Auditable Data Provenance in Streaming Data Processing

Afonso Bate
afonso.bate@tecnico.ulisboa.pt
Advisors: Luis Veiga, Paulo Carreira

Instituto Superior Técnico, Universidade de Lisboa

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. We attempt to do this by inspiring ourselves in a solution that provides fine-grained provenance for a fully deterministic stream processing system implemented in Apache Flink. We improve this solution by extending Apache Spark to provide coarse-grained and fine-grained provenance for a stream processing system which includes deterministic and non-deterministic operators.

Keywords: Stream Processing · Data Provenance

Table of Contents

1 Introduction

Data processing has been around for millennia, 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 Auditable Stream Processing

Just like any other software system, stream processing applications are not perfect and some errors may occur, which lead to incorrect results. When these situations occur it's important to audit the data stream in question and trace data lineage to find what caused that incorrect computation. However, providing this data provenance in a stream processing system is not as easy as in a batch processing one due to the dynamic nature of stream processing systems.

Data provenance consists of a system's ability to trace the flow of data and the transformations it suffered from input to output in order to justify a certain result. This provenance is even harder to implement in systems that face massive amounts of data since a result can result from a wide range of input data and it is hard to keep track of all that source data. The state-of-the-art regarding provenance in stream processing systems shows that most of the solutions are still quite limited. Most solutions only identify the source data which led to a specific result, however, this can be insufficient to understand and justify the correctness or incorrectness of those same results. A few works propose more complete solutions, however, this area can still be studied and developed in more depth.

1.2 Objectives

The main goal of this work is to develop a stream processing system capable of auditing data streams and providing data provenance to justify and compre-

hend the results obtained by our system. In order to accomplish this we set the following objectives:

1. Survey the current state-of-the-art in stream processing engines, compile the set of key design decisions that make up those engines, and understand the shortcomings of currently available solutions.
2. Study the relevant related systems in lineage and provenance for stream processing systems, understand the set of key design decisions that make up those systems, and understand the shortcomings of currently available solutions.
3. Design an architecture that extends a stream processing engine in order to provide correct data provenance while reducing at most the implied overhead.
4. Implement a framework that will support our architecture.
5. Create a structured evaluation methodology for assessing that our future work fulfills our requirements when tested on real-life environments and with real-life datasets.

1.3 Roadmap

The rest of the document is organized in the following way: In Section 2, we analyze the related work. Section 3, describes our proposed architecture. In Section 4, we describe how to evaluate our solution in terms of system metrics, and what workloads will be used to test its performance. Lastly, Section 5 concludes our work.

2 Related Work

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. 1, which we believe describes the components of a stream processing system as a set of layers. 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.

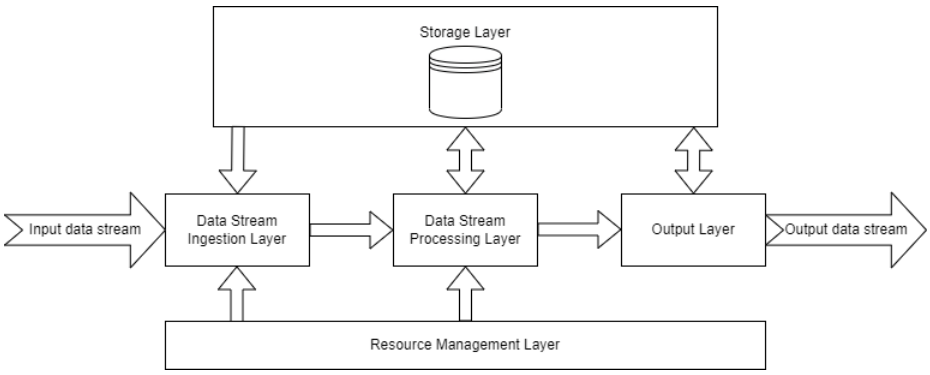

Fig. 1. Framework of a Stream Processing System

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 Kine-

[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/

sis 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 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.

[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/
[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/

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 [Isa+19; Liu+14] 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 understand what are its components. As we can see in Fig. 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.

[24] "Zookeeper" https://zookeeper.apache.org/
[25] "Twine" https://twinery.org/

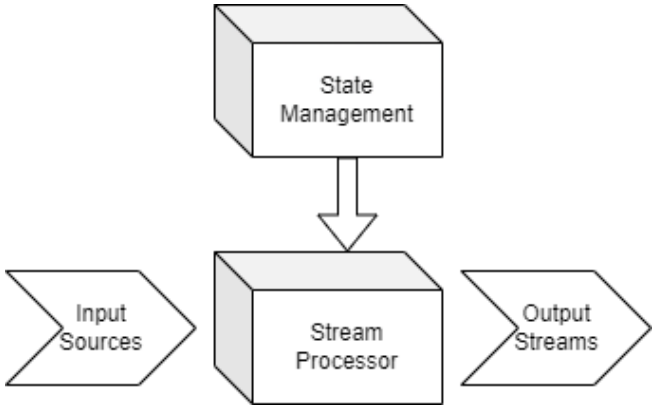

Fig. 2. Components of a Stream Processing Engine

Based on some works [Isa+19; Gor19; Baj+16; LS19] it's possible for us to 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 [Bor+20; Kar+18; Isa+19; Chi+16; SCS17; KYA17; LS19], 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 [Boc14], 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 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.

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

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. [Bon00]), 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. [ASB18] and Wang et al. [Wan+19]).

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

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 [Gul+22; Naj+19] 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 [RM19] 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 [RM19; Bor+20; Kar+18; Isa+19; Gor19; Chi+16; SCS17; Pal+21] 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 [Gul+22; Naj+19; Car+22; Bud+17; Naj+22] 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 *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.[Pal+21] 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,

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

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. [Gul+12], 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 [RM19; Kar+18; SCS17; Car+22] 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. [Bor+20] 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 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 [RM19; Bor+20; Kar+18; Isa+19; Gor19; Chi+16; Baj+16; LIX14; KYA17; LS19], 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 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-

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

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 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.

Table 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

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 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.3 we can see a representation of a classification of **Provenance** considering its granularity, according to [Tan+07].

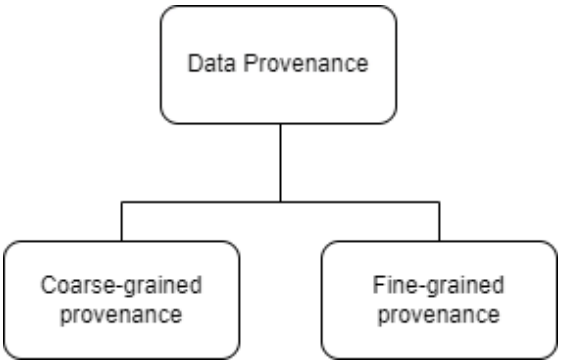

Fig. 3. Classification of Provenance regarding its granularity

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 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 [LMB09; Wan+15; YL21; Gla+14] 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. [PGP19] 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 [Pal+20], where the authors present **Ananke**, a framework that extends any fine-grained backward provenance tool and produces a live bipartite graph of fine-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.

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

s2p Ye and Lu [YL21] present **s2p**, a provenance solution for providing fine-grained and coarse-grained provenance in stream processing systems. Inspired by the philosophy of lambda architecture [Kir+15], 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. [Gla+14]. 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. [Tan+19]. 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 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 [YA22]. 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. [Zva+17; Zva+19] 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.

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

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. [HWA11] 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. [SWM12; SMW13]. 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. [Yam+22]. 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.

3 Architecture

In this section, we will present the architecture for our solution to the problem of auditable data provenance in streaming data processing. We will start by describing the requirements of our work, followed by an overview of our proposed solution, before giving an in-depth and detailed description of the basic concepts and implementation of our solution.

3.1 Requirements

Before we present in detail the architecture of our proposed solution, it's important to first understand the requirements that influenced our decisions and design.

As seen numerous times throughout this work and other works mentioned, providing provenance of data in stream processing is a complex and costly operation, in such a latency and throughput-dependent class of systems. To tackle this issue we follow the assumption that having to audit a stream and provide fine-grained provenance of specific data will be a rare event and executed offline. However, when that need arises our system must be able to audit the stream in a complete and accurate manner, even if that implies big memory and performance costs. We believe this trade-off is crucial to maximizing the system's overall performance, but without disregarding our main goal, which is to provide trustful and accurate provenance in a stream processing system.

3.2 Overview

Given the similarity of our main philosophy and the one followed by Ye and Lu when presenting s2p [YL21], we decided to inspire ourselves in their work. As mentioned when we first discussed their work, Ye and Lu developed a prototype of s2p in Apache Flink, which they called s2p-flink, and which allowed them to prove the correctness of their proposed solution. Our work will focus on following the same logic detailed in the creation of s2p, but with its implementation on

Apache Spark. We will extend Spark and implement our s2p-like solution internally following the idea of having two distinct phases, the **online provenance** phase to provide coarse-grained provenance, and the **offline provenance** phase responsible for fine-grained provenance. In our work, we will also attempt to overcome the s2p limitation of only providing correct provenance when a system consists of only deterministic operators.

3.3 Detailed Description

In this section, we will give a detailed description of the used systems and several components that constitute our proposed solution, which is represented in Fig.4. We will start by presenting the overall architecture of our stream processing system which can audit streams of data and provide provenance, followed by a description of both phases that respectively provide coarse-grained and fine-grained provenance. Finally, we will provide further information on our Provenance Management System. The relation between the Online phase, the Offline phase, and the Provenance Management System is depicted in Fig.5

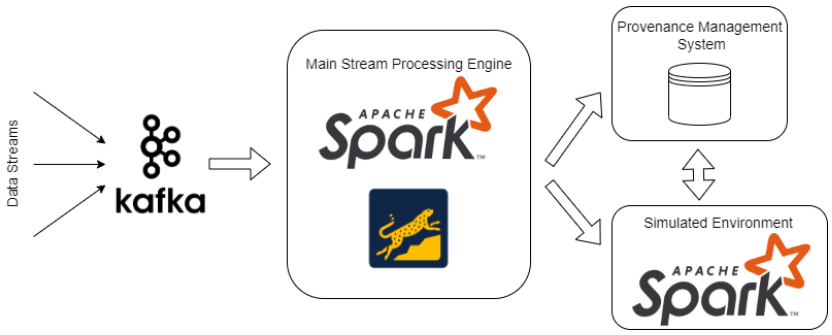

Fig. 4. Overview of our proposed solution's architecture

3.3.1 Stream Processing System

As mentioned previously, our proposed solution consists of a stream processing system where at the core, the stream processing engine chosen is Apache Spark. However, we perform an extension of Apache Spark in order to provide provenance for audited stream processing data.

To start, our proposed solution will rely on Apache Kafka to handle the input streams of data. Besides having the ability to manage several distinct input sources and grouping them to feed an input stream of data to Apache Spark, we will also take advantage of Kafka's internal data management to store every source data that enters our processing pipeline and produce unique IDs for it.

As mentioned before, at the core of our system we will have an Apache Spark extension. We will use Apache Spark Structured Streaming, a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. This allows us to express streaming computation the same way we would express batch computation for static data. Queries in Structured Streaming are processed with the use of micro-batches, which allows to achieve low end-to-end latency. The other main advantages of Spark Structured Streaming are being fault-tolerant and providing end-to-end exactly-once processing, which is guaranteed through checkpointing and Write-Ahead Logs.

The choice of using Apache Spark Structured Streaming is greatly influenced by its state store feature. A state store is used to store crucial intermediate data that needs to be maintained to correctly process streams of data. This feature could be implemented externally by us, however, the existence of built-in state store provider implementations in this engine is greatly beneficial to us, since we will leverage it to maintain a record of states which we will use to tackle the challenge of providing provenance for streams which include non-deterministic operations. We will use the RocksDB[32] state store provider, the more optimized built-in state management solution. RocksDB will allow us to store and update state information after the successful completion of the processing of a micro-batch which causes a change in the state.

We will also have a simulated environment where we will replay the stream of data from a specific point in order to obtain detailed information about that particular processing flow, which will allow us to provide fine-grained provenance for a specific result of interest.

Finally, we will need to implement a provenance management system that will be responsible for querying the provenance data, storing the detailed information regarding fine-grained provenance obtained by our replays in the simulated environment, and to serve as auxiliary storage for our state store.

3.3.2 Online Phase

Just like in s2p, our online phase for provenance corresponds to coarse-grained provenance, so there is no detailed description of how inner transformation affects the stream of data passing through the operators. In this phase, the main contribution is a mapping of the relationships between source data and result data.

To obtain this mapping we start by associating a unique ID to every input data before this data is ingested by source operators, which will give us the ability to access the stored input data when future provenance queries happen. We will take advantage of Apache Kafka to produce these unique IDs. Since Kafka manages internally the input data and produces an offset, which is a unique number, we can attach this to the input data and use it as its unique ID.

When the input data is processed and transformations occur we must propagate the unique ID to the newly formed data so we can keep track of the rela-

[32] "RocksDB" http://rocksdb.org/

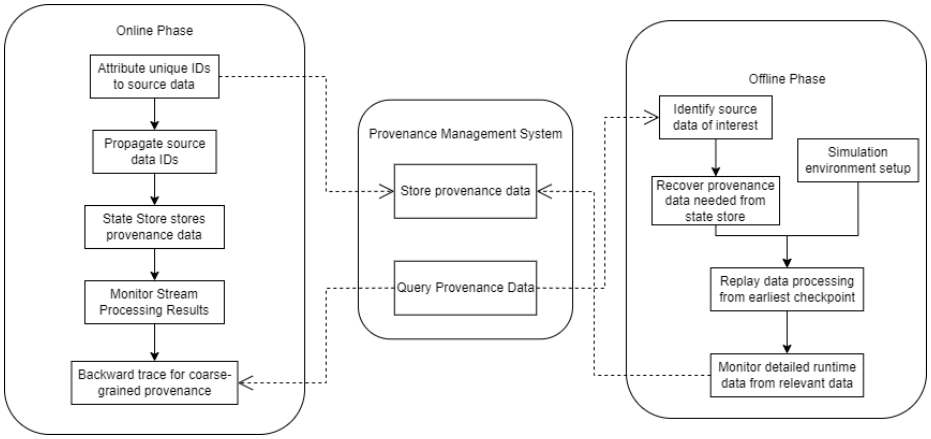

Fig. 5. Relation between Online phase, offline phase, and Provenance Management System

tionships between the data formed and the input data that led to its formation. With the use of piggybacking, the unique ID from source data is transmitted from data to data downstream through the use of operators in order when processing the data inside the stream. By extending the original data structure with a new list property, which saves the IDs from its ancestral source data upstream, we can observe in the result data a list with the IDs of all the ingested and processed source data that led to its creation, allowing us to know exactly which source data affected that result.

Despite this phase only being able to provide coarse-grained provenance, it is a crucial phase because it is here that the data we will later use to provide fine-grained provenance is stored. Even though the process of ID propagation at the time of transformations is sufficient to provide provenance for stateless operators, since data produced from these operators depends entirely on the input data received, the same cannot be said for stateful operators, which can have its processing affected by past events and non-deterministic data. For this reason, the intermediate and temporary data used by stateful operators must be stored so we can infer fine-grained provenance. This will be done through the use of RocksDB which can serve as a state store backend in Apache Spark. State store is a versioned key-value store that provides read and write operations. In Apache Spark Structured Streaming, the state store provider is used to handle the stateful operations across micro-batches. This means that for each micro-batch, all intermediate data needed for the replay of stateful operations will be stored in the state store.

Later, in the offline phase, if a stream needs to be audited, we can replay the processing of the data in our simulated environment knowing that by accessing the state store and retrieving the data that influenced the creation of

non-deterministic data we will produce the same results as the original processing.

3.3.3 Offline Phase

As for the offline phase, it's in this phase that fine-grained provenance will be provided. As explained before, the data acquired in the online phase allows us to replay the processing of specific data at a specific point in a simulated environment. Which, although costly, since we assume the need to replay data processing to be something rare, it will provide a good trade-off, because the original stream processing system won't be affected.

Our ability to provide fine-grained provenance relies on a combination of checkpointing and the RocksDB state store previously mentioned.

The process of obtaining fine-grained provenance information follows the next steps:

1. Initialize the simulation environment with our extended Apache Spark system;
2. Obtain the source data IDs list from the result data object we want to trace the lineage;
3. Determine which source data was processed first and what was its micro-batch;
4. After determining the first micro-batch to be processed, find in the state store the state at the time of its processing;
5. Configure the simulated environment with the state information extracted from the state store;
6. Replay the data processing in the simulated environment and track detailed information regarding data transformation.

In step number 3 we use a simple Algorithm 1 to determine which checkpoint is the earliest of those bound to the source IDs obtained previously.

Algorithm 1 Algorithm to determine earliest checkpoint

Require: Object source data set {(IDn, CPm)}
Ensure: Earliest checkpoint CPr needed to replay data processing

```
CPr ← MAX_CP
length ← length of {(IDn, CPm)}
for i ← 0 to length − 1 do
    if CPr is smaller than CPi in (IDi, CPi) then
        CPr ← CPi
    end if
end for
```

It's also important to note that, just like in the online phase, in this phase, we also generate unique IDs, but this time for every data formed in the processing

of our source data, these IDs will then also be added to a similar list structure in each data object, to identify all the pieces of data that led to its creation. We also store relevant information like temporary data and intermediate data, and we record the state values for stateful operators that contribute to the result data.

3.3.4 Provenance Management

Our provenance management system is extremely important to our proposed solution since it's here that most of the data, like intermediate and temporary data of interest, state information regarding stateful operators, parents' ID lists of stream data and the nearest checkpoint information associated with it, which are all needed for provenance. We will call all this stored data which is useful to provide provenance as **provenance data** from now on.

With all this data being stored by our Provenance Management system, it's important to differentiate it. Intermediate data formed in the processing of data is all stored in the offline phase and must be stored in a specific manner that preserves all the valuable information that comes with it. Hence, we will use the same data structure proposed by the authors of s2p to represent this data. Intermediate data is expressed as $<OP_{name}$, D_{uuid}, D_{value}, $\{P_{uuid}\}$, *Flag*$>$.

OP_{name} Operator's name.
D_{uuid} ID of this specific data.
D_{value} Actual value of this data.
$\{P_{uuid}\}$ Set of IDs from upstream data that this data object is related to.
Flag Denotes it as input or output.

The Provenance Management system is also responsible for querying the provenance data, which varies from the online to the offline phase. In the online phase, this query is the simple process of extracting the source ID list from the result data being queried and identifying the corresponding input data stream, represented by Algorithm 2, similar to the one presented by the authors of s2p.

Algorithm 2 Algorithm for provenance querying in the online phase

```
Require: Result data object of interest R
Ensure: Source data set S[0...i]
  S ← NULL
  if R.listOfIDs[] is not NULL then
      for all key ∈ R.listOfIDs[] do
          S.add(value)
      end for
  end if
```

The process of querying provenance data in the offline phase is more complex and is explained by Algorithm 3 proposed by the authors of s2p. This algorithm

produces a tree with result data as the root and the corresponding source data as the leaves. This tree can be reversed to obtain a Directed Acyclic Graph which depicts the data lifecycle from the input source to the result on the output stream with all the intermediate data in between. With this graph or tree, we can trace backward or forward to find data lineage.

Algorithm 3 Algorithm for provenance querying in the offline phase

Require: Result data object of interest R
Ensure: Tree T with result data as the root and the corresponding source data as the leaves

```
Q ← NULL
T ← NULL
Q.add(R)
while Q.size() is not 0 do
    treenode ← (TreeNode)Q.poll()
    T.add(treenode)
    parentIDs ← treenode.parentList
    if parentIDs is not NULL then
        for all id ∈ parentIDs do
            data ← GetData(id)
            Q.add(data)
        end for
    end if
    for all node ∈ T do
        if node.parentList contains treenode.ID then
            node.nextSet.add(treenode)
        end if
    end for
end while
```

4 Evaluation Methodology

In the previous section, we described the requirements and architecture of our proposed solution. In this section, we will present and explain the metrics and workloads used to analyze the performance and fulfillment of the requirements and architecture proposed.

4.1 Metrics

The following list represents the main performance metrics that we will consider during the evaluation phase of our implemented proposed solution.

Latency This metric represents the delay our system presents when processing data. The lower the latency, the more our system can handle data close to in

real-time. We aim to achieve low latency, although we expect some overhead caused by our provenance-related operations.

Throughput This metric measures how many units of data our system can process per unit of time. A system with high throughput can process greater amounts of data in less time. We aim to provide high throughput, however, some overhead can be expected and accepted, since our system takes longer to process data and release resources.

Resource Utilization As the name says, this metric evaluates the usage of resources by our implementation. We can divide this metric into two sub-metrics, which are **CPU** and **Memory** utilization.

It's important to note that the evaluation of these metrics will be done both for the regular streaming data processing with no audit operation and for the stream processing when an audit occurs to determine the provenance of some data. This way we can have a clear picture of the overhead caused by the provenance feature. These metrics are also important to compare the performance of our system with our main reference, s2p, but also with other implementations of solutions for the problem of provenance in stream processing which use the same metrics to evaluate their performance.

4.2 Workload

Next, we will present the workloads we will measure the previous metrics on. We will use several distinct workloads and data sets in order to obtain the most reliable results possible. To start we chose the same two datasets used in s2p's experimental evaluation:

Tweets Our first dataset consists of several Tweets from a period of six months and used on the data benchmark in [CCL10].

Movie Ratings Our second dataset is consisted of several ratings of movies and was used on the data benchmark [HK15].

These two datasets are then used in conjunction with three subject applications chosen from earlier works [Vei+16; Gul+19]. The first subject application is **WordCount**, which counts the times each word appears at regular intervals. The second subject application, **Grep**, finds the words in the input data sets that match any one in one given list of words. Lastly, we use the subject application **MovieRatings**, which finds movies with scores greater than four. Besides these micro-benchmarks, we will also use the ones proposed by [Wan+14] which are specific to streaming systems. They are **Search**, **Rolling Top Words**, **K-means** and **Collaborative Filtering**.

We will also utilize the use cases presented next, which are used in the performance evaluation of GeneaLog [PGP19] and Ananke [Pal+20], plus some macro-benchmarks commonly used to benchmark stream processing systems, to obtain a bigger result sample.

Linear Road We run two queries from the Linear Road Benchmark [Ara+04]. The first query detects broken-down vehicles through consecutive reports of zero speed and constant position, while the second one detects accidents from cars that are stopped at the same position.

Smart Grid From the smart grid domain we will also run two queries. The first one reports long-term blackouts by identifying meters with zero consumption for 24 hours. As for the second one, it detects anomalies through meters that report unusual consumption at midnight as compensation for the previous day.

Object Annotation In this use case, two queries receive information from a LiDAR and two cameras to enrich an in-vehicle computer vision system. It uses the Argoverse Tracking dataset [Cha+19], with 113 segments of 15-30s continuous sensor recordings of urban driving, plus 3D annotations of surrounding objects.

Vehicle Tracking This use case utilizes the GeoLife dataset [ZXM+10], composed of 18670 GPS traces of various vehicles over 4 years around Beijing. To simulate a large fleet driving simultaneously, it employs 10046 traces of cars driving a full day each.

Taxi Trip This dataset [DW16] contains data extracted from a four-year period of taxi trips in New York City, stored in CSV format. It contains two types of data: i) Fare data, which includes fare, tax, and tip amounts; ii) Ride data, which includes trip distance, trip duration, and pickup and dropoff location. Both types of data also include medallion number, hack license, and vendor ID. These there fields can be used to uniquely identify the taxi and its driver.

4.3 Setup

In order to evaluate our proposed solution with the previously mentioned workloads and to obtain results regarding the referred metrics, we need a setup where we can implement our desired framework.

We will first do a small deployment of our prototype in a server or a local cluster to simplify the implementation and testing of our stream processing system. Once the system is ready to be evaluated in a more demanding environment, closer to a real-life environment, we will use the Testground[33] for a larger deployment which simulates a deployment on a distributed infrastructure in a cloud provider.

5 Conclusion

Our work 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 also proposed an architecture for a system that

[33] "Testground" `https://docs.testground.ai/master/#/`

is 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. 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.

We design an architecture in which, even though some major overhead can be present, we know that the situations where that overhead occurs are rare and that is a tradeoff we gladly pay to achieve our main goal, which is to provide correct data provenance in a stream processing system. Our solution is inspired by a similar approach implemented in Apache Flink, which we will adapt and improve to extend Apache Spark and implement a feature of complete and correct data provenance in this stream processing engine.

We also propose a methodology to evaluate the correctness and performance of our proposed solution to assess its adequacy in realistic deployments and enable comparison against other approaches previously studied.

A Planning

The Gantt Chart presents the schedule of the main tasks and the respective deliverables and it will guide the work progress during the development of the thesis.

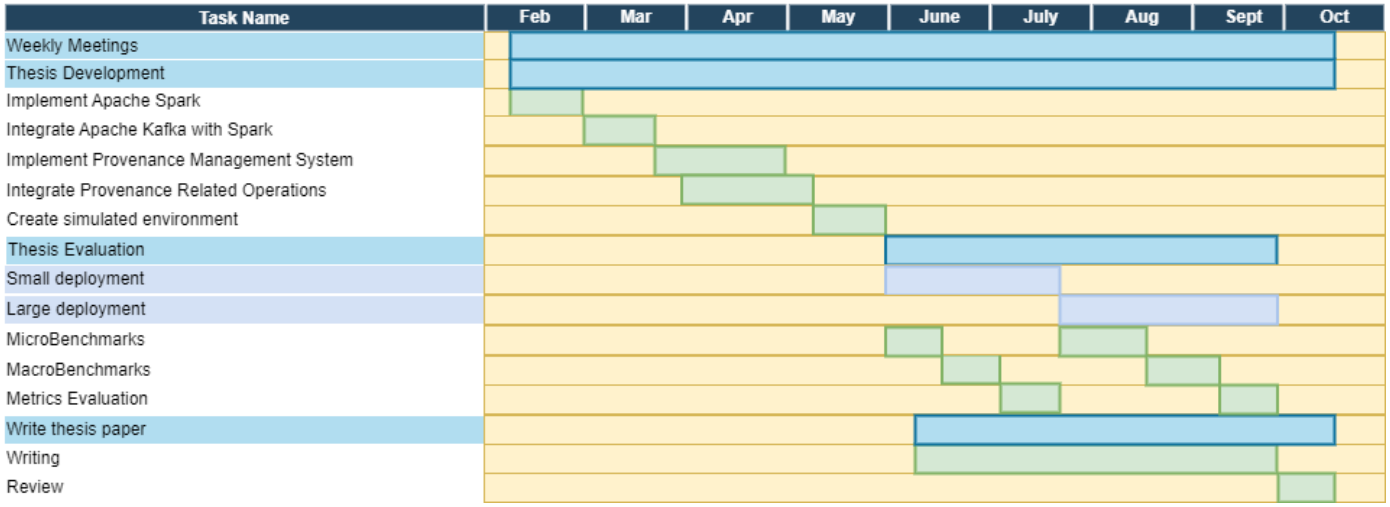

Fig. 6. Gantt Chart