# Auditable Data Provenance in Streaming Data Processing

**Afonso Basto de Almeida Ribeiro Bate**

Thesis to obtain the Master of Science Degree in

**Computer Science and Engineering**

Supervisors: Prof. Luís Manuel Antunes Veiga
Prof. Paulo Carreira

## Examination Committee

Chairperson: Prof. Diogo Manuel Ribeiro Ferreira
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Nuno Miguel Carvalho dos Santos

**November 2023**

# Acknowledgments

I would like to start by extending my deepest gratitude to my family, especially my parents and grandparents, for their unwavering support, boundless care, and for providing me with the tools that enabled me to embark on this educational journey culminating in this work. Without them this would not be possible.

To my beloved girlfriend, your love and encouragement were my pillars of strength during this challenging phase.

To my friends, thank you for the camaraderie and for being there during the good and bad times. Special thanks to those who walked this student life path beside me, understanding the trials and triumphs of this academic adventure.

I want to express my gratitude, as well, to my dissertation supervisors Prof. Luís Veiga and Prof. Paulo Carreira for their insight, support and sharing of knowledge that has made this work possible. I would also like to acknowledge that this work was supported by national funds through IAPMEI via SmartRetail project (ref. C6632206063-00466847).

Finally, I'm also grateful to anyone, known or unknown, who played a part in contributing to this work and my personal growth as a student and person.

To each and every one mentioned, and all who've been part of my journey, I offer a heartfelt and resounding thank you.

# Abstract

Stream processing is becoming more and more used in the field of Big Data analysis. The need to analyze unbounded data and data that needs to be processed in real-time has made batch processing obsolete in these cases. However, data processing systems are not perfect and errors or incorrect computations can lead to wrong results. When these wrong results occur we must inspect the flow from the input until the result while understanding all the transformations that lead to this outcome. This is useful both to understand why errors occur, but also to justify certain results. However, due to the dynamic nature of stream processing, data provenance in these systems poses a greater challenge. The works and proposed solutions for this challenge are in its majority incomplete, since they don't provide fine-grained provenance, and those who do, are limited because they are not a general solution to the problem. In this work, we present a survey of the current state-of-the-art in stream processing and data provenance and lineage. We also propose a solution to the problem of providing fine-grained provenance in a stream processing system. Our proposed solution comprises three interconnected pipelines, each with four Python modules. These pipelines empower real-time data provenance, efficient people tracking, zone counting, and instant alerts. Testing occurred in controlled local and distributed network environments, emphasizing performance metrics, like latency, throughput, and resource utilization. Our system excels in preserving data provenance, offering invaluable insights for complex real-world scenarios. The results illustrate the system's resilience and efficiency, particularly in preserving data provenance, thereby offering precise insights into complex real-world scenarios. This achievement marks a significant step toward ensuring reliable data tracing and management in demanding stream processing systems.

# Keywords

Stream Processing, Data Provenance

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1 Introduction

Contents

Data processing has been around for millennia in some form or another, but in recent times real-time information has become more valuable with the increased use of information-collecting devices. Nowadays, from Internet of Things sensors to smartphones, it's easier than ever to obtain information and the amounts of information gathered are greater than ever. However, this raw data is some times of limited value and some of that information loses value very fast after being collected.

The typical data analysis technique of batch processing does not satisfy these needs and this is where stream processing comes into play. Stream processing queries continuous data streams and provides results of those queries almost in real-time. One of the main advantages of stream processing over batch processing is the ability to extract meaningful and timely insights from unbounded data.

Stream processing can handle and process never-ending stream of events, while in batch processing you need to store the data, stop its collecting, and only then can you process it. Another big advantage is the use of fewer resources, especially memory resources since in batch processing the data needs to be stored in order to be processed, while in stream processing, data is processed after being collected and then it's discarded, all in quick succession. This allows stream processing to handle larger amounts of data. Real-time fraud and anomaly detection, Internet of Things edge analytics, and real-time personalization, marketing, and advertising, constitute some of the most common use cases for stream processing.

## 1.1 Motivation

In today's data-driven world, the continuous flow of streaming data from various sources has become ubiquitous. From financial transactions and social media updates to sensor readings and video feeds, data streams provide valuable insights in real time [1–4]. However, as the volume and complexity of streaming data continue to grow, so do the challenges associated with ensuring its reliability, integrity, and accountability [5].

Provenance, the ability to trace the origin and history of data, has gained prominence as a critical aspect of data management and analytics. While traditional data provenance techniques are well-established for batch processing scenarios, the unique characteristics of streaming data introduce novel complexities and demand innovative solutions [6].

The motivation behind this dissertation stems from the pressing need for auditable data provenance in streaming data processing. We are living in an era where the consequences of erroneous or biased decisions based on streaming data can be profound, ranging from financial losses and privacy breaches to public safety concerns. As data streams become central to critical decision-making processes in domains such as finance, healthcare, and security, the ability to provide transparent and verifiable evidence of data lineage becomes paramount.

In the realm of streaming data, ensuring the trustworthiness of information is a formidable challenge. Data can arrive from numerous sources, undergo multiple transformations, and be subject to various analytics in real time. Auditable data provenance serves as a safeguard against data corruption, tampering, or unauthorized alterations. By tracing the journey of data through the processing pipeline, we can pinpoint the source of errors or anomalies and restore confidence in the data's accuracy.

The ability to reproduce results and conduct forensic investigations is essential in scenarios where critical decisions are based on streaming data. Auditable data provenance ensures that results obtained in real time can be replicated and verified at a later stage. This capability is invaluable for incident response, forensic analysis, and auditing the performance of data processing pipelines.

However, real-time data processing systems are complex, comprising multiple components and algorithms. When unexpected results or anomalies occur, diagnosing the root cause can be daunting. Auditable data provenance acts as a diagnostic tool, enabling data engineers and analysts to retrace the processing steps, identify discrepancies, and debug the system effectively. Furthermore, it aids in anomaly detection by highlighting deviations from expected data flow patterns.

Another substantial challenge in the area of stream processing is the increase of regulatory frameworks imposing stringent data handling and privacy requirements, organizations must demonstrate compliance with data protection laws [7]. Auditable data provenance provides a means to track data access, sharing, and processing, facilitating compliance auditing and regulatory reporting. It enables organizations to answer critical questions about who accessed the data, how it was used, and whether privacy policies were adhered to.

By addressing the challenges of auditable data provenance in streaming data processing, we can contribute to the advancement of the field. Developing innovative techniques, algorithms, and systems for tracking, recording, and presenting data lineage in real-time environments extends the frontiers of data stream processing and strengthens its applicability across diverse domains.

In this context, we aim to explore, design, and implement auditable data provenance solutions tailored to the dynamic nature of streaming data. By addressing the motivations outlined above, this research aspires to provide a comprehensive framework for ensuring data trustworthiness, accountability, and transparency in the era of continuous data streams.

## 1.2 Shortcomings of current solutions

While the field of data provenance has made significant strides in ensuring the integrity and lineage of data in traditional batch processing scenarios, these advancements have not seamlessly translated to the realm of streaming data processing. Current solutions, although valuable, exhibit several shortcomings when confronted with the unique challenges posed by streaming data. This section discusses some of the key limitations of existing approaches.

**Latency and overhead** are, perhaps, the biggest challenge in stream processing. Existing data provenance solutions for streaming data often introduce significant latency and processing overhead [8–11]. In real-time applications, where timely decision-making is critical, these delays can be detrimental. Stream processing systems require low-latency solutions that can keep pace with the rapid influx of data while still recording comprehensive provenance information.

Besides the growing influx of data, the **volume of streaming data** is also increasing. And as the volume of streaming data grows, scalability becomes a major concern. Many current solutions struggle to scale efficiently with increasing data rates. Scalability issues can lead to dropped data, missed provenance records, and incomplete lineage tracking, undermining the reliability of the provenance system.

Another shortcoming in stream processing systems is the **handling of complex events**. Streaming data often involves complex event patterns and dependencies. Current solutions sometimes lack the flexibility to capture and represent these intricate relationships. This limitation hampers the ability to reconstruct the full data lineage accurately, especially in scenarios where events trigger cascading data transformations.

**Privacy and security** concerns associated with streaming data must also be adequately addressed by a solution. Recording fine-grained provenance information can inadvertently expose sensitive data or breach privacy regulations, necessitating the development of privacy-preserving provenance techniques.

**The dynamic semantics** of streaming data can also be a challenge, as data schemas, structures, and formats can change over time. Current solutions may rely on static schemas, making it challenging to adapt to evolving data semantics. A robust provenance system should be able to accommodate these dynamic changes seamlessly and be capable of capturing and representing provenance information from a wide variety of external systems and data sources. This will help ensure end-to-end lineage tracking in complex, heterogeneous environments.

And, finally, **resource usage** is the last common shortcoming in the existing stream processing solutions. In resource-constrained environments, such as edge computing or IoT devices, current solutions may consume excessive resources, including memory and processing power. This inefficiency can hinder the deployment of provenance systems in resource-limited scenarios.

## 1.3 Proposed solution

Understanding the shortcomings of the existing solutions for provenance in stream processing systems is crucial. It is by understanding these limitations that we can try to address them and come up with innovative approaches and solutions designed to overcome them and provide a solid foundation for ensuring data trustworthiness and transparency in the era of streaming data. By studying the existing solutions and even inspiring ourselves in some of them, we designed a framework for providing auditable provenance in a stream processing system that we believe presents acceptable performance in latency, throughput and resource utilization, while also addressing the scalability, handling of complex events, dynamic semantics and, privacy and security concerns.

Our proposed solution revolves around a sophisticated multi-pipeline framework designed for the continuous processing of video streams while prioritizing data integrity, transparency, and result accuracy. Recognizing that alerts are likely infrequent, the system prioritizes precision, even at the potential expense of overall performance, specifically, during alert confirmation. At the heart of this framework is a commitment to real-time data processing. It operates multiple pipelines that ensure immediate response to incoming data, allowing rapid detection and action when necessary. This responsiveness is crucial for identifying anomalies promptly. It's important to note that every pipeline is composed of 4 modules (Module 0, Module 1, Module 2 and Module 3), with mostly similar executions and functionalities across pipelines, in our use case, and by dividing each pipeline in modules with specific objectives we promote scablibility and facilitate the addition of new functionalities at a later time.

One of the fundamental aspects of the solution is Module 0, which plays a pivotal role in safeguarding data integrity. It achieves this by hashing both the data source and the code files of subsequent modules. This hash-based verification mechanism ensures that the results generated align faithfully with the underlying codebase.

Module 1, responsible for anomaly detection, is a proactive component of the system. It actively scans video frames for irregularities, such as instances where individuals unexpectedly stop being detected within the frame. If such an anomaly is detected, Module 1 triggers the automatic activation of the second pipeline. The second pipeline capitalizes on periodic checkpoints and stored frames to recreate and validate the results generated in the first pipeline. This replay mechanism is essential in determining the legitimacy of alerts, even if they are rare. The commitment to accuracy outweighs potential performance impacts.

To further ensure auditability and transparency, the third pipeline is available, although it is manually initiated by the user. This pipeline meticulously examines the outcomes produced in the second pipeline if an alert is indeed confirmed, providing an exhaustive view of data lineage and processing steps. It is a valuable resource for auditing and in-depth result analysis.

Throughout this intricate framework, Apache Kafka serves as the communication backbone, facilitating seamless data exchange and tracking between pipelines and modules. This choice underscores the commitment to transparency and data provenance.

To summarize, the proposed solution addresses the shortcomings of existing systems by emphasizing data integrity, real-time processing, and proactive anomaly detection. It balances performance considerations with the critical need for precise, verifiable results, even in situations where alerts are infrequent. This holistic approach ensures trustworthiness and transparency in complex streaming data processing scenarios.

## 1.4 Contributions

With our work we hope to make theoretical and practical contributions to further the knowledge on the field of data provenance in data stream processing. Those contributions are:

- An **in depth study and analysis of the state of the art** both in stream processing and data provenance. the areas of Stream Processing and Data Audit, Lineage, and Provenance. By studying the existing work and solutions in these areas we understand both their shortcomings and advantages, which helped us design an architecture framework for complete and correct auditable data provenance in a stream processing environment, while presenting acceptable performance and resource utilization.
- A practical **implementation and deployment of the proposed architecture** within a real-life use case. This real-world instantiation demonstrates the viability and efficacy of the auditable data provenance framework in addressing complex challenges in streaming data processing. The chosen use case embodies the complexity of processing continuous video streams in a dynamic setting, where individuals move within a physical space. This environment introduces multiple challenges, including real-time detection, tracking, and anomaly identification. By successfully implementing the proposed architecture in such a context, this work demonstrates its adaptability and problem-solving capabilities in challenging, dynamic scenarios.
- **Integration of diverse technologies** in our implementation, like computer vision for object detection, deep learning models for tracking, Apache Kafka for communication and state management for auditability, and providing the user with the capability to configure the system for a better performance and to analyze the obtained results via data and visual evidence.
- **Design of a test environment** to validate the correctness and evaluate the performance of our implementation, where stream processing relevant metrics like latency, throughput and resource utilization are analyzed, and the accuracy and completeness of our solution is verified.

## 1.5 Document Roadmap

This document is structured as follows: Chapter 2 presents and analyses our related work. Chapter 3 introduces and describes our solution, its architecture, data structures and algorithms, as well as the use case for the implementation of our solution, with pseudo-code snippets and architecture diagrams. Next, Chapter 4 explains our evaluation methodology and presents the obtained results. Finally, Chapter 5 is composed of a set of closing remarks and a set of improvements and future work.

# 2 Related Work

Contents

This section represents all the work we found to be relevant to the creation of our proposed solution. We will divide this section into the two main topics of our work: **Stream Processing** and **Data Audit, Lineage, and Provenace**.

## 2.1 Stream Processing

In this section, we will start by presenting the basic concepts of a stream processing system and its archetypal framework. Then we will present a taxonomy to classify stream processing engines. Finally, we will present some of the most used stream processing engines and compare them based on the taxonomy previously introduced.

### 2.1.1 Stream Processing Systems

In order to facilitate the discussion of the relevant related systems, it is important that we first understand the composition of a stream processing system. We start by presenting an archetypal framework, as described in Fig. 2.1, which we believe describes the components of a stream processing system as a set of layers. This illustration is inspired by the work [12]. Despite having diverse data domains and business logic, the layers we will present next are constant to the data processing pipeline of distinct stream processing systems. We will describe the importance of each layer to the overall framework of a stream processing system, how it is implemented, and the technologies which are used.

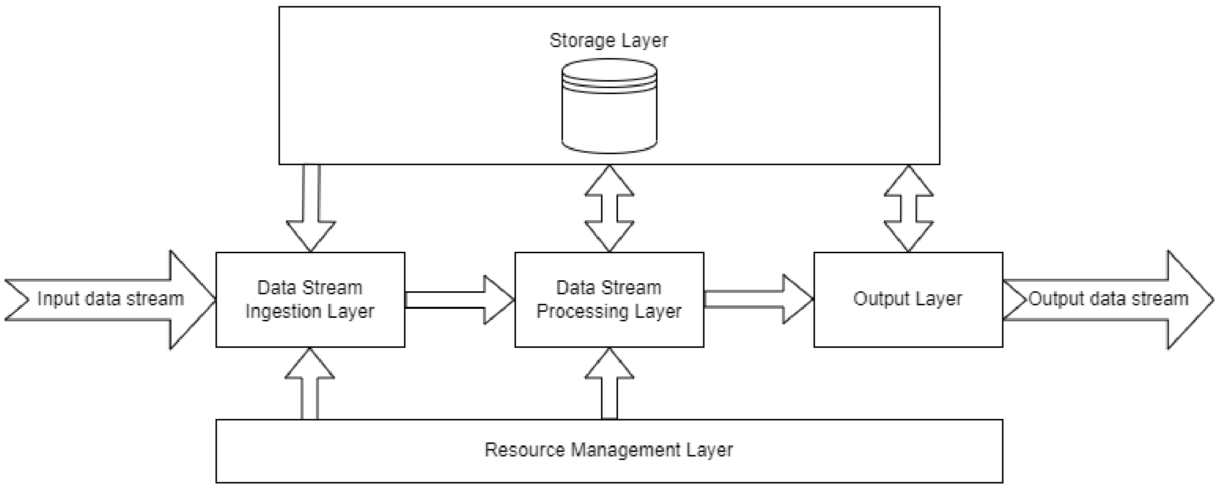
Storage Layer
Input data stream
Data Stream Ingestion Layer
Data Stream Processing Layer
Output Layer
Output data stream
Resource Management Layer

**Figure 2.1:** Framework of a Stream Processing System

**Data Stream Ingestion Layer:** Data ingestion in stream processing is the process of transferring the received data streams from its source to its processing or storage system in an efficient and correct manner. This layer is the doorway into the stream processing system and it deals with input data streams derived from different sources and parsed in distinct ways. These sources can be any element that can collect and transmit time-sensitive data. Some examples are: social network APIs, IoT devices, REST Web services, WebSockets, service usage logs and different stream processing systems. This layer must be prepared to receive data streams in different types of input, such as JSON objects, graphs, or plain text delimited by commas or tabs. The systems implemented in this layer are known as Stream Ingestion Systems or Queueing systems. MQTT[1], ActiveMQ[2], RabbitMQ[3], ZeroMQ[4] and NSQ[5] are well known queueing systems and Kinesis Data Firehose[6], IBM WebSphere MQ[7] and Microsoft Message Queuing[8] are commercial stream ingestion systems.

**Data Stream Processing Layer:** This layer is where the previously received data is processed. The data can be processed by disjoint applications or by a stream processing engine or even a combination of both. It's through processing that value is extracted from the data received. Today there is a large variety of stream processing engines, that, unlike traditional engines, which run a periodical analysis over finite stored data sets, can process data over dynamic unbounded data streams in real-time and in any given time interval. They can process high volume and velocity of streaming data. Some examples of modern stream processing Engines are Apache Storm[9], Apache Flink[10], Spark Streaming[11], Kafka Streams[12], IBM Streams[13],Amazon Kinesis[14], Azure Streams[15] and Google Cloud Dataflow[16]. We make a more in-depth analysis of some of these engines in Section 2.1.3.

**Storage Layer:** It's common for stream processing systems to store analyzed data, extracted knowledge, or even patterns found in different stages of data processing. This stored information is organized and indexed along with external knowledge or metadata to be used in future tasks of the processing of data streams. Data storage in a stream processing systems can be provided by a wide variety of solutions, like HDFS[17], a traditional file system, PostgreSQL[18], a distributed file relational

[1]"MQTT" https://mqtt.org/
[2]"ActiveMQ" https://activemq.apache.org/
[3]"RabbitMQ" https://www.rabbitmq.com/
[4]"ZeroMQ" https://zeromq.org/
[5]"NSQ" https://nsq.io/
[6]"Kinesis Data Firehose" https://aws.amazon.com/kinesis/data-firehose/
[7]"IBM WebSphere MQ" https://www.ibm.com/docs/en/ibm-mq/7.5?topic=mq-introduction-websphere
[8]"MSMQ" https://learn.microsoft.com/en-us/previous-versions/windows/desktop/msmq/ms711472(v=vs.85)
[9]"Apache Storm" https://storm.apache.org/
[10]"Apache Flink" https://flink.apache.org/
[11]"Apache Spark Streaming" https://spark.apache.org/docs/latest/streaming-programming-guide.html
[12]"Apache Kafka Streams" https://kafka.apache.org/documentation/streams/
[13]"IBM Streams" https://www.ibm.com/pt-en/cloud/streaming-analytics
[14]"Amazon Kinesis" https://aws.amazon.com/kinesis/
[15]"Azure Streams" https://azure.microsoft.com/en-us/products/stream-analytics/
[16]"Google Cloud Dataflow" https://cloud.google.com/dataflow
[17]"HDFS" https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[18]"PostgreSQL" https://www.postgresql.org/

database, Redis, a key-value store, VoltDB[19], an in-memory database, MongoDB[20], a document storage, Neo4j[21], a graph storage system, Cassandra[22], a NoSQL database, or CockroachDB[23] for NewSQL.

**Resource Management Layer:** This layer is responsible for the coordination of actions between compute and storage nodes, but is also responsible for managing the allocation and scheduling of resources in distributed systems, so that parallel processing of high volume and velocity of data stream are possible. Resource Management is even more important when building multi-cluster distributed streams. Some tools to support this are ZooKeeper[24] or Twine[25].

**Output Layer:** The final layer of a stream processing system is the one responsible for handling the results from the data stream processing pipeline. After the data is processed the produced results can be directed to monitoring dashboards, visualization tools, different workflows, and applications, or simply stored in temporary or permanent data storage for subsequent analysis. According to previous works [12,13] these visualization tools can be divided into four groups: (i) graph visualization tools for static and dynamic graph visualization, (ii) text visualization tools for static and dynamic text visualization, (iii) map visualization tools for geographic data exploring, (iv) multivariate data visualization tools for generic data types.

Now that we understand the framework of a stream processing system as a whole, we will focus on the stream processing engines, which are the core of the system, and how value is extracted from the data streams received.

### 2.1.2 Stream Processing Engines

Stream processing engines are complete processing systems that include the Dataflow Pipeline we described when we presented a stream processing system's framework. They are responsible for processing the data provided by the input data stream and extracting some value from that data. Nowadays, there is a wide variety of these stream processing engines, some being open-source platforms, while others are developed and commercially provided by major software companies like Google, IBM, Microsoft, or Amazon.

To better understand what stream processing engine better suits our solution, we must first under-

[19]"VoltDB" https://www.voltactivedata.com/
[20]"MongoDB" https://www.mongodb.com/
[21]"Neo4j" https://neo4j.com/
[22]"Cassandra" https://cassandra.apache.org/_/index.html
[23]"CockroachDB" https://www.cockroachlabs.com/product/
[24]"Zookeeper" https://zookeeper.apache.org/
[25]"Twine" https://twinery.org/

stand what are its components. As we can see in Fig. 2.2, these platforms are composed of four main components: **Input Sources**, where the data is read from, **Output Streams**, where the processed data is written, **Stream Processor**, where transformations and analysis are performed into the received data in order to extract value, and **State Management**, which keeps store of the progress and the results of former operations done by the processor. Based on some works [12, 14–16] it's possible for us to

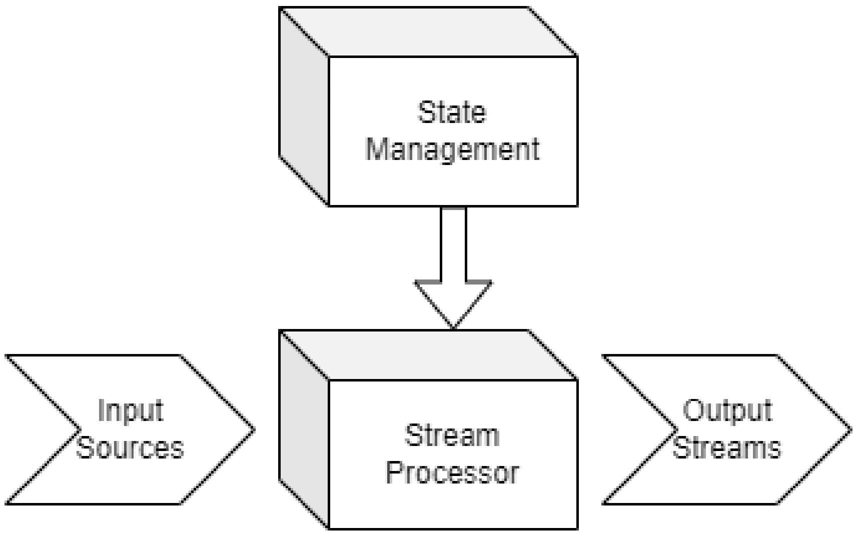

**Figure 2.2:** Components of a Stream Processing Engine

propose a taxonomy that takes into consideration the classification and main characteristics of stream processing engines, which are, **System Openness**, **Type of System**, **Architecture**, **Programming Model**, **Data Partitioning Strategy**, **State Management**, **Execution Semantics**, **Fault Tolerance** and **Deployment**. However, there are also properties and metrics that affect a stream processing engine's performance and usability. After reading and analysing some works regarding benchmarking of these engines [12, 16–21], we concluded that **Scalability**, **Performance** and **Resource Utilization**, must also be present in this taxonomy since they are a big factor when analyzing any stream processing system. This taxonomy will be later used to help us compare the state-of-the-art stream processing engines.

**System Openness:** Like any other software, stream processing engines can be either *Open-Source* software when their source code is available to the public, so it's easy to make modifications to the systems, or *Closed-Source* software, when their source code is not available to the public and can only be accessed with the payment of a high value and if an authenticated license is acquired. Even after meeting these conditions, there are a lot of restrictions on the usability and ability to make modifications to these kinds of software. Apache Spark, Apache Flink, and Apache Kafka are examples of open-source stream processing engines, while Amazon Kinesis and Azure Streams are closed-source.

**Type of System:** According to the work of Bockermann [22], stream processing engines can be divided into four major types of systems, which are *Query-based* systems, systems that originated from database research, *Online Algorithm Research* systems, systems which explores distinct algorithmic aspects of computing results from unbounded, streaming data sources, and finally, *General Purpose Streaming Data Processing* systems, those that allow custom streaming applications to be implemented and executed.

**Architecture:** As previously mentioned, stream processing engines are composed of several processes or nodes that must communicate and be coordinated with each other. There are two architecture models that a system can implement to do this, and they are, a *Master-Slave* model or a *Peer* architecture.
In a *Master-Slave* architecture, one node or process is chosen as the master, which will control the other nodes, who are known as slaves and will serve as their communication hub. stream processing engines like Kafka Streams or Apache Spark use this architecture.
The *Peer* architecture divides the capabilities and responsibilities equally amongst all of its composing processes or nodes. This architecture model is used by stream processing engines such as Apache Storm and Apache Flume[26].

**Programming Model:** Stream Programming Models define how a stream processing engine works in general. However, we will divide this category into four distinct categories, **Type of Streaming**, **Flow of Data**, **Storage System**, and **Application Language**.

**Type of Streaming** describes the method used by the system to process the data flow it receives. It can be categorized into two types, *Native* and *Micro-Batching*. *Native* models do real-time streaming data processing by continuously processing each tuple received in the data stream, individually and as soon as they arrive. This allows time-sensitive systems to receive valuable processed data quickly. Apache Flink and Kafka Streams are some prominent stream processing engines that use this model. *Micro-Batching* models split the input data stream into smaller batches through the use of windows. These windows are usually defined by time duration or by record count. These models are a middle ground between the typical *Batch Processing* and the processing done in Stream Programming *Native* models, they are useful to systems that need fresh data, but not real-time processed data. The most prominent stream processing engine which uses *Micro-Batching* is Apache Spark Streaming.

**Flow of Data** in this case refers to the type of data models a stream processing system uses to represent its data flow. It's common for some systems to use *Topologies* or graphs, like *Directed*

[26]"Apache Flume" https://flume.apache.org/

*Acyclic Graphs* or *Directed Computational Graphs* to define their data flow operations.

**Storage System** are responsible for the storage of data in stream processing engines. Some engines provide native *In-memory* storage, but most of the others rely on independent storage systems or *Data Bases* to fulfill this need.

**Application Language** is related to high-level languages supported by the stream processing engines. *Java*, *Scala*, *Python* and others, are examples of these high-languages. An engine's ability to support several high languages, can provide the developers with a greater choice of coding language, which can reduce the implementation time of processing pipelines.

**Data Partitioning Strategy:** In some application areas it's common for stream processing systems to have to handle big workloads, so managing the resources and partitioning the data within the used stream processing engine is critical to achieving efficiency. The two most commonly used partitioning strategies are *Hash* and *Range*, but there is a wide variety of others within all the stream processing engines.
The *Hash* partitioning uses a hash function to examine an input record, which produces a hash value. Records with the same values are allocated to the same data partition. As for the *Range* strategy, the partitions to where the data is allocated depend on the value of each tuple. Tuples with key values that fall within the same range will end up in the same partition.
Some engines, like Apache Storm, don't use one of these two strategies and instead use specific grouping strategies to distribute incoming tuples among bolt tasks. These grouping strategies can be a random distribution or based on a user-specified field, among others.

**State Management:** In stream processing the processes can be represented as a directed tree or a Directed Acyclic Graph composed of node operators and the output and source stream operators. Operators can be *Stateless* or *Stateful*.
The output produced by *Stateless* operators is uniquely dependent on the input received, while the output of *Stateful* operators, despite also being based on the input received, can potentially be affected by information obtained from previous operations and stored in internal data structures called states.

**Execution Semantics:** This feature of a stream processing system has a great effect on the balance between its reliability and cost. Some systems may want to focus on reliability and make sure that every message is received, while others are more focused on the cost of processing, and losing some messages won't affect their functionality. A system can use one out of three semantics regarding message processing, and they are *At-Most-Once*, *At-Least-Once*, and *Exactly-Once*.

The *At-Most-Once* semantics guarantees that a message will be delivered one or zero times. If an event is lost during routing, then there will be no other attempts for it to be delivered. This semantics has the least fault tolerance out of the three, but it's the simplest one.
The *At-Least-Once* semantics is commonly used by stream processing engines, it ensures an event is delivered a minimum of one time. This is possible because there will be continuous attempts to deliver a message until an acknowledgment of its delivery is received. This may lead to situations where events are delivered more than once, because the acknowledgment can be lost, and, therefore, processed more than once, resulting in an additional cost to the system. This semantics is used by Apache Storm.
The *Exactly-Once* semantics is the optimal solution to guaranteeing an event is delivered and processed. However, it's also the most complex. In this semantics, there is the certainty of an event being delivered exactly one time through the use of multiple acknowledgment checks, which provides the reliability that an event is delivered, but also avoids additional costs that would be caused by data processing repetition. Apache Flink and Apache Spark employ this semantics.

**Fault Tolerance:** Failures can occur in stream processing systems due to a variety of reasons, and when they happen it's important that the system remains operational during the time of recovery. This demonstrates the system's fault-tolerance capability. However, the ability of a system to recover from a failure requires additional resources and, therefore, an extra cost.
Fault Tolerance in stream processing engines can follow two distinct approaches, *Passive* or *Active*. The *Passive* approach can be implemented through checkpoints, upstream buffer, and source replay, while the *Active* approach can be achieved through the use of replicas.

**Deployment:** Stream processing engines can be deployed in three distinct manners. They can be deployed *locally* on a single machine, which is easy to implement but limits the system's scalability, velocity, and capability to handle a lot of data. Systems that need to handle high velocity and volume of data use *Cloud* or *Cluster* deployments. *Cluster* deployments, still face limited scalability, due to cluster size, while *Cloud* deployments have less limited scalability, but on the other hand, have a bigger probability of suffering from high latency.

**Scalability:** Scalability is an important property of any software system and one of the most studied ones (e.g. [23]), but even more for a stream processing system. With the evolution of technology and the increase of connectivity between people online, the amount of data available for processing is constantly growing and stream processing systems must be capable of handling that increase of data flow. Scalability is precisely the capability of a system to process a higher workload without service interruption. A system can *scale-up* when handling a bigger workload with the current resources, or *scale-out* when using the current resources as well as newly added ones to handle the increase in workload.

Scalability is a spectrum, some stream processing engines like Flink and Storm are less scalable, while Apex[27] is highly scalable, and Spark is in the middle of the spectrum.

Two big properties that influence a system's scalability are *Elasticity* and *Parallelization*. *Elasticity* guarantees a stream processing engine's low latency against the variation of workload. A system's resource needs may vary depending on the workload, an elastic system can scale its resources according to these needs. However, this creates the challenge of balancing between over-provisioning and on-demand scaling. Over-provisioning is costly but can handle surges in the workload, while on-demand scaling is not as robust to those surges, despite saving costs.

A lot of research has been done on how to increase *elasticity* in already existing stream processing engines (e.g.de Assuncao et al. [24] and Wang et al. [25]).

As for *Parallelization*, stream processing systems parallelize processing to provide high throughput and low latency despite the massive amount of data. However, the workload or available resources can change at runtime and this creates the challenge of how to continuously adapt the level of *parallelization* when these conditions change.

Frameworks like STRETCH [26, 27] propose a new concept of Virtual Shared-Nothing. This work shows that is possible to define parallel and elastic SPE operators that, by virtualizing the common Application Programming Interfaces based on Shared-Nothing parallelism, can leverage shared memory to first scale streaming applications up while allowing to rely on Shared-Nothing parallelism to later scale them out.

Röger and Mayer [28] propose a survey that overviews and categorizes the state of the art in stream processing *elasticity* and *parallelization*.

**Performance:** Stream processing systems are designed to handle heavy workloads of data to process online. For these systems, the main performance goals are low *latency* and high *throughput*, they characterize a system with quality of service. A system with low *latency* has the ability to process and react to new events in real-time, while *throughput* is a metric that measures the number of data units processed per unit of time.

A number of works [12, 14, 17–20, 28, 29] use these two metrics to evaluate the performance of the most notable stream processing engines in an effort to benchmark and compare them, while other works [26, 27, 30–32] propose or discuss solutions to best the performance of these engines regarding to these metrics. From all these works we can conclude that some engines prioritize

[27]"Apache Apex" https://apex.apache.org/

*latency*, others *throughput* and some try to find a balance between both. Some examples of this are: Storm and Flink, which by having a native programming model, prioritize low *latency* by handling data items immediately as they arrive, but present relatively high per-item cost; Spark, which is a batch-based processing system, prioritizes high *throughput* at the cost of the time an individual item spends in the data pipeline, despite achieving great resource-efficiency; and finally, Spark-Streaming[28], an extension of the Spark API, which by employing a micro-batching model balances *latency* and *throughput*.

The work of Palyvos-Giannas et al. [29] stands out. In this work, the authors propose *Lachesis*, a middleware system decoupled from any stream processing engine, which by manipulating the OS scheduler enforces its desired scheduling goals. The performance of several prominent stream processing engines, like Storm and Flink, with and without the use of *Lachesis* was compared, and the results clearly prove that the use of this scheduling middleware can provide higher *throughput*, as well as considerably lower average *latency*.

We also observed a current trend of fog and edge computing. Due to a lot of data streams originating from devices at network edges, like IoT devices, processing the incoming data right at the source can reduce a system's *latency*. However, this solution presents a limitation, since these edge devices present resource scarcity compared with cloud-based stream processing systems, so a high *throughput* is very difficult to achieve.

**Resource Utilization:** Besides latency and throughput, which can be directly perceived by users, the other metric from which a stream processing engine's performance can be evaluated is resource utilization. Despite only being observed at the level of the underlying stream processing system, this metric is crucial in the choice of engine. It's important for these systems to use distributed resources efficiently with minimal overhead. This resource utilization can be CPU or memory usage.

Concepts mentioned before like scalability, elasticity, and parallelization can all be used to improve resource utilization in stream processing systems. Such an example of this is the work done by Gulisano et al. [33], which present a stream processing engine, which reactively controls the average CPU utilization of the cluster hosting the operator graph. This elasticity management is combined with a novel parallelization technique to minimize the computational resources used.

There is also a considerable amount of works [18, 20, 28, 30] comparing and benchmarking stream processing engines, as well as relevant literature on the topic, that besides latency and throughput, also use resource utilization as an important metric. From these, the work by Bordin et al. [17] stands out. By collecting consumption metrics related to CPU, memory, and network utilization, extensive results were obtained and an in-depth comparison between Spark Streaming and Apache Storm is

[28]"Apache Spark Streaming" https://spark.apache.org/docs/latest/streaming-programming-guide.html

provided.

### 2.1.3 Relevant Related Systems

In this section we will go into more detail on the analysis of the stream processing engines we found to be more relevant. These systems were chosen due to their prominence in works regarding stream processing state-of-the-art and, because, they fit some of the desired criteria or features that we highlight in the previously presented taxonomy.

It's important to note that for the choice of relevant related systems, we took only into consideration those with open-source frameworks. From the observed works [12, 14–19, 21, 28, 34], we conclude that the most notable open-source stream processing engines are **Apache Spark**, **Apache Flink**, **Apache Storm** and **Apache Kafka Streams**.

**Apache Spark:** This stream processing engine is very versatile, since it can be used not only for stream processing, but also for batch processing and interactive queries, and is also efficient for large-scale data stream processing. It offers high-level APIs for Python, Java, Scala, R, and SQL. Apache Spark has the advantages of being a mature product, with a large community and several real-world use cases that prove its efficiency. It's fault-tolerant and supports advanced analysis. However, Apache Spark is not a true stream processing engine, it's a batch-processing system that performs very fast. It can present a latency of a few seconds in some cases and it can also consume a lot of memory. To handle stream processing, the developers of Apache Spark created a module named **Apache Spark Streaming**. Spark Streaming shifts Spark's batch-processing approach towards real-time requirements. This is achieved by dividing the stream of incoming data into smaller batches. Spark Streaming is the most popular open-source framework for micro-batch processing. It presents all of the advantages of Spark since it's part of its framework and runs on top of a common Spark cluster. The incoming data is transformed into resilient distributed datasets, which are processed in order. However, data inside these resilient distributed datasets is processed in parallel, hence having no ordering guarantees. Later, the creators of Apache Spark also implemented a module called **Structured Streaming**[29]. This module, just like **Spark Streaming**, uses micro-batches and expresses streaming computation like one would express batch computation. Instead of RDDS, this module uses Dataset/DataFrame APIs. This module is an upgrade over the first one and brings Apache Spark closer to real stream processing. Apache Spark and its modules have the great benefit of assuring end-to-end exactly-once processing guarantee.

**Apache Flink:** Flink is a native stream processor that can run stateful streaming applications and can be used both for batch and stream processing to compute unbounded or bounded data streams

[29]"Apache Spark Structured Streaming" https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

from several sources, which is possible due to Flink's approach to batches as data streams with finite boundaries. Flink ingests streaming data from many sources, processes it, and distributes it across various nodes. Flink has the advantages of providing exactly-once processing guarantees, presenting high throughput with low latency, dynamic analysis and optimization of tasks, and an easy-to-use User Interface. However, the disadvantages are that it can present some scaling limitations, only supports Scala and Java, and there's limited support from the community.

**Apache Storm:** This stream processing engine probably has the best technical solution for true real-time processing. Storm can handle large quantities of data while providing results with low latency. Its architecture is based on spouts and bolts, spouts being the origins of information that transfer this information to several bolts that are themselves linked with other bolts. This topology forms a DAG, where spouts and bolts are nodes connected through streams, which are the edges that direct the flow of information. Storm has a simple API, supports a wide variety of languages, and is very flexible and extensible. Even though Storm presents several advantages, it also has some disadvantages when compared with other stream processing engines, like the lack of a built-in windowing or state management feature, only providing at-least-once processing semantics, and not providing any guarantee regarding ordering.

**Apache Kafka Streams:** This is a stream processing Java API provided by Apache Kafka that can be used to build real-time streaming data pipelines and applications, giving developers the ability to filter, join, aggregate and group without writing any code. Kafka Streams is very easy to integrate with other existing applications, which offers low latency and replaces the need to have standard message brokers. Nonetheless, it also presents some disadvantages like limited analytics, lack of point-to-point queuing and other messaging paradigms, and struggling when there is an increase in the number of queues in a Kafka cluster.

In Table 2.1, we can see a comparison of the previously mentioned stream processing engines. It's important to note that some elements of our taxonomy, despite representing important characteristics and metrics of a stream processing engine, are not present in this table because they are not relevant to the context of our work at this time or due to there being no relevant differences between the mentioned engines regarding these elements.

## 2.2 Data Audit, Lineage and Provenance

With the increase of data sharing and the massive amounts of data available to be processed, stream processing systems face the challenge of keeping track of data sets' sources and what influenced their state along their lifecycle. This puts into cause the trustworthiness of a system and its data quality because it's hard to trust results or data processed when you don't know where the data ingested came

**Table 2.1:** Comparison of Stream Processing Engines

| Stream Processing Engines | | | | |
|---|---|---|---|---|
| | Spark | Flink | Storm | Kafka |
| Architecture | Master-Slave | Master-Slave | Peer | Master-Slave |
| Type of Streaming | Micro-Batch | Native | Native | Native |
| Storage System | HDFS, Cassandra and others | HDFS, S3, RocksDB and others | HDFS, Cassandra, MongoDB and others | In-memory |
| Application Language | Java, Scala and Python | Java, Scala and Python | Java, Scala, Ruby, Python and others | Java and Scala |
| Data Partitioning Strategy | Hash and Range | Hash | Grouping | Hash |
| State Management | Yes | Yes | Yes | Yes |
| Execution Semantics | Exactly-Once | Exactly-Once | At-Least-Once | Exactly-Once |

from or how it is transformed along the stream processing pipeline.

In Section 2.1 we analyzed stream processing and the existing stream processing engines as we detailed their framework, and most relevant characteristics and metrics. However, no widely deployed and used stream processing engine incorporates a feature or mechanism that allows one to trace back the processed data path and transformations it suffered. To tackle this challenge, we must first understand the concepts of **Provenance** and **Lineage**.

These concepts do not have a unanimous definition, however, in the context of our work we will interpret them as the following:

**Data Provenance** Is the policy we want to add to a system to ensure we know the origin of the processed data sets, their steps from their origin until the end result, and the transformations they suffered.

**Data Lineage** Is the mechanism we want to implement in a system to provide Data Provenance by tracing each data set's course and changes.

In Fig.2.3 we can see a representation of a classification of **Provenance** considering its granularity, according to [35].

**Coarse-grained Provenance** This type of granularity refers to provenance information gathered at the level of stream or sets of stream events. It's typical for details regarding data transformations to be hidden. With coarse-grained provenance we only know what source data led to a specific output, we have no information about what happened between those stages.

**Fine-grained Provenance** This type refers to provenance information gathered at the level of individual stream events, which provides more detail on what transformations might have originated that

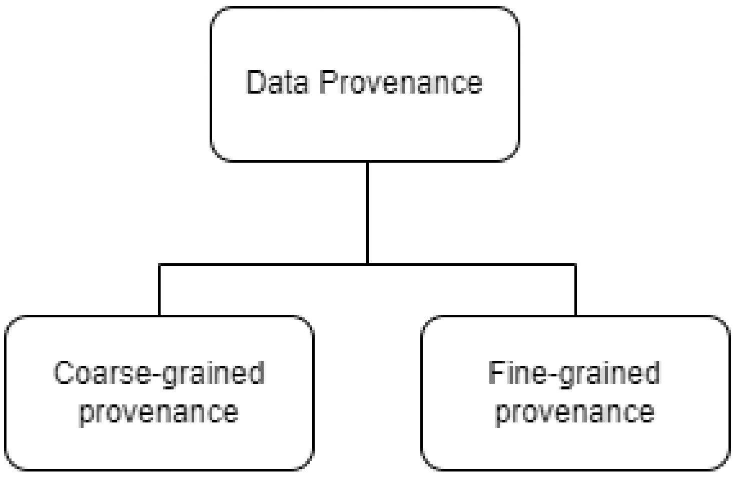

**Figure 2.3:** Classification of Provenance regarding its granularity

particular data item. Fine-grained provenance gives us the information that coarse-grained provenance does not. With this type of provenance, besides knowing the source data that led to a certain output, we also know all its values at intermediate stages of the data processing pipeline and what transformations influenced those values.

It's also important to note that lineage tracing can be distinguished into two types according to where the tracing starts:

**Backward Tracing** Begins at the latest version and is used to find the origin of information and data sets.

**Forward Tracing** Starts at the source of the data and is used to find which results originated from a specific source tuple.

As we can see, data provenance is important to guarantee data quality by understanding the data's lifecycle, but that is not its only benefit. It can also be used to check the data's integrity, and help understand and justify the occurrence of some errors by providing an audit trail.

However, as we can see from several works [8–11] on the topic of data provenance, this is not an easy feature to implement or to add to a stream processing system, as it poses numerous challenges. To trace data's lineage is an inherently heavy process and ties the efficiency of the whole system to its own, since it can implicate an increase in latency, a reduction of throughput and complications in memory storage. We can conclude the main challenges are:

**Storage** To identify data provenance it is not sufficient to store the final results of the processing and the data sources, it is also crucial to keep a record of all the intermediate data objects and dependencies in an efficient manner. In stream processing systems where the workload is already substantial and the input data enters at a high rate, all the intermediate data and dependencies implicate that the total data is multiple times larger than the source data.

**Latency and Throughput** Stream processing systems handle a high rate of incoming data and one of the most desired attributes of one of these systems is the capability to handle a big workload while also giving near real-time results, however, the implementation of a data provenance feature will cause significant overhead in the system's performance. The additional computation to trace data's lineage will cause delay, which means higher latency and higher throughput since the system's resources will be occupied for longer periods of time.

**Determinism** A deterministic system is a system that, given the same input, always produces the same output. Due to the dynamic nature of stream processing systems, it's not possible for us to assume that a given input will always produce the same output. To be able to reproduce a flow that leads to the output of a given data set and trace its lineage, it's important to keep track of intermediate data and dependencies not only during the execution. We must then store the state, and the data associated to it, at the time of the data processing, to ensure a replay of tasks will always produce the same output.

### 2.2.1 Relevant Related Systems

Through extensive research we were able to find works that propose solutions to the previously mentioned challenges while still providing correct data provenance in stream processing systems and in this section we will summarize them.

**GeneaLog** The work by Palyvos-Giannas et al. [36] presents **GeneaLog**, a fine-grained provenance system for support in deterministic stream processing engines. The main contributions of GeneaLog are regarding storage cost. By leveraging a small, fixed-size set of meta-attributes, common to every standard data streaming operators, for each tuple processed by a stream processing system, it is possible to reduce the per-tuple memory overhead that usually occurs in data provenance. Besides this, GeneaLog also leverages the memory management of the process to identify which source tuples contribute to the application output and which ones do not, in order to discard those that do not contribute and subsequently save temporary storage that would be wasted by that unnecessary data.

In this work, prototypes of GeneaLog were implemented on top of the Liebre[30] and Flink stream processing engines. The correctness and performance of these prototypes were evaluated and the results allowed to conclude that GeneaLog provides correct data provenance while also minimizing throughput and latency overheads when compared with other state-of-the-art provenance systems.

**Ananke** A different approach is followed in [37], where the authors present **Ananke**, a framework that extends any fine-grained backward provenance tool and produces a live bipartite graph of fine-

[30]"Liebre SPE" https://vincenzo-gulisano.github.io/index

grained forward provenance. This framework was built to tackle the issue of the lack of streaming-based tools for forward lineage tracing. Ananke has the benefit of not only providing to the user the source tuples that contribute to every output, but also identifying which of those source tuples can still generate future distinct outputs, preventing duplication, which can reduce the memory cost, since there is no need to store the tuples that can no longer generate distinct results or store the same results more than once. This is possible, through the leveraging of native operators of the underlying stream processing engine, by enabling specialized-operator-based and modular implementations that use those operators.
The authors implement two variations of the framework in Flink, one showing how Ananke's algorithm can be parallelized, which allows for the income of higher amounts of provenance data, and the other focused on optimizing the labeling of the expired source data as fast as possible. The authors proved Ananke's correctness and results regarding rate, latency, throughput, memory, and CPU utilization were obtained and they show that, despite presenting small overheads, this framework can provide live forward lineage tracing with similar overheads to the state-of-the-art in backward lineage tracing and outperform Genealog, the system presented before and the system that Ananke extends and uses to provide backward lineage tracing.

**s2p** Ye and Lu [10] present **s2p**, a provenance solution for providing fine-grained and coarse-grained provenance in stream processing systems. Inspired by the philosophy of lambda architecture [38], the design of this solution combines online provenance, used to trace and map the lineage from source data to result data, thus providing coarse-grained provenance, with offline provenance, used to provide detailed information regarding intermediate results or transformation processes, which provides fine-grained provenance.
The authors follow the logic that, in a stream processing system, abnormal results are rare, and the results that require an in-depth analysis are even rarer; so there is no need to track in detail the transformations and lineage for every input data, an approach followed by several systems, which causes major overheads and costs to the system. Instead, s2p targets detailed lineage of only a limited set of data considered to be relevant by replaying that data in an independent cluster. This solution also takes into account operator states, considering the semantics of each operator when analyzing the relationship among data and considering state transformation of stateful operators together with the data transformation process in lineage analysis. Another beneficial feature of this solution is managing data locally and only aggregating some chosen data if a lineage query happens, which contributes to reducing the cost of data transformation in the system.
A prototype of s2p was implemented on Flink and three experiments were conducted. Although these experimental evaluations were conducted in a resource-limited environment, the results show

that s2p causes an increase in end-to-end cost, a decline in throughput, and a limitation in memory storage. However, by comparing these results with other existing provenance solutions, the authors were able to conclude that the runtime overhead achieved is acceptable, taking into consideration that this solution targets more provenance-related data. It's also important to note that s2p is limited since it can only provide detailed lineage results for a stream processing engine consisting entirely of deterministic operators.

**Ariadne** One of the earlier systems we encountered that tackles the challenges of fine-grained provenance for stream processing systems is **Ariadne**, presented by Glavic et al. [11]. The authors introduce an approach that by modifying the behavior of operators, also known as operator instrumentation, can provide fine-grained provenance. The Reduced-Eager operator instrumentation is an approach that consists of eagerly propagating a form of lineage during query execution and lazily reconstructing lineage independent of the execution of the original network.

This approach implemented in Ariadne has the disadvantage of having a greater cost for storing and reconstructing tuples. However, due to the compressed representations used, this cost is offset thanks to better performance in terms of runtime and latency. This approach also gives the user the power to only request the lineage tracing of specific results and has the advantage of being able to correctly handle non-deterministic operators. The Replay-Lazy and Lazy-Retrieval techniques are also implemented to provide additional optimizations to decouple lineage computation from stream processing.

The authors conducted an experimental evaluation which by using a variety of parameters and workloads assesses the computational cost and latency of the system. The results allow the authors to validate the correctness and effectiveness of Ariadne's implementation and prove that, although presenting minor overheads, the system clearly outperforms query rewrite, the state-of-the-art at the time of the work's publication.

**SAC** Another system that claims to enable interactive data provenance in stream processing systems is presented by Tang et al. [39]. The system is called **Spark-Atlas-Connector** or **SAC** and extends Apache Atlas[31]. SAC can be easily implemented in Spark, needing no modifications to the stream processing engine or additional users' inputs, in order to provide an efficient query interface to manage the captured data lineage. The system also presents the advantage of providing data lineage tracing to all the processes in the stream processing pipeline and supports different data storage, as well as distinct stream processing paradigms. SAC also has the ability to provide a visual representation of data lineage, which allows the user to understand the flow of data from its source

[31] "Atlas," https://altas.apache.org/

to the output stream.
This system achieves efficient data lineage tracking for more than 100GB per day, a conclusion that was taken from the results of this system's real-world deployment. However, this system has the major difference of focusing on coarse-grained provenance, while all the previous works mentioned focus on providing fine-grained provenance.

**Visualization Tool** Another work with a similar feature of providing a visual representation of lineage to the users is proposed by Yazici and Aktas [40]. The authors propose a real-time visualization method of data lineage through the use of graphs. The authors also implemented the use of forward and backward tracing to identify post or prior relationships of any data tuple. Two other relevant features in this work are first: i) the ability to summarize the provenance data acquired to only the more relevant aspects since this data can be of a very large scale; ii) the ability for users to compare lineage graphs, which can be very useful in understanding anomalies.
The implemented prototype visualization tool and the visualization methods it uses were evaluated through an experimental study using two distinct data sets. The obtained results proved the visualization methods proposed present an insignificant processing overhead and are scalable.

**Lineage Tracing Framework** Zvara et al. [41, 42] present a **lineage tracing framework** design for batch and streaming processing systems. The authors propose this solution with the goal of detecting inefficiencies in lineage tracing to increase performance and reduce the overhead in these systems. Lineage is found by wrapping each record and capturing record-by-record causality. The authors also sample incoming records randomly to reduce overhead, which contributes to efficiency optimization. The main advantages of this solution are its suitability for batch and streaming data processing, as well as being able to trace lineage in multiple systems, with the same framework. To perform an experimental evaluation, two distinct prototypes were implemented on Spark, one for batch processing and another for stream processing.
Through these prototypes and their results, it was shown that this solution does improve efficiency and reduces tail-latency. However, tracing the lineage for all the data proves to be too expensive and major overheads occur.

**Provenance Inference Algorithm** As mentioned previously, storage is one of the main challenges of provenance, and fine-grained provenance consumes an additional amount of storage. Huq et al. [6] propose a solution to this problem through the creation of a **provenance inference algorithm**, which uses a temporal data model and coarse-grained provenance.
The temporal data model consists of adding a temporal attribute, like a timestamp, to each data item,

which allows us to obtain the overall state of a database at any given time. This, combined with the ability to reconstruct the window which was used for the original processing, obtained by the leveraging of coarse-grained provenance, ensures reproducibility and allows the algorithm to infer the fine-grained provenance of data. This approach is dataset independent and the more the sliding processing windows overlap, the more storage consumption it reduces.
However, this approach also presents some limitations, like only providing accurate lineage information if the processing windows always produce the same number of output tuples. The approach also only provides completely accurate lineage information in a system almost infinitely fast, something that is unlikely achievable in a real-world system. The probability of the inferred fine-grained provenance being inaccurate is high for real-world systems since these systems present processing delays that will cause errors in the algorithm.

**Stream Ancestor Function** Other works proposing a solution for fine-grained provenance in stream processing, which tackles the issue of storage consumption are presented by Sansrimahachai et al. [43, 44]. The solutions presented in this work are based on a reverse mapping function called **Stream Ancestor Function**. This function identifies dependency relationships for any data tuple from the data stream, hence providing fine-grained provenance and ensuring the reproducibility of data processing in a system. However, this solution still presented a storage issue. To solve this, the authors optimized the function and were able to reduce the storage cost by eliminating the need to store every intermediate stream element and by enabling provenance queries to be performed dynamically.
An experimental evaluation was conducted by the authors. This evaluation proved the solution's correctness and assessed the solution's influence on storage consumption and throughput. The results showed that this solution, besides reducing storage consumption, also provides acceptable processing overheads.

**Augmenteed Lineage** Finally, we also find it important to mention the work of Yamada et al. [45]. This work presents the concept of **augmented lineage**, a technique that ensures lineage traceability of complex data analysis including User Defined Functions for processing in the areas of Artificial Intelligence and Machine Learning. According to the authors, the presented framework can be extended to stream processing environments. Since this framework proved to be efficient, in the future it can present a contribution to lineage tracing in stream processing.

## 2.3 Summary

Throughout this chapter we presented a survey of the current state of the art in stream processing as well as a wide variety of works that discuss data provenance in general, and some even propose solutions for the challenge of providing data provenance in stream processing systems. We compare the capabilities of the most successful stream processing systems to help us decide what features we want to implement in our work. We also compare the proposed solutions for the problem of data provenance in stream processing systems to identify their limitations and important contributions to our work.

The analysis and understanding of the works and technologies presented in this chapter allowed us to understand how we want to design our solution and the methodology to evaluate its correctness and performance. Inspired by some of these works, we conclude that our system must be able to effectively provide stream processing functionalities while keeping a record of important data to use in rare cases where data streams need to be audited to obtain data provenance. By assuming the situations where the data stream will need to be audited are rare, we shift the priority of our solution to the correctness and completeness of the results in those situations. Hence, the main goal of our solution is to provide correct data provenance in a stream processing system, despite the occurrence of some overheads. This is a trade-off we feel will benefit the overall performance of our system.

# 3

# Solution

## Contents

In this section, we present the architecture for our solution to the problem of auditable data provenance in streaming data processing. Our solution focuses on providing auditable data provenance for a video streaming processing system, but its idea and architecture can be applied to other kinds of stream processing systems.

We will start by describing the use case of our work, followed by an overview of our proposed architecture, before giving an in-depth and detailed description of the basic concepts and implementation of our solution, as well as its requirements.

## 3.1 Use Case

Users interact with the system by providing a video stream as input. The system's primary pipeline includes modules for real-time people detection and tracking, area drawing and area counting. As frames are processed, the system identifies individuals, tracks their movement, and calculates the number of people within designated areas.

During real-time processing, the system constantly monitors for suspicious situations. If an anomaly is detected, such as a person disappearing unexpectedly, the system generates an alert. This alert triggers the secondary pipeline. The secondary pipeline replays the video frames starting from the time when the suspect person was first detected. It mirrors the processing steps of the normal pipeline with the purpose of checking if the same alert is produced. If the same alert is triggered, the system stores permanently the relevant video frames and state relevant data. This process is depicted in Figure 3.1.

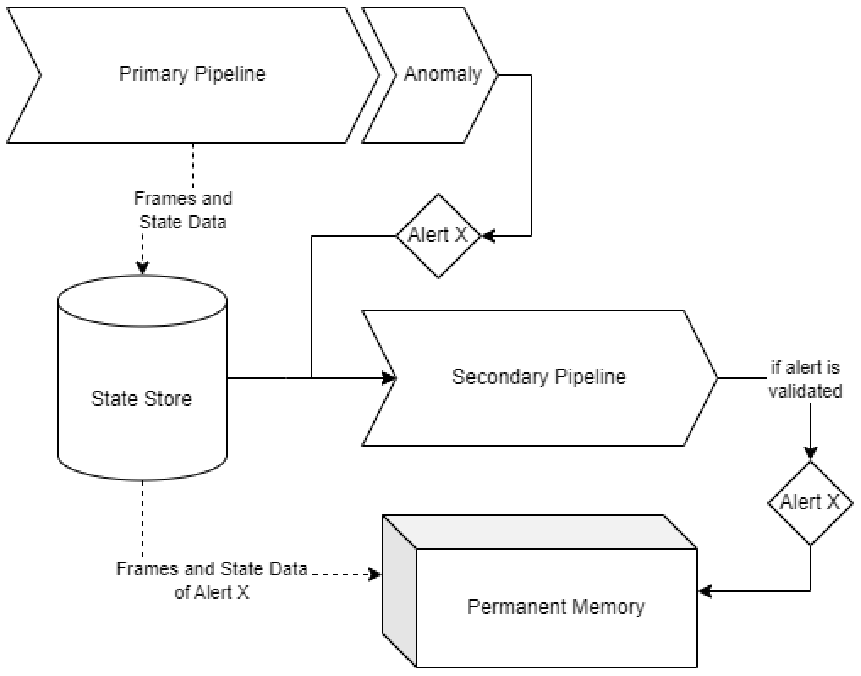

**Figure 3.1:** Interaction between Online Phase pipelines

The verified alerts and the information it contains will also be registered, and with the information stored during the alert validation, the system will have the capability to deterministically reproduce the data stream through a third similar pipeline at any time in the future for audit or investigation purposes.

## 3.2 Architecture Design Requirements

The architectural design of a stream processing system must be defined by robust principles in order to meet the goal of establishing comprehensive data provenance. While the system's use case may encompass various functionalities, the primary objective remains the creation of a system capable of delivering provenance in diverse stream processing scenarios. This section delves into the core requirements that govern our architectural design, emphasizing the principles vital for ensuring data lineage, accountability, and the ability to validate results in any stream processing scenario. The requirements are as follows:

**Data Provenance and Lineage:** At the heart of the design lies the necessity to establish and meticulously maintain data provenance. The primary goal is to capture and retain lineage information for all data that undergoes processing. This is accomplished by storing only the needed data to allow the replay of certain stream segments, which ensures accountability and facilitates auditing capabilities. The architecture is geared towards maintaining the data needed to obtain a meticulous audit trail, to document each step of processing, data transformation, input, and output, for valuable stream segments. This guarantees complete transparency in the system's operations, while also reducing the amount of resources needed by keeping only absolutely necessary data.

**Code Integrity:** Guaranteeing the integrity and authenticity of the underlying code and algorithms stands as a pivotal component of the architecture. To this end, a hash-based verification process is employed. This process verifies that the results align precisely with the code employed in the processing steps of every pipeline.

**Dependency Management:** Managing dependencies effectively, especially within the context of continuous streaming, is crucial. The architectural design comprehends the intricacies of dependencies between frames and data objects. It achieves this by meticulously and periodically preserving state information for replication.

**Replicability for Validation:** Replicability is a cornerstone requirement for the validation of results. The presence of secondary and tertiary pipelines allows for the recreation of results to confirm their accuracy, thereby offering a dependable means of verification. This wouldn't be possible without the storage of critical data, such as the periodically stored state information or the received data stream segments saved by an alert triggering.

**User Involvement:** While not the primary focus, user engagement remains a significant factor for provenance validation. The system affords users the capacity to review, validate, and affirm the lineage and precision of results.

**Scalability:** The architecture is engineered to cater to the scalability demands inherent in stream processing. It accommodates the increasing volumes of data while simultaneously upholding the tracking of data lineage and provenance. The architecture design is easily scalable by following an approach where each pipeline is divided into modules with different functionalities, allowing the user to adapt the system to its needs by adding or removing modules without compromising the capability of providing data provenance for the stream processing.

**Alerting:** The alerting mechanism we implement in our system allows the system to detect potential anomalies in the stream processing. Customizable alerts enhance the system's capability to detect and validate anomalies in real-time without interrupting the continuous processing of the data stream. This mechanism is also used to only store valuable information for replaying in case of an anomaly, therefore preventing a more resource-costing storage solution.

**Manual Auditing:** Specific auditing processes, such as those necessitating the involvement of the third pipeline, can be initiated manually. This feature proves valuable for conducting comprehensive audits or fulfilling specific verification requirements and gives the users the ability to analyze the data stream, and the data resultant of its processing, at a later time .

## 3.3 Architecture Design

As mentioned before, our system consists of three pipelines with similar modules and functionalities, but distinct purposes. To more easily present the architectural design of our system we present a representation in Figure 3.2, and we will divide this section into subsections for each pipeline, encapsulating the communication between pipelines and modules, and storage solutions. Our work, being influenced by the work developed in s2p by Ye and Lu [10], is divided in an online phase and an offline phase, where the two first pipelines are part of the online phase, and the last pipeline represents the offline phase.

### 3.3.1 Primary Pipeline or Continuous Video Processing Pipeline

The architectural design of the continuous video processing pipeline is centered around real-time detection, tracking, and analysis of video streams while ensuring the integrity and provenance of processed results. The pipeline consists of four sequential modules that work in harmony to achieve accurate processing, anomaly detection, and evidence generation. The modules are the following:

**Module 0 (Init):** This module, pictured in Algorithm 3.1, is responsible for hash-based code verification. At the pipeline's initiation, this module plays a crucial role in ensuring the integrity of subsequent

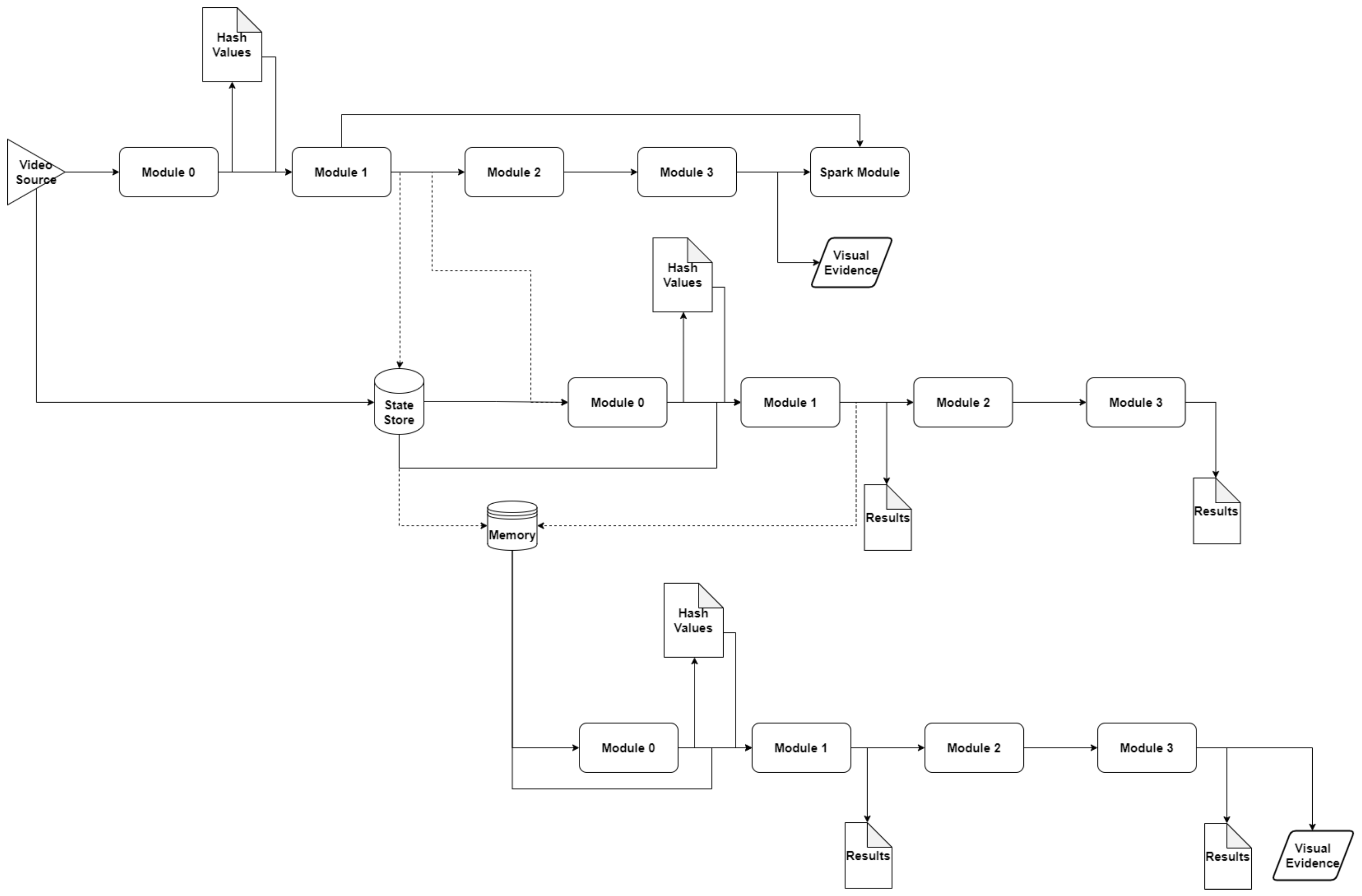

**Figure 3.2:** Architecture Design Diagram

modules. It computes hash values for the code files of each module in the Online Phase. These hash values serve as verifiable fingerprints of the code at that moment. After calculated, these values are combined with the hash value of the video source. The hashing of this combination will be stored to be cross-referenced later to guarantee that those results produced by the pipelines are consistent with the code that generated them, and that the pipelines have not suffered any alterations.

**Module 1 (Detection):** This module, depicted in Algorithm 3.2, is responsible for the real-time detection and tracking of individuals within each video frame. Utilizing state-of-the-art object detection techniques, it identifies persons in the video stream and tracks their movements over consecutive frames. Additionally, this module actively monitors whether individuals abruptly disappear within a frame or disappear after approaching the frame's edges[1]. If an individual unexpectedly disappears within the frame, instead of disappearing at one of the frame's edges, an alert is generated. This alert serves as a trigger for the secondary pipeline, ensuring timely responses to suspicious events. Importantly, this module also maintains a state store that captures the current state of the people detection and tracking model. At specific intervals, the module stores the model's results and state, as well as an amount of data previously received, allowing for potential replays of the processing steps. Finally, this module also generates a register with the number frame of when every person was first and last detected.

**Module 2 (Visual representations):** In this module, detailed in Algorithm 3.3, the focus shifts to providing visual assistance. Detected persons are highlighted with bounding boxes, determined by the coordinates calculated in the previous module, and drawn around them in each frame. This bounding box will also include the unique identifier assigned to each person. This visual representation aids users in understanding the detection outcomes, enhancing transparency and enabling verification of the system's accuracy. This module's outcomes lay the groundwork for subsequent analysis and evidence generation in the pipeline.

**Module 3 (Counting):** Module 3, depicted in Algorithm 3.4, takes charge of critical tasks involving people counting and area analysis. It defines and draws designated areas within each frame where people count analysis will occur. These areas can be dynamically adjusted to accommodate varying store layouts. By partitioning the frame into specific regions of interest, the system accurately tracks and counts individuals within each area. The count results are continuously updated as individuals move within and across areas. The outlined areas help users understand the zones under scrutiny for people counting. This visual feedback enhances transparency and allows operators to verify the accuracy of area definitions. This module, being the last of the pipeline, is also responsible for outputting the produced analysis to the user. This analysis includes a register of when each person

[1] It's important to clarify that this anomalous disappearances which generate alerts, are only an exemplification of an alert trigger, because the intended trigger are suspicious activities like theft, however to implement the technology for theft detection would be of an extreme complexity which falls out of the scope of this work.

entered or left an area and a video with the drawn bounding boxes and areas.

Besides these modules, which are common to all the pipelines, the primary pipeline includes an Apache Spark based module at the end, which is responsible for aggregating the results of all the previous modules to provide a final result to the user. The result of this module is an extensive list of the detected person's activities. It includes information regarding when every person entered and left the store, and entered and left a specific area within the the store. By using Apache Spark Structured Streaming, this result is continuously being updated by the processing of the incoming data stream.

The architectural design of the continuous video processing pipeline encapsulates state-of-the-art techniques in object detection, tracking, and people counting. By incorporating modules for anomaly detection, visualization, and user interaction, the system provides an auditable trail of events with user engagement. This design ensures that results are consistent with the code and processing steps, and that anomalies are addressed swiftly with verifiable evidence. This approach enhances the system's reliability, transparency, and accountability in real-time video processing scenarios.

As for the data flow inside this pipeline, just as the communication between modules and pipelines in the whole system, it is handled by Apache Kafka, and is depicted in Figure 3.3.

As pictured, the video source, which will provide the data stream, is received by the Module 0 (v1). This triggers the system to start. Module 0, depicted in Algorithm 3.1, starts by hashing the video source as well as the code files of the modules that are part of the Online Phase, and puts the resulting hash values in a file (hx). Finally, it hashes this same file for integrity purposes and sends the resulting hash value (hy), along with the video source received previously (x) to the following module. The hashing in our system was done with the use of the Python library, *hashlib*[2].

**Algorithm 3.1:** Module 0 of Primary Pipeline

```
Data: VIDEO_PATH, KAFKA_SERVER
Result: hash_value
Initialize KafkaProducer;
Hash video source and code files of the pipeline's following modules;
write($hash\_values$) to $HashFile$;
$hash\_value \leftarrow hashing(HashFile)$;
producer.send($hash\_value, VIDEO\_PATH$);
```

Module 1, pictured in Algorithm 3.2, will use the received video source to receive the video frames and process them using the chosen object detection and tracking technology, which was implemented through the Python libraries *ultralytics*[3] and *supervision*[4]. Once again for replay purposes, this module will have to periodically store the states of the Object Detection Model and Predictor on the State Store, as well as a predetermined amount of the last frames received and all of the alerts detected (ss), which are represented by an id, and include the frame number or timestamp and the id of the persons

[2]https://docs.python.org/3/library/hashlib.html
[3]https://pypi.org/project/ultralytics/
[4]https://pypi.org/project/supervision/0.8.0/

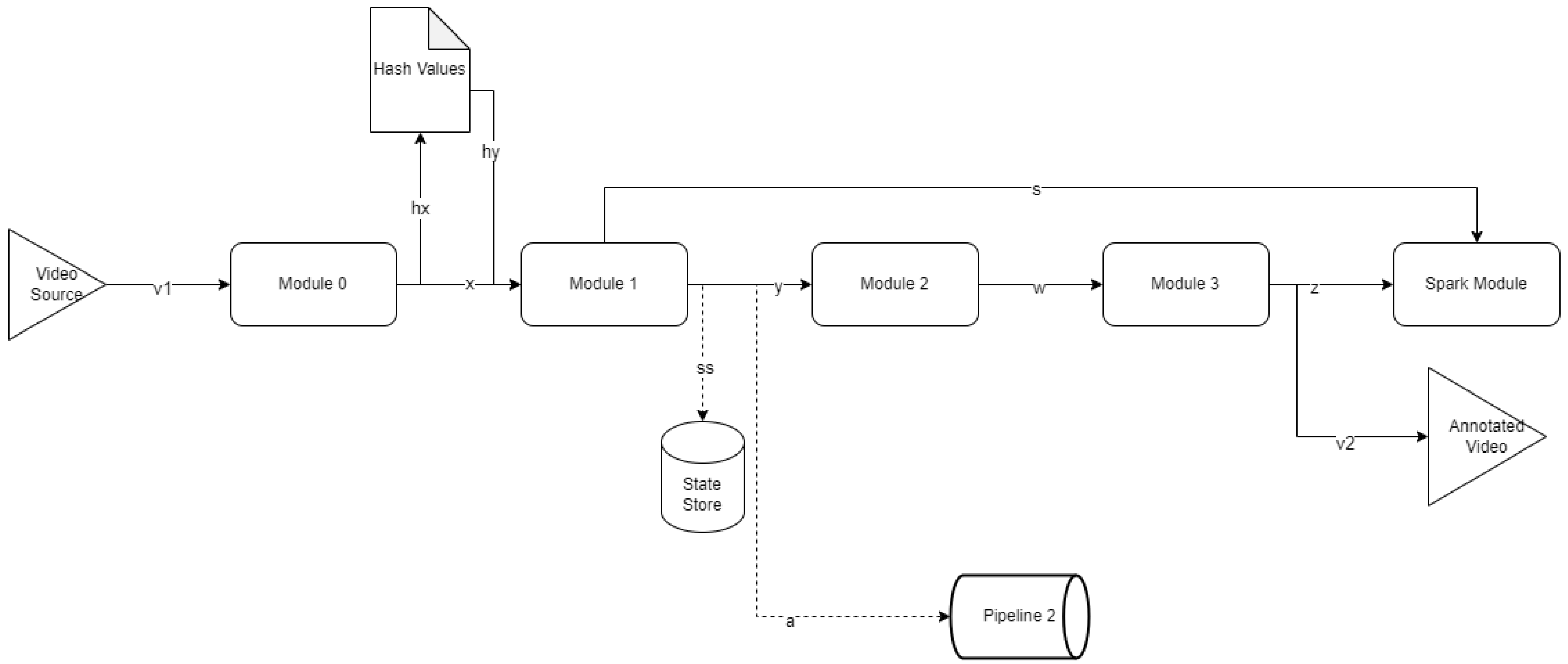

**Figure 3.3:** Data Flow in Primary Pipeline

involved in the suspected alert. These alerts are also sent to the secondary pipeline (a) with additional information, triggering its activation. This additional information consists of the entry and exit time of the person, and the coordinates of the person when they entered the store. Besides this data, Module 1, will also send the results representing the entering and leaving of people in the store and the hash value, produced by Module 0, to the Spark Module (s). Once the module finishes processing a frame, it sends it, along with the obtained detections, the frame count and the hash value received from the previous module (y), to the next module. The following data communications are very straightforward.

**Algorithm 3.2:** Module 1 of Primary Pipeline

```
Data: OBJECT_DETECTION_MODEL, KAFKA_SERVER
Result: detections, alerts, list of time when people entered and left store
Initialize KafkaConsumer;
Initialize KafkaProducer;
for $msg$ in $consumer$ do
    $hash\_value \leftarrow msg.hash\_value$;
    $video\_path \leftarrow msg.video\_path$;
    break;
$frame\_count \leftarrow 1$;
Initialize empty dictionary for alerts;
Initialize $OBJECT\_DETECTION\_MODEL$ on $video\_path$;
Initialize $DataFrame$ to store persons by id and in which frame they entered and left the store;
for $result$ in $model.track$ do
    $detections \leftarrow Detections(result)$;
    Update $DataFrame$ with $detections$;
    if $frame\_count$ equals $checkpoint$ then
        Store $model$ and $predictor$ on the state store;
    Compare $current\_frame$ with $previous_frame$;
    if Someone disappears unexpectedly then
        Create an $alert$ with key-value pair of frame number and id(s) of who disappeared
          unexpectedly;
        Add $alert$ to dictionary;
        Serialize and add detected alert to json file in state store;
        Send $alert$ to secondary pipeline;
    Send $frame$, $frame\_count$, $detections$ and $hash\_value$ to the following module;
    Send updated $DataFrame$ and $hash\_value$ to the Spark Module;
    Increment $frame\_count$;
Signal following module that data stream was interrupted;
Signal following pipeline that data stream was interrupted;
Terminate KafkaConsumer and KafkaProducer;
```

Module 2, depicted in Algorithm 3.3, draws the bounding boxes around the detected people and sends the annotated frames, as well as the detections, the frame count and hash value (w) received from the Module 1, to the following module.

**Algorithm 3.3:** Module 2 of Primary Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize PersonBoundingBoxAnnotator;
for msg in consumer do
    $frame \leftarrow msg.frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    if Signal informing of data stream interruption is received then
        Send signal to following module;
        break;
    Draw bounding boxes around every detected person in this $frame$;
    Send $annotated\_frame$, $frame\_count$, $hash\_value$ and $detections$ to the following module;
Terminate KafkaConsumer and KafkaProducer;
```

And finally, Module 3, pictured in Algorithm 3.4, will use the data received from the previous modules to draw the designated areas and count the people in each area and will send those results and the received hash value (z) to the Spark Module. Finally, the annotated frames will be used to produce a video with the drawns areas and the detected persons (v2).

**Algorithm 3.4:** Module 3 of Primary Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames, list of time when people entered and left store zones
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize BoundingBoxAnnotator;
Define $zones$;
Initialize $DataFrame$ to store persons by id and in which frame they entered and left each zone;
for msg in consumer do
    $annotated\_frame \leftarrow msg.annotated\_frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    if Signal informing of data stream interruption is received then
        break;
    Draw $zones$ in $annotated\_frame$;
    Update $DataFrame$ with $detections$ per $zone$;
    Send updated $DataFrame$ and $hash\_value$ to the Spark Module;
    Output $annotated\_frame$ to the user;
Terminate KafkaConsumer and KafkaProducer;
```

### 3.3.2 Secondary Pipeline or Validation Pipeline

The secondary pipeline represents a critical component of the system's architecture, activated exclusively in response to alerts triggered by the primary pipeline. Its main purpose is to validate the results produced by the primary pipeline, ensuring the accuracy and reliability of the detected anomalies, and disregarding possible false-positives. The secondary pipeline comprises a set of interconnected modules, each mirroring the functions of the primary pipeline while utilizing preserved state information and stored images to replicate results.

However, the continuous processing of image and video streams presents a formidable challenge, characterized by real-time demands and intricate dependencies. In the context of our system, where seamless surveillance and timely anomaly detection are paramount, the primary pipeline provides crucial real-time insights. However, the relentless nature of the video stream makes it impossible to precisely replicate the same processing conditions encountered in the initial alert-triggering situation.

The essence of the secondary pipeline lies in its recognition of the inherent complexities and dependencies that emerge during continuous video processing. While it may not be possible to rewind and recreate the exact sequence of dependencies leading to an alert, the secondary pipeline takes an ingenious approach. It acknowledges that, due to the unbroken nature of video streams, processing conditions may vary slightly with each iteration. Consequently, results may differ to some extent.

The key value proposition of the secondary pipeline is its commitment to accuracy and reliability in alert validation. By processing the same images and frames but possibly generating slightly different results, it offers a unique advantage. If, despite these subtle differences, both the primary and secondary pipelines trigger an alert for the same situation, a strong and replicable validation signal emerges. This occurrence underscores the alert's validity, essentially confirming it as a genuine alert situation rather than a false positive.

In essence, the secondary pipeline becomes an effective filter for the alerts generated by the primary pipeline. It leverages the inherent variations of continuous video processing to validate alerts under real-world conditions. This innovative approach ensures that alerts are not dismissed due to minor discrepancies but are instead validated through the convergence of multiple processing paths.

As mentioned before, this pipeline, represented in Figure 3.4, is activated when an alert (a) is received from the primary pipeline. Once an alert is received by Module 0, this module, pictured in Algorithm 3.5, will hash the video source (ss1) and the code files of the modules that are part of the Online Phase, just like in the primary pipeline, these hash values will be placed in a file (hx), which will then be hashed. The resulting hash will be used to guarantee the integrity of the system. If the resulting hash value does not match the value obtained and stored by the primary pipeline, the system will be interrupted. If not, the resulting hash value will be sent to the following module (hy) and system will proceed normally. This module will also send the information obtained with the alert, consisting of the suspect person id, the alert id, the entry an exit time, and the entry coordinates, to Module 1 (x). Besides this, also the Object Detection Model and Predictor states, which were stored in the State Store in the periodically checkpoint done in the primary pipeline, and the frames stored are obtained by the Module

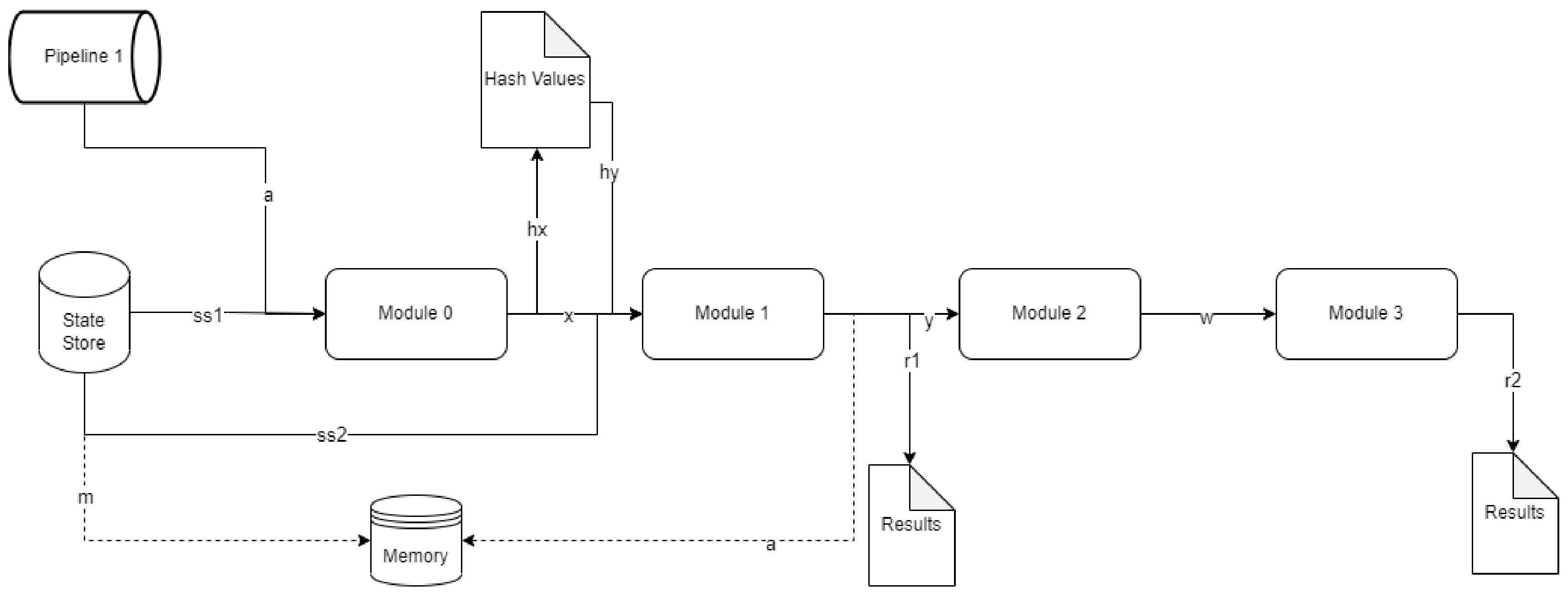

**Figure 3.4:** Data Flow in Secondary Pipeline

1 (ss2) to replay the processing of the data stream from the earliest avaialable frame where the person was detected as similarly as done by the primary pipeline.

**Algorithm 3.5:** Module 0 of Secondary Pipeline

```
Data: KAFKA_SERVER
Result: hash_value
Initialize KafkaProducer;
Initialize KafkaConsumer;
for $msg$ in $consumer$ do
    $person \leftarrow msg.person$;
    $entry\_frame \leftarrow msg.entry\_frame$;
    $exit\_frame \leftarrow msg.exit\_frame$;
    $entry\_coordinates \leftarrow msg.entry\_coordinates$;
    $alert\_id \leftarrow msg.alert\_id$;
    if Signal informing of data stream interruption is received then
        Send signal to following module;
        break;
    Hash video source and code files of the pipeline's following modules;
    write($hash\_values$) to $HashFile$;
    $hash\_value \leftarrow hashing(HashFile)$;
    if Obtained $hash\_value$ differs from $hash\_value$ from Primary Pipeline then
        Interrupt system;
    producer.send($person$, $entry\_frame$, $exit\_frame$, $entry\_coordinates$, $alert\_id$, $hash\_value$,
     $VIDEO\_PATH$);
Terminate KafkaConsumer and KafkaProducer;
```

It's important to note that the information obtained with previous pipeline's alert is crucial in easing the workload in this pipeline. With the entry coordinates of the suspected person calculated in the primary pipeline, we can cross reference with the results obtained in this pipeline for the same timestamp or frame, and hence we can identify, with a greater degree of certainty, the same suspect individual.

**Algorithm 3.6:** Module 1 of Secondary Pipeline

```
Data: OBJECT_DETECTION_MODEL, KAFKA_SERVER
Result: detections, alerts, list of time when suspects entered and left store
Initialize KafkaConsumer;
Initialize KafkaProducer;
for $msg$ in $consumer$ do
    $hash\_value \leftarrow msg.hash\_value$;
    $video\_path \leftarrow msg.video\_path$;
    $person \leftarrow msg.person$;
    $entry\_frame \leftarrow msg.entry\_frame$;
    $exit\_frame \leftarrow msg.exit\_frame$;
    $entry\_coordinates \leftarrow msg.entry\_coordinates$;
    $alert\_id \leftarrow msg.alert\_id$;
    Allocate memory space to save results for $alert\_id$;
    if Signal informing of data stream interruption is received then
        Send signal to following module;
        break;
    Identify last checkpoint made before $entry\_frame$;
    Get Object Detection Model and Predictor states, and frames corresponding to the
      checkpoint from state store;
    Initialize OBJECT_DETECTION_MODEL;
    Initialize $DataFrame$ to store suspect person's entry and leaving times;
    $frame\_count \leftarrow checkpoint\_time$;
    for $result$ in $model.track$ do
        $detections \leftarrow Detections(result)$;
        if $frame\_count$ equals $entry\_frame$ then
            Identify suspect person using $entry\_coordinates$;
            $person \leftarrow person\_id$;
            Update $DataFrame$ with $detections$;
        if $frame\_count$ greater than $entry\_frame$ then
            Update $DataFrame$ with $detections$;
            Compare $current\_frame$ with $previous_frame$;
            if Suspect disappears unexpectedly and $frame\_count$ equals $exit\_frame$ then
                Create an $alert$ with id of the suspect, entry time and exit time;
                Store that alert along with frames and checkpoint necessary for replay in
                  permanent memory space for this $alert\_id$;
        Send $person\_id$, $frame$, $frame\_count$, $detections$, $alert\_id$ and $hash\_value$ to the
          following module;
        Increment $frame\_count$;
    Save results of $DataFrame$ along with $hash\_value$ in file at memory space for $alert\_id$;
Terminate KafkaConsumer and KafkaProducer;
```

**Algorithm 3.7:** Module 2 of Secondary Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize PersonBoundingBoxAnnotator;
for msg in consumer do
    $person\_idgetsmsg.person\_id$ $frame \leftarrow msg.frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    $alert\_id \leftarrow msg.alert\_id$;
    if Signal informing of data stream interruption is received then
        Send signal to following module;
        break;
    Draw bounding boxes around every detected person in this $frame$;
    Send $person\_id$, $annotated\_frame$, $frame\_count$, $detections$, $alert\_id$ and $hash\_value$ to the
      following module;
Terminate KafkaConsumer and KafkaProducer;
```

**Algorithm 3.8:** Module 3 of Secondary Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames, list of time when suspect entered and left store zones
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize BoundingBoxAnnotator;
Define $zones$;
Initialize $DataFrame$ to store frame number or timestamp of when suspect entered and left a
  zone;
$old\_alert \leftarrow 1$;
for msg in consumer do
    $person\_idgetsmsg.person\_id$;
    $annotated\_frame \leftarrow msg.annotated\_frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    $alert\_id \leftarrow msg.alert\_id$;
    if Signal informing of data stream interruption is received then
        break;
    if $alert\_id$ not equals $old\_alert$ then
        Store $DataFrame$ in a file at the allocated memory for $alert\_id$;
        Initialize $DataFrame$ to store frame number or timestamp of when suspect entered and
          left a zone;
    Draw $zones$ in $annotated\_frame$;
    Update $DataFrame$ with $detections$ per $zone$ for the suspect;
    $old\_alert \leftarrow alert\_id$;
Terminate KafkaConsumer and KafkaProducer;
```

After Module 1, depicted in Algorithm 3.6, processes the received data, if the results obtained confirm the alert which originally triggered the pipeline, the newly obtained alert, consisting of the suspects id and the timestamp or frame number of the alert, will be stored in permanent memory (a), along with the videos frames, Object Detection Model and Object Detection Predictor (m), stored in State Store and needed for the replay. Module 1 will also store its results for this alert replay, storing in a file the exact coordinates of every detected person in every processed frame (r1), and it will send the obtained results and the hash value (y), received previously, to the next module.

Module 2, described in Algorithm 3.7, will perform the same operations as done in the primary pipeline and send the frames with the drawn bounding boxes, along with the hash value and object detections received from Module 1, to Module 3. Finally, Module 3, depicted in Algorithm 3.8, will draw the designated areas in the frames and perform the processing of the received data, providing a result of people per area by frame, stored in a file (r2).

### 3.3.3 Third Pipeline or Replay Pipeline

The ability to replicate and verify results stands as a critical pillar of data provenance in a system, ensuring reliability and accountability. The third pipeline within our architectural design is engineered to deliver precisely this capability, ensuring that results can be audited and validated with precision.

The primary function of the third pipeline is to replicate the processes carried out in the secondary pipeline. This replication can only be initiated when the secondary pipeline confirms the presence of an anomaly, originally detected by the primary pipeline. When this confirmation is received, the video section that corresponds to the alert is permanently stored, alongside the essential state information required to reproduce the entire process.

Unlike the primary pipeline, which continually processes an uninterrupted video stream, both the secondary and third pipelines focus on replicating specific video sections. This ensures that dependencies on previous frames are minimized, allowing for the recreation of exactly identical results. The third pipeline's ability to replay the same video section, using identical state information, is instrumental in establishing result consistency by producing the same exact results as the validation pipeline.

It's important to note that the third pipeline is not automatically triggered upon confirmation from the secondary pipeline, unlike the relation between the primary and secondary pipeline. Instead, it remains dormant until manual activation is warranted, typically for audit purposes. This means that in the event of an anomaly confirmation, the associated video section and state information must be securely stored, and ready to be revisited and re-validated at a later time.

In essence, the third pipeline allows the user to re-validate a previously detected alert without interfering with the continuous functionality of the primary and secondary pipeline, and its data flow is represented in Figure 3.5.

The third pipeline, in contrast with the other pipelines, is manually activated. The user must indicate an alert stored in memory, as well as an id corresponding with a suspect identified in that alert, to trigger

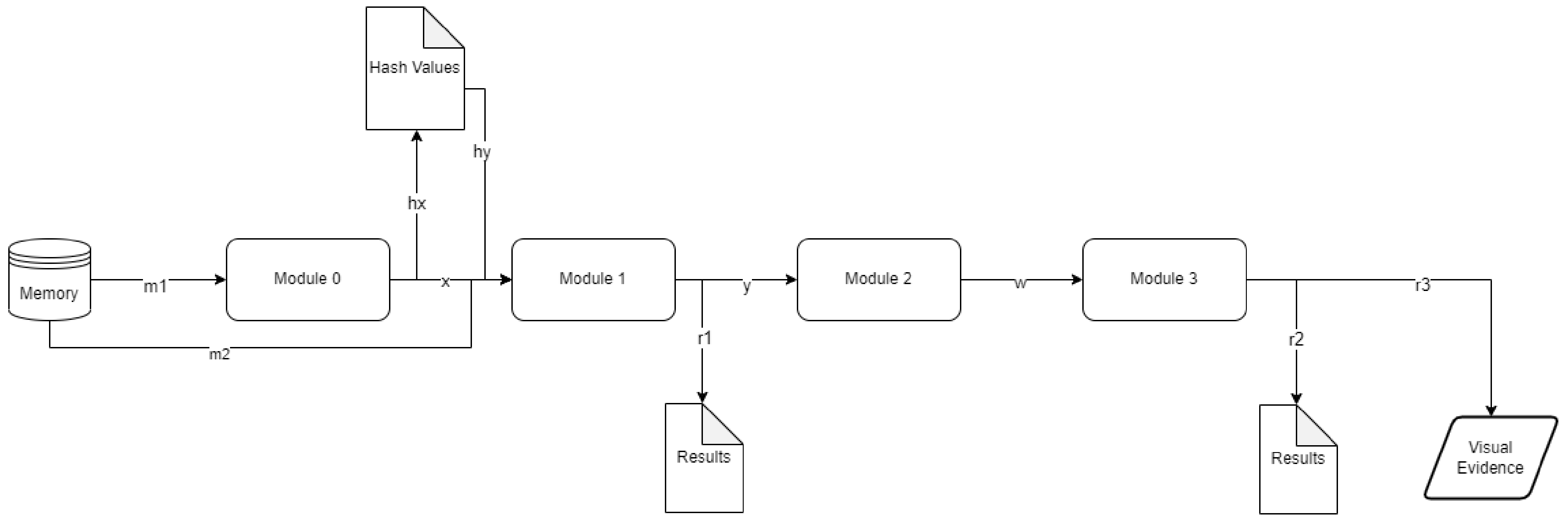

**Figure 3.5:** Data Flow in Third Pipeline

the replay of the data stream processing. After this data is selected by the user, Module 0 will compare it with the alerts stored in memory to check its existence (m1). If the existence of this alert is confirmed, it will obtain from memory the entry time and exit time of the suspect, as well as the video source (m1), to hash it and the code files of the following modules. Just like in the previous pipelines, these resulting hash values will then be put in a file (hx) and the hash value of the hashing of that same file will be sent to the following module (hy), as well as each video frame (x). Module 1 will receive this data and will get the Object Detection Model and Predictor associated with the alert being investigated (m2), which is stored in memory.

After processing the data received it will produce detailed results of detections with the coordinates of each detected person's exact coordinates at a certain time or frame. This will be put in a file (r1), for later analysis, and sent along with the previously received hash value to the following module (y). Module 2 will perform the same operations as done in the primary and secondary pipelines, and send the frames with the drawn areas, along with the hash value and object detections received from Module 1, to Module 3.

Finally, Module 3 will perform the processing of the received data, providing a result of people per area by frame, stored in a file (r2). In contrast with the two previous pipelines, after Module 3 processes the data stream, it stores in memory the video frames with annotated areas and bounding boxes identifying every detected person (r3). This will allow the user to perform a visual analysis of the results, confirmed by the secondary pipeline, by watching the resulting video or each individual frame.

**Algorithm 3.9:** Module 0 of Third Pipeline

```
Data: KAFKA_SERVER, alert_id chosen by user, person_id chosen by user
Result: hash_value
Initialize KafkaProducer;
Initialize KafkaConsumer;
if User given $alert\_id$ and $person\_id$ is confirmed by information in memory then
    Get video source, $entry\_time$ and $exit\_time$ from memory;
    Hash video source and code files of the pipeline's following modules;
    write($hash\_values$) to $HashFile$;
    $hash\_value \leftarrow hashing(HashFile)$;
else
    Terminate system
producer.send($person\_id$, $entry\_frame$, $exit\_frame$, $alert\_id$, $hash\_value$);
Terminate KafkaConsumer and KafkaProducer;
```

**Algorithm 3.10:** Module 1 of Third Pipeline

```
Data: OBJECT_DETECTION_MODEL, KAFKA_SERVER
Result: detections, alerts, list of time when suspects entered and left store
Initialize KafkaConsumer;
Initialize KafkaProducer;
for $msg$ in $consumer$ do
    $hash\_value \leftarrow msg.hash\_value$;
    $person\_id \leftarrow msg.person\_id$;
    $entry\_frame \leftarrow msg.entry\_frame$;
    $exit\_frame \leftarrow msg.exit\_frame$;
    $alert\_id \leftarrow msg.alert\_id$;
    break;
Allocate memory space to save results for $alert\_id$;
if Signal informing of data stream interruption is received then
    Send signal to following module;
    break;
Identify last checkpoint made before $entry\_frame$;
Get Object Detection Model and Predictor states, and frames corresponding to the checkpoint
  from memory;
Initialize OBJECT_DETECTION_MODEL;
Initialize $DataFrame$ to store suspect person's coordinates at all times;
$frame\_count \leftarrow checkpoint\_time$;
for $result$ in $model.track$ do
    $detections \leftarrow Detections(result)$;
    $detected \leftarrow$ False;
    if $frame\_count$ equal or greater than $entry\_frame$ then
        Filter out person with id different than $person\_id$;
        Update $DataFrame$ with $detections$;
        if $person\_id$ detected then
            $detected \leftarrow$ True;
    Send $detected$, $person\_id$, $frame$, $frame\_count$, $detections$, $alert\_id$ and $hash\_value$ to the
      following module;
    Increment $frame\_count$;
Save results of $DataFrame$ along with $hash\_value$ in file at memory space for $alert\_id$;
Terminate KafkaConsumer and KafkaProducer;
```

**Algorithm 3.11:** Module 2 of Third Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize PersonBoundingBoxAnnotator;
for msg in consumer do
    $detected \leftarrow msg.detected$;
    $person\_idgetsmsg.person\_id$;
    $frame \leftarrow msg.frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    $alert\_id \leftarrow msg.alert\_id$;
    if detected is True then
        Draw bounding boxes around suspect person in this $frame$;
    Send $person\_id$, $annotated\_frame$, $frame\_count$, $detections$, $alert\_id$ and $hash\_value$ to the
      following module;
Terminate KafkaConsumer and KafkaProducer;
```

**Algorithm 3.12:** Module 3 of Third Pipeline

```
Data: KAFKA_SERVER
Result: annotated_frames, list of time when suspect entered and left store zones
Initialize KafkaConsumer;
Initialize KafkaProducer;
Initialize BoundingBoxAnnotator;
Define $zones$;
Initialize $DataFrame$ to store frame number or timestamp of when suspect entered and left a
  zone;
for msg in consumer do
    $detected \leftarrow msg.detected$;
    $person\_idgetsmsg.person\_id$ $annotated\_frame \leftarrow msg.annotated\_frame$;
    $frame\_count \leftarrow msg.frame\_count$;
    $hash\_value \leftarrow msg.hash\_value$;
    $detections \leftarrow msg.detections$;
    $alert\_id \leftarrow msg.alert\_id$;
    if detected is True then
        Draw $zones$ in $annotated\_frame$;
        Update $DataFrame$ with $detections$ per $zone$ for the suspect;
        Store $annotated\_frame$ at the allocated memory for $alert\_id$;
Store $DataFrame$ in a file at the allocated memory for $alert\_id$;
Terminate KafkaConsumer and KafkaProducer;
```

## 3.4 Data Structures

Data structures are fundamental building blocks in computer science and software development, providing efficient ways to organize, store, and manipulate data. They have a wide range of applications across various domains and are essential for optimizing algorithms and data processing. In this section we will elaborate on the usage of data structures in our system, while maintaining data lineage, ensuring data quality, and facilitating efficient querying and replaying.

In our system the data structures present one common aspect, the presence of hash values. As mentioned previously, Module 0 is a module with the main functionality of hashing the files necessary for the processing of the incoming data stream. With the use of the SHA-256 algorithm, this module hashes the video source and the files of each of the subsequent modules, it then saves these values in a designated file. At the end of the pipeline these values are then used to provide a correlation between the incoming data and the produced results by integrating the hash value of the file containing the hash value of the video source and the checksum of the pipeline modules. This process is common to the three pipelines and ensures a robust and reliable method for ensuring data integrity in the system.

However, in the second and third pipeline the use of hashing is enhanced due to the need of providing an in-depth look into each module and their intermediate results to better understand the final results. In these pipelines the checksum of each module's file is added to a file similar to the one used in the primary pipeline, but containing only the module's checksum and the video source hash value. The logic and objective are the same of the one used in the primary pipeline but prioritizing the granularity that distinguishes these pipelines from the primary one.

Another reason for data structures being fundamental components of our system, is the crucial role they play by representing events, like entries and exits of individuals in specific areas of a store or in the store itself. In our use case the data structures used are characterized by having the following composition $<Id_{person}, Id_{zone}, Time_{entry}, Time_{exit}>$.

- $Id_{person}$: Id of individual in the store.
- $Id_{zone}$: Id of store zone. 0 for the store itself and integers greater than 1 for areas inside the store.
- $Time_{entry}$: Time of entry of an individual in a store zone defined by timestamp or frame number.
- $Time_{exit}$: Time of exit of an individual in a store zone defined by timestamp or frame number.

For the second and third pipeline additional data is included in these data structures to provide a more in-depth understanding of the processing that leads to the final results. This data is comprised of 4 integers, $x_1$, $y_1$, $x_2$, $y_2$, which represent the 4 corners of the bounding box drawn when a person is detected in a frame.

Another crucial type of event represented by data structures is the alert. In our system an alert is composed by $<Id_{person}, Time_{entry}, Time_{exit}, x1_{entry}, x2_{entry}, y1_{entry}, y2_{entry}, , Id_{alert}>$.

- $Id_{person}$: Id of individual in the store.
- $Time_{entry}$: Time of entry of an individual in a store zone defined by timestamp or frame number.

- $Time_{exit}$: Time of exit of an individual in a store zone defined by timestamp or frame number.
- $x1_{entry}$, $x2_{entry}$, $y1_{entry}$, $y2_{entry}$: Coordinates of person's bounding box at time of entry.
- $Id_{alert}$: Id of alert.

Regarding the data needed to be stored in the state store to enable the replay functionality of the second and third pipelines, data structures also play an important part. As mentioned before, the primary pipeline periodically stores data in the state store. The periodicity of this checkpoint can be set by the user and will have an impact on how the second and third pipeline perform. In our use case, three key elements must be stored in the state store:

**Object Detection Model:** Exporting the object detection model used to a TorchScript file allows the system to deploy the same object detection model with all the alterations it may have suffered in a new pipeline more easily, ensuring consistent results in different pipelines.

**Object Detection Predictor:** The predictor in the object detection model is responsible for processing the image features and producing predictions. It plays a critical role in transforming image features into actionable object detection results. It's a crucial component that enables the model to identify and locate objects in real-time within input images or video frames. Just like for the Object Detection Model, saving the state of this model's predictor into a serialized format, allows us to ensure consistent results when replaying the data stream processing in the second or third pipeline.

**Alerts:** To keep a log of all the alert situations flagged by the primary pipeline, a JSON file stores all the key-value pairs which represent an alert, where the key is a combination of the frame number or timestamp of the alert and the identifier of the video, which can be the hash of the video source or a unique identifier, and the value is the id of the person which became unexpectedly undetected or multiple ids in case there is more than one person unexpectedly becoming undetected.

Besides these elements, it is fundamental to keep a user-defined quantity of video frames in the state store. This user-set frame retention ensures that a history of video frames is continuously stored, allowing for the secondary pipeline to replay from the desired checkpoint and, hence, enabling essential capabilities like result validation, auditing, and anomaly investigation. In case of an alert validation, some of these frames will then be stored in the system's permanent memory for posterior use by the third pipeline. This functionality, depicted in Figure 3.6, optimizes resource management within our stream processing system.

Despite the object detection models being deep learning models, the neural network's parameters are fixed through the processing of data. However, due to the nature of our system and the existence of multiple pipelines running simultaneously, we implemented this checkpoint to ensure the correct object detection model and predictor are always used and to prevent a situation where a change in the object

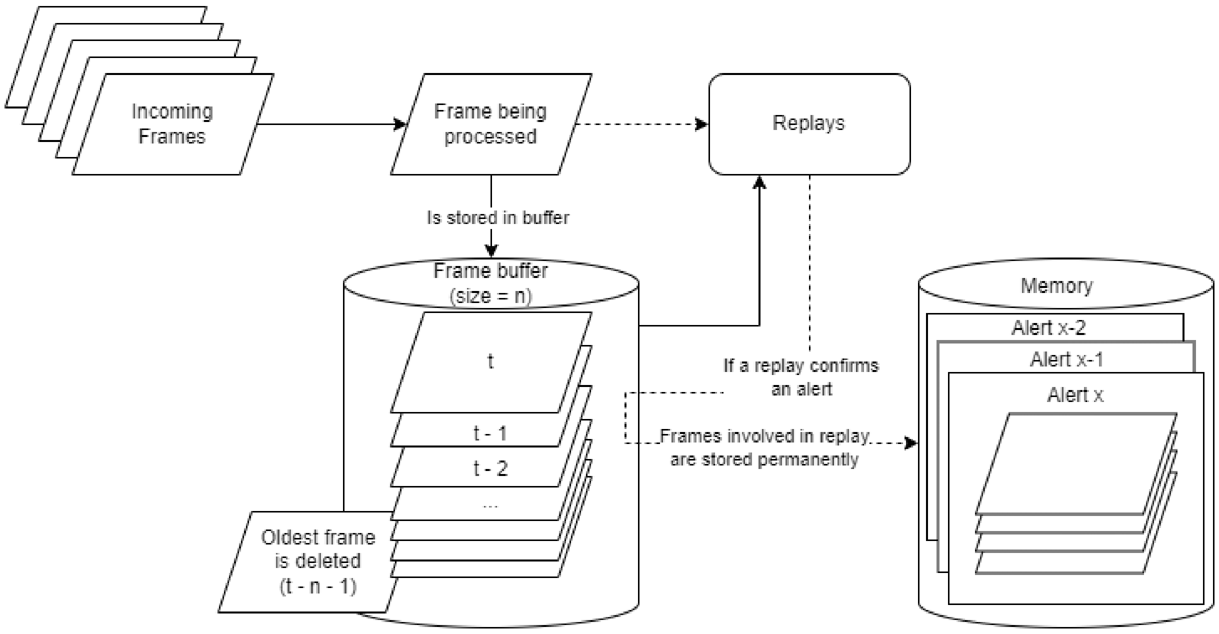

**Figure 3.6:** Process of frame storage

detection module will cause the execution of the second and third pipeline for queued alerts to be done with the new module, instead of with the intended old one, which could lead to inconsistent results. It is also important to note that this data should be stored in a serialized format, like Torchscript or JSON, or even Pickle in a python-to-python case, like ours. In a python environment, the libraries *pickle*[5], *jsonpickle*[6] and *shutil*[7] are very useful.

This usage of data structures is fundamental for ensuring provenance and traceability in our system by ensuring:

**Granular Data Representation:** The data structures encapsulate granular information about each event, including the exact time or frame number in the video, the unique identifier of the person, the identifier of the area (store or specific sections), and the action (entry or exit). This level of detail is essential for reconstructing the sequence of events accurately.

**Temporal Traceability:** By recording the precise time or frame number of each event, the system can establish a temporal order for entries and exits. This temporal traceability is critical for understanding the chronological flow of activities, detecting anomalies, and providing a comprehensive audit trail by allowing the system to replay the data stream processing from the most adequate checkpoint.

**Spatial Traceability:** The inclusion of area identifiers in the data structures allows our system to track which specific regions of the store individuals are entering or leaving. This spatial traceability

[5]https://docs.python.org/3/library/pickle.html
[6]https://pypi.org/project/jsonpickle/
[7]https://docs.python.org/3/library/shutil.html

is valuable for monitoring traffic patterns, identifying areas of interest, and addressing security or operational concerns in different sections of the store.

**Action Classification:** Distinguishing between entry and exit actions within the data structures enables the system to categorize and analyze events effectively. This classification helps in understanding customer behaviors, counting foot traffic, and detecting unusual activities, such as individuals entering restricted zones or theft.

**Provenance Verification:** Data structures that capture these event details serve as the basis for provenance verification. They allow our system to confirm the origin and history of each event, ensuring that the recorded events are consistent with the processing logic and that there are no unauthorized modifications or discrepancies.

**Replay and Audit:** In scenarios where anomalies or suspicious events occur, the data structures enable replay and audit capabilities. You can trace back to the original events, replay the processing steps, and validate the results, while being certain of the integrity of the data and the processing, thanks to the use of hashing. This is especially critical in addressing false positives or investigating security incidents.

**Cross-Referencing:** The data structures facilitate cross-referencing between different modules or pipelines within the system. For example, if an alert is triggered in one module and the secondary pipeline is activated, the data structures ensure that the events replayed in the secondary pipeline correspond to the same events detected in the primary pipeline, thus maintaining consistency and trustworthiness.

**Data Integrity:** By using structured data representations, our system can maintain data integrity throughout the processing workflow. This includes hash-based verification of code files, ensuring that processing results are consistent with the original code.

As a whole, the data structures representing entry and exit events, along with their associated details and the usage of a state store to allow accurate and trustworthy replay of data stream processing, serve as the backbone of our system's provenance tracking and auditability. They provide the means to maintain a reliable record of events, detect anomalies, and verify the correctness of processing steps. This level of data granularity and traceability is essential for ensuring the transparency, accountability, and trustworthiness of the stream processing system.

## 3.5 Summary

In this chapter, we embarked on a comprehensive exploration of the architecture design of our proposed solution and its implementation, beginning with the elucidation of our real-world use case within the realm of video surveillance and data provenance. We discerned the specific requirements and objectives necessitated by our use case, laying the groundwork for the subsequent design and development phases.

By presenting a detailed architectural design, we revealed the underlying framework of our solution, highlighting the vital role of the three pipelines, each composed of four distinctive modules. These modules, carefully crafted to fulfill unique functionalities such as object detection, tracking, and data visualization, form the bedrock of our system.

Crucially, we've provided an in-depth elucidation of how data navigates through this intricate system, illustrating the flows, transformations, and key data structures that underpin our data processing pipeline. These explanations serve to underscore the cohesiveness and coherence of our design, which is finely tuned to meet the stringent requirements of our use case, and enable complete and correct data provenance.

In essence, this chapter serves as the foundational guide to understanding the mechanics of our system, equipping us with the knowledge needed to undertake its evaluation and further analysis. As we transition from architectural insights to the practical evaluation phase, the subsequent chapters will shed light on the performance, robustness, and resilience of our system in real-world scenarios.

# 4 Evaluation

**Contents**

This chapter delves into the comprehensive evaluation of the system designed for auditable data provenance in stream processing. The system's performance and efficiency are examined under controlled conditions and real-world scenarios. The evaluation unfolds in multiple phases, allowing us to analyze different aspects of the system's behavior in distinct situations.

The initial phase of our evaluation process begins with local testing, which provides valuable insights into latency, throughput, and resource utilization, and also aims to show the precision and correctness of our solution. Here, we employ a set of Apache Spark Structured Streaming[1] benchmarks, specifically designed to simulate real-world workloads with varying degrees of complexity and randomness.

Subsequently, the integration of Prometheus[2] and Grafana[3] offers an in-depth analysis of latency and throughput metrics within the online phase of the system. Every individual module within the first two pipelines is scrutinized, along with an evaluation of the complete pipeline's performance. Simultaneously, Node Exporter[4] allows us to closely monitor CPU and memory utilization to understand the system's resource requirements.

The two-phase testing strategy encompasses local and networked scenarios. The local testing phase isolates the system, providing insights into its core performance without the confounding variables of network overhead. In contrast, the networked testing phase introduces real-world network conditions, allowing us to assess the system's behavior in a more challenging environment. Together, this comprehensive evaluation process provides a complete perspective on the system's performance, enabling us to draw meaningful conclusions and insights that contribute to the viability of the implementation of our designed architecture.

## 4.1 Local Testing

The initial phase of the evaluation process focuses on local testing, aiming to assess the system's core performance in an isolated environment. This phase serves as a foundational step in understanding the system's behavior while eliminating potential confounding variables, such as network latency.

Local testing provides a unique advantage by isolating the system from the potential network overhead that can distort performance metrics. This isolation enables us to establish a solid baseline of the system's capabilities, untainted by external factors. By evaluating the system's performance under controlled local conditions, we gain a foundational understanding of its capabilities and challenges. The insights gleaned during this testing phase lay the groundwork for subsequent evaluations, including those conducted in more complex networked scenarios.

[1] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[2] https://prometheus.io/
[3] https://grafana.com/
[4] https://prometheus.io/docs/guides/node-exporter/

Regarding the used testbed, in the first phase of testing, the evaluation was conducted on a local machine with the following specifications:

**Operating System:** Ubuntu 22.04 LTS

**CPU Model:** Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (Octa-core)

**Clock Speed:** 2800 MHz

**Total Memory:** 11 GB

**Used Memory:** 5.9 GB

**Available Memory:** 3.8 GB

This machine's octa-core CPU, ample memory, as compared to typical cloud cluster, and Linux operating system provided a suitable environment for conducting the initial benchmarks. It can be regarded as an example of a mid/high level machine part of a typical cluster. These hardware specifications were essential in understanding the system's performance in a controlled local environment.

### 4.1.1 Benchmark Testing

To establish a benchmark for assessing system performance, we implement two Apache Spark Structured Streaming benchmarks. These benchmarks, designed with the explicit inclusion of random factors, are engineered to replicate real-world workloads, where the processing of data is seldom predictable. The inclusion of random factors have the aim of simulating the need to store arbitrary data, which is random or susceptible to changes over time, and is crucial for the enabling of data provenance and assess deterministic replayability.

The first benchmark, or benchmark A, while relatively simple in terms of processing operations, introduces random variables into the computation. This deliberate introduction of randomness offers insights into the system's ability to handle unexpected variations in data, a critical aspect of real-time data processing. We meticulously measure the system's response time and resource utilization during this benchmark.

The second benchmark, benchmark B, inspired by the work presented by Chintapalli et al. [46], is more complex, simulating advanced operations. In addition to introducing randomness, these operations stress-test the system's processing capabilities, mimicking scenarios where advanced data transformations are needed.

This being said, to test the benchmarks we created two datasets comprised of thousands of events each, which were fed to each benchmark by a Kafka stream. Regarding benchmark A, each event was represented as $<id, value, time_{event}, num_{random}, hash>$, where $num_{random}$ is a random value between 1 and 100 which occurred once every 100 events. For the remaining events this value was always 0. Benchmark A performed the following operations:

- Sum $value$ with $num_{random}$, creating the variable $final\_value$;
- Store events with $num_{random}$ bigger than 0;
- Calculate the average value of $final\_value$;

For benchmark B, each event was represented as $<id_{event}, id_{user}, id_{page}, id_{ad}, type_{ad}, type_{event}, time_{event}, price, hash>$, where the random value was a combination of $price$ and a calculated variable, $id_{campaign}$, which gathers several ads under one campaign. Benchmark B performed the following operations:

- Filter out events with $type_{event}$ different than $"purchase"$;
- Filter each event to only include the variables $id_{ad}$, $time_{event}$, $hash$ and $price$;
- Calculate $id_{campaign}$, which is directly dependent of $id_{ad}$;
- Store price of campaigns across time;
- Join events with respective $id_{campaign}$;
- Calculate total profit of each $id_{campaign}$;

The source code of each benchmark is presented in Appendix A and the results obtained for each benchmark, along with resource utilization results obtained during the normal usage of the machine, can be seen in Table 4.1:

**Table 4.1:** Average benchmark testing results

| | Time to read events | Time to process events | CPU Usage | RAM Usage |
|---|---|---|---|---|
| **Benchmark A** | 1.98 s | 1.14 s | 85.5% | 98.33% |
| **Benchmark B** | 2.01 s | 1.61 s | 87.9% | 98.83% |
| **Normal Usage** | - | - | 4.5% | 50.5% |

It's important to acknowledge that the benchmarks and our system are not perfectly comparable due to several key distinctions. Firstly, the benchmarks are built on Apache Spark Structured Streaming, which inherently operates on a micro-batching model. This means that data is processed in discrete chunks or micro-batches, while the modules of your system are tailored to handle incoming data frame by frame. These contrasting processing paradigms impact how latency and throughput are measured and can lead to variations in the results.

Additionally, the nature of the data being processed sets our system apart. While the benchmarks operate on raw data and execute relatively simple operations, our system is designed to tackle more computationally intensive tasks, such as processing images. The richness and complexity of image data can substantially influence resource utilization and overall performance, introducing challenges that may not be reflected in simpler benchmark scenarios.

Recognizing these differences is crucial to ensuring a fair and meaningful evaluation. While our benchmarks offer valuable insights into the system's performance, it's essential to interpret the results with these distinctions in mind, particularly when comparing resource utilization metrics.

However, despite these differences, it's still good practice to obtain and analyze these metrics to better understand the performance of less demanding and even alternative data processing solutions in our testing environment. Even though the testing is done in a controlled environment, the results of our system's testing are still subject to external factors related to the environment itself, and the analysis of the benchmarks' performance is critical to understand how these results are being affected by those factors, allowing us to have a more profound understanding of our system's performance.

### 4.1.2 System Testing

In this section, we delve into the comprehensive analysis of the primary and secondary pipelines' performance metrics:

**Latency**

**Throughput**

**Resource Usage** (CPU & RAM)

The data for this analysis was collected from processing a 5-minute video stream at a rate of 25 frames per second and was meticulously acquired through Prometheus, Grafana, and Node Exporter, enabling a multifaceted evaluation of the system's capabilities.

The data collected allowed us to calculate the following latency measurements useful for the latency analysis:

**Average Latency:** The average latency measures how long, on average, it takes for a frame to traverse the modules within the pipelines. It provides insight into the typical processing time for each frame.

**Total Latency through 15 Seconds:** Calculating the total latency over a 15-second window allows us to understand the cumulative delay experienced by frames during short intervals.

**Percentiles for Latency**

**25th Percentile:** This metric denotes the latency value below which 25% of frames fall, providing insight into the efficiency of the best performance periods in processing.

**50th Percentile (Median):** The median latency showcases the point where half of the frames have been processed. It highlights the central tendency of the latency distribution.

**75th Percentile:** The 75th percentile indicates the latency value below which 75% of frames are processed. It reveals the performance of the majority of the data.

**90th Percentile:** Highlighting the latency that 90% of frames fall below, the 90th percentile exposes the performance for the bulk of the data, relevant for a regular SLA.

**95th Percentile:** This metric underlines the latency value below which 95% of frames are processed, offering insights into the system's performance for the majority of frames, relevant for a stringent SLA.

As for the throughput analysis, the measurement done was:

**Throughput Per Second:** Throughput is a key metric that illustrates the number of frames processed per unit of time. It provides an understanding of the system's overall efficiency in handling incoming data.

Finally, for resource utilization, two measurements were obtained:

**CPU Usage:** CPU utilization data reveals how intensively the system's processors are working during the processing of video frames.

**RAM Usage:** The memory utilization metrics showcase the memory consumption of the system during data processing.

These analysis were conducted individually for each of the three modules within the two pipelines that compose the online phase, allowing for a fine-grained examination of how each module contributes to the overall system performance. Furthermore, we assessed the pipelines as a whole to gain insights into the end-to-end performance of the system.

The results for latency testing are presented on the tables Table 4.2, Table 4.3, Table 4.6 and Table 4.7 and in the sample images on Appendix B.

The results for throughput testing are presented on the tables Table 4.4 and Table 4.5, and in the sample images on Appendix B.

As for the results regarding resource usage during testing, they were on average:

**CPU Usage:** 92.1%

**RAM Usage:** 70.7%

**Table 4.2:** Latency results for local testing on Primary Pipeline

| | Avg Lat | Max Avg Lat | Min Avg Lat | Avg Lat 15s | Max Lat 15s | Min Lat 15s |
|---|---|---|---|---|---|---|
| **Module 1** | 0.061 s | 0.139 s | 0.024 s | 2.95 s | 4.18 s | 1.24 s |
| **Module 2** | 0.041 s | 0.047 s | 0.037 s | 1.89 s | 2.34 s | 1.49 s |
| **Module 3** | 0.017 s | 0.024 s | 0.012 s | 0.72 s | 0.94 s | 0.50 s |
| **Pipeline** | 0.292 s | 0.359 s | 0.228 s | 12.72 s | 13.57 s | 11.63 s |

**Table 4.3:** Latency results for local testing on Secondary Pipeline

| | Avg Lat | Max Avg Lat | Min Avg Lat | Avg Lat 15s | Max Lat 15s | Min Lat 15s |
|---|---|---|---|---|---|---|
| **Module 1** | 0.213 s | 0.240 s | 0.191 s | 14.96 s | 15.27 s | 14.68 s |
| **Module 2** | 0.038 s | 0.043 s | 0.035 s | 2.77 s | 2.96 s | 2.48 s |
| **Module 3** | 0.021 s | 0.025 s | 0.015 s | 1.45 s | 1.77 s | 1.15 s |
| **Pipeline** | 0.363 s | 0.394 s | 0.330 s | 25.59 s | 26.54 s | 24.15 s |

**Table 4.4:** Throughput (frames per second) results for local testing on Primary Pipeline

| | Avg Throughput | Max Avg Throughput | Min Avg Throughput |
|---|---|---|---|
| **Module 1** | 4.48 | 5.07 | 3.99 |
| **Module 2** | 4.44 | 5.13 | 3.60 |
| **Module 3** | 4.46 | 5.00 | 4.07 |
| **Pipeline** | 4.46 | 5.00 | 4.07 |

**Table 4.5:** Throughput (frames per second) results for local testing on Secondary Pipeline

| | Avg Throughput | Max Avg Throughput | Min Avg Throughput |
|---|---|---|---|
| **Module 1** | 4.70 | 5.21 | 4.13 |
| **Module 2** | 4.70 | 5.20 | 4.27 |
| **Module 3** | 4.71 | 5.27 | 4.27 |
| **Pipeline** | 4.71 | 5.27 | 4.27 |

**Table 4.6:** Latency Percentiles for Primary Pipeline and its Modules in local testing

| Percentile | Module 1 (s) | Module 2 (s) | Module 3 (s) | Pipeline (s) |
|---|---|---|---|---|
| **25th** | 0.020 | 0.023 | 0.007 | 0.221 |
| **50th** | 0.031 | 0.038 | 0.011 | 0.256 |
| **75th** | 0.041 | 0.054 | 0.030 | 0.287 |
| **90th** | 0.047 | 0.080 | 0.048 | 0.329 |
| **95th** | 0.055 | 0.092 | 0.059 | 0.371 |

**Table 4.7:** Latency Percentiles for Secondary Pipeline and its Modules in local testing

| Percentile | Module 1 (s) | Module 2 (s) | Module 3 (s) | Pipeline (s) |
|---|---|---|---|---|
| **25th** | 0.249 | 0.035 | 0.021 | 0.448 |
| **50th** | 0.291 | 0.057 | 0.032 | 0.537 |
| **75th** | 0.338 | 0.079 | 0.042 | 0.589 |
| **90th** | 0.404 | 0.092 | 0.052 | 0.641 |
| **95th** | 0.442 | 0.098 | 0.064 | 0.706 |

## 4.2 Distributed Pipeline Testing

In the pursuit of a comprehensive evaluation of our system's capabilities, we move into the second phase of our evaluation process, which we call Distributed Pipeline Testing. In this stage, both pipelines that compose the online phase of our system are deployed on separate computing nodes within the same local network. While maintaining the controlled environment that ensures stable testing conditions, this approach allows us to mimic real-world scenarios more closely.

Running our pipelines on distinct machines over a local network serves a crucial purpose. It enables us to assess the system's performance under conditions that resemble operational setups common in practical applications. This testing phase will shed light on how our system handles distributed workloads, emphasizes collaborative processing, and thrives in a networked environment.

As for the testbed used in this stage, we required two distinct machines, which we will refer to Machine A and Machine B. Machine A is the same machine used for the first phase of the evaluation process, while Machine B has the following specifications:

**Operating System:** Ubuntu 22.04 LTS (via WSL)

**CPU Model:** 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz (Octa-core)

**Clock Speed:** 2611.253 MHz

**Total Memory:** 7.6 GB

**Used Memory:** 582 MB

**Available Memory:** 6.8 GB

This machine's characteristics are similar, but not equal to the specifications of Machine A, which allows us to understand the behaviour of our system in different machines, and the fact that Machine B's operating system is running through a virtualization may cause some overhead, which can be beneficial to see how the system behaves on sub-optimal conditions. Since this testing phase implied running each of the pipelines in two different machines on the same local network, a factor of complexity was added, which was the communication between those pipelines through a network. Kafka continued to serve as the communication backbone for the sending of messages containing alerts, detected by the primary pipeline, to the secondary pipeline. However, the secondary pipeline required access to more than just alert messages. It needed access to the state information stored by Module 1 of the primary pipeline. These files contained the critical data required to replicate the processing of frames, ensuring that the replayed data closely mirrored the original data processing. This, in turn, allowed the secondary pipeline to validate and corroborate the alerts initially identified by the primary pipeline.

To facilitate this data transfer, we employed SFTP (SSH File Transfer Protocol), since Kafka is not adequate for transferring files. Machine B, responsible for running the primary pipeline, transferred these files to Machine A, the host of the secondary pipeline. Machine A played a vital role in managing the state store that stored the essential information necessary for replaying and thoroughly assessing data processing. This meticulous data transfer mechanism allowed us to create a testing environment more akin to real-world scenarios, strengthening our ability to validate the effectiveness and efficiency of our data processing system.

Regarding the metrics and measurements for performance analysis in this testing phase, they are exactly the same as in the previous stage, and are acquired following the same process and with the the same tools.

The results for latency testing are presented on the tables Table 4.9, Table 4.10, Table 4.13 and

Table 4.14 and in the sample images on Appendix B.

The results for throughput testing are presented on the tables Table 4.11 and Table 4.12, and in sample images on Appendix B.

**Table 4.8:** Resource Usage results during distributed pipeline testing

| | Machine A normal | Machine A testing | Machine B normal | Machine B testing |
|---|---|---|---|---|
| **CPU Usage** | 4.5% | 60.6% | 0.5% | 58.7% |
| **RAM Usage** | 50.5% | 68.5% | 10.2% | 29.7% |

**Table 4.9:** Latency results for distributed pipeline testing on Primary Pipeline

| | Avg Lat | Max Avg Lat | Min Avg Lat | Avg Lat 15s | Max Lat 15s | Min Lat 15s |
|---|---|---|---|---|---|---|
| **Module 1** | 0.061 s | 0.118 s | 0.024 s | 3.14 s | 4.74 s | 1.23 s |
| **Module 2** | 0.032 s | 0.058 s | 0.021 s | 1.71 s | 2.78 s | 0.95 s |
| **Module 3** | 0.023 s | 0.038 s | 0.014 s | 1.186 s | 1.60 s | 0.65 s |
| **Pipeline** | 1.064 s | 2.460 s | 0.353 s | 51.89 s | 143.00 s | 18.61 s |

**Table 4.10:** Latency results for distributed pipeline testing on Secondary Pipeline

| | Avg Lat | Max Avg Lat | Min Avg Lat | Avg Lat 15s | Max Lat 15s | Min Lat 15s |
|---|---|---|---|---|---|---|
| **Module 1** | 0.131 s | 0.165 s | 0.124 s | 14.93 s | 15.11 s | 14.56 s |
| **Module 2** | 0.022 s | 0.025 s | 0.020 s | 2.69 s | 2.71 s | 2.19 s |
| **Module 3** | 0.015 s | 0.016 s | 0.013 s | 1.67 s | 1.88 s | 1.42 s |
| **Pipeline** | 0.244 s | 0.298 s | 0.233 s | 27.952 s | 28.97 s | 26.61 s |

**Table 4.11:** Throughput results for distributed pipeline testing on Primary Pipeline

| | Avg Throughput | Max Avg Throughput | Min Avg Throughput |
|---|---|---|---|
| **Module 1** | 3.53 | 4.40 | 2.33 |
| **Module 2** | 3.63 | 4.93 | 2.40 |
| **Module 3** | 3.42 | 4.73 | 1.87 |
| **Pipeline** | 3.42 | 4.73 | 1.87 |

**Table 4.12:** Throughput results for distributed pipeline testing on Secondary Pipeline

| | Avg Throughput/s | Max Avg Throughput/s | Min Avg Throughput/s |
|---|---|---|---|
| **Module 1** | 7.63 | 8.07 | 5.87 |
| **Module 2** | 7.64 | 8.08 | 6.07 |
| **Module 3** | 7.65 | 8.11 | 5.93 |
| **Pipeline** | 7.65 | 8.11 | 5.93 |

**Table 4.13:** Latency Percentiles for Primary Pipeline and its Modules in distributed pipeline testing

| Percentile | Module 1 (s) | Module 2 (s) | Module 3 (s) | Pipeline (s) |
|---|---|---|---|---|
| **25th** | 0.020 | 0.021 | 0.011 | 0.401 |
| **50th** | 0.030 | 0.034 | 0.025 | 0.499 |
| **75th** | 0.041 | 0.046 | 0.046 | 0.618 |
| **90th** | 0.047 | 0.067 | 0.071 | 0.680 |
| **95th** | 0.051 | 0.079 | 0.082 | 0.755 |

Table 4.14: Latency Percentiles for Secondary Pipeline and its Modules in distributed pipeline testing

| Percentile | Module 1 (s) | Module 2 (s) | Module 3 (s) | Pipeline (s) |
|---|---|---|---|---|
| 25th | 0.113 | 0.020 | 0.018 | 0.226 |
| 50th | 0.128 | 0.030 | 0.028 | 0.252 |
| 75th | 0.143 | 0.040 | 0.039 | 0.278 |
| 90th | 0.152 | 0.046 | 0.046 | 0.296 |
| 95th | 0.159 | 0.048 | 0.048 | 0.323 |

## 4.3 Analysis of Experimental Results

In this section we will perform a qualitative and quantitative analysis of the results obtained in the two distinct phases described previously in this chapter. We'll start by analyzing the Local Testing phase and how it performed in comparison with the benchmarks, which were ran in the same controlled environment. Once we understand the performance of our system in a fully local testing environment, we will analyze the results of the Distributed Pipeline Testing phase and see how our system's performance is affected by the transition to a testing environment more similar to a real world situation.

Before any analysis it's important to point out the limitations of our system's testing. Due to the lack of videos available and acceptable to use for our use case, all the testing was done with the same video as data source for stream processing. This video is very susceptible to alert situations, due to the placement of the camera and due to containing several people in an ample space, which resulted in people overlapping each other frequently and making it impossible for the tracking model to detect some people which were still in the frame. This made the occurrence of alerts more frequent than what would be considered normal in a real world situation where our system would be applied. This higher frequency of alerts demanded an higher effort from our system, which naturally was reflected in the performance of our system and results of its testing. This increase on the rate of alerts also implied that the frequency of periodic check-pointing of the Object Detection Model and Predictor states be increased too, which also impacted the system's performance. However, this situation allows us to test our system in a high stress environment and we can interpret the obtained results as closer to the results of a worst case scenario, which permits us to conclude that the performance of our system in a real world situation would be considerately better than the performance in our testing scenario.

Starting with our quantitative analysis, as mentioned before, the initial testing phase involved rigorous local testing, where both pipelines that made up the online phase of our system were executed on a singular machine. In this controlled environment, we aimed to glean insights into the system's performance when subjected to the inherent intricacies of data processing. When comparing the Local Testing results with the Benchmark Testing results, the only directly comparable performance metric is resource usage. By comparing these values, we can see that regarding CPU usage, our system uses slightly more CPU. This is expected, as Spark Structured Streaming often leverages CPU power efficiently, and our system requires additional computational power due to the more complex and computationally intensive nature of video frame processing. For RAM usage, we can see our system is more efficient. This might indicate that our system manages memory resources efficiently and is well-optimized for its specific task. However, due to different data types and operations involved in each testing scenario, this cannot be directly comparable.

The subsequent testing phase hinged on the deployment of pipelines on different machines. This change in architecture introduced an element of communication across a local network, further mimicking real-world deployment conditions.

We will start by comparing the resource usage in both phases of testing. As we can see in Table 4.8, the CPU usage in the Distributed Pipeline Testing phase is considerably lower than in the Local Testing phase, which is expected since in Local Testing the machine has to handle both pipelines' processing. This indicates that distributing the workload across two machines helps reduce CPU utilization. Regarding RAM usage, we can see that performance of Machine B during Distributed Pipeline Testing is substantially better, while the performance of Machine A is only marginally better when compared to the RAM utilization of that same machine during Local Testing. This can be explained by the role played by Machine A in the second phase of testing. Since Machine A is working as the state store of our system, it has to endure the bigger memory demands of our system.

When comparing latency between the two testing phases, we can see that the results for the Primary Pipeline is both stages is very similar when comparing the individual performance of each module. However, there is an increase on the latency of the overall pipeline from the first to the second stage of testing. One contributing factor was the placement of the Kafka server, which resided on Machine A during the Distributed Pipeline Testing phase. This introduced network latency, not present in the local testing, affecting the pipeline's latency. Simultaneously, the concurrent SFTP file transfers between the laptops, though unrelated to the pipeline, shared network resources and possibly system resources. These transfers could have impacted the pipeline's overall performance. This is also reflected in the slight worsening of throughput performance in this pipeline from one testing phase to the other. These real-world conditions emphasize the need to consider external factors when evaluating system performance and adapting it for consistency and predictability.

When comparing the performance results of the Secondary Pipeline on the two testing stages we can see an improvement of these results on the transition from the Local Testing phase to the Distributed Pipeline Testing phase. This improvement can be attributed to the different environment and resource allocation. In the second testing phase the Secondary pipeline remained on the same machine which eliminated the simultaneous processing of the Primary Pipeline on that machine. However, it's also important to note that the Kafka server, responsible for communication between the pipelines and modules, continued to run on this machine. This change in the environment, with reduced competition for resources, likely contributed to the enhanced performance of the Secondary Pipeline both latency and throughput wise.

When looking at the images on Appendix B we can note, specially on the sample of results for average latency, the impact of the checkpointing that happens on Module 1 of the Primary Pipeline, which enables data provenance. Since the checkpointing in our use case is frequent, happening once every 100 frames, the spikes in latency are notable. In the images **??** and **??**, we can see that the latency increase caused by checkpointing is of around **300%**. The images **??** and **??** show that in the whole pipeline the impact is of around **24%** on Local Testing and **200%**. This of course is also reflected in the throughput of the system. From this we can conclude how enabling data provenance affects our system. However, we firmly believe that in a more realistic situation with less frequent alerts and less need for frequent checkpointing, this impact on the system's performance will be less notable, since the checkpointing and alerts will happen at a lower rate, causing the performance of our solution to be more similar to the lower values in our result samples, hence more optimized.

As for qualitative analysis, the only analysis provenance related worth making is the comparison between the Secondary Pipeline and the Third Pipeline. This comparison is particularly meaningful because the Third Pipeline essentially replays the operations of the Secondary Pipeline under identical conditions. The objective is to ensure that both pipelines produce the exact same results. While it might be tempting to compare the Primary Pipeline to the others, it's important to understand that such a comparison doesn't result in any meaningful insights. The reason is that the Primary Pipeline, although responsible for different results, shares the responsibility of detecting alerts with the Secondary Pipeline. The consistent detection of alerts across these pipelines, despite variations in their outcomes, is a strong indicator of a genuine alert. This alert detection process is predominantly influenced by the object detection modules, which falls outside the scope of our provenance analysis. Therefore, the core of our qualitative analysis centers around ensuring the fidelity of results between the Secondary and Third Pipelines, a critical aspect for validating the reliability and accuracy of our alert detection system.

The results of this comparison between the Secondary and Third pipelines offers highly positive insights. In both testing phases, it becomes abundantly clear that the Third Pipeline consistently generates identical results to the Secondary Pipeline. This observation is of paramount importance because, in our system, particularly in this replay stage, we prioritize result accuracy over performance. While optimal performance is always a consideration, the primary emphasis in this context is on the unwavering reliability of results. The ability to consistently reproduce the same results reinforces our confidence in the integrity of the system. This finding underscores that, in scenarios where data provenance and result accuracy are critical, the Third Pipeline is a robust and dependable component, reaffirming that it will consistently produce the precise outcomes of the Secondary Pipeline.

In conclusion, the evaluation of our system has shed light on key insights into its performance, resource utilization, and data integrity. Through local testing and distributed pipeline testing, we've obtained a comprehensive understanding of its behavior. While local testing demonstrated efficient resource utilization and prompt data processing, the transition to distributed pipeline testing exposed some intricacies. The presence of a Kafka server on one machine and concurrent SFTP transfers did introduce some latency into the pipeline execution. However, it's essential to note that the individual module performance remained largely consistent, indicating their resilience. Furthermore, the qualitative analysis affirmed that the Third Pipeline consistently reproduces the results of the Secondary Pipeline. This speaks to the system's unwavering commitment to result accuracy, which is crucial in scenarios where data provenance is paramount, while still presenting robust performance and resource efficiency.

# 5 Conclusion

## Contents

In this work we addressed the topic of auditable data provenance in streaming data processing. We started by presenting a survey of the current state of the art in stream processing as well as a wide variety of works that discuss data provenance in general. By analysing this literature we proposed an architecture for a solution of a stream processing system which ensures auditable data provenance, which we then tested and evaluated.

## 5.1 Conclusions

In our work we aimed to explore, design, and implement auditable data provenance solutions tailored to the dynamic nature of streaming data. By acknowledging the increasing use of streaming data processing and the pressing need for auditable data provenance, and addressing its main challenges, this research aspires to provide a comprehensive framework for ensuring data trustworthiness, accountability, and transparency in the era of continuous data streams.

We studied several works in the areas of both stream processing and data provenance, and it was notable that the existing solutions present limitations, such as data provenance solutions introducing latency and overhead into stream processing systems, scalability issues due to growing influx of data, inability to handle the processing of complex events with dynamic semantics, excessive consume of resources, and even privacy and security concerns.

By understanding the advantages and shortcomings of the existing solutions, we designed an architecture framework for complete and correct auditable data provenance in a stream processing environment which was applied to a real-life use case. By processing continuous video streams in a dynamic setting, we showed our system can handle multiple complex challenges and be integrated with diverse technologies, like computer vision for object detection, deep learning models for tracking, Apache Kafka for communication and state management for auditability. A test environment was also designed to validate the correctness of our solution and evaluate its performance.

Our proposed solution's architectural framework comprises three distinct pipelines, each consisting of four Python modules. These modules share common functionalities: the first ensures script integrity by hashing subsequent modules, the second detects and tracks individuals, the third draws bounding boxes around detected persons, and the fourth defines areas within frames and counts individuals within those areas. In the initial pipeline, the second module also monitors the disappearance of individuals, generating alerts for suspicious occurrences. This triggers the secondary pipeline, which will replay the frame processing using data from the primary pipeline's state store. The secondary pipeline may not replicate the exact same results due to frame dependencies. However, if a secondary alert confirms the initial alert, it signifies a true alert, prompting data transfer from the state store to permanent memory. This will enable the third pipeline, an offline component, that reproduces processing from the secondary pipeline, offering detailed insights and visual evidence. The primary and secondary pipelines operate online, while the third pipeline functions offline. Communication among modules and pipelines occurs through Apache Kafka, and a concluding Apache Spark Structured Streaming module assembles a continuously updated list of entries and exits in store areas or the store itself.

Our evaluation process began with the development of two Apache Spark Structured Streaming benchmarks. Subsequently, we conducted a comprehensive assessment of our system, focusing on latency, throughput, and resource utilization, involving two key stages. The first stage featured local testing, where both pipelines were evaluated on a single machine, providing insights into controlled performance. In the second stage, the pipelines operated on separate machines within the same local network, introducing external factors. As anticipated, the shift to distributed pipeline testing resulted in some latency increases, yet the system's overall performance remained satisfactory. Crucially, through qualitative analysis, we established that our system offers dependable and accurate data provenance in the context of a demanding stream processing environment.

## 5.2 Improvements and Future Work

There is, of course, room for improvements in our system. Starting with our tests, which could have benefited from a wider variety of videos to serve as data source for stream processing. Videos not so susceptible to alerts would have allowed to understand how our system would perform in normal stress environments.

Looking at the system itself, there are two major improvements that come to mind:

**Enhanced Confirmation Logic:** Introducing a windowing mechanism to the alert confirmation process could lead to resource savings. This would enable the system to confirm multiple alerts occurring within the same window in a single replay, streamlining the process.

**Database Integration:** Integrating an external database technology into our system would significantly enhance data management. It would improve the storage of files, data required for replays, and results, making the system more efficient and capable of handling larger volumes of data effectively.

Finally, we believe that an appropriate long-term objective for our system's evolution is the integration of additional data sources. Given the specific focus of our system, an exciting avenue of exploration is the inclusion of multiple video sources covering the same store area. This expansion allows us to investigate how our system can effectively process data from various camera angles simultaneously. Moreover, we can explore the integration of other sensor types throughout the store. Combining data from these diverse sources has the potential to yield even more comprehensive and nuanced results. These expanded capabilities will help our system adapt and evolve to address a broader spectrum of challenges in the realm of data provenance and real-time analytics.

# Bibliography

[1] F. Gürcan and M. Berigel, "Real-time processing of big data streams: Lifecycle, tools, tasks, and challenges," in *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2018, pp. 1–6.

[2] E. Mehmood and T. Anees, "Challenges and solutions for processing real-time big data stream: A systematic literature review," *IEEE Access*, vol. 8, pp. 119 123–119 143, 2020.

[3] A. N. Navaz, S. Harous, M. A. Serhani, and I. Taleb, "Real-time data streaming algorithms and processing technologies: A survey," in *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*, 2019, pp. 246–250.

[4] T. Kolajo, O. Daramola, and A. Adebiyi, "Big data stream analysis: a systematic literature review," *Journal of Big Data*, vol. 6, p. 47, 06 2019.

[5] Z. Zhang, X. Chen, J. Ma, and X. Tao, "New efficient constructions of verifiable data streaming with accountability," *Annals of Telecommunications*, vol. 74, 01 2019.

[6] M. R. Huq, A. Wombacher, and P. M. Apers, "Inferring fine-grained data provenance in stream data processing: reduced storage cost, high accuracy," in *Database and Expert Systems Applications: 22nd International Conference, DEXA 2011, Toulouse, France, August 29-September 2, 2011, Proceedings, Part II 22*. Springer, 2011, pp. 118–127.

[7] R. Eiss, "Confusion over europe's data-protection law is stalling scientific progress," *Nature*, vol. 584, pp. 498–498, 08 2020.

[8] H.-S. Lim, Y.-S. Moon, and E. Bertino, "Research issues in data provenance for streaming environments," in *Proceedings of the 2nd SIGSPATIAL ACM GIS 2009 International Workshop on Security and Privacy in GIS and LBS*, 2009, pp. 58–62.

[9] J. Wang, D. Crawl, S. Purawat, M. Nguyen, and I. Altintas, "Big data provenance: Challenges, state of the art and opportunities," in *2015 IEEE international conference on big data (Big Data)*. IEEE, 2015, pp. 2509–2516.

[10] Q. Ye and M. Lu, “s2p: provenance research for stream processing system,” *Applied Sciences*, vol. 11, no. 12, p. 5523, 2021.

[11] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul, “Efficient stream provenance via operator instrumentation,” *ACM Transactions on Internet Technology (TOIT)*, vol. 14, no. 1, pp. 1–26, 2014.

[12] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, “A survey of distributed data stream processing frameworks,” *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[13] S. Liu, W. Cui, Y. Wu, and M. Liu, “A survey on information visualization: recent advances and challenges,” *The Visual Computer*, vol. 30, pp. 1373–1393, 2014.

[14] D. V. Gorasiya, “Comparison of open-source data stream processing engines: spark streaming, flink and storm,” 2019.

[15] F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, and S. Sakr, “Big data 2.0 processing systems: Taxonomy and open challenges,” *Journal of Grid Computing*, vol. 14, pp. 379–405, 2016.

[16] D. K. Lal and U. Suman, “Towards comparison of real time stream processing engines,” in *2019 IEEE Conference on Information and Communication Technology*. IEEE, 2019, pp. 1–5.

[17] M. V. Bordin, D. Griebler, G. Mencagli, C. F. Geyer, and L. G. L. Fernandes, “Dspbench: A suite of benchmark applications for distributed data stream processing systems,” *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.

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

[19] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, “Benchmarking streaming computation engines: Storm, flink and spark streaming,” in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016, pp. 1789–1792.

[20] A. Shukla, S. Chaturvedi, and Y. Simmhan, “Riotbench: An iot benchmark for distributed stream processing systems,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[21] Z. Karakaya, A. Yazici, and M. Alayyoub, “A comparison of stream processing frameworks,” in *2017 International Conference on Computer and Applications (ICCA)*. IEEE, 2017, pp. 1–12.

[22] C. Bockermann, “A survey of the stream processing landscape,” 2014.

[23] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd international workshop on Software and performance*, 2000, pp. 195–203.

[24] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[25] L. Wang, T. Z. Fu, R. T. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 573–588.

[26] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, and P. Tsigas, "Stretch: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4221–4238, 2022.

[27] H. Najdataei, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, and V. Gulisano, "Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 7–18.

[28] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–37, 2019.

[29] D. Palyvos-Giannas, G. Mencagli, M. Papatriantafilou, and V. Gulisano, "Lachesis: a middleware for customizing os scheduling of stream processing queries," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 365–378.

[30] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo, "Runtime adaptation of data stream processing systems: The state of the art," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, 2022.

[31] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, 2017.

[32] H. Najdataei, V. Gulisano, P. Tsigas, and M. Papatriantafilou, "pi-lisco: parallel and incremental stream-based point-cloud clustering," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 460–469.

[33] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[34] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *Proceedings of the 18th International Database Engineering & Applications Symposium*, 2014, pp. 356–361.

[35] W. C. Tan *et al.*, "Provenance in databases: Past, current, and future." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 3–12, 2007.

[36] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou, "Genealog: Fine-grained data streaming provenance in cyber-physical systems," *Parallel Computing*, vol. 89, p. 102552, 2019.

[37] D. Palyvos-Giannas, B. Havers, M. Papatriantafilou, and V. Gulisano, "Ananke: a streaming framework for live forward provenance," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 391–403, 2020.

[38] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE international conference on big data (big data)*. IEEE, 2015, pp. 2785–2792.

[39] M. Tang, S. Shao, W. Yang, Y. Liang, Y. Yu, B. Saha, and D. Hyun, "Sac: A system for big data lineage tracking," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1964–1967.

[40] I. M. Yazici and M. S. Aktas, "A novel visualization approach for data provenance," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 9, p. e6523, 2022.

[41] Z. Zvara, P. G. Szabó, G. Hermann, and A. Benczúr, "Tracing distributed data stream processing systems," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2017, pp. 235–242.

[42] Z. Zvara, P. G. Szabó, B. Balázs, and A. Benczúr, "Optimizing distributed data stream processing by tracing," *Future Generation Computer Systems*, vol. 90, pp. 578–591, 2019.

[43] W. Sansrimahachai, M. J. Weal, and L. Moreau, "Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems," in *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 2012, pp. 1–12.

[44] W. Sansrimahachai, L. Moreau, and M. J. Weal, "An on-the-fly provenance tracking mechanism for stream processing systems," in *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. IEEE, 2013, pp. 475–481.

[45] M. Yamada, H. Kitagawa, T. Amagasa, and A. Matono, "Augmented lineage: traceability of data analysis including complex udf processing," *The VLDB Journal*, pp. 1–21, 2022.

[46] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh *et al.*, “Benchmarking streaming computation engines at yahoo.” Technical report, 2015.

# A

# Benchmarks

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import shutil
import os
import time

checkpointLocation = "/home/bate/smart-retail-example/V1/benchmark0/spark/
    output/check"
checkpointLocationAux = "/home/bate/smart-retail-example/V1/benchmark0/spark/
    aux/check"
outputPath = "/home/bate/smart-retail-example/V1/benchmark0/spark/output"
outputPathAux = "/home/bate/smart-retail-example/V1/benchmark0/spark/aux"

upper_bound = 100
```

```python
window_duration = "10 minutes"
sliding_interval = "5 minutes"

if os.path.exists(checkpointLocation):
    shutil.rmtree(checkpointLocation)

if os.path.exists(outputPath):
    shutil.rmtree(outputPath)

if os.path.exists(checkpointLocationAux):
    shutil.rmtree(checkpointLocationAux)

if os.path.exists(outputPathAux):
    shutil.rmtree(outputPathAux)

if os.path.exists(checkpointLocationTest):
    shutil.rmtree(checkpointLocationTest)

if os.path.exists(outputPathTest):
    shutil.rmtree(outputPathTest)

spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()

df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "bench0_spark") \
    .option("startingOffsets", "earliest") \
    .load()

df_s = df.selectExpr("CAST(value AS STRING)")

schema = StructType() \
    .add("id", IntegerType()) \
    .add("number", IntegerType()) \
    .add("event_time", LongType()) \
    .add("hash", StringType()) \
    .add("flag", IntegerType())

df_f = df_s.select(from_json(col("value"), schema).alias("data")).select("
    data.*")

df_sum = df_f.withColumn("final_value", col("number") + col("flag"))

df_state_store = df_sum.filter("flag > 0").select("id", "event_time", "
    final_value")

df_timestamp = df_sum.withColumn("timestamp", to_timestamp(from_unixtime("
    event_time")))

eventsDF = df_timestamp.withWatermark("timestamp", "5 minutes")#.
    withWatermark("timestamp_lat", "5 minutes")

joinedDF = eventsDF.join(df_state_store, on=["campaign_id", "event_time"],
    how="inner").repartition(1).dropDuplicates()

joinedDF_watermark = joinedDF.withWatermark("timestamp", "5 minutes")

finalDF1 = joinedDF_watermark.select("ad_id", "price", "timestamp")

finalDF1 = eventsDF.withColumn("window", window("timestamp", window_duration,
     sliding_interval))#.withColumn("window_lat", window("timestamp_lat",
    window_duration, sliding_interval))

finalDF = finalDF1.groupBy("window", "hash").agg(avg("final_value").alias("
    avg_value")).repartition(1)

finalDF = finalDF.withColumn("window_start", col("window.start"))
finalDF = finalDF.withColumn("window_end", col("window.end"))

finalDF = finalDF.drop("window").select("avg_value", "window_start", "
    window_end", "hash")

df_state_store \
    .writeStream \
    .format("csv") \
    .option("path", outputPathAux) \
    .option("header", "true") \
    .option("mode", "append") \
    .option("checkpointLocation", checkpointLocationAux) \
    .start()\
    .awaitTermination(timeout=30)

finalDF \
    .writeStream\
    .format("csv")\
    .option("path", outputPath) \
    .option("header", "true") \
    .option("mode", "append")\
    .option("checkpointLocation", checkpointLocation)\
    .start()\
    .awaitTermination(timeout=100)
```

**Listing A.1:** Benchmark A - Normal Pipeline

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import shutil
import os

checkpointLocation = "/home/bate/smart-retail-example/V1/benchmark0/spark/
    prov/check"
outputPath = "/home/bate/smart-retail-example/V1/benchmark0/spark/prov"
ss_path = "/home/bate/smart-retail-example/V1/benchmark0/spark/aux"
window_duration = "10 minutes"
sliding_interval = "5 minutes"

if os.path.exists(checkpointLocation):
    shutil.rmtree(checkpointLocation)

if os.path.exists(outputPath):
    shutil.rmtree(outputPath)

schema_ss = StructType() \
    .add("id", IntegerType()) \
    .add("event_time", IntegerType()) \
    .add("final_value", IntegerType())

spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()

df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "bench0_spark_prov") \
    .option("startingOffsets", "earliest") \
    .load()

dfSS = spark.readStream \
    .format("csv") \
    .schema(schema_ss) \
    .option("header", "true") \
    .load(f"{ss_path}/*.csv")

df_s = df.selectExpr("CAST(value AS STRING)")

schema = StructType() \
    .add("id", IntegerType()) \
    .add("number", IntegerType()) \
    .add("event_time", IntegerType()) \
    .add("hash", StringType())

df_f = df_s.select(from_json(col("value"), schema).alias("data")).select("
    data.*")

df_timestamp = df_f.withColumn("timestamp", to_timestamp(from_unixtime("
    event_time")))

```

```
eventsDF = df_timestamp.withWatermark("timestamp", "5 minutes")

dfSS = dfSS.withColumn("timestamp", to_timestamp(from_unixtime("event_time"))
    ).withWatermark("timestamp", "5 minutes")

result_df = eventsDF.join(dfSS, on=["id", "event_time", "timestamp"], how="
    left")

joinedDF = result_df.withColumn("number", coalesce(col("final_value"), col("
    number"))).drop("final_value")

finalDF1 = joinedDF.withColumn("window", window("timestamp", window_duration,
     sliding_interval))

finalDF = finalDF1.groupBy("window", "hash").agg(avg("number").alias("
    avg_value"), sum("number").alias("sum"), count("number").alias("count"),
    collect_list("number").alias("numbers"), collect_list("event_time").alias
    ("event_times"), collect_list("id").alias("ids")).repartition(1)

finalDF = finalDF.withColumn("event_times_str", concat_ws(",", "event_times")
    ).withColumn("numbers_str", concat_ws(",", "numbers")).withColumn("
    ids_str", concat_ws(",", "ids"))

finalDF = finalDF.withColumn("window_start", col("window.start"))
finalDF = finalDF.withColumn("window_end", col("window.end"))

finalDF = finalDF.drop("window").select("avg_value", "sum", "count", "ids_str
    ", "event_times_str", "numbers_str", "window_start", "window_end", "hash"
    )

finalDF \
    .writeStream\
    .format("csv")\
    .option("path", outputPath) \
    .option("header", "true") \
    .option("mode", "append")\
    .option("checkpointLocation", checkpointLocation)\
    .start()\
    .awaitTermination(timeout=100)
```

**Listing A.2:** Benchmark A - Replay Pipeline

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import shutil
import os
import time

checkpointLocation = "/home/bate/smart-retail-example/V1/benchmark/spark/
    output/check"
checkpointLocationAux = "/home/bate/smart-retail-example/V1/benchmark/spark/
    aux/check"
outputPath = "/home/bate/smart-retail-example/V1/benchmark/spark/output"
outputPathAux = "/home/bate/smart-retail-example/V1/benchmark/spark/aux"

upper_bound = 100
window_duration = "10 minutes"
sliding_interval = "5 minutes"

if os.path.exists(checkpointLocation):
    shutil.rmtree(checkpointLocation)

if os.path.exists(outputPath):
    shutil.rmtree(outputPath)

if os.path.exists(checkpointLocationAux):
    shutil.rmtree(checkpointLocationAux)

if os.path.exists(outputPathAux):
    shutil.rmtree(outputPathAux)

if os.path.exists(checkpointLocationTest):
    shutil.rmtree(checkpointLocationTest)

if os.path.exists(outputPathTest):
    shutil.rmtree(outputPathTest)

spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()

df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "bench_spark") \
    .option("startingOffsets", "earliest") \
    .load()

df_s = df.selectExpr("CAST(value AS STRING)")

schema = StructType() \
    .add("user_id", IntegerType()) \
    .add("page_id", IntegerType()) \
    .add("ad_id", IntegerType()) \
    .add("ad_type", StringType()) \
    .add("event_type", StringType()) \
    .add("event_time", LongType()) \
    .add("hash", StringType()) \
    .add("id", IntegerType()) \
    .add("price", FloatType())

df_f = df_s.select(from_json(col("value"), schema).alias("data")).select("
    data.*")

df_aux = df_f.withColumn("incoming_time", current_timestamp().cast("long") *
    1000000)

df_filter = df_f.filter(col("event_type") == "purchase")

df_projection = df_filter.select("ad_id", "event_time", "hash", "price")

df_campaign = df_projection.withColumn("campaign_id", when(col("ad_id") \% 10
     == 0, 10).otherwise(col("ad_id") \% 10))
```

```

df_state_store = df_campaign.select("campaign_id", "event_time", "price")

df_timestamp = df_campaign.withColumn("timestamp", to_timestamp(from_unixtime
    ("event_time")))

eventsDF = df_timestamp.withWatermark("timestamp", "5 minutes").drop("price")

joinedDF = eventsDF.join(df_state_store, on=["campaign_id", "event_time"],
    how="inner").repartition(1).dropDuplicates()

joinedDF_watermark = joinedDF.withWatermark("timestamp", "5 minutes")

finalDF1 = joinedDF_watermark.select("ad_id", "price", "timestamp")

finalDF1 = joinedDF.withColumn("window", window("timestamp", window_duration,
     sliding_interval))

finalDF = finalDF1.groupBy("window", "campaign_id", "hash").agg(sum("price").
    alias("total_profit")).repartition(1)

finalDF = finalDF.withColumn("window_start", col("window.start"))
finalDF = finalDF.withColumn("window_end", col("window.end"))

finalDF = finalDF.drop("window").select("campaign_id", "total_profit", "
    window_start", "window_end", "hash")

df_state_store \
     .writeStream \
     .format("csv") \
     .option("path", outputPathAux) \
     .option("header", "true") \
     .option("mode", "append") \
     .option("checkpointLocation", checkpointLocationAux) \
     .start()\
     .awaitTermination(timeout=30)

finalDF \
     .writeStream\
     .format("csv")\
     .option("path", outputPath) \
     .option("header", "true") \
     .option("mode", "append")\
     .option("checkpointLocation", checkpointLocation)\
     .start()\
     .awaitTermination(timeout=100)
```

**Listing A.3:** Benchmark B - Normal Pipeline

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import shutil
import os

checkpointLocation = "/home/bate/smart-retail-example/V1/benchmark/spark/prov
    /check"
outputPath = "/home/bate/smart-retail-example/V1/benchmark/spark/prov"
ss_path = "/home/bate/smart-retail-example/V1/benchmark/spark/aux"
window_duration = "10 minutes"
sliding_interval = "5 minutes"

if os.path.exists(checkpointLocation):
    shutil.rmtree(checkpointLocation)

if os.path.exists(outputPath):
    shutil.rmtree(outputPath)

schema_ss = StructType() \
    .add("campaign_id", IntegerType()) \
    .add("event_time", IntegerType()) \
    .add("price", FloatType())

spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()
```

```

df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "bench_spark_prov") \
    .option("startingOffsets", "earliest") \
    .load()

dfSS = spark.readStream \
    .format("csv") \
    .schema(schema_ss) \
    .option("header", "true") \
    .load(f"{ss_path}/*.csv")

df_s = df.selectExpr("CAST(value AS STRING)")

schema = StructType() \
    .add("user_id", IntegerType()) \
    .add("page_id", IntegerType()) \
    .add("ad_id", IntegerType()) \
    .add("ad_type", StringType()) \
    .add("event_type", StringType()) \
    .add("event_time", IntegerType()) \
    .add("hash", StringType())

df_f = df_s.select(from_json(col("value"), schema).alias("data")).select("
    data.*")

df_filter = df_f.filter(col("event_type") == "purchase")

df_projection = df_filter.select("ad_id", "event_time", "hash")

df_campaign = df_projection.withColumn("campaign_id", when(col("ad_id") \% 10
     == 0, 10).otherwise(col("ad_id") \% 10))

df_timestamp = df_campaign.withColumn("timestamp", to_timestamp(from_unixtime
    ("event_time")))

```

```
eventsDF = df_timestamp.withWatermark("timestamp", "5 minutes")

joinedDF = eventsDF.join(dfSS, on=["campaign_id", "event_time"], how="inner")
    .repartition(1).dropDuplicates()

finalDF1 = joinedDF.withColumn("window", window("timestamp", window_duration,
     sliding_interval))

finalDF = finalDF1.groupBy("window", "ad_id", "hash", "campaign_id").agg(
    collect_list("price").alias("prices"), collect_list("event_time").alias("
    event_times")).repartition(1)

finalDF = finalDF.withColumn("event_times_str", concat_ws(",", "event_times")
    ).withColumn("prices_str", concat_ws(",", "prices"))

finalDF = finalDF.withColumn("window_start", col("window.start"))
finalDF = finalDF.withColumn("window_end", col("window.end"))

finalDF = finalDF.drop("window").select("ad_id", "campaign_id", "
    event_times_str", "prices_str", "window_start", "window_end", "hash")

finalDF \
     .writeStream\
     .format("csv")\
     .option("path", outputPath) \
     .option("header", "true") \
     .option("mode", "append")\
     .option("checkpointLocation", checkpointLocation)\
     .start()\
     .awaitTermination(timeout=100)
```

**Listing A.4:** Benchmark B - Replay Pipeline

# B

# Result Samples

The images with the sample results of our testing phase can be seen in the AnnexB directory of the following Github repo: https://github.com/afonsobate/Thesis
Please keep in mind that these images are merely samples of the results obtained in our testing phases.