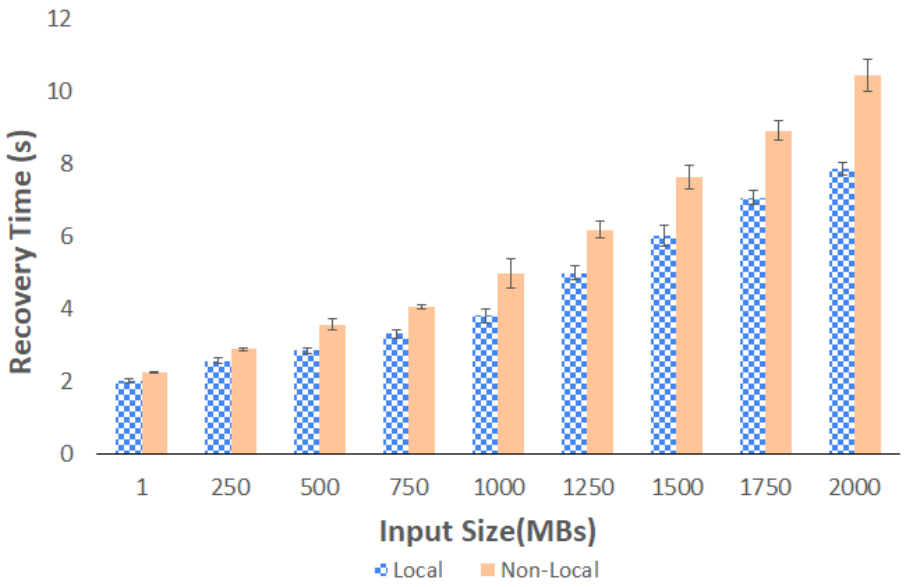

Figure 5.11: Result representation of *recovery time* during local and remote state recovery

Chapter 6

Related Work

Stream data processing is an important technology posing unique challenges and opportunities. Röger and Mayer's survey [38] highlights the challenges of processing high-velocity data streams, achieving low-latency processing for real-time analytics, and the difficulties of scaling stream processing systems to handle large volumes of data. They provide an extensive review of methods to parallelize data-stream processing queries, including techniques such as data partitioning, pipeline parallelism, and load balancing. These techniques distribute the workload across multiple processing nodes, allowing stream processing systems to scale out to handle larger data volumes and provide low-latency processing.

6.1 Achieving Completeness: In-Order Stream Processing

Handling out-of-order stream data in window processing is an important consideration in stream-processing systems. Watermarks are a key mechanism for addressing this challenge. Akidau et al. [2] provide a comprehensive analysis of different watermarking implementations in two widely-used systems, Apache Flink and Google Cloud Dataflow. The authors analyze the tradeoffs between cost and complexity of different watermarking approaches, as well as their effectiveness in ensuring temporal completeness. In this thesis, we contribute an application-specific watermarking strategy ensuring completeness in the processing of financial tick data.

In Truviso [31], the authors present the model of order-independent systems, which specifically addresses the issue of handling late-arriving tuples after a window closes. Truviso produces partial results when the system receives tuples older than the latest emitted punctuation (progress indicator tuple), leading to correctness on an eventually consistent basis. These late results are used to consolidate previously emitted results of closed windows in a lazy fashion, either for live continuous queries or on-demand when queried. Similar to P-TAP, Truviso is designed to handle streams from multiple data sources that are internally in-order and

out-of-order with respect to each other. However, in contrast to P-TAP, Truviso focuses more on data stream processing with partial results, targeting a different application domain.

Schneider et al. [40] proposed an approach for auto-parallelizing stateful distributed streaming applications that addresses the challenge of ensuring in-order processing of events. To detect missing and out-of-order events, the authors used periodic pulses and sequence numbers, respectively. Periodic pulses were sent on parallel instances with the same sequence number and merged with tuples to guarantee that at least one piece of information necessary for stream progression happens per epoch. P-TAP employs similar concepts at a high level, but is implemented at application-level over unmodified Apache Flink. Rather than assigning sequence numbers to each tuple, we leverage their grouping into batches along with metadata to reduce the number of required broadcasting messages. Additionally, our periodic timers are triggered lazily per keyed state until each key catches up with the latest *safe-to-report* batch ID.

Tangwongsan et al. [43] introduced FiBA, an aggregation algorithm specifically designed for sliding window aggregation that efficiently handles out-of-order tuples. FiBA leverages B-trees with finger searching and position-aware partial aggregates, enabling it to match a theoretical lower bound. While our problem requires the use of tumbling windows, our windowing mechanism could be extended to support sliding windows by incorporating additional metadata. FiBA focuses on developing an optimal sliding window aggregation mechanism without restrictions on the degree of disorder, and it accomplishes this without relying on watermarking. However, it delegates decisions such as closing windows and purging state to the corresponding framework employed. As such, the FiBA algorithm can complement our windowing mechanism.

6.2 Processing of Financial Tick Data

In the context of DEBS 2022 GC, several solutions have been proposed to address the problem of detecting breakout events using tick data [6] [32] [37] [3] [34] [47]. Many of these solutions leverage open-source frameworks such as Apache Flink and Apache Spark, which are specifically designed to handle large volumes of data efficiently. While some approaches attempted to develop custom solutions [6] [34], designed from scratch to maximize performance, they often lack essential features such as fault-tolerance that are inherent in the aforementioned frameworks. Additionally, the GC encouraged participants to address non-functional requirements, leading some solutions to focus on aspects such as easy deployment using Docker or graphical representation of results using tools like Grafana and Prometheus. Our solution emphasizes on meeting the functional requirements and how we can achieve completeness while not sacrificing performance.

Within the domain of financial analytics, several works have focused on developing prediction market forecasting mechanisms using various machine learning

models, reinforcement learning, genetic algorithms, and more [9] [39]. Frischbier's paper [22] offers valuable insights into the challenges posed by financial data at scale (including data management, IT governance, and compliance) and how these challenges can be addressed. The solution in this thesis builds upon this body of knowledge and, to the best of our knowledge, is the first to achieve complete in-order event processing in financial tick data despite unpredictable delays in multiple data-ingestion paths within the parallel query.

6.3 Recovery of SPSs

Fault-tolerance and state migration have emerged as crucial topics in stream processing systems, gaining significant importance and traction in recent years [33, 26, 25, 41, 8]. While significant progress has been made in understanding how to checkpoint state remotely (e.g., the work of Kwon, Balazinska and Greenberg [33], which uses HDFS for this purpose) and using it to recover to a consistent state of a streaming job, handling efficiently very large state (of sizes that cannot possibly fit in the memories of backup nodes via active replication [25]) in such systems is still a challenging and multi-faceted problem.

One work closely related to the approach taken in this deliverable is Rhino [8]. Rhino is a library designed for the efficient management of very large distributed state in stream processing systems based on the streaming dataflow paradigm. It introduces two protocols: a handover protocol and a replication protocol. The handover protocol facilitates the migration of running operators among workers, while the replication protocol enables proactive and asynchronous incremental state checkpointing on a set of workers. The protocols are considered to be tailored for resource elasticity, fault-tolerance and runtime query optimizations.

While Rhino shares the core goal and similar ideas with our proposed implementation, there are significant differences in the approaches. Rhino is a library that implements its protocols and functionalities from scratch, without leveraging the existing functionality of stream processing systems. Notably, the functionality provided by both Rhino protocols essentially replicates features already implemented in systems like Flink and HDFS. In contrast, our solution takes a different approach by minimizing intrusion through the combination and coordination (cross layer) of existing systems. By leveraging the existing functionality of Flink and HDFS, our approach aims to minimize redundant logic and build upon established frameworks.

Chapter 7

Conclusions & Future Work

This chapter concludes the thesis, summarizing the key contributions discussed in previous chapters, while suggesting potential improvements for future work.

In this thesis, we have explored two challenges related to stream processing in the financial domain. The first challenge focused on developing an efficient Tick Analysis Platform (TAP) that leverages event aggregation and complex event processing to compute trend indicators and detect patterns in real-time tick data. Our objective was to achieve in-order processing of tick data to accurately capture market trends and maintain the integrity of derived analytics. To address this challenge, we proposed two applications: S-TAP and P-TAP.

In Chapter 3, we introduced S-TAP, a solution developed for the 2022 DEBS Grand Challenge. S-TAP effectively calculated trend indicators and identified patterns resembling those used by real-life traders. However, we identified the handling out-of-order events and the mapping between batches of events and the corresponding window-closings as significant correctness challenges. While S-TAP parallelizes most operators, it maintains a single instance of the source operator and batch-unpack logic, eventually limiting the achievable parallelism.

To overcome the limitations of S-TAP, in Chapter 4 we presented P-TAP. P-TAP improved upon its predecessor and achieved parallel ingest of tick data streams while preserving the same ordering guarantees, by leveraging Flink's broadcast state pattern to disseminate control information about order. Our results when analyzing our dataset demonstrate that P-TAP achieves a speedup of up to ∼1.50x (with default settings for parallelism) vs. S-TAP, increasing to ∼1.73x when fine-tuning P-TAP by selecting a SlotSharingGroup configurations that better aligning operator tasks with the available physical parallelism. It is important to note that while P-TAP exhibits superior performance in scenarios where parallelism is fully utilized, S-TAP remains a more efficient solution when not saturated at the source (ingest). Therefore, we recommend a hybrid design based on the level of available parallelism.

Furthermore, as the second challenge in this thesis, we have focused on the

rapid recovery of Stream Processing Systems (SPSs) to ensure uninterrupted analysis in the face of failures. In Chapter 5, we focused on achieving fast recovery by aligning recovery tasks with externally stored state. Building upon the state-of-the-art incremental distributed checkpointing capabilities of the Apache Flink SPS, we extended them to enhance recovery efficiency. Our primary focus was on controlling task recovery decisions and task placement within the Apache Flink SPS, as well as influencing Hadoop Distributed File System (HDFS) block placement on data nodes.

However, there are still opportunities for further exploration and improvement for both challenges addressed in this thesis. Regarding the TAPs, additional optimizations should be considered before moving the solution to production. The proposed solutions suggest a sophisticated data pipeline along with additional external components (e.g., Kafka for data ingest). Optimizing such a pipeline can be a challenging procedure due to the large number of parameters that Flink offers concerning resource management, network communication, and more. Another interesting avenue would be to leverage the windowing mechanism utilized for this application. While our work focused on developing a financial application, this mechanism can be used to ensure in-order guarantees in different application domains.

Additionally, as mentioned in Chapter 5, some of the ideas of that section can be further explored in future work. It is crucial to determine the mapping between HDFS and the Local RocksDB sst tables to identify all the state files required for recovery accurately. Furthermore, an additional HDFS affinity placement policy is needed to enforce files originating from the same local instance of RocksDB to be checkpointed on the same machines, ensuring locality even in more complex pipelines with multiple operators. Another improvement would involve implementing a mechanism as shown in Fig. 7.1. In this case, assuming that DNs and TMs are co-located to achieve locality, throughout the recovery process, instead of performing a copy from HDFS to the local instance of RocksDB, a reference to it could be made. We intend to focus on achieving these improvements in our follow-on research.

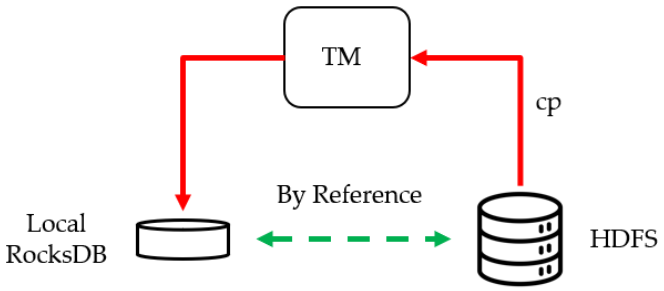

Figure 7.1: HDFS - Flink zero copy mechanism. When recovering from failure, instead of copying the recovery state from HDFS simply modify its reference

Bibliography

[1] Debs 2022. *Debs 2022: Call for Grand Challenge Solutions.* `https://2022.debs.org/call-for-grand-challenge-solutions/`[Online].

[2] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow. *Proc. VLDB Endow.*, 14(12):3135–3147, jul 2021.

[3] Cecilia Calavaro, Gabriele Russo Russo, and Valeria Cardellini. Real-time analysis of market data leveraging apache flink. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 162–165, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows, 2015.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[6] Luca De Martini, Alessandro Margara, and Gianpaolo Cugola. Analysis of market data with noir: Debs grand challenge. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 139–144, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[8] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2471–2486, New York, NY, USA, 2020. Association for Computing Machinery.

[9] M A H Dempster and C M Jones. A real-time adaptive trading system using genetic programming. *Quantitative Finance*, 1(4):397, jul 2001.

[10] FlatIcon. *"Graph Icons"*. https://www.flaticon.com/[Online].

[11] Flink. *Checkpoints-State Backend*, 2021. https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/checkpoints/.

[12] Flink. *Incremental checkpointing*, 2021. https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html.

[13] Apache Flink. *Allowed Lateness*. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/windows/#allowed-lateness[Online].

[14] Apache Flink. *Async I/O*. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/asyncio/[Online].

[15] Apache Flink. *DataStream API*. https://nightlies.apache.org/flink/flink-docs-release-1.10/dev/api_concepts.html[Online].

[16] Apache Flink. *Stateful Computations over Data Streams*. https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/general_arch.html[Online].

[17] Apache Flink. *Stateful Stream Processing*. https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/stateful-stream-processing/[Online].

[18] Apache Flink. *Stream processing: An Introduction to Event Time in Apache Flink*. https://www.ververica.com/blog/stream-processing-introduction-event-time-apache-flink[Online].

[19] Apache Flink. *Timely Stream Processing*. https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/time/[Online].

[20] S. Frischbier, J. Tahir, C. Doblander, A. Hormann, R. Mayer, and H.-A. Jacobsen. Debs 2022 grand challenge data set: Trading data, 2022. https://doi.org/10.5281/zenodo.6382482.

[21] S. Frischbier, J. Tahir, C. Doblander, A. Hormann, R. Mayer, and H.-A. Jacobsen. The DEBS 2022 Grand Challenge: Detecting Trading Trends in Financial Tick Data. In *Proc. of the 16th ACM Int. Conference on Distributed and Event-Based Systems*, DEBS '22, 2022.

[22] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth. *Managing the Complexity of Processing Financial Data at Scale - An Experience Report*, pages 14–26. 01 2020.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. volume 37, pages 29–43, 12 2003.

[24] Apache Hadoop. *Apache Hadoop Official Website*. `https://hadoop.apache.org/`[Online].

[25] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, page 779–790, USA, 2005. IEEE Computer Society.

[26] Gabriela Jacques-Silva, Bugra Gedik, Henrique Andrade, and Kun-Lung Wu. Language level checkpointing support for stream processing applications. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 145–154, 2009.

[27] S. Kalogerakis. Migrating state between jobs in apache spark. Diploma work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2020.

[28] Stefanos Kalogerakis, Antonis Papaioannou, and Kostas Magoutis. Efficient processing of high-volume tick data with apache flink for the debs 2022 grand challenge. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 156–161, New York, NY, USA, 2022. Association for Computing Machinery.

[29] P. J. Kaufman. *Trading Systems and Methods*. Wiley Publishing, 5th edition, 2013.

[30] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *Proc. of 6th International Workshop on Networking Meets Databases (NetDB 2011)*.

[31] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 1081–1092, New York, NY, USA, 2010. Association for Computing Machinery.

[32] Emmanouil Kritharakis, Shengyao Luo, Vivek Unnikrishnan, and Karan Vombatkere. Detecting trading trends in streaming financial data using apache flink. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 145–150, New York, NY, USA, 2022. Association for Computing Machinery.

[33] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1):574–585, August 2008.

[34] Kevin Li, Daniel Fernandez, David Klingler, Yuhan Gao, Jacob Rivera, and Kia Teymourian. A high-performance processing system for monitoring stock market data stream. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 166–170, New York, NY, USA, 2022. Association for Computing Machinery.

[35] Medium. *"Managed Key State in Flink"*. `https://medium.com/@sruthisreekumar/managed-key-state-in-flink-30ed45103b3a`[Online].

[36] Antonis Papaioannou and Kostas Magoutis. Amoeba: Aligning stream processing operators with externally-managed state. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Quan Pham, Quang Nguyen, Ryte Richard, Shekhar Sharma, and Xavier Ruiz. Detecting technical trading patterns in financial data with apache flink: Debs grand challenge 2022. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 151–155, New York, NY, USA, 2022. Association for Computing Machinery.

[38] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys*, 52(2):847–872, 2019.

[39] Santosh Kumar Sahu, Anil Mokhade, and Neeraj Dhanraj Bokde. An overview of machine learning, deep learning, and reinforcement learning-based techniques in quantitative finance: Recent progress and challenges. *Applied Sciences*, 13(3), 2023.

[40] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[41] Zoe Sebepou and Kostas Magoutis. CEC: Continuous eventual checkpointing for data stream processing operators. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 145–156, 2011.

[42] J. Tahir, C. Doblander, R. Mayer, S. Frischbier, and H.-A. Jacobsen. The debs 2021 grand challenge: Analyzing environmental impact of worldwide lockdowns. In *Proc. of the 15th ACM Int. Conf. on Distributed and Event-Based Systems*, DEBS '21, 2021.

[43] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Optimal and general out-of-order sliding-window aggregation. *Proc. VLDB Endow.*, 12(10):1167–1180, jun 2019.

[44] TowardsDataScience. *"How Flink stores your state"*. `https://towardsdatascience.com/heres-how-flink-stores-your-state-7b37fbb60e1a` [Online].

[45] Dawid Wysakowicz Ververica. *"A beginner's Guide to checkpoints in Apache Flink"*. [Online].

[46] Juliane Verwiebe, Philipp Grulich, Jonas Traub, and Volker Mark. Correction to: Survey of window types for aggregation in stream processing systems. *The VLDB Journal*, 05 2023.

[47] Suyeon Wang, Jaekyeong Kim, Yoonsang Yang, Jinseong Hwang, Jungkyu Han, and Sejin Chun. Real-time stock market analytics for improving deployment and accessibility using pyspark and docker. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 171–175, New York, NY, USA, 2022. Association for Computing Machinery.