TECHNICAL UNIVERSITY OF CRETE

Migrating State Between Jobs in Apache Spark

by

Kalogerakis Stefanos

A thesis submitted in partial fulfillment of the requirements for the Diploma of Electrical and Computer Engineering

Thesis Committee

Associate Professor Antonios Deligiannakis, *Supervisor*
Professor Minos Garofalakis
Associate Professor Vasileios Samoladas

December 29, 2020

Abstract

Nowadays, data is being generated at an unprecedented rate and impacts every aspect of our everyday life. As this amount increases, more and more organizations try to incorporate techniques to handle that data in real-time and evolve their business strategy. One critical challenge is ensuring fault-tolerance and high availability in our data. On different occasions, the heterogeneous systems responsible for data processing must disrupt their operation and update their infrastructure. In some other cases, system failures can occur. Therefore, migration techniques that prevent data loss are getting increasingly important.

In this thesis, we propose a state migration algorithm implemented on Apache Spark's Structured Streaming API. This powerful API offers a fast, scalable solution for processing complex workloads and ensures fault tolerance through its checkpointing mechanism. The algorithm handles state among different jobs and covers various scenarios where users might wish to split, merge, or remotely deploy workflows in each job with no data loss. In that way, users have complete control over workflow operators and can impact their execution at will. Additionally, to prove that our implementation works, we used Rapidminer Studio workflow designer to present complete and detailed test-cases for the cases mentioned above.

Περίληψη

Στις μέρες μας, νεα δεδομένα παράγονται συνεχώς σε ένα πρωτοφανή ρυθμό επηρεάζοντας όλες τις πτυχές της καθημερινότητάς μας. Καθώς ο όγκος τους συνεχίζει να αυξάνεται, όλο και περισσότεροι οργανισμοί προσπαθούν να ενσωματώσουν τεχνικές για την διαχείριση και επεξεργασία των δεδομένων αυτών σε πραγματικό χρόνο προκειμένου να εξελίξουν τις στρατηγικές της επιχείρησης τους. Μια σημαντική πρόκληση, είναι η εξασφάλιση ότι τα δεδομένα διαθέτουν ανοχή σε σφάλματα και υψηλή διαθεσιμότητα. Σε διαφορετικές περιπτώσεις, τα ετερογενή συστήματα που είναι υπεύθυνα για την επεξεργασία των δεδομένων πρέπει να διακόψουν την λειτουργία τους για αναβαθμίσουν τις υλικοτεχνικές υποδομές τους. Σε άλλες περιπτώσεις, μπορεί να συναντηθούν σφάλματα συστήματος. Για αυτόν τον λόγο τεχνικές migration για την αποτροπή απώλειας και μεταφοράς δεδομένων γίνονται όλο και πιο σημαντικές.

Στα πλαίσια της συγκεκριμένης διπλωματικής εργασίας, παρουσιάζεται ένας αλγόριθμος μεταφοράς κατάστασης υλοποιημένος στο Structured Streaming API του Apache Spark. Αυτό το ισχυρό API προσφέρει μια γρήγορη και επεκτάσιμη λύση για την επεξεργασία περίπλοκων workflows και εξασφαλίζει ανοχή σε σφάλματα μέσω του μηχανισμού checkpoint που διαθέτει. Ο αλγόριθμος διαχειρίζεται την κατάσταση μεταξύ διαφορετικών jobs, και καλύπτει πληθώρα σεναρίων, όπου οι χρήστες δύνανται να διαχωρίσουν, να ενώσουν ή να εκτελέσουν απομακρυσμένα workflows σε κάθε job δίχως απώλεια δεδομένων. Με αυτόν τον τρόπο, υπάρχει πλήρης έλεγχος των operators του κάθε workflow και η εκτέλεση μπορεί να επηρεαστεί με τρόπο που επιθυμεί ο εκάστοτε χρήστης. Προκειμένου να αποδείξουμε ότι η υλοποίηση λειτουργεί, χρησιμοποιήσαμε το Rapidminer Studio για τον σχεδιασμό των workflows, όπου παρουσιάζονται πλήρη και λεπτομερή test-cases για τις όλες περιπτώσεις που αναφέρθηκαν προηγουμένως.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Antonios Deligiannakis for his immediate assistance and guidance throughout this thesis. I would also like to thank the members of committee, Prof. Minos Garofalakis and Prof. Vasilis Samoladas, for their support.

Furthermore, I would like to personally thank SoftNet Lab members Nikos Giatrakos, Giorgos Stamatakis and Xenia Arapi. The completion of this thesis would not be possible without their assistance.

Last but not least, I am deeply thankful to my family and friends for always supporting me throughout my studies.

Contents

List of Figures

List of Code Snippets

Chapter 1

Introduction

In the era of big data, an unfathomable amount of data is created, processed, and transferred every second. In 2020, an average person is estimated to create approximately 2.5 quintillion bytes per day. Another astonishing statistic states that most of the world's data(appr. 90%) has been created in only the past two years. This is justified as the IoT industry is booming and plays an integral part in our everyday lives.

As this amount increases, it is crucial to find ways to analyze those data on-the-fly and extract useful information, leading to groundbreaking discoveries in different fields of study. However, processing such large amounts of data is difficult, and performing traditional computations is not doable under those circumstances. For those reasons, big data frameworks, such as Apache Spark, have been created, allowing users to process large amounts of data with quickness, accuracy, and efficiency.

While big data frameworks are getting more popular and continuously evolving, new software tools are developed on top of big data frameworks to make them more user friendly. Normally big data frameworks demand programming knowledge, which is a limiting factor for many users. This is not the case with Rapidminer Studio, which is an easy to use data science platform, and among other things, enables interaction with Big Data frameworks without writing any code.

An important aspect of those frameworks is ensuring fault-tolerance and high availability in data between execution interruptions. On different occasions, the heterogeneous systems responsible for data processing must disrupt their operation and update their infrastructure. In some other cases, system parameters or resource allocation must change for more effective data processing. Therefore, migration techniques that prevent data loss are getting increasingly important.

In this thesis, a State Migration Algorithm is proposed using Apache Spark's Structured Streaming API and Rapidminer Studio designer. The algorithm takes full advantage of the powerful checkpointing mechanism that Structured Streaming provides and offers a flexible solution that allows users to completely reformat their workflows,

alter their parameters or even transfer them to a remote cluster. All of the above is accomplished with no data loss.

1.1 Thesis Outline

- **Chapter 2-Background Knowledge:** This chapter presents basic theoretical knowledge of key concepts used in this thesis
- **Chapter 3-Tools:** Provides a comprehensive analysis(key features, architecture) for all tools used in the implementation afterward. Each subsection is devoted to a different technology, including HDFS, Apache Kafka, Apache Livy, Rapidminer, and Apache Spark. In Apache Spark's subsection, Structured Streaming API is also scrutinized, the API used for the implementation.
- **Chapter 4-Implementation:** The fourth chapter examines all the aspects of the current implementation. For a better understanding, the chapter is divided into four subsections, all of which examine design choices and changes made in Rapidminer Studio Extension. The proposed State Migration algorithm is also analyzed in the last subsection.
- **Chapter 5-Experimental Execution:** This chapter includes complete test cases for some key scenarios that State Migration Algorithm can deploy. All the different workflows, setups, and properties are presented step by step for these scenarios, which include splitting, merging, or remotely executing workflows.
- **Chapter 6-Challenges and future work:** The final chapter mentions some limitations and challenges in the current implementation and recommends steps for future interest

Chapter 2

Background Knowledge

2.1 Streaming

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, and their manipulation can occur on the fly, without download. Those data can be generated by various sources at different rates and volumes and can also combine into a single stream. Real-time stock exchange, sensor monitoring, website tracking activity are only some examples of streaming data applications. Steaming is also essential in big data, as it provides real-time analytics, data ingestion, and data integration.

2.1.1 Bounded and Unbounded Streams

We can examine Streaming in two different paradigms; **bounded and unbounded**. An unbounded stream is a stream, which does not have a defined ending point, and events must be processed on the fly, as it is impossible to wait for all input data. In that case, all events are the same, and users can distinguish them by their received time. Bounded streams, on the contrary, have a predefined start and end time. All data in that time frame can be ingested before processed or analyzed even further. Data is handled in batches; that is why bounded stream processing is also known as batch processing.

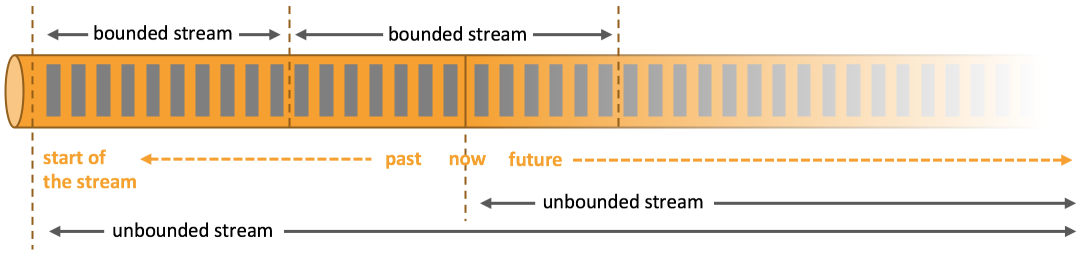

FIGURE 2.1: Bounded and Unbounded Streaming

2.1.2 Window Models

In Streaming, multiple models exist regarding how time affects the streams and time manipulation in general. The most common which are analyzed below are the **Landmark and Sliding window model**.

- **Landmark model:** In this model, we consider all the events from the moment we initiate monitoring in a stream. Let us consider a time point 0, which is the time when the monitoring starts. Each window will maintain events in the interval $[0, now]$ and grow as the stream progresses.
- **Sliding window model:** In this model, we consider only events contained in a predefined window range. Specifically, let us consider a fixed size window w and the current time point t. Each window will hold only the events existing in the interval $[t-w+1, t]$ and discard older events. As time passes, the window maintains its size and slides. The number of events considered is (obviously) depending on the window size.

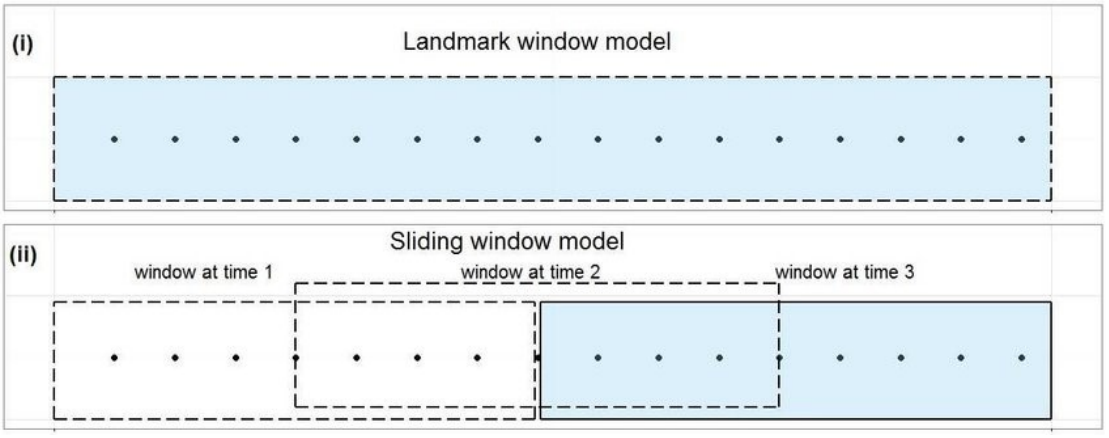

FIGURE 2.2: Landmark and Slinding Window Models

2.2 Hashing and String hashing

A *hash function* is any function that will deterministically map data of arbitrary size to smaller fixed-size values.

One of the applications of a hash function is efficient string comparison. The hash function converts a string into an integer, and the condition that must apply in the following: In the case of two equal strings, also their hash values must be equal. However, it is important to highlight that **the opposite statement does not have to hold**(if hash values are equal, then the strings are not equal) as collisions can occur. This method improves the complexity of the classic brute-force approach for letter by letter comparison between two strings with $O(1)$ complexity (brute-force complexity $O(\min(length(string1), length(string2)))$)

2.2.1 Polynomial rolling hash function

The *polynomial rolling hash function* is a common way to define the hash of a string. This family of hash functions treats the symbols of a string as coefficients of a polynomial with a positive variable **P** and computes its value modulo a positive constant **M**. Given a string s of length n :

$$\text{hash}(s) = s[0] + s[1] \cdot P + s[2] \cdot P^2 + ... + s[n-1] \cdot P^{n-1} \mod M \tag{2.1}$$

$$= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod M, \tag{2.2}$$

The right choice of P and M values is critical in order to get a decent hashing function. P is typically a prime number roughly equal to the number of the characters within the input alphabet. M is employed to avoid manipulating larger values(perform all computations modulo M) and should be a large number; hence the probability of a single comparison between two random strings colliding is inversely proportional to $M(about \approx 1/M)$. However, this value should be sufficiently small to multiply two values using 64-bit values and avoid overflows(chosen value $M = 10^9 + 9$ is a good choice).

In order to boost no-collision probability, users often deploy different techniques, including the selection of larger modulo and multiple hashes(for each string, compute multiple hashes and compare their pairs one by one).

2.3 JSON files

The JSON file is a file that follows the *JavaScript Object Notation (JSON)* format to store data structures and objects consisting of attribute-value pairs or other serializable structures. JSON is a standard data interchange format that is also human-readable and text-based, making it a ubiquitous data format. It serves a variety of purposes but is mainly used to exchange data between web apps and servers.

```
{
    "firstName":"Stefanos",
    "lastName": "Kalogerakis",
    "address": {
        "streetAddress": "myAdress",
        "city": "Chania",
        "postalCode": "73134"
    }
}
```

Code 1: JSON Property file example

Chapter 3

Tools

3.1 HDFS - Hadoop Distributed File System

Hadoop Distributed File System is an open-source, highly fault-tolerant data storage file system that operates on commodity hardware. It is optimized when handling big data, as the design purpose was to overcome issues like reliability and speed that traditional databases could not.

3.1.1 Key Features of HDFS

1. **Handles big data:** HDFS accommodates applications with data sets typically gigabytes to terabytes in size and provides a solution that traditional file systems could not. It does this by segregating the data into manageable blocks, which allow fast processing times and can deliver more than 2GB of data per second, thanks to its cluster architecture.

2. **Fault-tolerant:** Data is replicated across multiple systems and is always accessible even in case of failure

3. **Scalable:** Resources can be managed (and scaled) in each system accordingly. HDFS includes vertical and horizontal scalability mechanisms.

4. **Portable:** HDFS is designed to be portable across multiple heterogeneous hardware platforms and be compatible with a variety of underlying operating systems.

3.1.2 Architecture

HDFS incorporates a master-slave topology. The master node is known as **NameNode**(only one per cluster), whereas (multiple) slave nodes are called **DataNodes**.

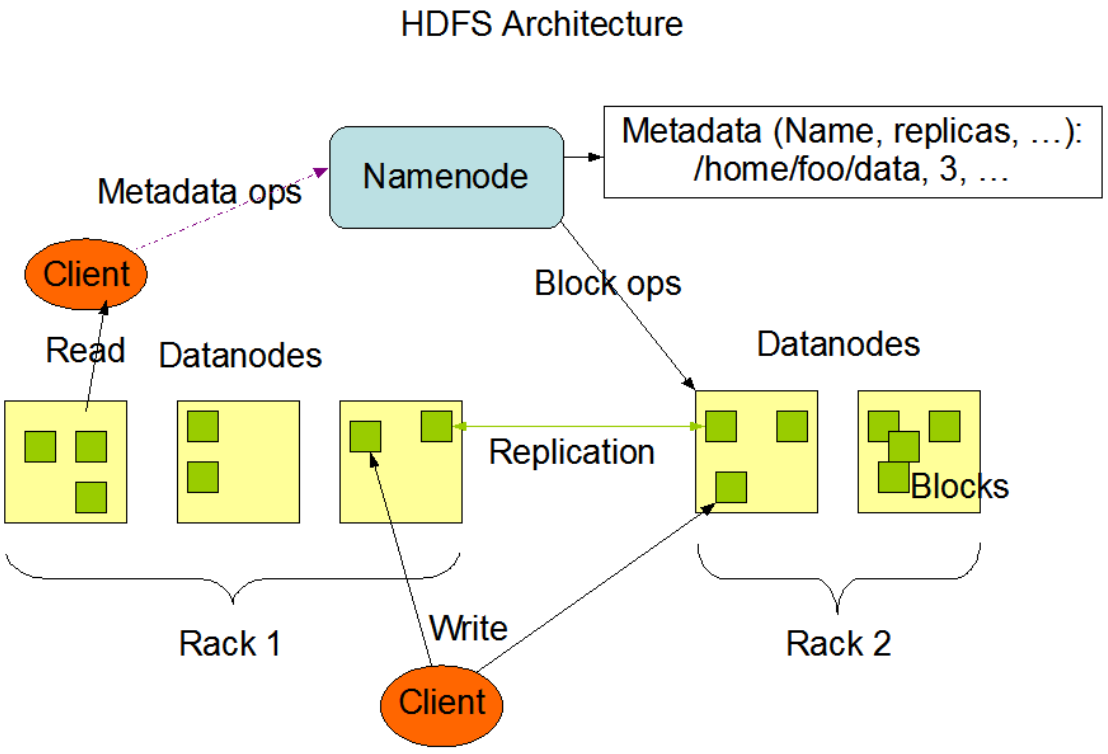

FIGURE 3.1: HDFS - Architecture

3.1.2.1 NameNode

NameNode 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, etcetera.

3.1.2.2 DataNodes

DataNodes are accountable for storing actual data in blocks assigned by the NameNode. HDFS overcomes the issue of DataNode failure by creating replicas(copies) of the data. The default replication is three, and it is strongly advised not to go under three.

DataNodes are supposed to send a heartbeat or signal to ensure that every node is working.

3.1.3 Data Manipulation

Just like any other DFS, each system file is stored as a sequence of blocks on DataNodes. Each block's **default size is 128MB** in Apache Hadoop 2.x(64MB in Apache Hadoop 1.x). Block size is modifiable through the configuration. Before the NameNode can store and manipulate data, files need to divide into smaller block-sized chunks stored as independent units.

3.1.3.1 Block Division

The number of blocks depends on the initial size of the file. All but the last block are the same size as the configured block size(128MB by default), while the last one is what remains of the file. For example, an 320MB file splits into two blocks of 128MB, and a third block stores the remaining 64MB.

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

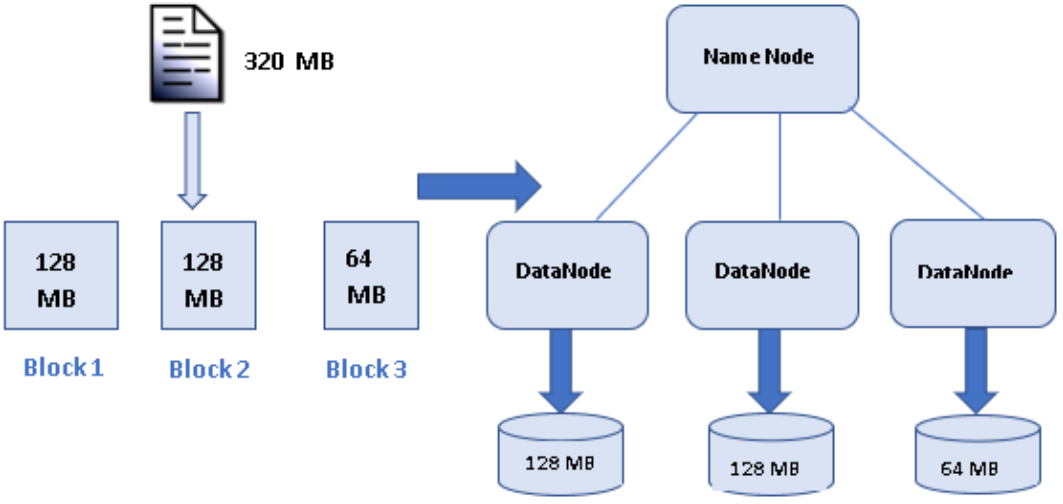

FIGURE 3.2: HDFS - Block Division

Note that the selection for a default block size of 128MB, transpired to balance the tradeoff between overhead and processing time perfectly. In case the block size is smaller, too many data blocks along with metadata can cause increased overhead, whereas in case the block size is very large, the processing for each block increases.

3.1.3.2 Replication

The *data replication* method is responsible in order to handle unexpected failures and data loss. NameNode creates several copies of every data block (3 by default, can be modified through configuration), distributed across different DataNodes. Replicas are not distributed randomly, as HDFS has rack awareness policies to ensure high availability and fault tolerance while maximizing network bandwidth.

HDFS Rack Awareness policies include:

1. One DataNode can store one replica of a data block.
2. The same Rack cannot assign over two replicas of a single block.
3. The number of racks used inside an HDFS cluster must be smaller than the number of replicas.

3.2 Apache Kafka

Apache Kafka is an open-source, reliable, high-throughput publish/subscribe messaging system also described as a distributed event log where all the new records are immutable and appended at the end of the log.

Kafka is nowadays a ubiquitous solution for real-time ingestion, analytics, and processing streaming data. It is also compatible with all popular Big Data platforms, including Spark, Storm, and Flink.

3.2.1 Publish/Subscribe messaging

Publish/Subscribe is a messaging pattern in which the sender (also known as the publisher) does not send data directly to specific receivers (also known as subscribers). The publisher classifies the messages without knowing if there are any subscribers interested in the particular type of message. Similarly, the receiver subscribes to receive a specific class of messages without knowing if any senders are sending those messages. Many Pub/Sub systems contain a broker or event bus where all messages are published. This broker enables a loose connection between sender and receiver, which improves flexibility and scalability in the system.

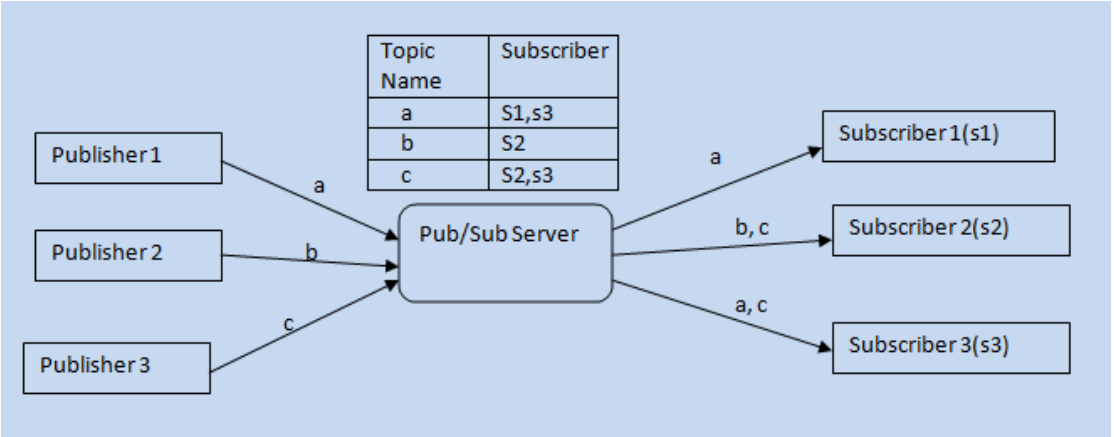

FIGURE 3.3: Publish/Subscribe messaging pattern

3.2.2 Architecture

Before breaking down the architecture even further, it is essential to introduce some fundamental concepts utilised later on.

- **Topic:** is an ordered collection of events, stored in a durable way
- **Producers:** are the publisher clients, producing messages and assigns them to different topics
- **Consumers:** are the subscriber clients, consuming messages from topics and maintaining their position in the stream of data

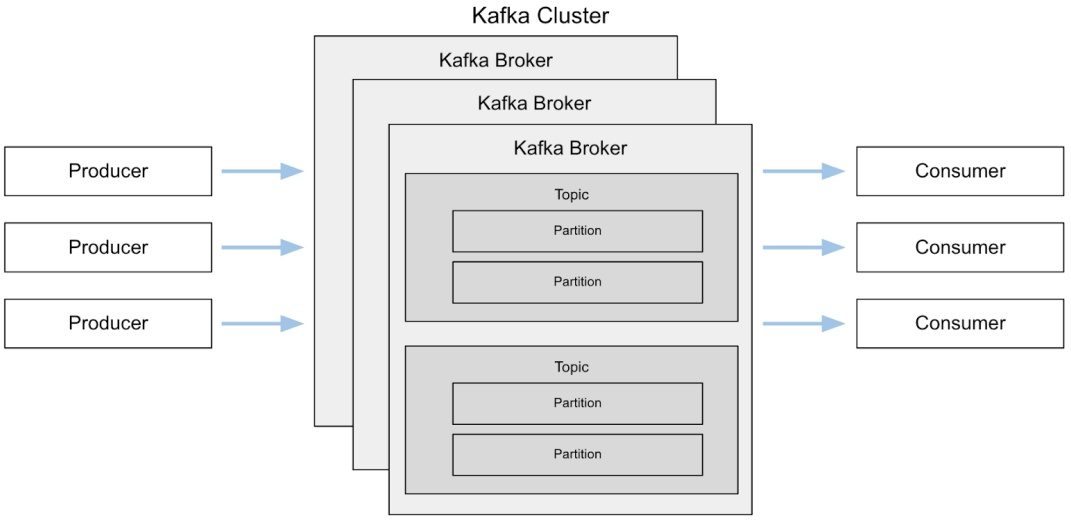

FIGURE 3.4: Apache Kafka - Architecture

On a high level, Kafka consists of 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 collecting messages from producers, assigning offsets, and committing messages to disk. It is also responsible for responding to consumers' fetch requests and sending messages. Within the cluster, one broker will work as a cluster controller and is responsible for tasks such as monitoring broker failures.

Following the publish/subscribe messaging pattern, producers create new messages and send them to a specific topic. Topic classifies data being sent and can be further broken down into subsets, known as **partitions**. Each partition maintains separate commit logs, and the order of messages can be guaranteed only across the same partition. Splitting a topic into multiple partitions makes scaling easy as different consumers can read each partition. In that way, partitions and consumers can be distributed across different servers and achieve higher throughput. Then, consumers read messages by subscribing to different topics. All messages are read in the order they were produced, which is accomplished by keeping track of the offsets(sequential ID of specific messages in specific partitions). Consumers always belong to a specific consumer group, and all partitions of a topic are distributed evenly in each member.

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

3.3 Apache Livy

Apache Livy is an open-source REST-based web service that enables easy interactions with a Spark Cluster. Livy extends Spark capabilities, offering additional multi-tenancy and security features via a simple REST interface or an RPC client library.

3.3.1 Key Features of Livy

1. Long-running Spark Contexts may be used for multiple Spark jobs by multiple clients. Each context can own a different configuration, and every context or driver can have multiple executors associated with it

2. Job results can be retrieved over REST asynchronously or synchronously

3. Multi-user support via user impersonation. The Livy server has access to resources and files on behalf of the user that submits the corresponding requests. In that way, permission escalation is avoided(important for multi-tenant environments)

4. Multiple jobs and clients can share Cached RDDs or Dataframes
5. Multiple Spark Contexts can be managed simultaneously and the Spark Contexts run on the cluster in their containers (YARN/Mesos) instead of the Livy Server, for improved fault tolerance and concurrency
6. Jobs submission can be delivered through precompiled jars, snippets of code, or via client API
7. Ensure security via secure authenticated communication and wire encryption

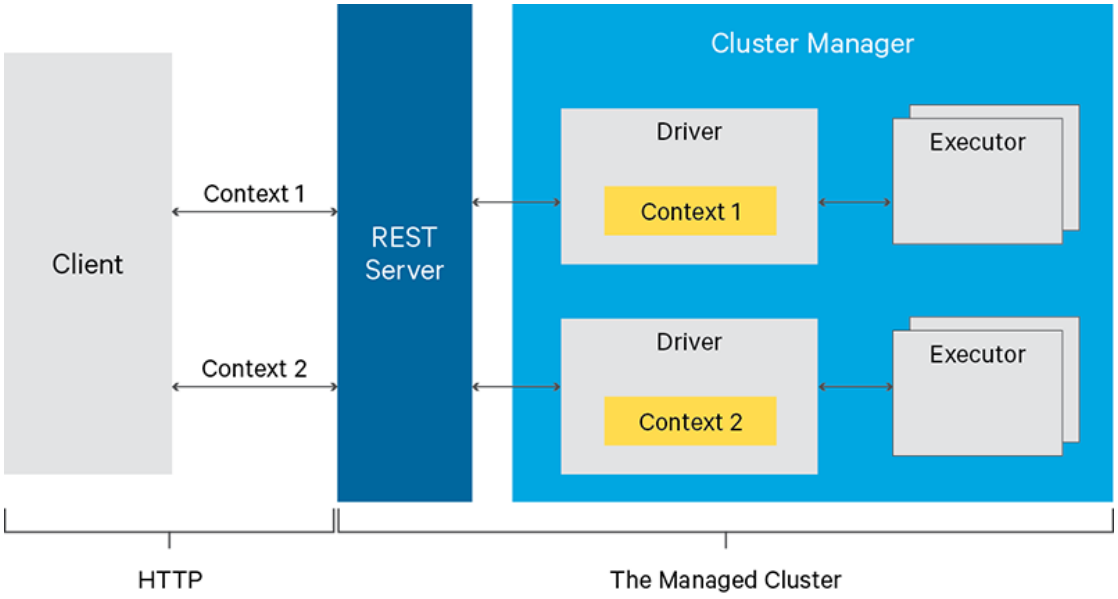

FIGURE 3.5: Apache Livy - Architecture

There two modes to interact with the Livy interface:

- **Interactive Sessions** offer a running session where users can submit statements. Given that resources are available, the demanded statements execution proceeds and users can obtain the output. A typical use case is to experiment with data or perform quick calculations. Requests are submitted under **/sessions/** directory.
- **Jobs/Batch** offers a more compact solution to submit code packages like programs. A typical use case is a regular task equipped with arguments(input files, output directory) and workload executed in the background. Requests are submitted under **/batches/** directory.

The available requests for Apache Livy REST API are **GET, POST, DELETE**. Thorough information regarding accepted parameters and expected responses in the different

modes are available in the documentation. The following example demonstrates an indicative **DELETE request in batch mode**:

```
curl -H "X-Requested-By: skalogerakis" -X DELETE
↪  http://clu01.softnet.tuc.gr:8999/batches/27
```

Code 2: Apache Livy - Delete request in batch mode

3.4 Rapidminer

RapidMiner is a data science software platform that unites data preparation, machine learning, and predictive model deployment. It is suitable for non-programmers, as its powerful analytical solutions are based on templates that ensure speed and error minimization. Rapidminer's core of the platform is open source and was formerly known as YALE(Yet Another Learning Environment). Some **key features** include:

1. Easy code-free environment
2. Data Loading
3. Data Transformation
4. Data Modelling
5. Data Visualization
6. Modular component schema(Using Operators)

3.4.1 RapidMiner Studio

RapidMiner Studio is an easy-to-use visual workflow designer that streamlines data science tasks right from the fast prototyping of concepts to the development of mission-critical predictive models. The intuitive GUI simplifies the development of predictive analytic workflows by providing mechanisms to simulate data science-related tasks. It also provides by default numerous data connectors that can extract, access, and load information from any data source, any format, at any scale. Tools provided by default in RapidMiner Studio include:

- **Turbo Prep:** For ETL(extract, transform, load) processing via a web-based user interface

- **Auto Model:** Create predictive models using automated machine learning and data science best practices
- **Deployments:** Offers one-click deployment of models

Extensions can add extra functionality to RapidMiner Studio, and are made available on the *RapidMiner Marketplace*.

3.4.1.1 Fundamental Components

As mentioned beforehand, RapidMiner Studio provides an intuitive GUI to design visual workflows. Those workflows are called **Processes** and are designed in a plug-and-play fashion by wiring multiple **Operators** through **Ports**. Each **Operator** is a component responsible for performing tasks(data processing, transformations, etcetera.) within the process, and its output represents an input of the next one. Operator properties are modifiable through the **Parameters** section.

3.4.2 Rapidminer Streaming Extension

Rapidminer Studio does not support the processing of streaming workflows by default. *Rapidminer Streaming Extension* enables the creation of optimized, streaming workflows in Flink, Spark, or Kafka.

The extension adds extra architectural components that reflect the philosophy of Rapidminer Studio and allow workflow design without extra coding. **Streaming Nest** is a new key Operator that serves as a *subprocess*. That means that the developer can encapsulate families of Operators(**Stream Transformation Component**) inside after double-clicking the corresponding Streaming Nest. Operators can connect with arrows, and their direction showcases the data flow of each workflow.

Stream Transformation Component includes Operators that can perform stream transformations provided by Big data platforms. All these Operators were designed as an abstraction layer and can operate without defining the Big Data platform. Some of these operators are:

- Aggregate Stream
- Duplicate Stream
- Join Streams
- Connect Streams

- Filter Stream
- Kafka Sink
- Kafka Source
- Map Stream
- Select Stream

The **Connection Component** connects the operator between the abstraction layer of the Operators mentioned above and the cluster of different Big Data platforms.

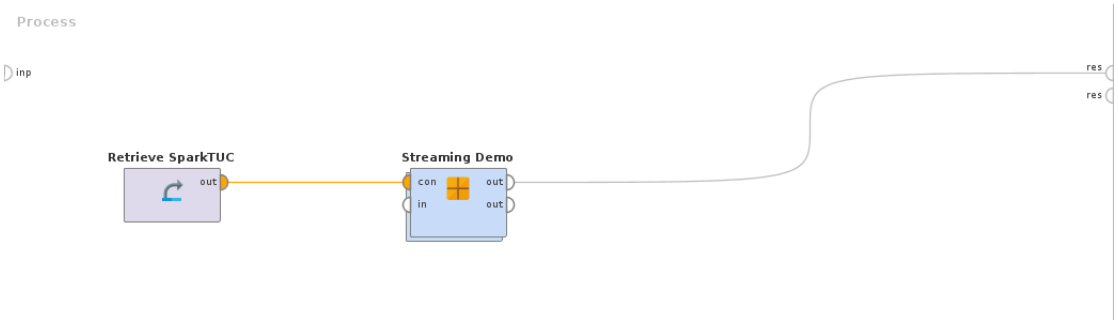

FIGURE 3.6: Rapidminer Studio - Streaming Nest Operator Example

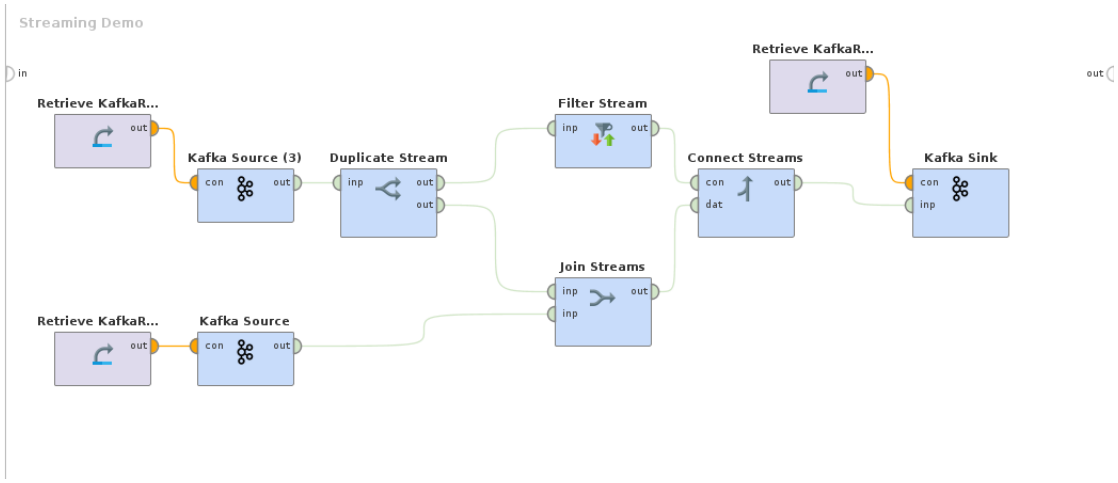

FIGURE 3.7: Rapidminer Studio - Streaming Extension Operators Examples

3.5 Apache Spark

Apache Spark is an open-source cluster computing framework for real-time processing and analysis of a large amount of data. Some of its key features, including real-time low latency processing, fault-tolerance, powerful caching, and ease-of-use, make the framework incredibly popular among data scientists.

3.5.1 Spark components

The following components comprise the *Spark ecosystem*:

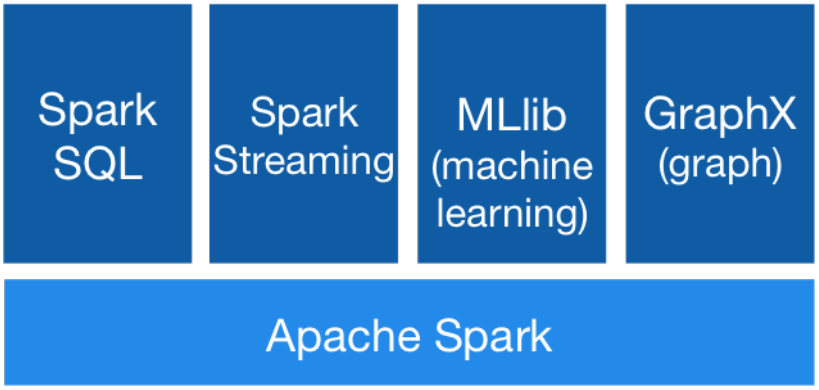

FIGURE 3.8: Apache Spark - Ecoystem

- **Spark Core Engine:** The core engine of the entire spark ecosystem and takes care of processing. Everything works on top of Spark Core Engine.
- **SparkSQL:** used for structured data and semi-structured data processing
- **Spark Streaming:** A lightweight API, used for real-time analytics. Both batch processing and real-time processing is allowed.
- **Spark MLlib:** Used for ML applications and predictive analytics
- **GraphX:** Graph type of analytics, data is represented as Graphs

3.5.2 Architecture

Spark is based on two important abstractions

- **RDDs(Resilient Distributed Dataset):** are the fundamental units of data in Spark that can be divided and executed parallel across different worker nodes of the cluster.
- **DAG(Directed Acyclic Graph):** DAG is the Spark Architecture background scheduling layer that implements scheduling in a stage-oriented manner. (sequence of computations performed on data)

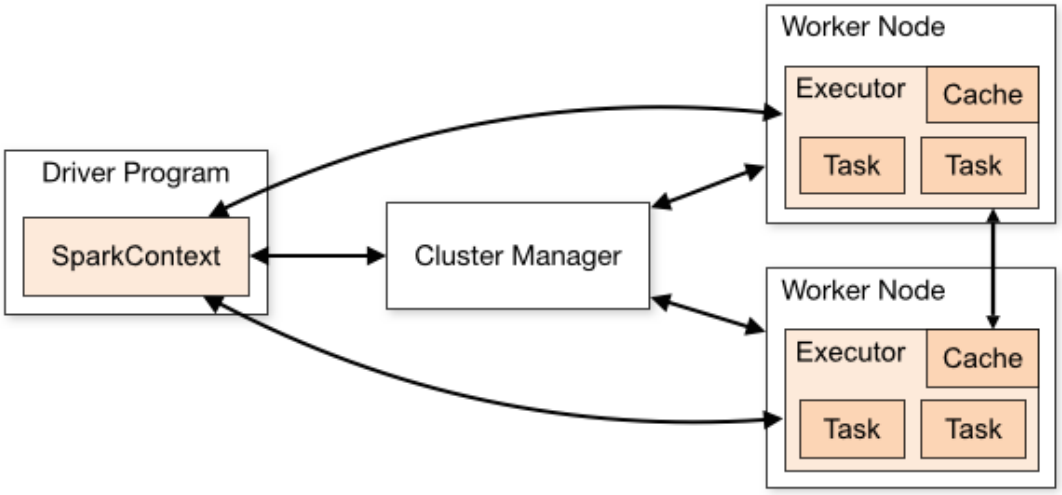

FIGURE 3.9: Apache Spark - Architecture

Spark uses a master-slave architecture that consists of a driver that runs on a master node and multiple executors which run across the worker nodes in the cluster.

Master Node has a **Driver Program**. Every spark application has a driver program, and that internally contains a **Spark Context**, which is the entry point of the application for any spark functionality.

This Spark Context interacts with the cluster manager to handle jobs. The **cluster manager** is responsible for acquiring resources on the Spark cluster and allocating them to a Spark job. The driver program and the Spark Context take care of executing the job across the cluster. At a high level, a job is split into multiple tasks, and afterwards will be distributed over the slave or worker nodes. Whenever Spark performs transformations or uses methods of Spark context, then RDDs are distributed across multiple nodes. Worker nodes are the slave nodes who execute the tasks on the partition RDDs and then return the result to the spark context.

In a nutshell, Spark Context takes the job, divides the jobs into multiple tasks, and sends them on the worker nodes. These tasks then transform on partition RDDs, and the result returns to the main spark context.

Increasing the number of workers can lead to faster execution time, as memory caching increases, and jobs can be executed concurrently in multiple nodes.

3.5.3 Structured Streaming

Apache Spark's Structured Streaming is a processing engine built on top of the strong foundations of Spark SQL. The main idea behind Structured Streaming is that the user should perform streaming analytics without reasoning about streaming and corner cases.

This powerful and high-level API deals with those limitations and provides a fast, scalable, and fault-tolerant solution for managing complex data and workloads. Structured Streaming supports two kinds of Stream processing; **micro-batch and continuous**. For this thesis, we explore the **micro-batch engine** as Continuous Stream Processing does not support stateful operations.

3.5.3.1 Programming Model

Conceptually, Structured Streaming treats data as entries of a dynamic(and infinite) input table. New entries in the stream behave as rows appended to the input table. The users define queries on the input table as if it was a batch-like query on a static table, and a final **Result Table** is generated.

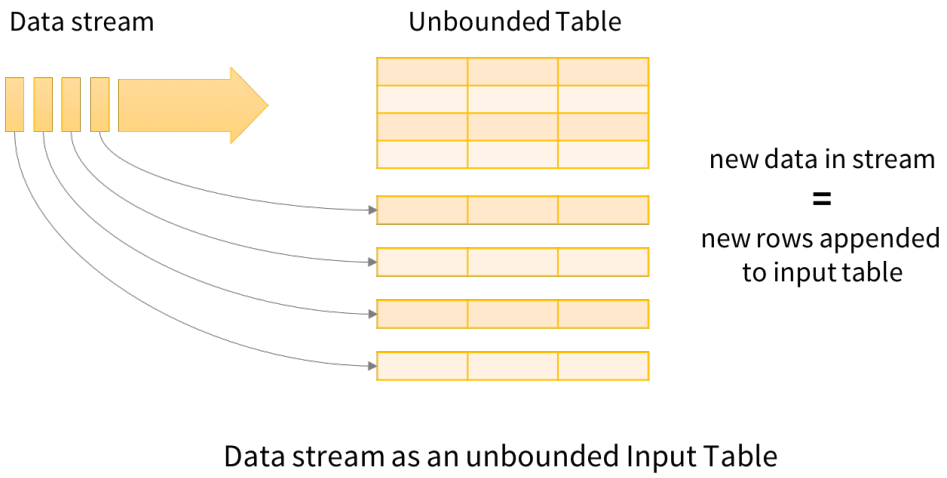

FIGURE 3.10: Structured Streaming - Programming Model

Internally, Spark maintains the minimal intermediate state required(not the entire input stream) and incrementally updates the final Result Table after new entries arrive, respectively. When changes occur in the results table, modified results are written to the **external sink**. **Triggers** control the frequency Spark appends new entries to Input Table and eventually updates the Result Table.

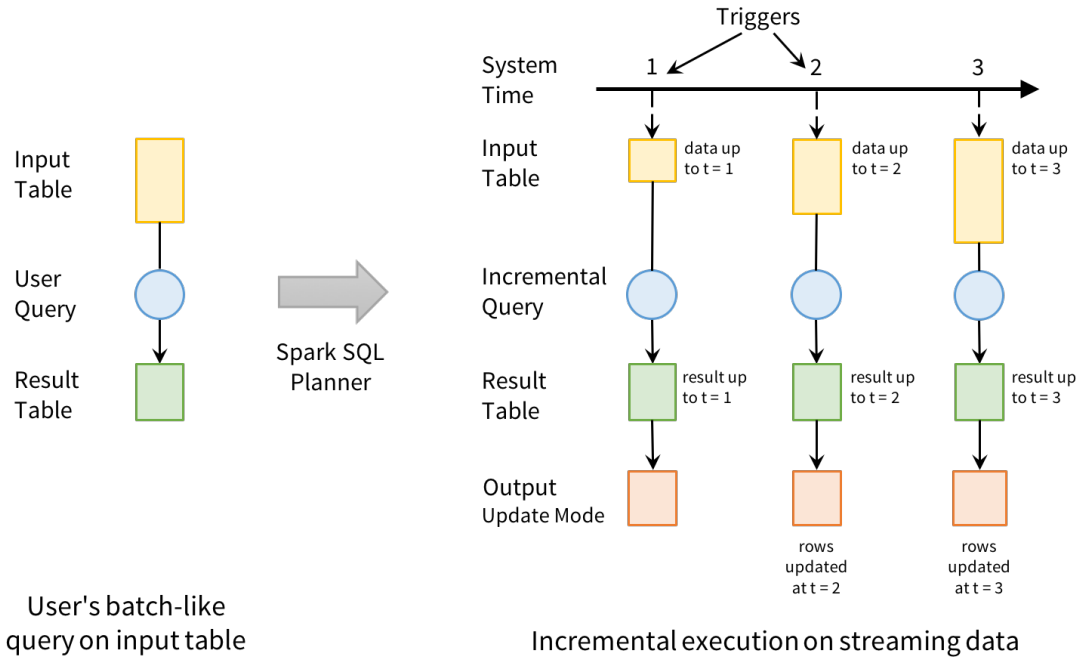

FIGURE 3.11: Structured Streaming - Processing Model Example

The **output** is the final part of the model, which specifies the changes to write into an **external system(HDFS, S3)**. Three modes are currently available:

- **Complete mode:** The whole result table are written to external storage every time
- **Append mode:** The default mode, in which only newly appended rows to the result table since the last trigger, are written to external storage
- **Update mode:** In this mode, only the updated rows that in the Result table are written to external storage

3.5.3.2 Transformations and Window Operations on Event-time

Spark's *Dataset and DataFrame* APIs are embedded on Structured Streaming. Both Datasets and DataFrames represent distributed collections of data with different properties and methods.Such methods include data transformation(map, flatmap) and SQL-type operations(select, groupBy) along with running aggregations(average, max).

```
Dataset<Row> windowAggr = query.groupBy(
  functions.window(column("timestamps"), "10 minutes", "5 minutes")
).count();
```

Code 3: Structured Streaming - Window Aggregation Example

The developer can create Datasets or DataFrames with the use of **readStream()** from different sources. This example illustrates how to read data from the connection 145.2.1.3:1234

```
1 Dataset<Row> lines = spark
2   .readStream()
3   .format("socket")
4   .option("host", "145.2.1.3")
5   .option("port", 1234)
6   .load();
```

Code 4: Structured Streaming - Dataset creation example listening from a socket

A widespread way to handle data is by performing operations on different types of *windows on event-time*(generation time of the event). The code above demonstrates a sliding window example performing count aggregation(10 minutes window, slide every 5 minutes). In Structured Streaming, window operations are special group-By aggregations. Note that in this case, every time window is a group, and input events can affect different windows. So in some instances, multiple table rows need to be updated.

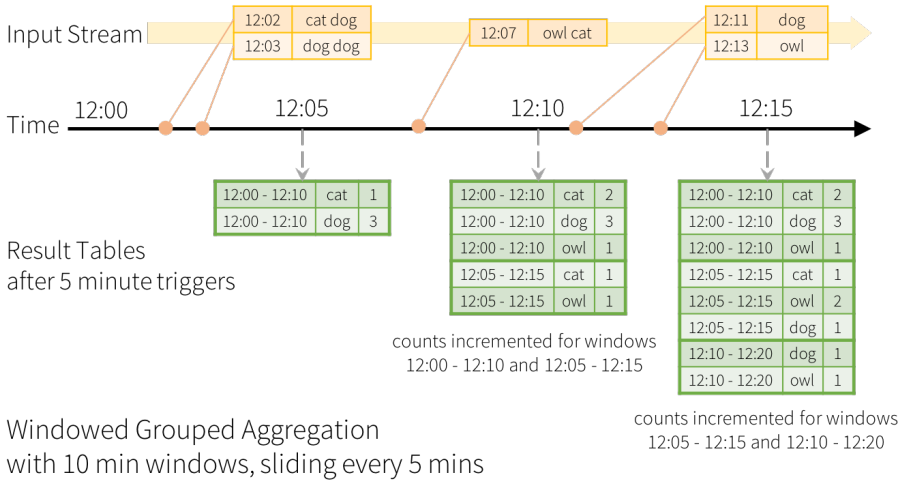

FIGURE 3.12: Structured Streaming - Window count using sliding window model

3.5.3.3 Stateful Incremental execution

Stateful Stream Processing is stream processing with state. The main difference with stateless processing is that records can combine and are not independent. State is a collection of keys and their current value pairs. A streaming query is stateful in the following cases:

- Stream Aggregation
- Arbitrary Stateful Streaming Aggregation
- Stream-Stream Join
- Stream Deduplication
- Streaming Limit

During the execution of stateful streaming queries, SparkSQL internally *maintains an intermediate state* for fault tolerance. The intermediate state is stored versioned inside the Spark executors memory and also backed to the user-defined checkpoint location using write-ahead logs. **Checkpoint location** is a path to a fault-tolerant file system like HDFS. Every trigger reads the previous state and saves the updated state both in memory and the write-ahead log. In case of failure, the latest completed state restores from the checkpoint location, and the query resumes its execution from the point of failure. The API also ensures exactly-once guarantees for stateful stream processing when input sources are replayable, and streaming sinks are idempotent to handle reprocessing.

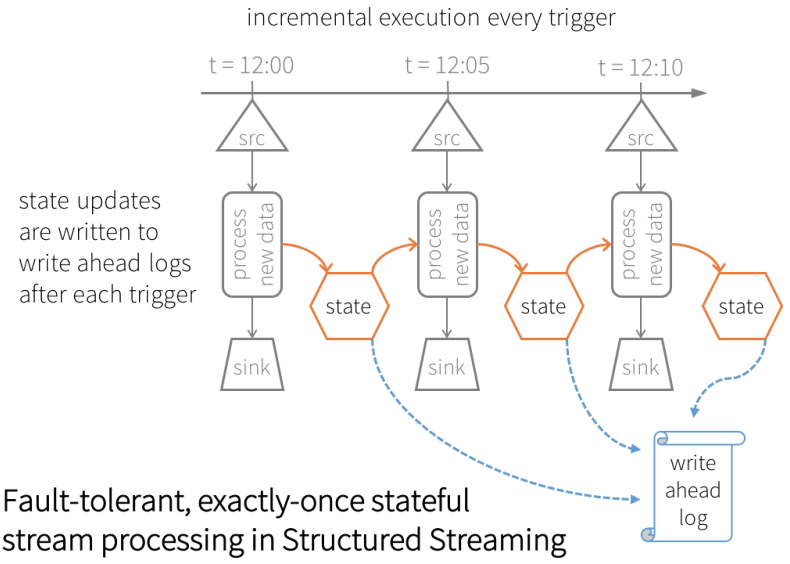

FIGURE 3.13: Structured Streaming - Incremental Stateful Query Updates

3.5.3.4 Handling Late Events

Handling late and out-of-order data is also supported automatically; Spark preserves state, enabling late events to update older window entries. However, it is crucial to limit state; otherwise, it would increase indefinitely. In order to achieve that, **watermarking** is introduced, which is essentially a moving threshold of how late events are expected to be and when to drop old state. The system keeps track of the max event time, and the watermark is a trailing threshold that trails behind this max event time. Data within the trailing gap are allowed to aggregate.

In contrast, data older than watermark is too late and dropped. Windows older than watermark automatically delete to limit the amount of intermediate state in-memory. Watermarks only applied on stateful operations and ignored otherwise.

In the previous windows aggregation example, the method **.withWatermark("timestamp", "10 minutes")** adds a 10-minute watermark threshold, and the following photo illustrates the impact of the watermark.

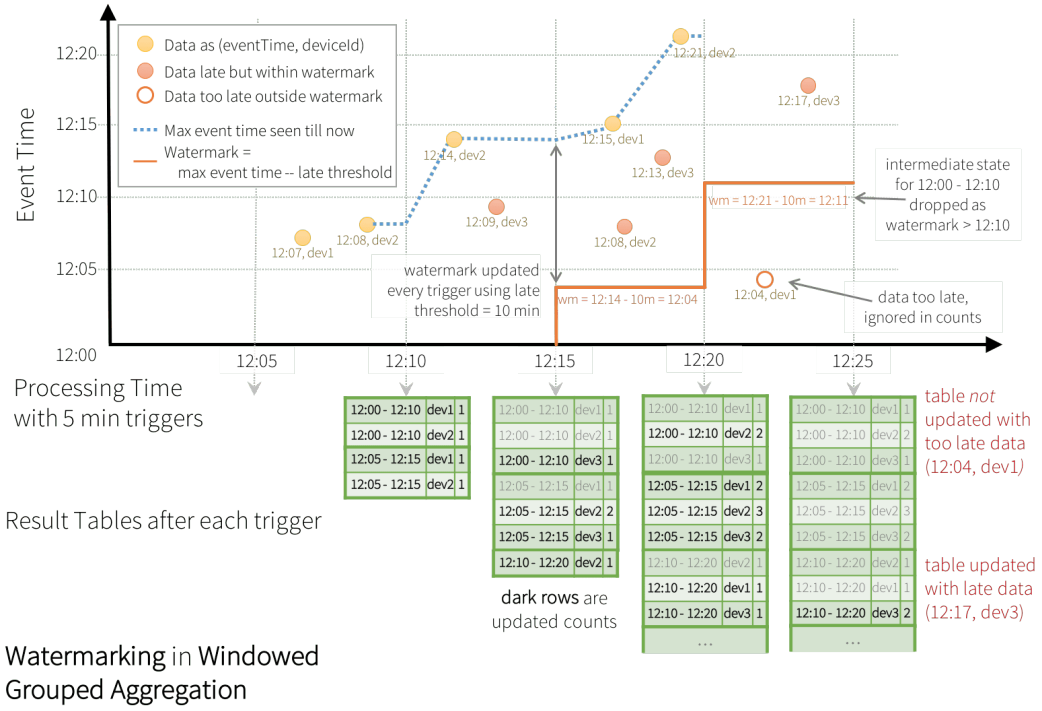

FIGURE 3.14: Structured Streaming - Late Event manipulation

3.5.3.5 Starting Streaming Queries

After specifying the final Dataset or DataFrame, the final step is to initiate the streaming computation. **Dataset.writeStream()** accomplishes that by returning the required DataStreamWriter interface with the following properties:

- **Output sink information:** Specifies the sink format (file, Kafka, console, memory), output location path, etcetera
- **Output mode:** Specifies the content written to the external sink. Append is the default mode. Check the Programming Model section for more details regarding the available modes.
- **Query name:** Specifies an ID-like distinct query name. This property is optional.
- **Trigger interval:** Specifies the trigger interval. This property is optional, and if absent, the system checks for availability of new data right after the completion of previous processing
- **Checkpoint location:** Specifies a directory location in an HDFS-compatible, fault-tolerant file system to store all checkpoint information. It is important to

state that not all sinks guarantee the same end-to-end fault-tolerance (some sinks are meant for debugging purposes). This property is optional.

To start executing the continuous query, the developer must also use the **start()** method, which produces a final StreamingQuery object. It can be used later for monitoring purposes. In many cases, to prevent the execution from stopping while the running query is alive, the method **awaitTermination()** is called.

```
1  Dataset<Row> example = ...
2
3  example
4    .writeStream()
5    .outputMode("append")
6    .queryName("randomness")
7    .format("json")
8    .option("checkpointLocation", "randomPath/dir")
9    .option("path", "NewRandomPath/dir")
10   .start();
```

Code 5: Structured Streaming - Complete Streaming Query Example

3.5.3.6 Checkpoint and State internals

In the previous sections, we examine both state and checkpoint high-level concepts. In this section, we perform a more in-depth analysis regarding the internals of checkpointing and state.

In the earlier code snippet, the following line performed the checkpoint.

```
    .option("checkpointLocation", "randomPath/dir")
```

Code 6: Structured Streaming - Checkpointing Mechanism Property

Each query defines a checkpoint location, and while the query is alive, Spark constantly writes metadata to the checkpoint path. Just as a reminder, the checkpoint location is a physical directory pointing to a distributed file system. Checkpoint location stores four types of data:

- **Source Files:** that maintain information about all input sources processed in the query. For instance, a Kafka source preserves information about partitions and offsets.

- **Offsets:** Files that maintain offset details for each particular batch(watermark, timestamp, configurations). Internally it's represented as org.apache.spark.sql.execution. streaming.OffsetSeqLog

- **Commits:** Files that contain commit information about batch metrics .Internally it's represented as org.apache.spark.sql.execution. streaming.CommitLog

- **Metadata:** File that contains metadata information regarding our query as a query ID. Internally it's represented as org.apache.spark.sql.execution.streaming. CommitMetadata

- **State:** These files exist only after stateful stream processing. State data is stored as LZ4-compressed objects(delta and snapshot files) that contain key-value pairs for each state.

Chapter 4

Implementation

In this chapter, we examine all the different aspects and steps of the implementation. These steps can be further divided into five subsections analyzed below: **Streaming Extension Details, Data Parser, Operators, Custom JSON Property File, State Migration Algorithm**. It is essential to state that all our workflows were designed and executed in Rapidminer Studio and Rapidminer Streaming extension, so all our techniques were deployed in this environment.

4.1 Streaming Extension Details

Before scrutinizing our implementation, it is important to highlight some high-level details concerning Rapidminer Streaming Extension. As discussed earlier, the new Streaming Extension Operators were designed as an abstraction layer and can operate independently without defining the Big Data platform at first. The connection component is accountable for the interaction with Big Data platforms, and the connection is achieved via REST APIs. In Spark's case, Apache Livy REST API is employed. So when referring to changes in Spark configuration, later on, we actually refer to modification in Apache Livy configuration or the request sent in Spark at a later stage. Apache Livy operates on batch mode, and the requests are jar files carrying information about Spark cluster configurations and the Operators executed in the requested workflow.

4.2 Data Parser

Firstly, it is essential to ensure that our Operators always have sufficient data resources to execute successfully. One of the first observations after using Apache Spark's Structured

Streaming API, was that on some occasions, a new batch did not generate despite the new data provided as input. It came up when very few data entries were provided as input. For that reason, the design of a custom data parser was necessary, to provide a continuous data stream to Apache Kafka; our primary messaging system. This data parser receives as input .txt or .csv files and "publishes" all entries in a topic defined by the user.

```
1  Stream<String> FileStream = Files.lines(Paths.get(CsvFile));
2  KafkaProducer<String, String> finalProducer = producer;
3
4  FileStream.forEach(line -> {
5  /**
6  * Synchronous Kafka producer to make sure we will not lose any data
7  */
8
9      try {
10         RecordMetadata metadata = finalProducer.send(new
           ↪  ProducerRecord<>(KafkaTopic, line)).get();
11     } catch (Exception e) {
12         LOGGER.log(Level.SEVERE, "Exception occured",e);
13         throw new RuntimeException(e);
14     }
15 });
```

Code 7: Data Parser Code Snippet

Input files are randomly generated files in a specific format to match the requirements of the operators in Rapidminer Studio. Data format follows the pattern "word":"<character _value>" ," value":" <integer_value>"

4.3 Operators

Rapidminer Streaming extension offers a variety of choices regarding Operators with several attributes in each case. All Operators existed beforehand, and the goal was to add some extra properties to complement each one without modifying their core functionality. It is also apparent that the final goal, which is to restart and migrate State, will be achieved by handling stateful operations. On that basis, not all Operators can serve our purpose. More precisely, only **Aggregate Stream, Join Stream, and**

Connect Stream will operate in our final implementation. It is essential to notice that all of these Operators follow the **Sliding Window Model**, and their State perseveres for the duration of window time. The user determines the duration of the window.

Another critical detail is each Operator's name. While the user can rename at will all Operators, by default, Rapidminer Studio does not allow the usage of the same name in more than one instance in the same Process. That means that **all the operators in the same Process possess ID-like unique names**, which is incredibly helpful, as explained in the following sections.

4.4 Custom JSON Property File

The implementation's fundamental challenge was to enable dynamic user interaction, and extra functionality as the whole migration approach utilizes user-defined paths. However, that could not get directly accomplished through Rapidminer's Studio GUI. The design of a custom JSON file with the following properties serves our purpose:

JSON file properties

Jobs:

- Data Type: List
- Description: Contains multiple different jobs with different properties
  - **job name**
    - Data Type: String
    - Description: The name of the job that the properties below apply
  - **checkpoint location**
    - Data Type: String
    - Description: Full checkpoint path where all the required information for restarting is stored
  - **merge**
    - Data Type: Boolean/String("true", "false" values allowed)
    - Description:Flag property that Enables/disables merge_jobs
  - **merge_jobs**
    - Data Type: comma-delimited Strings
    - Description: When enabled, moves all the available (full) paths to the checkpoint location

- **remote**
  - Data Type: Boolean/String("true", "false" values allowed)
  - Description: Flag property that Enables/disables remote_connection
- **remote_connection**
  - Data Type: String in the format "host:port"
  - Description: When enabled, initiates the new Spark Job in the given configuration
- **remote_location**
  - Data Type: String
  - Description: Copy the contents of the checkpoint location on a different DFS location(full path) and use that as a new checkpoint location. Use that property only when the checkpoint location already exists and will not be initiated in this execution. Disable the property by leaving its value empty " " or by removing the property at all.

```
{
    "job name":"newOperators",
    "checkpoint location": "hdfs://45.10.26.123:9000/apps/",
    "merge":"false",
    "merge_jobs":["hdfs://45.10.26.123:9000/apps/Checkpoint/"],
    "remote":"true",
    "remote_connection":"45.10.26.123:58090"
}
```

Code 8: JSON Property file example

4.5 State Migration Algorithm

When a new Spark Job(Streaming Nest with spark connection) starts executing, the first step relies on processing the custom JSON file as described earlier. All *job_names* included in the file are scanned, and all the related properties of the one matching the new running job name are fetched. Some of these properties are critical and required for the successful execution of the job, so our job must find a JSON file match.

The first property to investigate is the *checkpoint location*. This property is executed in the algorithm's last steps but needs to be examined first for context purposes. *Checkpoint location* points to the full path in a persistent storage DFS(in current implementation HDFS), where all the information required for the running workflow to

restart will be stored. More specifically, each stateful Operator creates a subdirectory to store its State by utilizing the Operator's name, since each Operator in Rapidminer Studio has a unique name(in the same Process).

However, this unique operator name cannot directly apply to name a directory due to HDFS name constraints. According to HDFS documentation, all characters in URLs that are not a–z, A–Z, 0–9, '-', '.', '_' or ' ' must first convert to URL encoding and so plenty of special characters including spaces heavily adopted by users will cause issues. For that purpose, the following **polynomialRollingHash** transformed our unique operator name into a unique ID number.

```
static String polynomialRollingHash(String str)
{
    // M:1/M equals collision probability
    // P number of symbols
    final  int p = 10000;
    final int m = (int)(1e9 + 9);
    long power_of_p = 1;
    long hash_val = 0;

    //Loop to calculate the hash value in each string character
    for(int i = 0; i < str.length(); i++)
    {
        hash_val = (hash_val + str.charAt(i) * power_of_p) % m;
        power_of_p = (power_of_p * p) % m;
    }
    return String.valueOf(hash_val);
}
```

Code 9: Polynomial Rolling Hash Function

Each Operator's target path to store its State follows the pattern **checkpoint location\polynomialRollingHash(operatorName)**. By inspecting the hash function's result, the user will not be able to distinguish the actual operators between them. To provide the user with the option to identify each Operator, when an operator name contains parentheses, the contents between them concatenate with the hash result using underscore(_) to form the target path. For example, if **operatorName="SKtest(mytest)"** and **polynomialRollingHash(operatorName)="9123223"** the final path for that operator will be **checkpoint_location\9123223_mytest**.

In case this target path already exists, the state recovers and execution resumes, whereas if it does not exist, Spark creates a new target path, and execution starts while preserving state information. When in production and not in debug mode, an additional directory is created with the formatted **checkpoint location\polynomialRollingHash (operatorName)_RES** to store the result files produced during the execution.

On the other hand, *remote and remote_connection* take effect while handling spark configuration. When the *remote* property is enabled ("true" as value), *remote_connection* is used. *Remote_connection* overwrites the current connection configuration by setting a new host and port. Remember that *remote_connection* property follows the pattern "host:port".This property can prove very useful as each job can be initiated or continued in a different cluster.

Merge, and merge_jobs properties can be applied to concatenate directories in the same HDFS. In case the *merge* property is enabled("true" as value), all the directories included in the *merge_job* property move their content to the *checkpoint location* path. In many scenarios, the user might wish to move the State of a specific operator and continue executing different workflows with different operators.

Finally, the *remote_location* property copies the checkpoint location's contents to a different HDFS. After copying all the contents to the new HDFS, it sets our *checkpoint location = remote location* to handle restart directly from that location. Note that the *checkpoint location* directory should already exist and not get created for the first time during this execution. When the value of this property is empty " " or does not exist at all, it is ignored.

The following pseudocode sums all steps and properties analyzed beforehand.

Algorithm 1: State migration algorithm

```
1  Properties job_properties = {job name, checkpoint location, remote_location,
     merge, merge_jobs ,remote, remote_connection}
2  Input  json_config = {jobName | Φ(jobName) ⊆ job_properties }
3  foreach job do
4      if cur_job_name exists in json_config then
5          fetch cur_job_properties ← Φ(cur_job_name)
6          if remote then
7              Create job in host:port
8          end
9          if merge then
10             foreach merge_jobs directory do
11                 move directory into checkpoint location
12             end
13         end
14         if remote_location exists || not empty then
15             Copy checkpoint location to remote_location
16             Set checkpoint location ← remote_location
17         end
18         foreach stateful operator do
19             if \checkpoint location\polynomialhash(uniqueOperatorName) exists
                then
20                 restore state from directory
21             else
22                 create directory and store state
23             end
24             continue  execution
25         end
26     end
27 end
```

Chapter 5

Experimental Execution

In this chapter, various execution scenarios showcase how the implementation can adapt to any situation. The following examples cover handling (split, merge) workflow and remote state migration cases.

5.1 Detailed workflow description

Rapidminer Studio and streaming extension Operators are applied to design all the workflows. For this demonstration, only **Aggregate state Operators** are incorporated so that all the workflows remain as simple as possible(even the complex ones).

5.1.1 Complex Process(Streaming Nest name - complex_Demo)

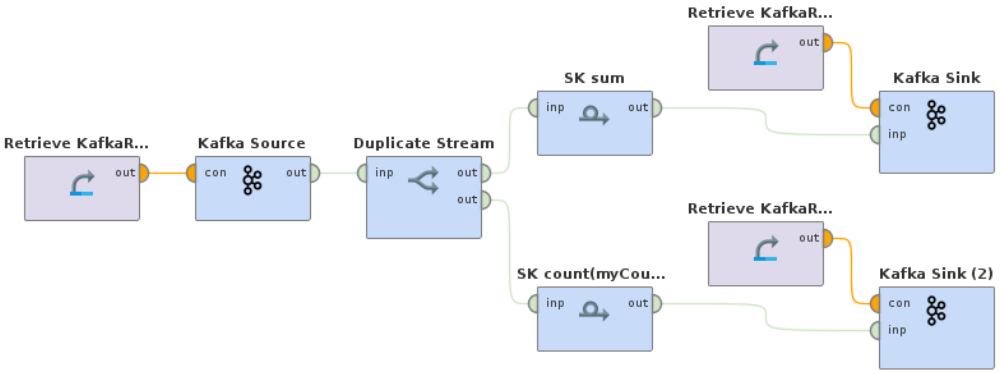

FIGURE 5.1: Complex Process - complex_Demo Workflow

Operator Parameters:

- **Kafka Source:**
  - topic: SKinput
- **SK sum:**
  - key: word
  - value key: value
  - window length: 2
  - function: Sum
- **SK count(myCount):**
  - key: word
  - value key: value
  - window length: 5
  - function: Count
- **Kafka Sink:**
  - topic: SKout
- **Kafka Sink (2):**
  - topic: SKout2

5.1.2 Simple Process(overview)

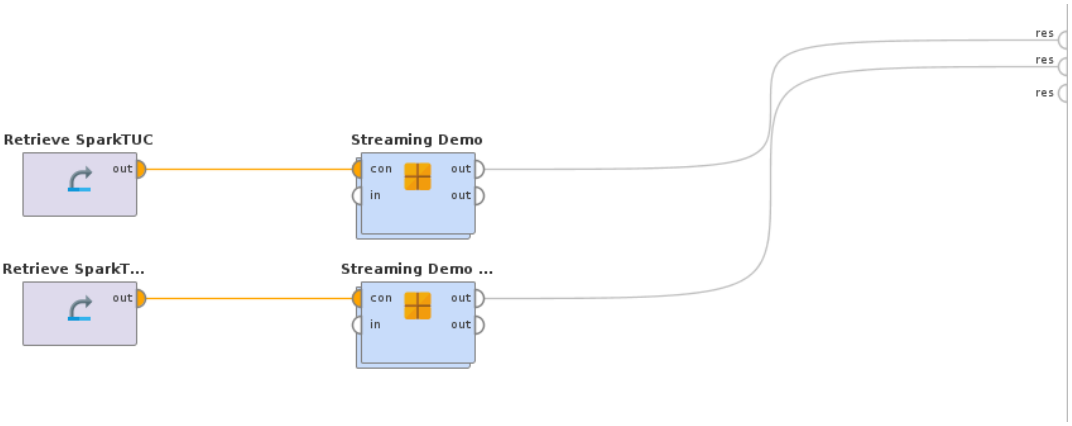

FIGURE 5.2: Simple Process - Overview

The Process above consists of two different Streaming Nest with the names simpleCount_Demo and simpleSum_Demo, both of which are analyzed below

Streaming Nest name - simpleCount_Demo

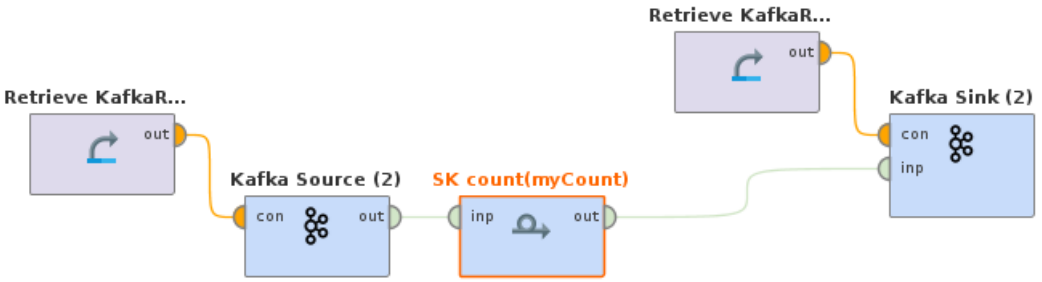

FIGURE 5.3: Simple Process - simpleCount_Demo Workflow

Operator Parameters:

- **Kafka Source (2):**
  - topic: SKinput
- **SK count(myCount):**
  - key: word
  - value key: value
  - window length: 5
  - function: Count
- **Kafka Sink (2):**
  - topic: SKout2

Streaming Nest name - simpleSum_Demo

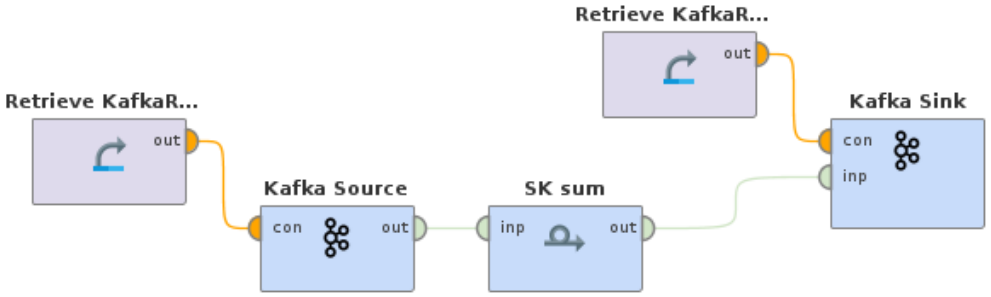

FIGURE 5.4: Simple Process - simpleSum_Demo Workflow

Operator Parameters:

- **Kafka Source:**
  - topic: SKinput
- **SK sum:**
  - key: word
  - value key: value
  - window length: 2
  - function: Sum
- **Kafka Sink:**
  - topic: SKout

5.2 Setup

Extra logic must be added to the original code to verify that our checkpoint mechanism works. The main reason lies in the architecture and working principles of Structured Streaming API, in conjunction with windowing and watermarking used by the stateful Operators in Rapidminer Studio. We add the following code to *SparkAggregateTransformerTranslator.java*.

```
my_stream.writeStream()
    .format("console")
    .outputMode("complete")
    .option("truncate", false)
    .option("checkpointLocation",
    ↪  ""+this.checkpointName+finalOperatorName+"")
    .start();
```

Code 10: Additional Code for debugging

The format employed is **console** (only for debugging purposes, as mentioned in the documentation) to print all results in the console. As outputMode, the **complete** option outputs the entire result table, every time that new entries trigger the system(and a new batch generates).

Another important detail is that throughout the tests, the custom Data Parser examined in the previous section is in use so that all operators have a sufficient amount of data to process every time.

5.3 Execution

5.3.1 Split case

In this scenario, the job initiated is *complex_Demo* with the following properties.

```
{
    "job name":"complex_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal⌋
    ↪  ogerakis/DemoComplexCheckpoint/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
    "remote_location":""
}
```

Code 11: Split Case - complex_Demo properties

After a while, the running job gets terminated, and the console output is

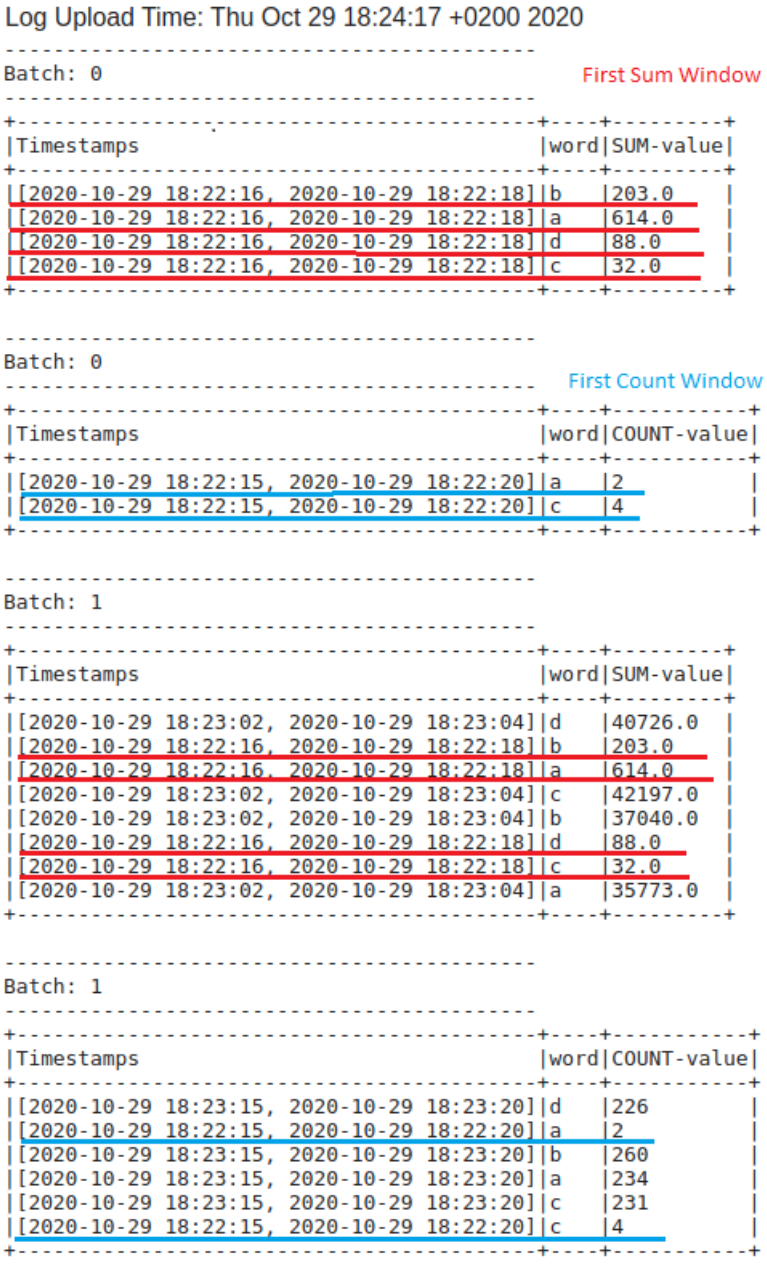

FIGURE 5.5: Split Case - Initial Execution

As previously mentioned, after the generation of every new batch, the whole result table is printed as well as the time window with the corresponding timestamps to monitor both new and old entries. In the output above, after the first execution, the red values represent the sum operator's first window values, whereas the values in blue represent the first window values of the count operator.

The checkpoint location provided in JSON is inspected to confirm that the checkpoint worked as expected.

```
skalogerakis@clu04:~$ hadoop fs -ls /user/skalogerakis/DemoComplexCheckpoint/
Found 2 items
drwxrwxrwx   - livy hadoop          0 2020-10-29 18:23 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount
drwxrwxrwx   - livy hadoop          0 2020-10-29 18:22 /user/skalogerakis/DemoComplexCheckpoint/168998164
skalogerakis@clu04:~$
```

FIGURE 5.6: Split Case - HDFS checkpoint directory

For reasons of completeness, directory contents of a random operator are listed.

```
skalogerakis@clu04:~$ hadoop fs -ls /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount
Found 5 items
drwxr-xr-x   - livy hadoop          0 2020-10-29 19:03 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount/commits
-rw-r--r--   3 livy hadoop         45 2020-10-29 18:22 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount/metadata
drwxr-xr-x   - livy hadoop          0 2020-10-29 19:03 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount/offsets
drwxr-xr-x   - livy hadoop          0 2020-10-29 18:22 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount/sources
drwxr-xr-x   - livy hadoop          0 2020-10-29 18:23 /user/skalogerakis/DemoComplexCheckpoint/165830172_myCount/state
skalogerakis@clu04:~$
```

FIGURE 5.7: Split Case - HDFS operator checkpoint directory

The same *complex_Demo* job restarts, and this time the execution of both sum and count operators resumes from the last successful batch submitted. So the values of the previous execution are preserved.

Log Upload Time: Thu Oct 29 18:36:58 +0200 2020

Batch: 2 — First Sum Window (Prev. execution)

Timestamps	word	SUM-value
[2020-10-29 18:23:02, 2020-10-29 18:23:04]	d	40726.0
[2020-10-29 18:22:16, 2020-10-29 18:22:18]	b	203.0
[2020-10-29 18:23:56, 2020-10-29 18:23:58]	b	45001.0
[2020-10-29 18:22:16, 2020-10-29 18:22:18]	a	614.0
[2020-10-29 18:23:02, 2020-10-29 18:23:04]	c	42197.0
[2020-10-29 18:23:02, 2020-10-29 18:23:04]	b	37040.0
[2020-10-29 18:23:56, 2020-10-29 18:23:58]	d	40487.0
[2020-10-29 18:22:16, 2020-10-29 18:22:18]	d	88.0
[2020-10-29 18:23:56, 2020-10-29 18:23:58]	c	51642.0
[2020-10-29 18:22:16, 2020-10-29 18:22:18]	c	32.0
[2020-10-29 18:23:02, 2020-10-29 18:23:04]	a	35773.0
[2020-10-29 18:23:56, 2020-10-29 18:23:58]	a	46147.0

Batch: 2 — First Count Window (Prev. execution)

Timestamps	word	COUNT-value
[2020-10-29 18:24:05, 2020-10-29 18:24:10]	b	236
[2020-10-29 18:24:05, 2020-10-29 18:24:10]	d	210
[2020-10-29 18:23:15, 2020-10-29 18:23:20]	d	226
[2020-10-29 18:22:15, 2020-10-29 18:22:20]	a	2
[2020-10-29 18:24:05, 2020-10-29 18:24:10]	c	196
[2020-10-29 18:23:15, 2020-10-29 18:23:20]	b	260
[2020-10-29 18:23:15, 2020-10-29 18:23:20]	a	234
[2020-10-29 18:23:15, 2020-10-29 18:23:20]	c	231
[2020-10-29 18:24:05, 2020-10-29 18:24:10]	a	198
[2020-10-29 18:22:15, 2020-10-29 18:22:20]	c	4

FIGURE 5.8: Split Case - Restart complex_Demo workflow

As the next step, the **Process Simple** will execute containing *simpleCount_Demo* and *simpleSum_Demo* Streaming Nest jobs consecutively. The *simpleCount_Demo* contains

the same count operator from the complex workflow, whereas *simpleSum_Demo* contains the sum operator. The properties for each job are defined below.

```
{
    "job name":"simpleCount_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal
    ↪  ogerakis/DemoComplexCheckpoint/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
    "remote_location":""
}
```

Code 12: Split Case - simpleCount_Demo properties

```
{
    "job name":"simpleSum_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal
    ↪  ogerakis/DemoComplexCheckpoint/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_location":""
}
```

Code 13: Split Case - simpleSum_Demo properties

Despite splitting our workflow, each operator's execution can resume independently from the last point of failure and continue with new entries in the result table.

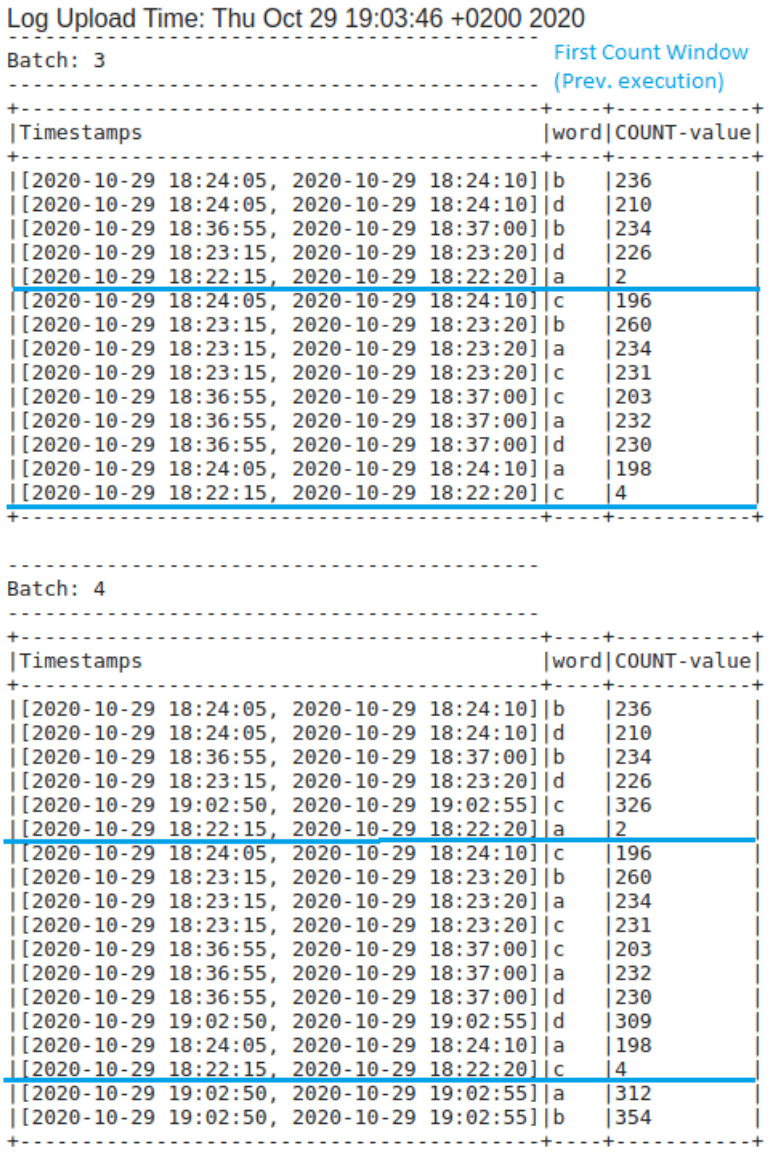

FIGURE 5.9: Split Case - Restart simpleCount_Demo workflow

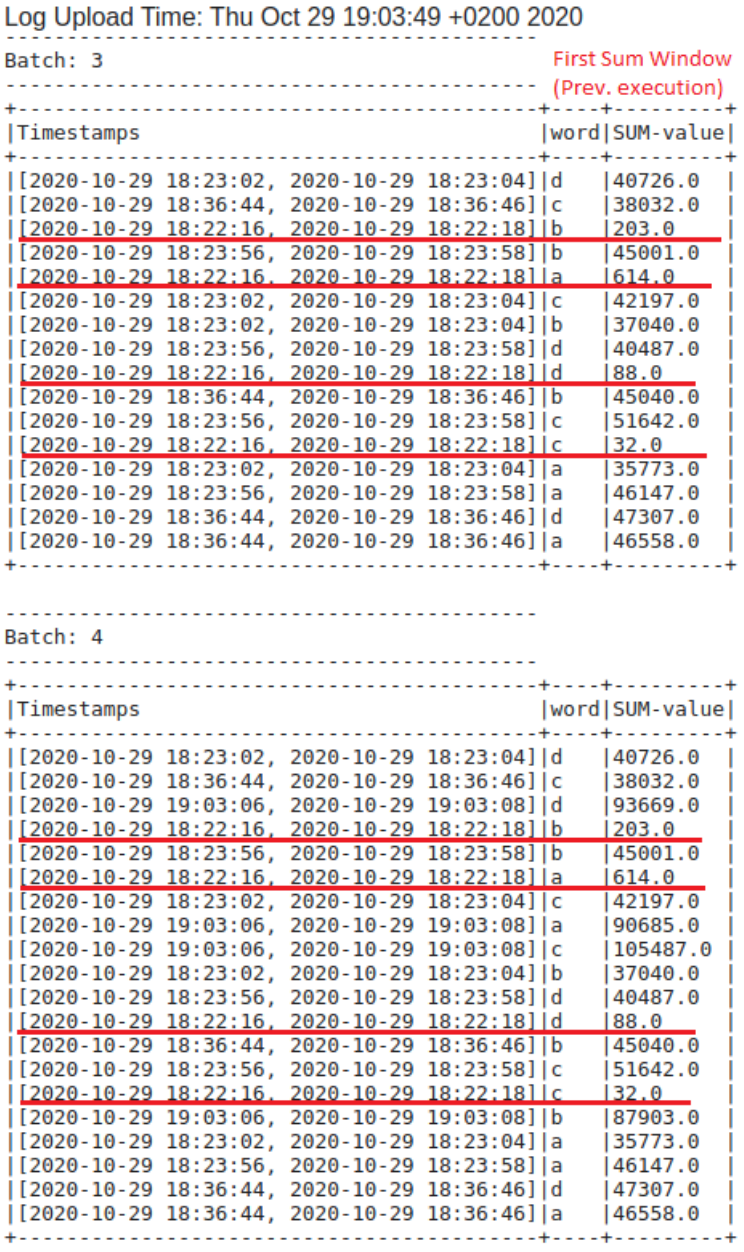

FIGURE 5.10: Split Case - Restart simpleSum_Demo workflow

Note that the user can modify the window length parameter during restarts without causing any errors to maintain a longer/shorter window state.

5.3.2 Merge

This execution scenario includes executing simpler workflows and merging some of the operators to run a more complex workflow. For that reason, the **Process Simple** executes with the following properties for *simpleCount_Demo* and *simpleSum_Demo* jobs, respectively.

```
{
    "job name":"simpleCount_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal⌋
    ↪  ogerakis/DemoSimpleCountCheckpoint/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
    "remote_location":""
}
```

Code 14: Merge Case - simpleCount_Demo properties

```
{
    "job name":"simpleSum_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal⌋
    ↪  ogerakis/DemoSimpleSumCheckpoint/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_location":""
}
```

Code 15: Merge Case - simpleSum_Demo properties

Since the checkpoint location points to a different place from the previous example, all aggregate computations should start from the beginning. The output console confirms the hypothesis.

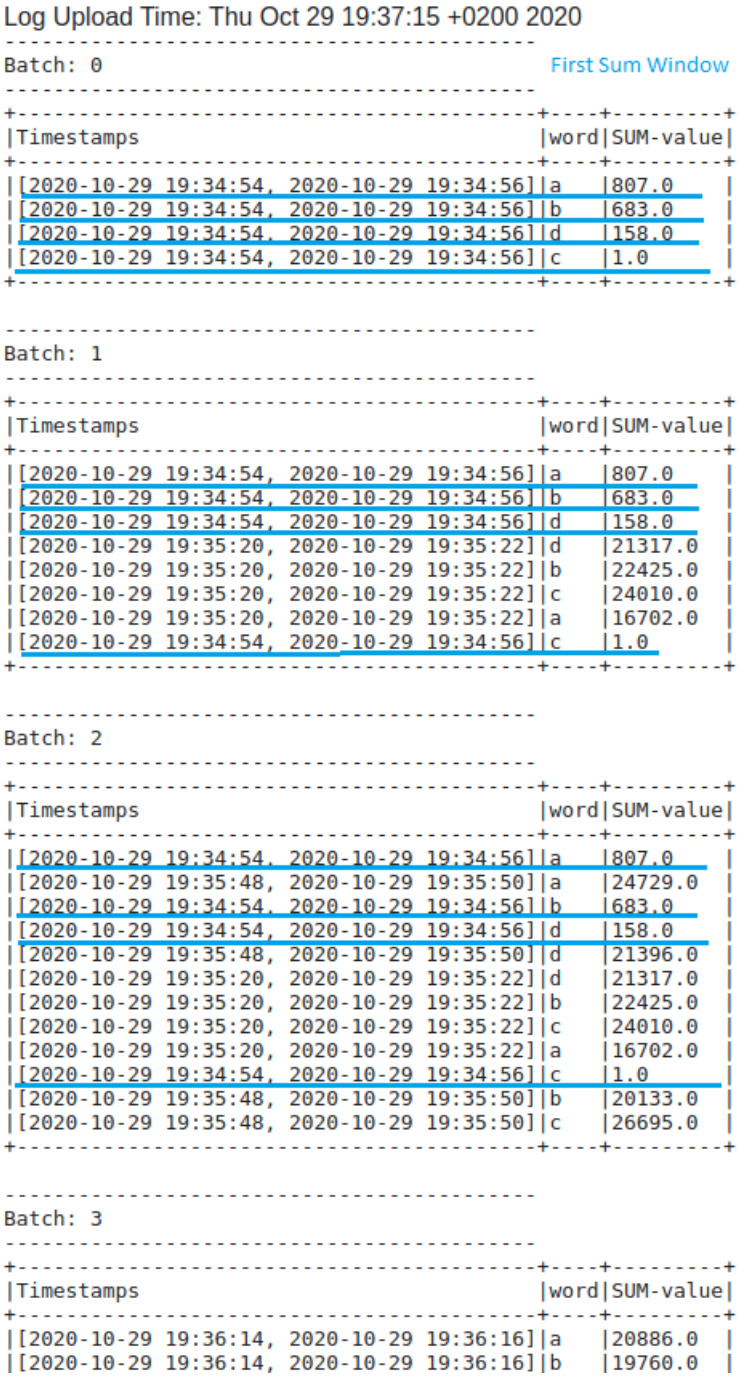

FIGURE 5.11: Merge Case - Initial simpleSum_Demo workflow execution

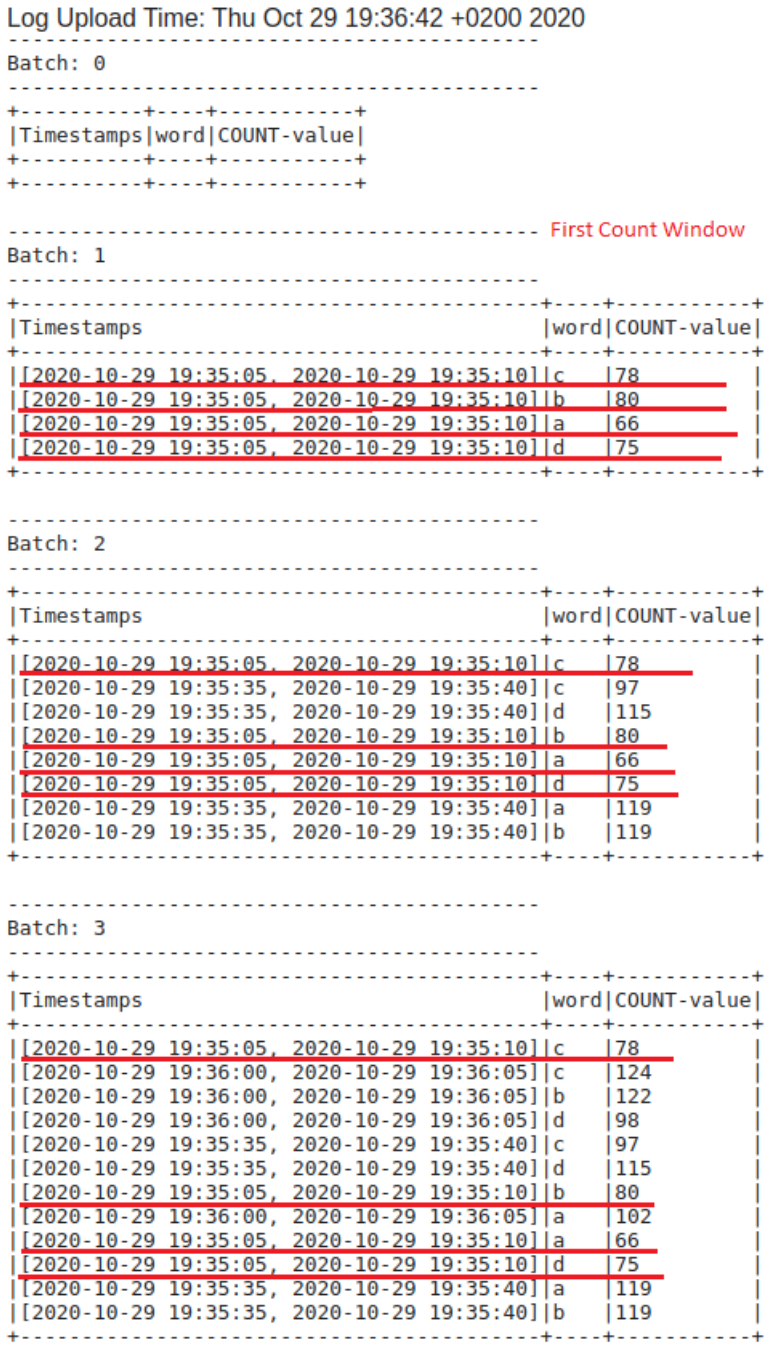

FIGURE 5.12: Merge Case - Initial simpleCount_Demo workflow execution

The **Complex Process** then deploys, containing the count and sum operators from *simpleCount_Demo* and *simpleSum_Demo*, respectively. To successfully restore and resume the execution, all operators must exist under the same directory, used as the checkpoint location. For that purpose, the jobs mentioned above execute with the following properties.

```
{
    "job name":"complex_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal
    ↪  ogerakis/MergeCheckpointDemo/",
    "merge":"true",
    "merge_jobs":["hdfs://clu01.softnet.tuc.gr:8020/user/skalogerakis/
    ↪  DemoSimpleSumCheckpoint/168998164/","hdfs://clu01.softnet.tuc.
    ↪  gr:8020/user/skalogerakis/DemoSimpleCountCheckpoint/165830172_
    ↪  myCount/"],
    "remote":"false",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
    "remote_location":""
}
```

Code 16: Merge Case - complex_Demo properties

Count aggregate operator terminated in batch 3, whereas sum aggregate ended in batch 4. Both initiate the **Complex Process** execution from lastbatchID + 1 while maintaining the previously generated values(Figure 4.14).

Also the HDFS path, hdfs://clu01.softnet.tuc.gr:8020/user/skalogerakis/MergeCheckpointDemo/ the merged Operators also verify the correctness of the approach.

```
skalogerakis@clu04:~$ hadoop fs -ls /user/skalogerakis/MergeCheckpointDemo
Found 2 items
drwxrwxrwx   - livy hadoop          0 2020-10-29 19:34 /user/skalogerakis/MergeCheckpointDemo/165830172_myCount
drwxrwxrwx   - livy hadoop          0 2020-10-29 19:35 /user/skalogerakis/MergeCheckpointDemo/168998164
skalogerakis@clu04:~$
```

Figure 5.13: Merge Case - HDFS checkpoint directory

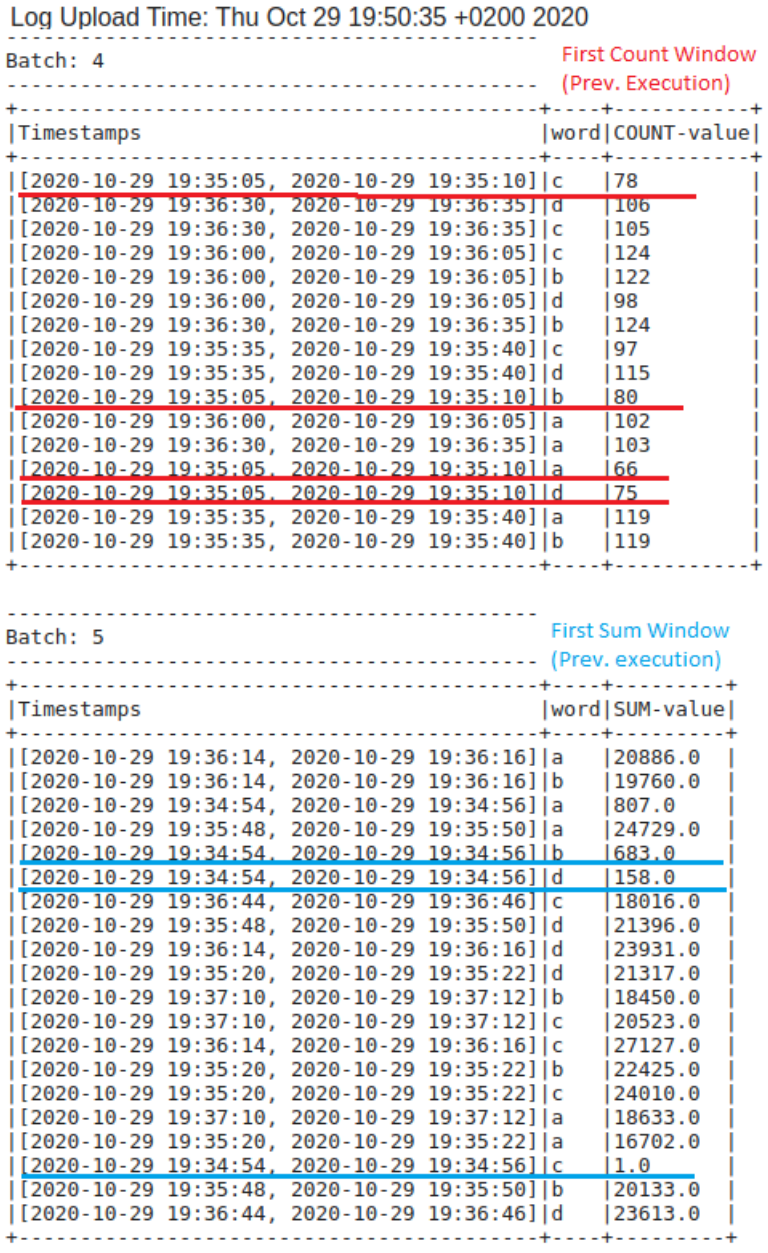

FIGURE 5.14: Merge Case - Restart complex_Demo workflow

5.3.3 State and job migration

The implementation supports both state and job migration, which can occur independently—the property *remote_location* copies (and transfers) state from one HDFS cluster to another. Checkpoint location copies contents from the original HDFS into a *remote_location* path, located in a different cluster.

For this example, we execute *simpleCount_Demo* Streaming Nest(found in **Process Simple**) with the following properties.

```
{
    "job name":"simpleCount_Demo",
    "checkpoint location": "hdfs://clu01.softnet.tuc.gr:8020/user/skal
    ↪  ogerakis/DemoMigration/",
    "merge":"false",
    "merge_jobs":[""],
    "remote":"false",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
    "remote_location":""
}
```

Code 17: Remote Case - simpleCount_Demo properties

Execution initiates from the beginning and after batch one is terminated.

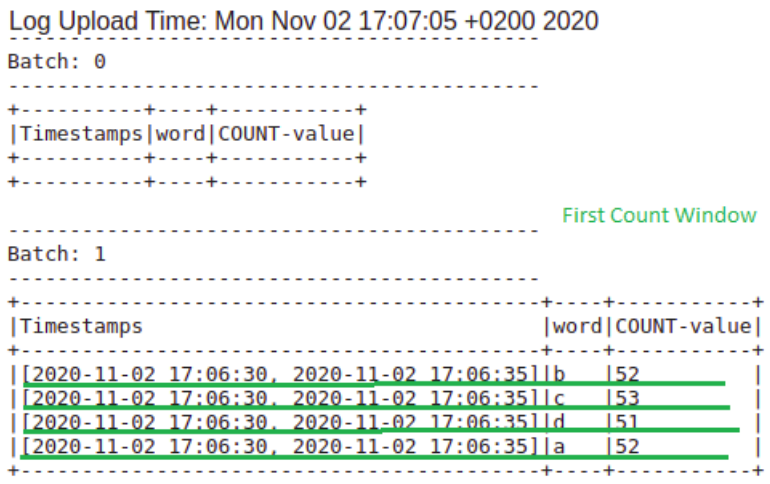

Figure 5.15: Remote Case - Initial workflow execution

Then, the *remote_location* property changes, and the job restarts.

```
{
    "remote_location":"hdfs://45.10.26.123:9000/apps/Checkpoint/DemoRe
    ↪  moteLocationMigration/"
}
```

Code 18: Remote Case - Modified remote_location property

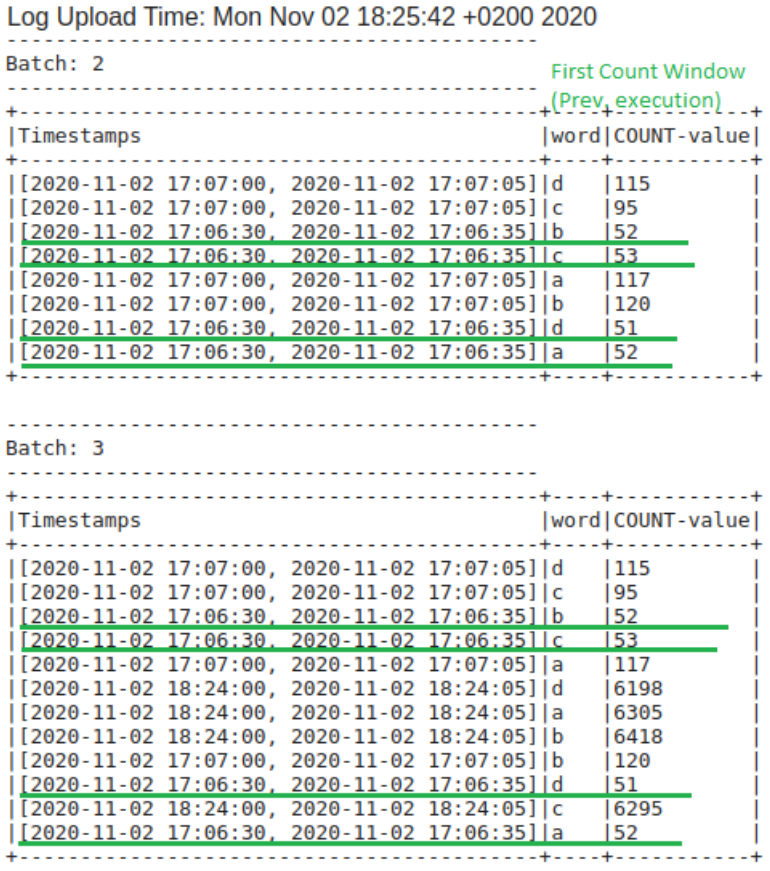

FIGURE 5.16: Remote Case - Remote workflow restart

The execution works as expected, and the first batch generated is batch 2, resuming from the last successfully executed batch. Note that Spark configuration did not change from the previous examples during that test, and the latest job submission transpires in the same cluster as previously. This fact proves that Spark and HDFS are independent and can operate under different clusters with success.

Job migration between clusters can happen with the use of *remote* and *remote_connection* parameters. When remote is enabled, property *remote_connection* will overwrite Spark's existing host and port configurations.

For instance, the following properties will execute the job in a remote spark cluster with configurations port=clu01.softnet.tuc.gr, host=8999.

```
{
    "remote":"true",
    "remote_connection":"clu01.softnet.tuc.gr:8999",
}
```

CODE 19: Cluster migration property

Nevertheless, both clusters must operate under the same(or similar) Spark version. Spark and Structured Streaming API can undergo significant changes from one version to another, leading to unexpected behaviour and failures during the execution.

5.3.4 Production

As mentioned at the beginning of this chapter, our examples execute on the console, used only for debugging purposes. The following code showcases one of the possible alternatives for production mode.

```
my_stream.writeStream()
         .format("json")
         .outputMode("append")
         .option("path",
         ↪  ""+this.checkpointName+finalOperatorName+"_RES")
         .option("checkpointLocation",
         ↪  ""+this.checkpointName+finalOperatorName+"")
         .start();
```

Code 20: Production - JSON output file

For each operator, in this case, a new directory is created with the name **/finalOperatorName_RES**(more information regarding finalOperatorName in Implementation Chapter), where only newly appended rows in the dynamic result table since the last trigger are outputted in JSON files.

```
skalogerakis@clu04:~$ hadoop fs -ls /user/skalogerakis/ProductionCheckpoint
Found 2 items
drwxr-xr-x   - livy hadoop          0 2020-10-30 11:05 /user/skalogerakis/ProductionCheckpoint/165830172_myCount
drwxr-xr-x   - livy hadoop          0 2020-10-30 11:06 /user/skalogerakis/ProductionCheckpoint/165830172_myCount_RES
```

Figure 5.17: Production Case

Chapter 6

Challenges and Future Work

State and job migration stages were the most challenging sections of the implementation.

During state migration, *remote_location* property induces massive overhead slowing down the execution. As a reminder, *remote_location property* copies an HDFS directory's contents into a remote and new HDFS directory. A documented problem concerning small files in HDFS can provoke such problematic behavior.

Firstly, the checkpointing mechanism in Structured Streaming API produces thousands of small files. Moreover, as stated beforehand, in the HDFS description, it is optimized to operate under large files, with small files causing lots of seeks, which proves very inefficient. Small files are considered those that are significantly smaller than the default HDFS block size. If we are storing many small files, the matter is that HDFS cannot handle lots of files. The primary reason is that each file, directory, and block in HDFS serves as an object within NameNode memory, each of which occupies 150 bytes. For example, 10 million files, each using only a block, would use about 3 gigabytes of memory. From that point, scaling up much beyond proves extremely inefficient and infeasible in many cases.

Note that our initial approach *merge_locations* property also copied the contents of directories into the chosen final directory of the same HDFS, which, however, caused the same issue. Transferring the initial directories into the final target directory, solved the problem using Hadoop move command. This solution worked instantly, without additional overhead to the execution.

Compatibility was undoubtedly challenging and caused problems during the job migration stage. As mentioned in the last chapter, only clusters operating under the same Spark versions(or similar versions) can perform job migration because Spark can undergo major updates between updates. During our tests, two different clusters were accessible, using Spark versions 2.4.5 and 2.3.2, respectively. The implementation operated as expected after fetching the correct dependencies for each version. Nevertheless, no implementation would simultaneously work for both versions, and so a complete example, including the *remote* property, could not be executed properly.

Future work could introduce an optimal way to manage small file problems in HDFS so that the implementation could operate under production. Some proposed solutions, incorporate different compaction techniques to handle and discard some of the small files, use of different DFS(HopsFS claims that is designed on top of HDFS to tackle small file issue) or even use of a different type of storage(such as HBase, Cassandra, ScyllaDB).

Bibliography

[1] Json - introduction. URL https://www.w3schools.com/js/js_json_intro.asp.

[2] Json files. URL https://en.wikipedia.org/wiki/JSON.

[3] Bounded-unbounded streams. URL https://flink.apache.org/flink-architecture.html.

[4] Window models. URL https://www.researchgate.net/figure/Window-models-illustrations-i-landmark-ii-sliding-iii-damped_fig1_325977106.

[5] Streaming data. URL https://aws.amazon.com/streaming-data/.

[6] String hashing and polynomial rolling hashing. URL https://cp-algorithms.com/string/string-hashing.html?fbclid=IwAR3CATNn2O-PPeIML3W17SF9C4JU8peLE5y9Hldf9Q5ntMd4WyZHifnL21Q.

[7] Polynomial rolling hashing. URL https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/.

[8] What is hdfs. URL https://phoenixnap.com/kb/what-is-hdfs.

[9] Hdfs tutorial. URL https://www.simplilearn.com/tutorials/hadoop-tutorial/hdfs.

[10] Apache hdfs-overview. URL https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[11] Hdfs image. URL https://www.datasciencecentral.com/profiles/blogs/hadoop-for-beginners.

[12] Apache kafka explained. URL https://www.youtube.com/watch?v=JalUUBKdcA0&t=1029s&ab_channel=Finematics.

[13] Apache kafka overview. URL https://kafka.apache.org/.

[14] Apache hdfs-overview. URL https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[15] Apache livy overview. URL https://livy.apache.org/.

[16] Apache livy tutorial. URL https://www.statworx.com/de/blog/access-your-spark-cluster-from-everywhere-with-apache-livy/.

[17] Rapidminer studio review. URL https://reviews.financesonline.com/p/rapidminer-studio/.

[18] Rapidminer studio wiki. URL https://en.wikipedia.org/wiki/RapidMiner.

[19] Rapidminer streaming extension. URL https://zenodo.org/record/4064296#.X46VC9AzZqM.

[20] Apache spark presentation. URL https://www.youtube.com/watch?v=9U4ED7KQwlE&feature=emb_title&ab_channel=Databricks.

[21] Apache spark overview. URL https://spark.apache.org/.

[22] Structured streaming-basic analysis. URL https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html.

[23] Structured streaming intro. URL https://www.pavanpkulkarni.com/blog/18-spark-structured-streaming-intro/.

[24] Structured streaming stateful processing. URL https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html.

[25] Checkpoint storage in structured streaming. URL https://www.waitingforcode.com/apache-spark-structured-streaming/checkpoint-storage-structured-streaming/read.

[26] Internals stateful stream processing. URL https://databricks.com/session_eu19/the-internals-of-stateful-stream-processing-in-spark-structured-streaming.

[27] Structured streaming stateful processing. URL https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html.

[28] Structured streaming api- overview. URL https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[29] Hdfs small file problem. URL https://blog.cloudera.com/the-small-files-problem/.

[30] Hdfs small files best practises. URL https://www.agilelab.it/management-of-small-files-on-hdfs-problem-analysis-and-best-practices/.