TÉCNICO LISBOA

SDD4 Streaming

Tiago Mourão Lopes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Dr. Sérgio Ricardo de Oliveira Esteves

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Bruno Miguel Brás Cabral

January 2021

Acknowledgements

I would like to thank my parents for their encouragement and care over all these years, for always being there for me through thick and thin.

I would also like to acknowledge my dissertation supervisor Prof. Luís Veiga for the insight, support and sharing of knowledge.

Last but not least, to all my friends that helped me grow as a person and were always there for me during the good and bad times in my life and to my work colleagues that have taught me so much in so many levels, as a person as well as a professional, which will surely help in the years to come. For all the support given especially this year full of issues due to Covid, thank you, for without it this project may have not been finished.

Thank you.

Resumo

Com a quantidade cada vez maior de dispositivos que se conectam à Internet, sendo Internet das Coisas (IOT) um exemplo disso, aumenta também a quantidade de dados que precisam ser processados. O Stream Processing foi criado com o único propósito de lidar com grandes volumes de dados e provou repetidamente que pode lidar com isso com sucesso. No entanto, ainda é necessário melhorar ainda mais a escalabilidade e o desempenho desses sistemas.

Este trabalho propõe o SDD4Streaming, uma solução que vem mitigar estes problemas específicos dos Stream Processing Engines. Os motores atuais já implementam soluções de escalabilidade, mas com o tempo vimos que isso não é suficiente e são necessárias mais melhorias. Portanto, SDD4Streaming emprega uma extensão do sistema atual para melhorar o uso de recursos, de forma que as aplicações usem os recursos de que precisam para processar dados no menor tempo possível, aumentando assim o desempenho e ajudando outros aplicativos que estão sendo executados em paralelo no mesmo sistema.

Palavras-Chave: Processamento em fluxo, Apache Flink, Gestão de Recursos, Escalabilidade, Desempenho

Abstract

With the ever-increasing amount of devices getting connected to the internet, being Internet of Things (IOT) a good example of this, so does the amount of data that needs to be processed. Stream Processing was created for the sole purpose of dealing with high volumes of data and it has proven time and time again that it can successfully handle it. However, there is still a necessity to further improve scalability and performance on these systems.

This work presents SDD4Streaming, a solution that comes to solve these specific issues on the Stream Processing Engines. Current engines already implement scalability solutions but with time we've seen that this is not enough and further improvements are needed. So SDD4Streaming employs an extension of the current system to improve resource usage, so the applications use the resources it needs to process data in a timely manner, and so increasing the performance and help other applications that are running in parallel in the same system.

Keywords: Stream Processing, Apache Flink, Resource Management, Scalability, Performance

Contents

List of Figures

Chapter 1

Introduction

The increasing amount of devices connected with each other created a big demand for systems that can cope with the high volume that needs to be processed and analyzed according to certain criteria. Great examples of this are Smart Cities [1], operational monitoring of large infrastructures, and Internet of Things (IoT) [2]. Since most of this data is most valuable closest to the time it was generated, we need systems that can, in real-time, process and analyze all of the data as quickly as possible and for this to happen the technology Stream Processing was created.

1.1 Motivation

First, we should explain what the Stream Processing paradigm is. It is equivalent to dataflow programming [3], event stream processing [4], and reactive programming [5] but simplifies software and hardware parallelization by restricting the parallel computation that can be performed. This is done by, for a given sequence of data (a stream), apply a series of operators to each element in the stream.

At a high level, each technology that implements this paradigm will be similar, meaning that we can find common elements between them but the way they behave is different due to the way they decide to handle each of the issues inherent to streaming.

Even though this paradigm simplifies the processing of variable volumes of data through a limited form of parallel processing, it still has quite some issues that need to be tackled in order to have a resilient and performant system. Since the volume of data is ever-changing, the system needs to be able to adapt in order to accommodate and process this data accordingly in a timely manner while also being resilient so no data is lost while any stream is being processed.

Many of the technologies nowadays already address many of these issues, namely systems owned by the Apache Software Foundation, such as Apache Spark and Apache Flink to name a few, that handle reliability and scalability to a certain degree. But as with any type of software, especially one as high key as this one, there are always issues that could not be handled by the system itself either by limitations it has or it gives a mechanism for the user to handle it themselves.

1.2 Goals and Contributions

As we are going to cover in the next chapters, there are various solutions for this type of issue. However, most of them cannot handle the unpredictability of these types of systems. These solutions handler their optimizations before the application starts executing and so are not fully able to adapt to the variable state of the system.

With SDD4Streaming we intend to handle this issue by doing the optimizations during runtime. This brings extra complexity since we're handling a complex system that is executing but we're able to adapt to unexpected behaviors that may occur and adapt said system to accommodate its current load.

The individual work goals are:

- Investigate the state of the art and previous researches in SPEs, scalability, elasticity and bottleneck fixes;
- Check how Flink works, what metrics can be obtained from those systems and how they are obtained;
- Resulting from the previous study, design and create an architecture that allows for runtime job adaptation;
- Rescale the overall job while the lower level components are adapted automatically;
- Avoid the occurrences of bottlenecks in a job;
- Make use of the already existing mechanisms to avoid data loss while altering the system in runtime;

1.3 Document Organization

This document is structured as follows: Chapter 2 presents and analyses our related work. Chapter 3 introduces and describes SDD4Streaming, its architecture, data structures, and algorithms. Chapter 4 covers the implementation of our solution, with code snippets and figures representing the relation between the classes as well as with the client application and Flink. Next, Chapter 5 explains our evaluation methodology and presents the results obtained. Finally, Chapter 6 provides a set of closing remarks and a set of improvements and future work.

Chapter 2

Related Work

In this section, we will cover the research work and industry references that can be considered relevant to our initial objective. The following Section 2.1 will explain how Streaming Engines are structured and how they work in a high level. In Section 2.2 we will cover a set of systems defined as relevant. Finally, Section 2.3 will address some of the web technologies of interest in this area.

2.1 Stream Processing

Stream Processing can be decomposed in various dimensions/aspects, taking into account they are parallel and distributed data processing systems, that need to be addressed to create a functional system with a good quality of service. These dimensions are shown in Figure 2.1. We have **Distributed Architecture**, **Programming Model**, **Scheduling**, **Monitorability**, **Scalability**, **Real-Time Processing**, **Fault Tolerance**.

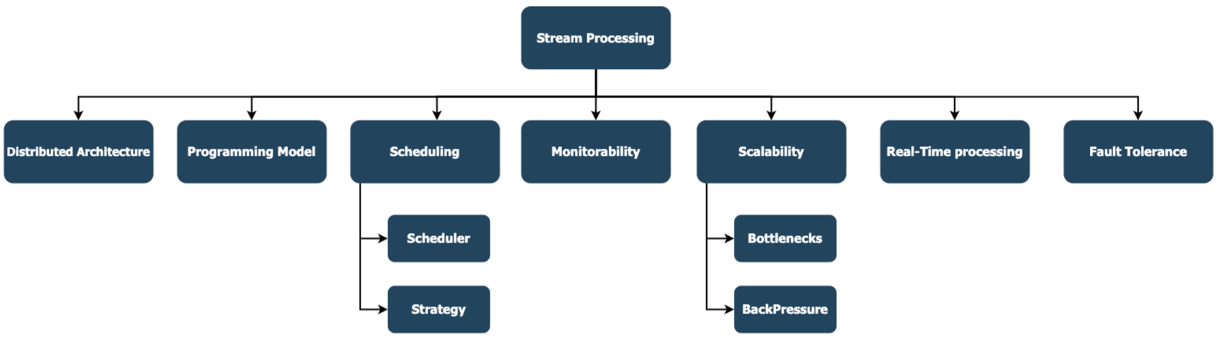


Figure 2.1: Stream Processing dimensions

Distributed Architecture: A stream processing engine is a type of system that often needs to be distributed among multiple machines/processes, hence, it should have an architecture that enables this [6]. These systems usually allow for the creation of jobs specific to a certain application that wants to use the services the system provides. To carry out actions according to the requisites of an application, the system uses a distributed architecture to distribute work over various machines and coordinate the data (input/output) as shown in Figure 2.2.

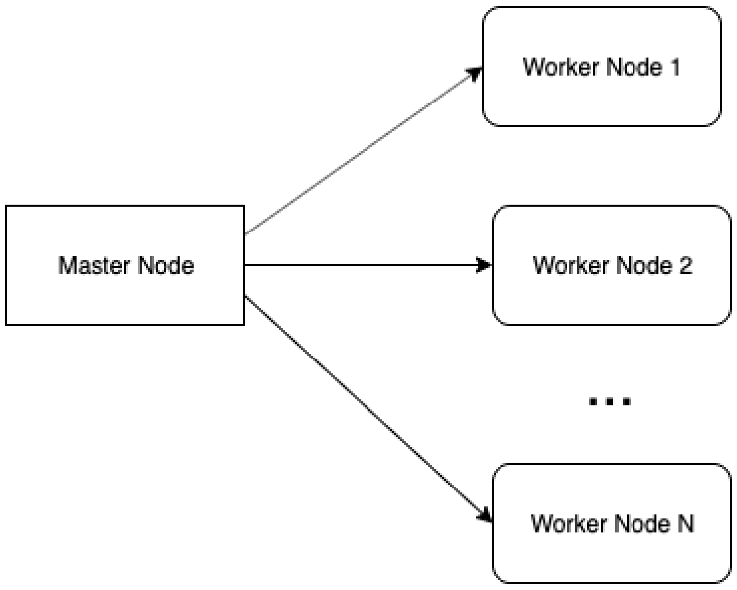


Figure 2.2: Generic Distributed Architecture

Thus, such a system can be depicted as a Master node that communicates with N Worker nodes which the amount can vary which is handled by scalability mechanisms (explained in its respective dimension).

The Master node is the entry point of input data into the application, which from here it will be sent into one or more worker nodes. So it will in a generic way act as a load-balancer for the worker nodes.

The Worker node is where the application data processing is done with the input received from the master. Usually, these nodes have a queue of data which is picked up by the N processes it has running for this purpose. Again this amount can change due to scalability reasons.

An example of a Distributed Architecture [7] that at the application layer uses the Spark Streaming framework is one for IoT Smart Grids Monitoring that needs to handle a great volume of data which cannot be handled a centralized system.

In a distributed environment, coordinating and managing a service has become a difficult process. To facilitate this, numerous technologies have been using *Apache ZooKeeper*.

Apache ZooKeeper [8] is used for maintaining centralized configuration information, naming, providing distributed synchronization, and providing group services in a simple interface so that developers don't have to write it from scratch. Apache Kafka also uses ZooKeeper to manage configuration. ZooKeeper allows developers to focus on the core application logic, and it implements various protocols on the cluster so that the applications need not implement them on their own.

Programming Model: All stream processing engines need a way for the user to create an application that uses said engine and how to interact with its components. For this, usually, the system will have libraries available in some programming languages (e.g. Java or Python) that a programmer can use for their intended purposes.

Such a library allows access to a multitude of functionalities that the programmer can do as for example create a Directed Acyclic Graph (DAG) or just a simple pipeline through the use of the operators available (e.g. map, reduce, filter, etc) which is dependent on the system/engine. One important part of the programming model is to abstract components [9] to hide complexity from the programmer since he should focus on the application part only and not what supports it (that should be provided by the framework).

Besides creating pipelines, the programming model usually allows the programmer to customize the temporal window he wants to use. Windows are at the heart of processing infinite streams. Windows split the stream into "buckets" of finite size, over which we can apply computations. In Figure 2.3 we are able to see what a sliding window is and how it can be represented.

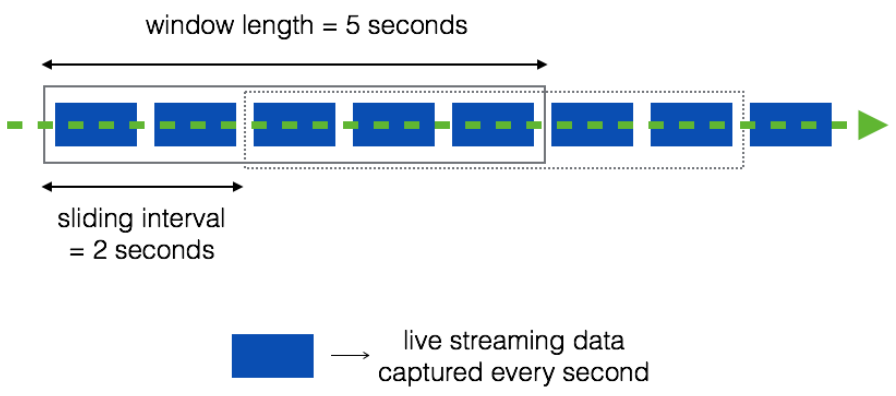

window length = 5 seconds
sliding interval = 2 seconds
live streaming data captured every second

Figure 2.3: Sliding Window Example (https://prateekvjoshi.com/2015/12/29/performing-windowed-computations-on-streaming-data-using-spark-in-python/)

There are multiple types of windows [10], and the most employed ones are Tumbling Window, Hopping Window, Sliding Window, and Session Window.

- Tumbling Window: A tumbling window has a fixed length. The next window is placed right after the end of the previous one on the time axis. Tumbling windows do not overlap and span the whole time domain, i.e. each event is assigned to exactly one window.

- Hopping Window: Like tumbling windows, hopping windows also have a fixed length. However they introduce a second configuration parameter: The hop size h. Instead of moving the window of length s forward in time by s we move it by h. This means that tumbling windows are a special case of hopping windows where s = h. If s $>$ h windows are overlapping and if s $<$ h some events might not be assigned to any window.

- Sliding Window: A sliding window, opposed to a tumbling window, slides over the stream of data. Because of this, a sliding window can be overlapping and it gives a smoother aggregation over the incoming stream of data - since you are not jumping from one set of input to the next, rather you are sliding over the incoming stream of data.

- Session Window: In contrast to the previous window functions session windows have a variable length. When using a session window function you need to specify a time threshold between consecutive events that must not be exceeded. The window will keep expanding as long as new events are coming in that are close enough in time.

Scheduling: When dealing with a system that processes a great amount of data, we need a way to manage and maintain that data going through in the least amount of time and in the most efficient way possible. This is usually done through scheduling which means controlling and making a workload the most efficient it can be. A workload, specifically for scheduling, usually is composed of tasks that represent a unit of work and a task manager which is responsible for all the tasks assigned to it.

For scheduling, we want to maintain a good load balance between the system resources because we don't want a machine overloaded, neither one with no inputs to process. To achieve this we need to have a good scheduling strategy.

The strategy defines how the tasks/work should be divided based on information gathered from previous jobs and on a set of rules pre-established. To give an example of a strategy, we have Round-Robin, one of the most commonly used strategies, where tasks are assigned to each process in equal portions and in circular order, handling all processes without priority.

While Round-Robin is a simple strategy, we can use more complex ones that take into account the network, for example, [11] which makes the scheduling more intelligent and in turn better and less prone to cause issues in the system. Another example would be having an adaptative control of stream processing [12] in order to not waste resources when in an extreme-scale scenario.

Monitorability: Whenever we want a system to have a good quality of service with a certain level of availability, we need a way to monitor said system so we can then act upon it when an issue occurs.

There are various ways and levels of monitorability that can be applied to infrastructure. An easy division can be, for example, network and system monitoring. Each has its functions, advantages, and disadvantages and the user needs to use them accordingly to his needs.

A simple monitoring process that can be done through the network could be, pinging all the machines in the system from time to time to check what the latency is in the requests and if there is any machine that is not responding.

For more advanced monitoring, we need to start combining both types of monitoring and system monitoring is the most important one for Stream Processing. By monitoring a system we can know, usually in real-time, what is going on in a specific machine in terms of software and hardware. This is useful since through the network we can only know the latency in general but through the system itself, we can learn what is causing such latency and so act upon it.

The most common way to access logs and information about a system in execution and the one numerous technologies use is through a REST API. For example Apache Flink has an API to fetch metrics from the system [1].

This API has numerous categories of data from the system as for example the general health of the system, such as if any machine went down and for how long. You can also check the progress of current jobs and how many resources are being used by the machines (e.g. CPU).

There are also solutions external to the stream processing engine specific to monitoring said engine. For example Sematext [2] is a solution for Full Stack Infrastructure Monitoring and Management. It can be

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/rest_api.html

[2] https://sematext.com/spm/

integrated on Apache Spark for example and by using its tools, we are able to monitor the system.

Scalability: For a system that is constantly dealing with data and with clients that are expecting a certain Quality of Service (QoS) [3] from this system, we need to have a degree of scalability to be prepared for any type of situation that might happen.

So scalability is the property that a system has, to be elastic [13] (ability to change itself) to accommodate the requirements it has and in the ever-changing amount of work it receives. This involves the change in the number of resources available and includes either growing whenever there is more work than resources available or shrinking when the amount of work decreases over time and we have more resources than the ones needed.

As an example, we can imagine an API where internally it has a load-balancer that redirects the requests to the worker machines which will then process the said request. This system supports 1000 req/s at a certain point in time and so with this, we have three situations that can happen. Either we are receiving fewer requests than our limit we can support and so wasting resources (e.g. paying unnecessary money, etc), have exactly the amount we support and this would be the perfect world for the system but it's not a real scenario that we should take into account as when it happens its usually for a really small amount of time. The third case is when the number of requests exceeds the limit of what the system supports and so a bottleneck shall occur and the QoS will decrease while latency increases.

To give example of systems, Aurora [14] and Medusa [15] are stream processing engines that try to be scalable but still have some issues which the article Scalable Distributed Stream Processing [16] explains what the issues are and how they could be solved.

Real-Time Processing: Due to the type of inputs that normally these systems receive (e.g. sensors, IoT [2]), they need to be processed and analyzed as fast as possible to get the most value out of them [17]. For example, sensors that check for traffic in a street, if the data gotten from them isn't processed as close as possible from the time they were generated we may be acting on an invalid state that had already happened in the past. One case of this would be checking the data from the sensors from a few seconds ago and what is currently happening is different and so the action will no according to reality.

So we need to be always processing as soon as any type of input is received and the systems need to be prepared for it. This not only involves the machines that will do the processing which needs to have the needed resources for the job, but also the databases which will need to store all the information being received and processed.

Fault-Tolerance: The most essential part of a streaming engine is processing/analysis of data, so data loss is something we don't want to happen depending on the type of data we are dealing with. For financial services, for example, we do not want to lose data since it is precious. But in most cases, it's preferable to discard data instead of delaying computation. To overcome these issues, we need to have a fault-tolerant system [18] and mechanisms that make it possible.

Data losses can happen due to numerous reasons, as for example a machine that has information

[3]https://www.networkcomputing.com/networking/basics-qos

stored and was in the middle of a workload and crashes. For this case, first of all, we need the system being monitored and so when the machine crashes, the said system can identify it and apply its mechanism to recover from it.

For recovery from a fault, there are numerous mechanisms to do so. Some of the most famous ones are *snapshots* [19] and *resilient data structures* [20].

For snapshots, we can store a certain state of a part of the system's state (e.g. a certain machine doing a part of a processing job) at a certain point in time which can, later on, be used to restore the system to a consistent state. This, for example, is used to quite a degree in operating systems, where the users can save the current information in their system as to recover sometime in the future in case something goes wrong in the system.

For the other, the resilient data structures, we can maintain at all times the current state and record all the previous operations done in a machine that is doing processing work. For example, we can have an immutable data structure which for every operation made a log of it is created. Through the execution time of the system we start getting a history of logs which in case of a failure/fault, we can use this history to recover the state.

2.2 Stream Processing Technologies

There are numerous technologies nowadays that can process a great amount of data in real-time. They all excel in one way or another so each does something better than the other so they are specific to each situation/business. Even though the technologies that will be explained next, some have different nomenclatures for their components, at a high level they all mean the same but with different names. For example, Master and Worker nodes may be called by different names depending on the system.

2.2.1 Apache Spark

Apache Spark [4] [21] is a cluster computing solution that extends the MapReduce model to support other types of computations such as interactive queries and stream processing. Designed to cover a variety of work-loads, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs) that enables running computations from inputs in memory. RDDs are immutable and partitioned collections of records, and provide a programming interface for performing operations, such as map, filter and join, over multiple data items.

For fault-tolerance purposes, RDD tracks the graph of transformations that were used to build it and reruns these operations on base data to reconstruct any lost partitions.

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates a new RDD when we apply any transformation. Applying transformation builds a lineage, including the entire parent RDDs of the final RDD(s). RDD lineage, also known as RDD operator graph or RDD dependency graph. It is a logical

[4]https://spark.apache.org/

execution plan i.e., it is a Directed Acyclic Graph of the entire parent RDDs of the final RDD.

The basic fault-tolerant semantics of Spark are:

- Since Apache Spark RDD is an immutable dataset, each Spark RDD remembers the lineage of the deterministic operation that was used on fault-tolerant input datasets to create it.
- If due to a worker node failure any partition of an RDD is lost, then that partition can be re-computed from the original fault-tolerant dataset using the lineage of operations.
- Assuming that all of the RDD transformations are deterministic, the data in the final transformed RDD will always be the same irrespective of failures in the Spark cluster.

But this system brings an issue which is the inability to ingest live streams of data. For this, it has a component named Spark Streaming which will be explained in the next subsection.

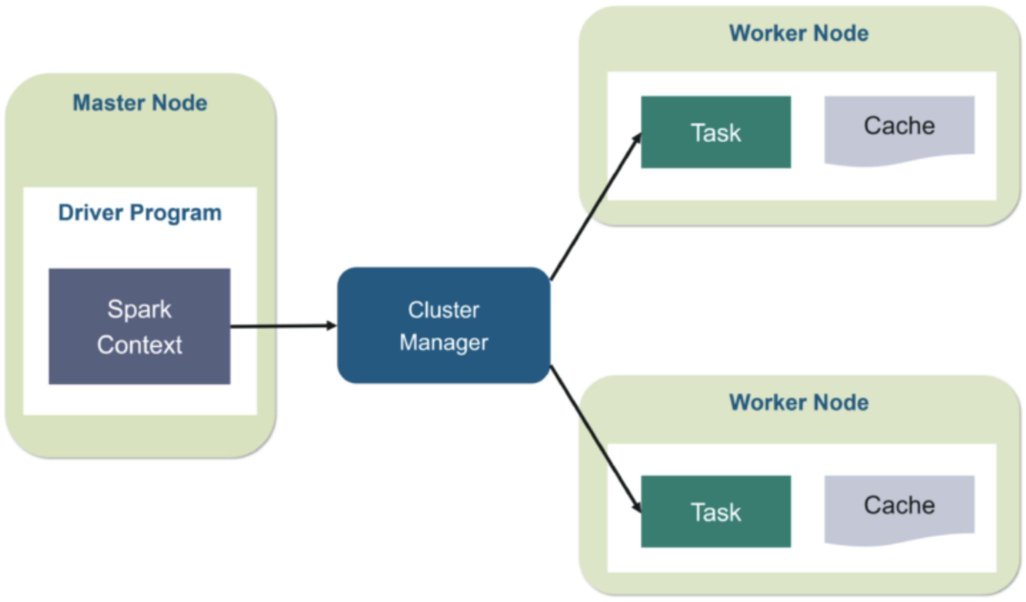


Figure 2.4: Apache Spark architecture (https://www.edureka.co/blog/spark-architecture/)

2.2.2 Apache Spark Streaming

Apache Spark [5] is composed of multiple components, from which the user can use in any type of combination he wants depending on his needs. The most important one from the list is Spark Streaming. This component works as an extension of the Spark Core functionalities and makes it so we can have live streaming of data across the system with its new added functionalities.

Spark Streaming offers scalable, fault-tolerant and high-throughput processing of live data streams from numerous types of sources. To be able to ingest live data streams, it has a **Discretized Stream** abstraction named ”DStream” which represents a stream of data divided into small batches and is built on top of Spark RDDs, which are Spark’s core data abstraction.

[5]https://spark.apache.org/streaming/

Dividing the data into small micro-batches allows for the fine-grained allocation of computations to resources, which in turn allows for a dynamic load-balancing. Let us consider a simple workload where partitioning of the input data stream needs to be done by a key and processed. In the traditional record-at-a-time approach, if one of the partitions is more computationally intensive than others, the node to which that partition is assigned will become a bottleneck and slow down the pipeline. The job's tasks will be naturally load-balanced across the workers where some workers will process a few longer tasks while others will process more of the shorter tasks in Spark Streaming.

Traditional systems have to restart the failed operator on another node to recompute the lost information in case of node failure. Only one node is handling the recomputation due to which the pipeline cannot proceed until the new node has caught up after the replay. In Spark, the computation discretizes into small tasks that can run anywhere without affecting correctness. So failed tasks we can distribute evenly on all the other nodes in the cluster to perform the recomputations and recover from the failure faster than the traditional approach.

2.2.3 Apache Flink

Apache Flink [6] [22] offers a common runtime for data streaming and batch processing applications. Applications are structured as arbitrary DAGs, where special cycles are enabled via iteration constructs. Flink works with the notion of streams onto which transformations are performed. A stream is an intermediate result, whereas a transformation is an operation that takes one or more streams as input, and computes one or multiple streams. During execution, a Flink application is mapped to a streaming workflow that starts with one or more sources, comprises transformation operators, and ends with one or multiple sinks. Although there is often a mapping of one transformation to one dataflow operator, under certain cases, a transformation can result in multiple operators. Flink also provides APIs for iterative graph processing, such as Gelly.

The parallelism of Flink applications is determined by the degree of parallelism of streams and individual operators. Streams can be divided into stream partitions whereas operators are split into sub-tasks. Operator sub-tasks are executed independently from one another in different threads that may be allocated to different containers or machines.

Apache Flink offers a fault tolerance mechanism to consistently recover the state of data streaming applications. The mechanism ensures that even in the presence of failures, the program's state will eventually reflect every record from the data stream exactly once. Note that there is a switch to downgrade the guarantees to at least once (described below). The fault tolerance mechanism continuously draws snapshots of the distributed streaming data flow. For streaming applications with a state that has little information, these snapshots are very light-weight and can be drawn frequently without much impact on performance.

The state of the streaming applications is stored at a configurable place (such as the master node, or HDFS). In case of a program failure, Flink stops the distributed streaming dataflow. The system then restarts the operators and resets them to the latest successful checkpoint. The input streams are reset to

[6] https://flink.apache.org/

the point of the state snapshot. Any records that are processed as part of the restarted parallel dataflow are guaranteed to not have been part of the previously checkpointed state.

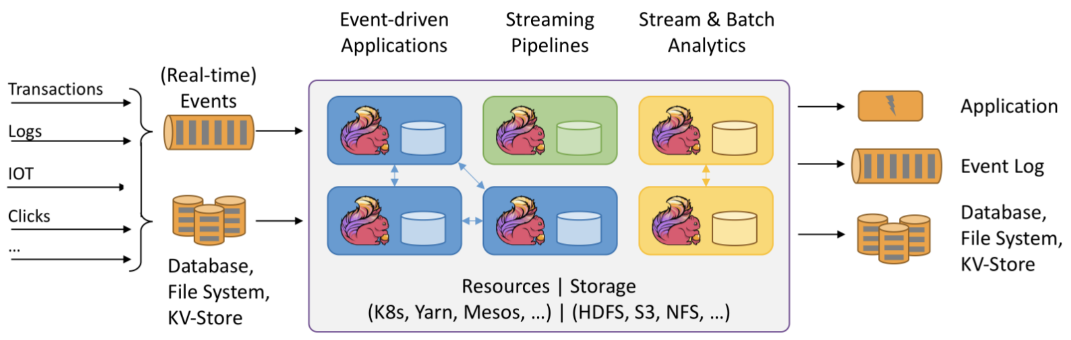

Event-driven Applications
Streaming Pipelines
Stream & Batch Analytics
Transactions
Logs
IOT
Clicks
...
(Real-time) Events
Database, File System, KV-Store
Resources | Storage
(K8s, Yarn, Mesos, ...) | (HDFS, S3, NFS, ...)
Application
Event Log
Database, File System, KV-Store

Figure 2.5: Apache Flink architecture (https://flink.apache.org/)

2.2.4 Google MillWheel

Google MillWheel [23] is a stream processing system built at Google that models computation as a dynamic directed graph of computations. MillWheel allows user's to write arbitrary code as part of an operation yet still transparently enforces idempotency and exactly-once message delivery. MillWheel uses frequent checkpointing and upstream backup for recovery.

Data in MillWheel is represented by (key, value, timestamp) triples. Values and timestamps are both arbitrary. Keys are extracted from records by user provided key extraction functions. Computations operate on inputs and the computations for a single key are serialized; that is, no two computations on the same key will every happen at the same time. Moreover, each key is associated with some persistent state that a computation has access to when operating on the key.

MillWheel also supports low watermarks. If a computation has a low watermark of t, then it's guaranteed to have processed all records no later than t. Low watermarks use the logical timestamps in records as opposed to arrival time in systems like Spark Streaming. Low watermark guarantees are not actually guarantees; they are approximations. Injectors inject data into MillWheel and can still violate low watermarks semantics. When a watermark violating record enters the system, computations can choose to ignore it or try to correct it. Moreover, the MillWheel API allows users to register for code, known as timers, to execute at a certain wall clock or low watermark time.

2.2.5 Amazon Kinesis Streams

You can use Amazon Kinesis Data Streams [7] to collect and process large streams of data records in real time. You can create data-processing applications, known as Kinesis Data Streams applications. A typical Kinesis Data Streams application reads data from a data stream as data records. These applications can use the Kinesis Client Library, and they can run on Amazon EC2 instances. You can send the processed

[7]https://aws.amazon.com/kinesis/data-streams/

records to dashboards, use them to generate alerts, dynamically change pricing and advertising strategies, or send data to a variety of other AWS services.

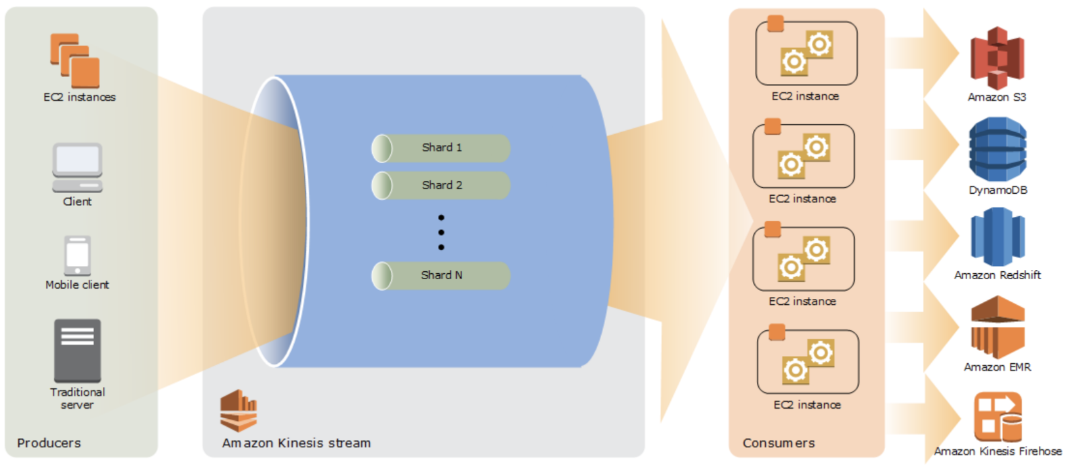


Figure 2.6: Amazon Kinesis architecture (https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html)

As shown in Figure 2.6, the system is based on producers that continually push data to Kinesis Data Streams, Kinesis Data Streams which is a set of shards, and the consumers process the data in real-time.

The important part here is the Kinesis Streams and how they work. They are composed of a set of shards. A shard is a uniquely identified sequence of data records in a stream. A stream is composed of one or more shards, each of which provides a fixed unit of capacity. The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards.

Each data record has a sequence number that is unique per partition-key within its shard. Kinesis Data Streams assigns the sequence number after you write to the stream using the client library provided. Through the Kinesis Client Library, someone can create an application using the supported languages and easily set up a simple streaming operation. This library is compiled into the application to enable fault-tolerant consumption of data from the stream. The Kinesis Client Library ensures that for every shard a record processor is running and processing that shard. The library also simplifies reading data from the stream. The Kinesis Client Library uses an Amazon DynamoDB table to store control data. It creates one table per application that is processing data.

2.2.6 Apache Storm

Apache Storm [8] [24] is a distributed real-time computation system for processing large volumes of high-velocity data. Storm is extremely fast, with the ability to process over a million records per second per node on a cluster of modest size. Enterprises harness this speed and combine it with other data access applications in Hadoop to prevent undesirable events or to optimize positive outcomes.

Apache Storm has two types of nodes, Nimbus (master node) and Supervisor (worker node). Nimbus is the central component of Apache Storm. The main job of Nimbus is to run the Storm topology. Nimbus

[8] https://storm.apache.org/

analyzes the topology and gathers the task to be executed. Then, it will distribute the task to an available supervisor. A supervisor will have one or more worker processes. The supervisor will delegate tasks to worker processes. The worker process will spawn as many executors as needed which will have the job of executing the task. Apache Storm uses an internal distributed messaging system for communication between nimbus and supervisors.

Since Storm cannot manage its cluster state, it depends on Apache ZooKeeper for this purpose. ZooKeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgments, processing status, etc.

The basic primitives Storm provides for doing stream transformations are ”spouts” and ”bolts”. Spouts and bolts have interfaces that you implement to run your application-specific logic.

As in Figure 2.7, the architecture of Apache Storm can be compared to a network of roads connecting a set of checkpoints. Traffic begins at a certain checkpoint (called a spout) and passes through other checkpoints (called bolts). The traffic is, of course, the stream of data that is retrieved by the spout (from a data source, e.g. a public API) and routed to various bolts where the data is filtered, sanitized, aggregated, analyzed, and sent usually to a dashboard where people can use.

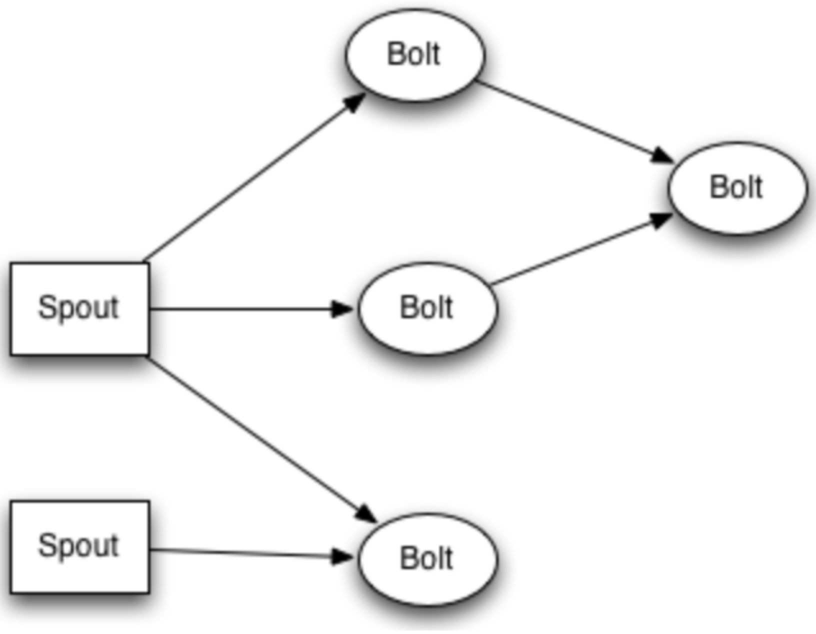


Figure 2.7: Apache Storm architecture

Storm is based on the ‘fail fast, auto restart’ approach that allows it to restart the process once a node fails without disturbing the entire operation. This feature makes Storm a fault-tolerant engine. It guarantees that each tuple will be processed ‘at least once or exactly once’, even if any of the nodes fail or a message is lost. Also, this is only possible because the nodes are stateless since all state is kept in Zookeeper or on disk.

2.2.7 Apache Heron

While maintaining API compatibility with Apache Storm, Apache Heron [9] [25] was built with a range of architectural improvements and mechanisms to achieve better efficiency and to address several of Storm issues highlighted in previous work. Heron topologies are process-based with each process running in isolation, which eases debugging, profiling, and troubleshooting. By using its built-in backpressure mechanisms, topologies can self-adjust when certain components lag.

Similarly to Storm, Heron topologies are directed graphs whose vertices are either Spouts or Bolts and edges represent streams of tuples. The data model consists of a logical plan, which is the description of the topology itself and is analogous to a database query; and the physical plan that maps the actual execution logic of topology to the physical infrastructure, including the machines that run each spout or bolt.

Even though Heron shares so many similar things with Storm since both came from the same company (Twitter and then passed onto Apache), Heron belongs to a newer generation and comes to fix some issues the older one had.

Resource isolation: Heron uses process-based isolation both between topologies and between containers within topologies, which is more reliable and easier to monitor and debug than Storm's model, which involves shared communication threads in the same JVM.

Resource efficiency: Storm requires scheduler resources to be provisioned upfront, which can lead to over-provisioning. Heron avoids this problem by using cluster resources on demand.

Throughput: For a variety of architectural reasons, Heron has consistently been shown to provide much higher throughput and much lower latency than Storm.

2.3 Relevant Research Work

In this subsection, two papers that come with solutions for performance and scalability issues on top of other pre-existing base stream processing systems.

In a higher level they include the management of resources but in a static manner where it won't change the system in runtime as well as managing the inputs and outputs in a way that they are able to know if there is a need to process something that doesn't differ much from the previous results.

2.3.1 Resource Management

When developing a stream processing application/job, the programmer will define a Directed Acyclic Graph (DAG) with all the operations that will be done upon the inputs received. The right choice for this topology can make a system go from very performant with high throughput, to very slow with high latency and bottlenecks.

So the paper proposes SpinStreams [26], a static optimization tool able to leverage cost models that programmers can use to detect and understand the inefficiencies of an initial application design.

[9]https://apache.github.io/incubator-heron/

SpinStreams suggests optimizations for restructuring applications by generating code to be run on a stream processing system. For testing purposes, the author used an Streaming Processing System (SPS) called Akka [27].

There are two basic types of restructuring and optimization strategies applied to streaming topologies:

- **Operator fission:** Pipelining is the simplest form of parallelism. It consists of a chain (or pipeline) of operators. In a pipeline, every distinct operator processes, in parallel, a distinct item; when an operator completes a computation of an item, the result is passed ahead to the following operator. By construction, the throughput of a pipeline equals to the throughput of its slowest operator that represents the bottleneck. A technique to eliminate bottlenecks is to apply the so-called pipelined fission, i.e. to create as many replicas of the operator as needed to match the throughput of faster operators (possibly adopting proper approaches for item scheduling and collection, to preserve the sequential ordering)

- **Operator fusion:** A streaming application could be characterized by a topology aimed at expressing as much parallelism as possible. In principle, this strategy maximizes the chances for its execution in parallel, however, sometimes it can lead to a misuse of operators. In fact, on the one hand, the operator processing logic can be very fine-grained, i.e. much faster than the frequency at which new items arrive for processing. On the other hand, an operator can spend a significant portion of time in trying to dispatch output items to downstream operators, which may be too slow and could not temporarily accept further items (their input buffers are full). This phenomenon is called backpressure and recursively propagates to upstream operators up to the sources

The SpinStreams workflow is summarized in Figure 2.8. The first step is to start the GUI by providing as input the application topology. It is expected that the user knows some profiling measures, like the processing time spent on average by the operators to consume input items, the probabilities associated with the edges of the topology, and the operator selectivity parameters. This information can be obtained by executing the application as is for a sufficient amount of time, so that metrics stabilize, and by instrumenting the code to collect profiling measures.

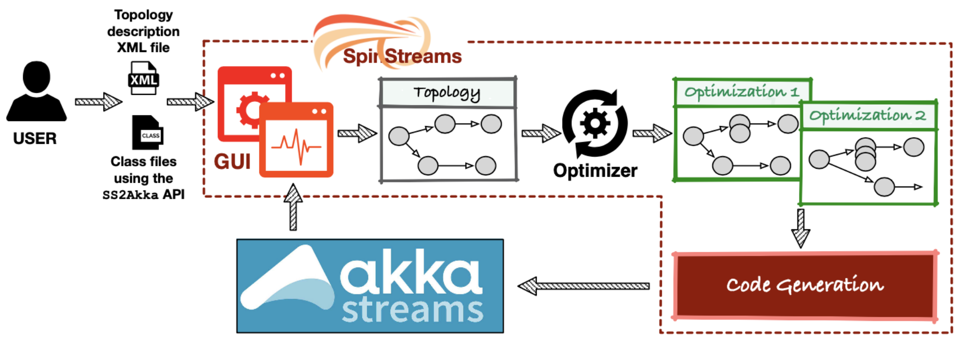


Figure 2.8: Spin Streams Workflow (from [26])

The main inputs to SpinStreams are:

- the structure of the topology and the profiling measurements expressed in an XML file. The syntax provides tags to specify the operators, with attributes for their name, the service rate (specifying the time unit), the pathname of the class file, the type (stateless, stateful, partitioned-stateful with the number of keys and the file with their probability distributions). Other tags specify the output edges and their probability, and the input/output selectivity;
- along with the XML file, the user provides, for each operator, a .class file obtained by compiling a source code written using a specific API. Such API is provided to allow the automatic code generation from the abstract representation used in SpinStreams to the code to be run on the target SPS. For Akka this API is called SS2Akka.

SpinStreams checks if the input topology satisfies the constraints (acyclicity and rooted graph) before creating a new imported entry that will contain all the versions prototyped for the topology. After, the user can request SpinStreams to introduce some specific optimizations such as identify and remove bottlenecks and/or try a fusion optimization by selecting sub-regions of the graph. SpinStreams proposes a set of candidates , ranked by their utilization factor in order to ease the process of selection of the sub-graph to be fused. Once chosen, the user starts the fusion optimization that produces a new topology.

2.3.2 Input and Processing Management

A stream processing application usually will be used for a certain type of data (e.g. data being generated by sensors in a smart city) and not for a range of applications. So with this, we can create an application that depends on the input it receives and based previous training (machine learning) it decided whether or not it should process them or just simply give the last result. For certain applications where the workflow output changes slowly and without great significance in a short time window, we are wasting resources inefficiently and making the whole process take a lot longer than it could take while remaining with a moderately accurate output.

To overcome these inefficiencies, SmartFlux [28] comes with a solution that involves looking at the input the system receives, train a model using Machine Learning with this model check and analyze if the input being received needs to be processed all over or not with a good confidence level. This is done through a middleware framework called **SmartFlux** which affects one part of a Stream Processing Engine which is the Workflow Management System since it wants to intercept the way the workflows are being processed.

In Figure 2.9 we can see the architecture that was designed for the Smart Flux solution.

The system has two different operating modes: i) training mode; and ii) execution mode. In the training mode, a workflow is executed synchronously and its collected metrics about the input impact and output deviation for each processing step that tolerates error. After a predetermined number of waves, a classification model is built with the previously collected data. The training mode is represented by the white curved arrows: the Monitoring component, that gets data from the adaptation components, feeds the Knowledge Base with statistical information about the data updated in the data store; then, the

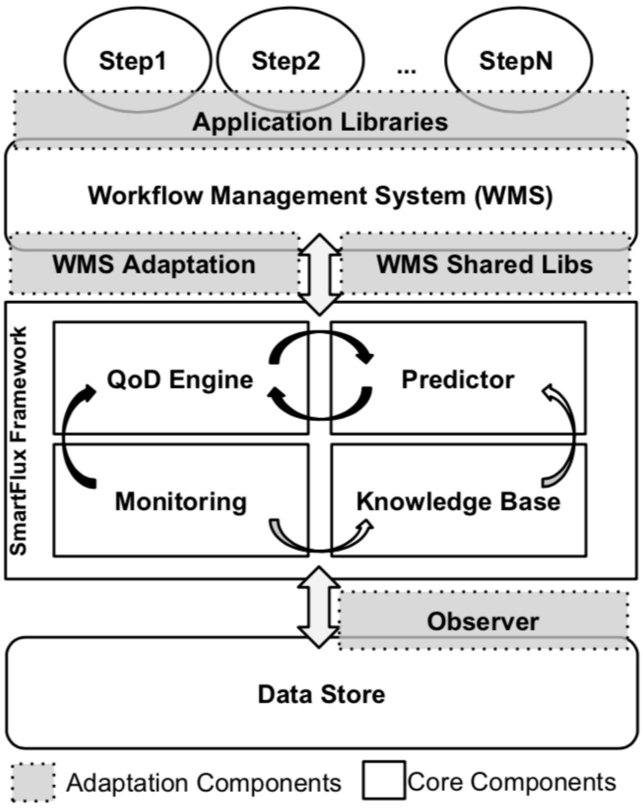


Figure 2.9: SmartFlux Middleware Architecture (from [28])

Predictor component builds a classification model based on the data sets with the metrics contained in the Knowledge Base (input impact, error).

The execution mode is represented by the dark curved arrows: the Monitoring component collects statistical information from data store R/W requests, and sends to the Quality of Data (QoD) Engine computed input impact metrics at each wave of data; after, the QoD Engine queries the Predictor with input impact data and gets in return the configuration of processing steps that should be executed.

Smart Flux has a learning process in order to bound the output error, arising from the delayed execution of processing steps, and to provide guarantees about the maximum deviation of workflow outputs. Specifically, it makes use of Machine Learning classification techniques to predict how input data affects the output of processing steps. This based on predictions that are naturally not perfect, and therefore called probabilistic guarantees; i.e., the capability to ensure that error bounds are respected within a confidence interval.

The solution has three possible execution phases:

- **Training Phase**: Unless a training set is given beforehand, a training phase starts taking place when the workflow is executed for the first time. During this phase, all processing steps of the workflow are executed synchronously (without any QoD enforcement). The duration of this phase is configured by users with a specified number of waves.

- **Test Phase**: Assess the quality of the trained model measuring: (i) accuracy, the proportion of instances correctly classified; (ii) precision, the number of classified instances that are truly of a class divided by the total number of instances classified as belonging to that class; and (iii) recall, the number of instances classified as a given class divided by the number of instances that are truly of that class.

- **Application Phase**: After a sufficiently accurate model is built, the application phase takes place and the workflow starts running asynchronously in an adaptive way. At each wave, the input impact ι is calculated for each step and fed to the classifier, which in return indicates which steps should be executed.

2.4 Analysis and Discussion

In this section, we will address the decisions made when selecting what technology to use for the solution.

All the technologies explained previously, have the basic functionalities typically required in a stream processing engine but our solution requires additional features. And since we need to choose one, it makes sense that it should be the one that better approximates our requirements.

First, the technology needs to be fully available to the public, and not something private to a specific company. For example, Google Milwheel is something that was created by Google for Google and so only they have access/use to the system.

Second, the technology must be able to execute in various types of infrastructures and not be restricted mostly to one or two. For example, Amazon Kinesis Streams is mostly supposed to be used in the AWS infrastructure with their technologies.

Third, we must be able to extract metrics from the system at any point in time about the performance and the status of the resources in use by the system. For example, such metrics may include CPU load, memory usage, resource management, etc.

And finally, we must be able to adapt the system at runtime, through some mechanism that said system provides to us. This may be, by a REST API for example or through an interface in the programming model.

Besides these criteria, we want to extend the use of a system that is widely used nowadays and that has a good community backing it. For these reasons, Apache Flink was chosen. This system is also one of the best performant one in the list [29].

Chapter 3

SDD4Streaming

SDD4Streaming acts as an extension of a Stream Processing Engine focusing on improving its scalability and overall performance. We seek to accomplish these improvements while trying to minimize the loss of output accuracy usually inherent when changing during runtime a complex stateful system.

We divided this chapter as follows. We start by covering the use case of our system in Section 3.1, where we will introduce some of our broader architectural decisions. Next, we will move to Section 3.2 where we will explain the SDD4Streaming data structure model. Following that, we will look into one of the most critical parts of our architecture in Section 3.3, it being the algorithms and mechanisms used for managing the system resources.

3.1 Architecture

As said before, Stream Processing involves processing a variable volume of time-sensitive data and with this, the system needs to be prepared to handle the issues coming from it. We can take as guaranteed from the underlying system many things such as reliability and low-level resource management (at the task level) and so we do not need to further address theses specific aspects.

Since Flink already handles these issues for us we don't need to go as low level for resource usage and altering the system as we would need otherwise, meaning that we will adapt the execution of jobs overall by changing the level of parallelism used. As on these systems, multiple jobs can be executed at the same time, each taking a percentage of the total resources and that the number of resources needed to process the input changes through time we have two ways to handle this.

When creating a job we can define the level of parallelism we want which changes how Flink handles tasks (transformations/operators, data sources, and sinks). The higher parallelism we have, the higher amount of data that can be processed at a time on a job but also creating an overhead on overall memory usage due to more data needing to be stored for savepoints/checkpoints.

Before we can act on the system we need the client application to provide us with some information that we will base our decisions on, and this will be our Service Level Agreement (SLA). The SLA is comprised of:

- Max Number of Task Slots: What is the maximum amount of parallelism or Task Slots allowed for the job in order to avoid a job scaling up indefinitely;
- Resource Usage: What is the maximum resource usage allowed in the system;
- Input Coverage: What is the minimum amount of inputs that should be processed.

This SLA will allow us to make decisions according to the type of performance the client wants the system to have. This is essential because every client has its idea of how the system should behave which we as an external element do not know of. This is the basis of our Resource Management component that on these values will check against the metrics obtained from the system and decide what to do in order to improve performance and scalability.

To try and mitigate overhead caused by our solution, we decided to separate responsibilities into the two following parts:

- SDD4Streaming Library: Responsible for communicating with our server and overriding Flink operator functions;
- Resource Management Server: Responsible for handling all metric related information as well as deciding the state of known jobs and their adaptation;

Like this, we can make the system where the client application is running, use its resources in a more efficient manner being focused solely on the computation it was designed to do. Most of our work is structured on the server where our major features are located.

On the two parts explained above we can further divide it into the following components:

- SDD4Streaming Library:
  - Middleware: Responsible for extending the Flink programming model in order to override the operator functions;
- Resource Management Server:
  - Metric Manager: Component responsible for handling all metric related information, from fetching it from Flink to storing it in our data structure for later use;
  - Resource Manager: Component responsible for making decisions based on the state of system through the use of the stored metrics and altering the system based on this;

In Figure 3.1 we have the relation between the client application and the two components above.

Now that we talked about the major issue we tackle with SSD4Streaming, we will now explain how the architecture behaves to accomplish this.

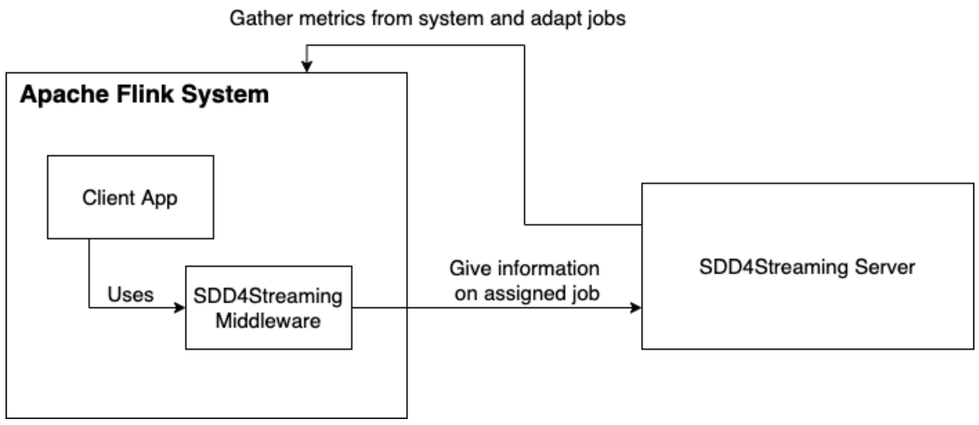


Figure 3.1: Relation between client application and our components

3.1.1 Extending Apache Flink

SDD4Streaming can gather information on the system and alter its execution on runtime by extending its existing functionalities. This mainly revolves around acting as a middleware for each operation available as well as using a REST API that gives us a way to gather information on the system and alter it.

One thing to note is that for our solution to work as expected it needs the user to specific (intended to be simple) modifications to its application. These will involve using our components instead of the ones Flink provides as well as provide us the initialization data with information about the overall system the client created. This will be explained in further detail in the Implementation section.

For the middleware component of the solution, we can create our own versions of the operators Flink provides so we can override their functionality. We do this so we can verify how the system is behaving and act upon it and so seamlessly handle resource management without us having to make any extra communication with the client application.

Besides this, we also have internal metric management with the information obtained from an API Flink provides. This API not only gives us metrics about each component running in the system (e.g. Jobs, Tasks, etc), it also allows us to adapt said system. With this, we can create an internal data structure that allows us to make decisions, and then we can also make actions on the system with the same API.

3.1.2 Metric Manager

SDD4Streaming's decisions and actions are based on information we gather from the system through our Metric Manager. This component is responsible for getting metrics on the current system from Flink and organize it according to our data structure.

Flink allows its clients to fetch system metrics through various means but for our use, we find that the REST API provided is the best way to achieve this. This API gives us details on every level and components of the system without us having to do extra work to identify these. Another way of getting this information from the system is through the Java Management Extensions (JMX) but for this, to work we need to know the port for each component running and there is no easy way to find out programmatically so we decided to avoid this solution.

For the API, we mapped internally each of the endpoints we find useful to have that give us the information we need. These are ones corresponding to the components the system has running at any point in time. We are able to gather information from the highest level being the job itself to the lowest one being the sub-tasks generated by Flink from the operators the client is using.

3.1.3 Resource Management

For SSD4Streaming, the Resource Management component is the most important one and where all the decisions on the system will happen. This component will make use of the metrics we get from our Metric Manager and make decisions depending on the system being compliant with the SLA or not.

To avoid performance issues we want the system to make the best of the resources it has available. So whenever needed, we will be making changes to the system to accommodate the ever-changing needs for processing the incoming data. We can affect the system in two different ways. Either by rescaling job we're assigned to or by suppressing inputs for the sake of performance at the cost of result accuracy.

Algorithm 1: Decision Algorithm

```
Input: taskInfo
Output: shouldProcessInput
1  if !JobGettingRescaled(taskInfo) OR !areMetricsAvailable() then
2      return true
3  if isJobDegraded(taskInfo) then
4      if shouldUpScaleJob() then
5          upscaleJob()
6          return true
7      if shouldSuppressInput() then
8          return false
9  if shouldDownScaleJob() then
10     downscaleJob()
11 return true
```

In Algorithm 1 above we have the pseudo-code of how we check the job related metrics and how we adapt the system accordingly. So for every input we receive, we will analyze the state of the job. First, we ask the Metric Manager, what are the current metrics for the job. If there are no metrics available or the job is getting rescaled we will return true (line 1) since there is nothing we can do at that point. If there are metrics currently available and the job is not getting rescaled then we will check the state of the job and if it is compliant with the defined SLA.

With this, we are able to make the decision of simply passing the control back to the user code and processing the input or the need to act upon the system first. If the system is running smoothly, before we pass control to the user code we will check if we can downscale the job (lines 9-10). If the system is running abnormally we need to change something but before deciding to do so, we need first check what needs to be changed exactly. We have three possible actions at this point depending on the decision made:

- Process Input
- Suppress Input
- Rescale Job

If we have enough resources available in the system we are able to simply rescale the job (lines 4-6). This will help solve bottlenecks and decrease the load on each task and so help reduce performance degradation. But we only rescale the current job if no other rescaling operation is happening on the job. If we do not have enough resources to do the operation then we need to follow a different approach. Our other approach is suppressing the input partially and so not processing it (lines 7-8). This will decrease the load and help reduce performance degradation. But this comes with the cost of reducing the accuracy of the output data which is why we have a rule for it in our SLA, where the user can declare what is the minimum accuracy (i.e the percentage of the input subject to processing/reflected in the output) required at all times. Lastly, if all of our other approaches are not possible, we will have to pass the control to the user code and allow him to process the input as normal (line 12). Even though this will make increase the load in the system, there is nothing more we can do without breaking our SLA with the user.

3.2 Data Structures

SDD4Streaming has two major sets of data structures necessary for its execution. These consist of data, the client application provides us about the overall system we will be executing in, as well as metrics we fetch from said system.

3.2.1 Initialization data

For us to do anything at all in the system we first need some initialization data, which comes directly from the client application using our solution. These will give us the means to query the system about its resources as well as allow us to dynamically change it. This structure has the following elements:

- Service Level Agreement: Details the optimal performance the clients wants the system to have;
- Job Name: Used to identify a running job;
- Server Base URL: The base url for where our webserver is running;
- Client Base URL: The base url for where the Job Manager the job is gonna be executed;
- Jar Name: Name of the jar used to create the Job;

These elements are required by our library in order to communicate with the server as well as for the latter to communicate with the Flink system for data gathering and job adaptation.

3.2.2 System Metrics

Apache Flink provides an extensive REST API that we can use to query or modify the system in various ways. We make use of this API to fetch metrics about the resources being used in the system.

To accomplish this we had to map the endpoints we find necessary from the API, being the data we have to send and receive for each one. On some elements unfortunately its not so simple and we can't directly find out the relation between components and for this we need to gather this information ourselves.

A job in Apache Flink has multiple levels and we are able to gather different information on each one of them. The levels important for our solution are as follows:

- JobManager: Orchestrator of a Flink Cluster;
- Job: The runtime representation of a Logical Graph [1] (also often called dataflow graph);
- TaskManager: Supervises all tasks assigned to it and interchanges data between then;
- Task: Node of a Physical Graph [2]. Its the basic unit of work.
- SubTask: A Task responsible for processing a partition of the data stream;

It's good to have a notion of all these levels even if we don't certain properties of some of them since we will need to go over all of them to gather all the metrics we require to function.

For our solution, we will use and store data about the JobManager, TaskManagers and the tasks still running from the known jobs. While on Flink's API these elements are not directly related to each other, we connect them in our data structure so we can easily check all elements required of the job in order to make decisions.

Now talking about what kind of data we can get from these elements, Flink through its API provides a good degree of information with different representations. We can gather information either on a collection of items (e.g. the metrics for all task managers in the system) or for a specific element which we then use to store in our internal structure.

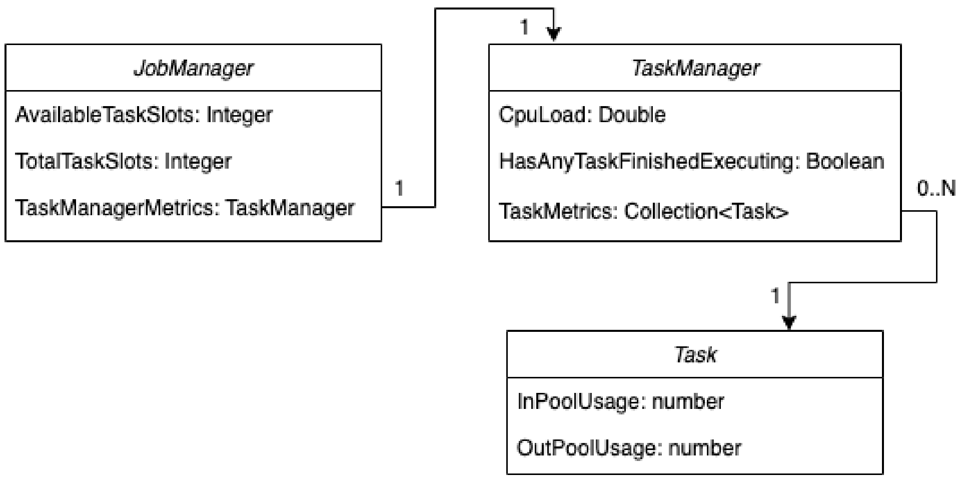
Our data structure will be comprised of these elements with the following relation JobManager $<-->$ TaskManager $<-->$ Task.

As shown in Figure 3.2, we see each structure we have and its relation with each other. For the JobManager, we will store the available and total amount of task slots it has available. This is necessary to know if we can scale a job up or not, because on Flink whenever a job does not have enough slots available for its parallelism level it will stay waiting until any slot frees up to achieve the level required.

We then have the TaskManager which will store the CPU load on the system. This load represents the average load between all the TaskManagers the job is affecting, meaning all the ones where tasks are being executing.

Finally we have Task which the TaskManager will store a collection of and each one will have the buffer usage for input and output which are used to identify backpressure issues (e.g. possible bottlenecks) [3].

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/glossary.html#logical-graph
[2] https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/glossary.html#physical-graph
[3] https://flink.apache.org/2019/07/23/flink-network-stack-2.html#backpressure



JobManager
AvailableTaskSlots: Integer
TotalTaskSlots: Integer
TaskManagerMetrics: TaskManager
1
1
TaskManager
CpuLoad: Double
HasAnyTaskFinishedExecuting: Boolean
TaskMetrics: Collection<Task>
0..N
1
Task
InPoolUsage: number
OutPoolUsage: number

Figure 3.2: Metric Data Structure

3.3 Summary

In this chapter, we presented a technical and architectural overview of SDD4Streaming. We started by giving a brief overview of what we are seeking to accomplish as well as our broader architectural decisions in Section 3.1. We looked at how parallelism affects jobs in Apache Flink, how we extended the system to handle resource management seamlessly and how we handle resource management. In Section 3.2, we introduced our data structures, what information we require from the client as well as what we store and use for each internal Flink components.

Chapter 4

Implementation

For our SDD4Streaming implementation [1], we try and take advantage of all the mechanisms Flink provides us while trying to abstract details to the clients in order to increase ease of use without having to change much of already existing applications.

Our solution consists of a library that the clients imports into its application as well as a web server. In Section 4.1 we explain how the library extends the Flink programming model and the requirements the clients need to comply with to use it. After, we explain how the webserver accomplishes resource and metric management and how the library communicates with it in Section 4.2. Finally, we provide an example application using our solution in Section 4.3 to show how the library is used.

4.1 SDD4Streaming Library

The library was made in Java, one of the two languages supported by Flink, the other one being Scala but since this one isn't used as much we decided to implement the solution using Java.

As we said in the previous part, our solution is available as a library that can be imported through the normal package managers used for Java namely Maven and Gradle. But before this can be done, we have a few restrictions that need to be complied with to have the solution as a dependency.

In the following sections we will explain our dependencies on Section 4.1.1, what we require for initialization from the clients and why in 4.1.2, and how our middleware is structured in 4.1.3.

4.1.1 Dependencies

Flink only supports Java 8 and 11, so in order to allow for compatibility with most applications the clients have while having the great features that have been introduced to the language such as Lambdas and Streams we chose to use Java 8. One other great thing about using this version is avoiding issues with build tools that come with the newer versions as well as some other licensing issues due to some changes Oracle has made a few years back.

[1] https://github.com/PsychoSnake/SDD4Streaming

Besides this, we are also restricting the Apache Flink version to 1.9.0 due to what our solution needs to handle. There was a major change involving one of our major features, being the dynamic rescaling of jobs. Flink decided to completely block the use of the CLI and the REST API for rescaling jobs [2]. This is due to it being an experimental feature and they wanted to make a major change to their internal scheduling mechanism.

Besides Apache Flink, we also have a few other dependencies such as okhttp3 and retrofit2 due to the need for us making HTTP requests to our web server. It was decided to use these libraries because of the abstractions they give us in defining and making our requests. We also use Jackson which is a high-performance JSON processor for Java, needed to convert the responses we get into Java objects we can use.

Besides the restrictions on the Java and Flink versions we did not do any kind of restriction to the other libraries and chose to use their latest stable version that is supported by Java 8.

In Listing 4.1 we can see all the dependencies we use and the corresponding versions for each in Gradle format:

Listing 4.1: Dependencies

```
ext {
    javaVersion = '1.8'
    flinkVersion = '1.9.0'
    scalaBinaryVersion = '2.11'
}

implementation "org.apache.flink:flink-java:${flinkVersion}"
implementation "org.apache.flink:flink-streaming-java_${scalaBinaryVersion}:${flinkVersion
    }"
implementation "org.apache.flink:flink-clients_${scalaBinaryVersion}:${flinkVersion}"

implementation 'com.google.guava:guava:29.0-jre'

implementation 'com.squareup.okhttp3:okhttp:3.11.0'
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
implementation 'com.squareup.okhttp3:logging-interceptor:3.8.0'
implementation 'com.squareup.retrofit2:converter-jackson:2.7.2'
implementation 'com.fasterxml.jackson.core:jackson-databind:2.11.0'
```

4.1.2 Library Configuration

For our internal components to work as expected we require the client application to send over the initialization data defined in Section 3.2.1. With this data we are able to create the HTTP client which will be explained in Section 4.1.4. With this client we tell the server that a new job is about to be created while passing the SLA and configuration data necessary to communicate with Flink.

[2]http://apache-flink-user-mailing-list-archive.2336050.n4.nabble.com/DISCUSS-Temporarily-remove-support-for-job-rescaling-via-CLI-action-quot-modify-quot-td27447.html

We have created an exception type of our own in order to tell the clients whenever something went wrong. This is used for the scenarios where the initialization parameters were not all passed and if communication with the server has failed.

4.1.3 Middleware

SDD4Streaming intercepts the Flink operator's execution to accomplish our goals. We do this by providing our own version of these operators so we can override their behavior and add our logic to them. This is our way of abstracting to the clients, and at the same time, for us to interact with the underlying system.

Flink provides various operators from affecting a certain data set such as FlatMap, Reduce to affecting a window as a whole such as WindowFilter, WindowReduce. When creating these operators, the user needs to pass a function that will handle all the processing for the said operator. Due to the simplicity of this architecture, we can create an adapted version of each operator, extending the corresponding class we're overriding and adding our logic to the processing method.

These new classes created by us will receive the original processing function from the clients for how they want to process the data so we can call that whenever we think data can be processed.

The way the clients can create our versions of these operators is by using a class we provide in the SddStreaming instance created. It works like this because we need to pass to each operator created data needed to communicate with our web server.

When our operators are executing we will make a request to the server asking if the input can be processed while passing the job name so we can identify which metrics to use. After, we will either call the client function or not depending on the response we get.

On Listing 4.2 we have an example of an operator we adapted.

Listing 4.2: Operator Adaptation Example

```java
public class SddFlatMapFunction<T, U> implements FlatMapFunction<T, U> {
  private final FlinkRestApiHandler flinkClient;
  private final FlatMapFunction<T, U> clientFunction;

  public SddFlatMapFunction(FlinkRestApiHandler flinkClient, FlatMapFunction<T, U>
      clientFunction) {
    this.flinkClient = flinkClient;
    this.clientFunction = clientFunction;
  }

  @Override
  public void flatMap(T value, Collector<U> out) throws Exception {
    boolean shouldProcessInput = this.flinkClient.shouldProcessInput();

    if (shouldProcessInput) {
      this.clientFunction.flatMap(value, out);
    }
  }
}
```

4.1.4 Request Handling

As said before we will be making requests to the API our server instance provides. This way we can give the initialization data to it as well as notify and check if input should be processed or not.

To use this API first we had to create a structure representing each of the endpoints we need and the corresponding request and response types. This was done using a combination of okhttp, retrofit and Jackson. With retrofit we're able to declare requests by creating an interface where each method will directly correlate to one endpoint.

Each of these methods will have annotations that represent the HTTP Method used and its path (e.g. @GET("/should-process/jobname")) and the Headers sent in the request (e.g. @Headers("accept: application/json")). Besides this we're also able to specify values to be replaced in the path, add query parameters and request body if needed through the arguments set in the method itself. Finally we can define what type of response body we'll get that is the same as the return value of each method. In Listing 4.3 we have an example of a request definition.

Listing 4.3: Request Definition Example

```
@GET("/should-process/{jobname}")
@Headers({"accept: application/json"})
CompletableFuture<OperationResponseBody> shouldProcessInput(@Path("jobname") String
    jobName);
```

So we've talked about how our requests are declared, now we're gonna explain how we handle the transformation of Java objects to JSON and the vice-versa. Since the data we need to send to Flink and the ones it provides comes as a JSON format we decided to use Jackson which is a very well known JSON processor for Java. To make this happen we need to create classes where for each property we'll add an annotation representing its equivalent in the JSON format (e.g. @JsonProperty("jobName")).

With these classes, Jackson can serialize/deserialize the data to be sent/received in each request and we can simply use them in the methods declared using retrofit. This works seamlessly because retrofit allows us to give it a converter to use which one already exists for Jackson. In Listing 4.4 we have an example of a JSON body and its equivalent in Java class format.

Listing 4.4: Java Class with Jackson annotations and JSON equivalent

```
{
    "clientBaseUrl": ...,
    "jobName": ...,
}

public class NewJobRequestBody {
    @JsonProperty("clientBaseUrl")
    public final String clientBaseUrl;
    @JsonProperty("jobName")
    public final String jobName;

    ...
}
```

Finally after talking about how to declare methods and how data is serialized, we can explain how the requests are made. For this indirectly use okhttp which is the library used by retrofit for HTTP requests. First we need to initialize retrofit with a base URL that we get from the clients, for the Flink cluster and provide it with a Converter Factory that handles the request and response body conversion from and to JSON.

After initializing the service we can create an HTTP client using the interface we explained above. With the instance we get from this we're already able to make requests just by calling the method corresponding to the endpoint we want. These methods return to us a generic CompletableFuture instance that allows us to make requests asynchronously. In Listing 4.5 we have an example of generating a Retrofit instance and making a request.

Listing 4.5: Flink client creation

```
Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://localhost:8081")
                .addConverterFactory(JacksonConverterFactory.create())
                .build();

FlinkRestClient flinkClient = retrofit.create(FlinkRestClient.class);

// This generates the request to be made
CompletableFuture<OperationResponseBody> response = this.flinkClient.shouldProcessInput(
    this.jobName);

// This makes the actual request
OperationResponseBody responseBody = responseBody = response.get();
```

4.2 SDD4Streaming Web Server

The bulk of our solution is in the server where we fetch the metrics for each known job, as well as decide if any of them should be adapted to the current load.

This server was created using Node.JS due to its ability to run in practically any machine and operating system as well as the low memory requirements when compared to Java for example. We decided to use Typescript, an extension of Javascript, that adds static typing which allows for the project to be easier to understand and maintain over a long time.

4.2.1 Initialization and Dependencies

In order for the server to be executed we require that the client uses specific versions of Node.JS and NPM which need to be ”^12.16.3” and ”6.14.4” respectively.

Besides this the only other thing we require is an environment variable for the port to be used for the server which is named ”PORT”. If this variable is not set we will default to port 3000.

When the server is started we start the MetricManager which will start fetching metrics for all known jobs, if there is any. Until we get a request to indicate that a new job is getting created, the server will mostly remain idle since we don't have any information of jobs and the Flink system.

When we get the request for a new job, we will store that job in our internal structure and send over a 200 response back to the library. In Listing 4.6 we have the type for the body of the request as well as the code for the route.

Listing 4.6: New Job request example

```
interface NewJobRequestBody {
    serviceLevelAgreement: ServiceLevelAgreement;
    jobName: string;
    clientBaseUrl: string;
    jarName: string;
}

app.post('/new-job', (req: express.Request, resp: express.Response) => {
    const jobInfo = req.body as NewJobRequestBody;
    SddStreaming.setJobInfo(jobInfo.jobName, jobInfo);
    resp.status(200).end();
});
```

Besides this request, the server is designed to receiver another one which our library in the Flink application will call for each operator in order to know if the input should be processed or not. In Listing 4.7 we have the code for this request in the server.

Listing 4.7: Operation request example

```
app.get('/should-process/:jobname', async (req: express.Request, resp: express.Response)
    => {
    const shouldProcessInput = await DecisionManager.shouldProcessInput(req.params.jobname
        );
    resp.status(200).json({ shouldProcessInput }).end();
});
```

4.2.2 Metric Manager

The Metric Manager is a component that runs independently of the other components in the server. When the server is started, it will tell the Metric Manager to initiate fetching metrics. To not overload the system with requests by fetching metrics per each input we get, we use an interval to know when to get metrics again.

Since we know that by default Flink updates the metrics it provides in the REST API at least in 10 second intervals, we based our interval on this. But due to our decision that we would only start the timeout after we finished fetching the current metrics we lowered the interval a bit to accommodate the processing time it will take and so we use 8 seconds.

When a new update to the metrics happens, we need to check if the job is still running because if it is not we should stop the update for that job and remove it from our internal structure. If the job is still running then we will gather the metrics for all the components we require, namely the JobManager, TaskManager, and Tasks.

For the JobManager is straightforward and we just need to make a request to its endpoint and get the metric values. But for the TaskManager and Tasks it gets a bit more complicated. A Flink System can have multiple TaskManagers and this can cause the parallel sub-tasks of a job be running on different managers.

In order to solve this we get the information about all sub-tasks executing on the Job and checking which manager is responsible for it. This way we can check how much of the TaskManager is the Job actually affecting and so the CPU load for each manager will be $cpuLoad*(numberOfSubTasksUsed/totalNumberOfSubTasks)$. By summing the values for each of the TaskManagers gotten from the expression we get the average of the CPU used.

Finally for the tasks we first get an overview of the job in order to check which ones are still running. If any of the tasks is not running then we will ignore it and set the flag on our TaskManager structure that at least one task has finished running. For the tasks in contrary to the TaskManagers we will have a collection instead of getting the average of all them combined.

All the metrics explained here are exposed through a public method so our Resource Manager can get the current metrics for a job.

4.2.3 Resource Management

Finally, we're gonna explain how our Resource Management component works, how we check how the system is handling the current load, and how we adapt it.

This component is executed whenever we get a request from the execution of an operator in our library. First of all we check if there are metrics available for the job as well as if the initialization data was provided. If these are missing then we will respond right away that the input should be processed since we don't have enough data to infer on the system. Also another condition to check the state of the job is if it is being rescaled or not, and if it is then we will simply return true.

If we have all the required data then we shall proceed to check how the system is behaving and compliant with the SLA we got. We first check if the job is degraded, meaning if the current cpu load is higher than the one specified in the SLA and if there is any bottleneck in the running tasks.

If the job is running smoothly and there is no degradation then we further check if we should down scale the job in order to free task slots for other jobs that may need it. But if we can't do the down scale, we will tell the library that the input should be processed.

If the job is actually degraded, then we have three options. We will first check if we can up scale the job in order to try and avoid bottlenecks in the tasks but can only do it if there are task slots available. If we can't up scale then we will check if we can ignore the input in order to improve performance while being compliant with the SLA. Finally if none of the other two options are possible as a last option we will need to tell the library to still process the input even though the system is currently not running optimally.

Due to the limitations of Flink in current versions, we can't call an endpoint just to do a rescaling of a job since they have disabled this functionality from v1.8.0, and so we will need to instead handle with savepoints in order to adapt the job. To do a scale up or down, we need information that only the client can give us and that's the name of the jar stored in the system that was used to initially create the job.

First, we will need to create a savepoint for the job while saying that the job should be stopped by sending a request to the corresponding endpoint. One thing to note is that we cannot create a savepoint if at least one of the tasks has finished running, and so we won't even try to rescale the job if this happens.

After this, we need to use the response we got from the previous request to check the status of the savepoint operation. When it completes we will get the location for said savepoint. This is important so we can create a job that should start from the savepoint created.

But before we can start the job we first need to get the id of the jar that has the contents of the job previously running.

Finally with the savepoint location and jar id gotten from the previous steps as well as the new parallelism level received we can start a new job.

In Listing 4.8 we have the code responsible for creating a savepoint.

Listing 4.8: Code for Creating a Savepoint

```
public static async rescaleJob(jobInfo: JobInfo, newParallelismLevel: number): Promise<
    boolean> {
        if (!jobInfo || !jobInfo.jobId || !jobInfo.jarName) {
            return false;
        }
        const triggerId = await RequestManager.createSavepoint(jobInfo.jobId, true);
        const savepointStateResponse = await RequestManager.getSavepointOperationState(
            jobInfo.jobId, triggerId);

        if (savepointStateResponse.status.id !== 'COMPLETED' || !savepointStateResponse.
            operation.location) {
            return false;
        }

        const savepointLocation = savepointStateResponse.operation.location;
        const jobJar = await RequestManager.getJar(jobInfo.jarName);
        if (!jobJar) {
            return false;
        }

        await RequestManager.startJobFromJar(jobJar.id, savepointLocation,
            newParallelismLevel);
        return true;
    }
```

4.3 Example Application

In Listing 4.9 we have an example of a simple application that counts each word given on a socket, that incorporates our solution.

Listing 4.9: Example Application with SDD4Streaming

```java
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment
    ();
SddStreaming streamingOptimizer = new SddStreaming(new ServiceLevelAgreement(0.6, 0.5,
    0.8),"http://localhost:3000", "http://localhost:8081", JOB_NAME, JAR_NAME);
OperatorCreator oc = streamingOptimizer.getOperatorCreator();

...

DataStreamSource<String> socket = env.socketTextStream("localhost", 9000);

text.uid("Source-UID")
  .flatMap(oc.createFlatMapOperator((FlatMapFunction<String, Tuple2<String, Integer>>) (
      value, out) -> {
    String[] splits = value.toLowerCase().split("\\W+");
    for (String split : splits) {
      if (split.length() > 0) {
        out.collect(new Tuple2<>(split, 1));
      }
    }
  })).uid("FlatMap-UID")
  .returns(TupleTypeInfo.getBasicAndBasicValueTupleTypeInfo(String.class, Integer.class))
  .keyBy(0)
  .reduce((ReduceFunction<Tuple2<String, Integer>>) (value1, value2) -> new Tuple2<>(
      value1.f0, value1.f1 + value2.f1)).uid("Reduce-UID")
  .print();

env.execute(JOB_NAME);
```

Chapter 5

Evaluation

To evaluate our system, we want to look at its core focus, the increase in performance while decreasing possible bottlenecks and a dynamic resource usage depending on the needs of the system at any point in time. Our tests will be based on how applications behave with our solution and compare them with just using what Flink provides to see how it much improvement it gives.

We start our evaluation by looking into the workloads used in Section 5.1. In Section 5.2, we document the dataset we used as well as the transformations necessary to make this data viable. We then move on to an analysis of the metrics we intended to gather in Section 5.3. After this we look at how we configured the testbed in Section 5.4. Finally are now ready to look into the results of the testing in Section 5.5.

5.1 Workload

For the workload we have an application that is able to demonstrate how our solution behaves in a application. This workload involves sending a variable volume of data to the job and check how our solution will scale the job and the overall performance of the system.

The producer of data will be Kafka [1], an open-source stream-processing software platform developed by the Apache Software Foundation that aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds.

Initially we will prepare Kafka with a big volume of data which when the application starts will read from. Since the volume of data is so high, after the application finishes processing it we will wait for a bit and check if our solution downscales the job since its load at the time will be very low. Finally after we finish waiting we will again send over a large volume of data to Kafka and see if our solution is able to adapt the job to support the new load The data used for testing will be explained in Section 5.2.

[1] https://kafka.apache.org/

5.2 Dataset

For the workload explained, we will use a dataset provided by the Univerty of Illinois System. This dataset represents the taxi trips (116GB of data) and fare data (75GB of data) for the year 2010 to 2013 in New York [2]. We don't actually need to use all this amount of data for our purposes so we decided to use a subset of year 2010.

5.2.1 Filtering and Data Cleanup

The dataset for the taxi rides/fares is pretty extensive coming in at a total of 191GB which for our purposes we don't need to use everything in order to cause a heavy load on the system. For this reason, we decided to use the latest available data which is for the year 2013 but before this can be used by the application we first need to do a bit of cleanup.

We need to look at the data and remove rows that are missing essential data since these will provide nothing for our results while also mapping the columns necessary for our execution or not.

5.3 Metrics

For each execution, we look to extract two key groups of data: system performance and overhead caused by our solution. The following list describes these in more detail:

System Performance:

- Resource Utilization: This metric assesses whether or not the solution is scaling the system accordingly. The resources used by tasks scale to keep up with the input rate;
- Latency: If the input is taking too long to be processed;
- Throughput: How much data is being processed per period of time;
- Accuracy: Observe how the accuracy of the applications varies over time.
- CPU Usage: Check percentage of CPU being used by tasks in the cluster as well as CPU reserved but not used(assess resource waste and costs).

Solution Overhead:

- CPU Load: This metric assesses how much of the CPU is affected by the execution of our solution;
- Memory Load: This metric assesses how much of the memory is affected by storing our data structures by our solution.

[2]https://uofi.app.box.com/v/NYCtaxidata

5.4 Testbed configuration

We designed our test runs to be executed in managed infrastructure (commonly known as cloud services). The Cloud Service used for this was the Google Cloud Platform (GCP). This service provides 300 credits in a free trial per account which for our use case is enough for us to do the necessary tests.

Our setup consisted of 3 VMs each with two vCPUs, 4 GiB of RAM and 20 GiB of storage. Each of these machines will be responsible for each part necessary for testing. One will host the Flink cluster where the job will run, the other will host the data that the job will read from and the third one will host our web server.

Besides this we also needed a way to gather the metrics from the system while our jobs were running. To accomplish this we decided to use a Metric Reporter [3], in specific the one chosen was Prometheus [4].

To give a brief overview of what this tool is. Prometheus is a time series database (TSDB) combined with a monitoring and alerting toolkit [Prometheus 2020]. The TSDB of Prometheus is a non-relational database optimized for storing and querying of metrics as time series and therefore can be classified as NoSQL. Prometheus mainly uses the pull method where every application that should be monitored has to expose a metrics endpoint in the form of an REST API either by the application itself or by a metrics exporter application running alongside the monitored application. The monitoring toolkit then pulls the metrics data from those endpoints in a specified interval.

This tool will be executed in our personal PCs as to not use more credits on GCP than we need to.

5.5 Results

For both tests Apache Flink was configured to have one JobManager and one TaskManager. The JobManager was configured to have available 1024 MiB while the TaskManager which is responsible for managing all the tasks, the units of work, has double that amount at 2048 MiB. The amount of available tasks are 50 and each of the tests we will start with a parallel level of 20, so using 20 of the 50 total slots

First we will go over the metrics we got from the test where we have the application running without our solution being used. All figures below for this test belong to the same time interval and have a duration of 23 minutes.

[3]https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/metrics.html#reporter
[4]https://prometheus.io/

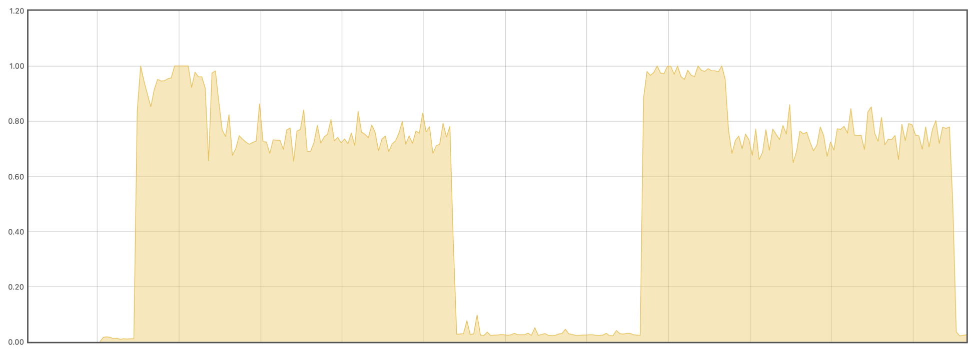


Figure 5.1: CPU Load on the TaskManager

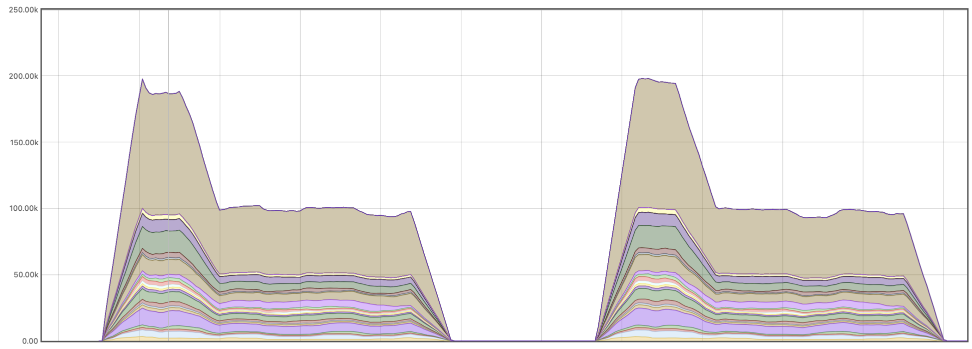


Figure 5.2: Amount of records getting processed by each sub-task per second (throughput)

We can see the CPU usage in Figure 5.1 for the workload used, that is pretty high and besides the first minutes where data is getting fetched from Kafka, the load is mostly constant overal with some spikes now and then. For throughput in Figure 5.2 we see something similar to the CPU load graph which makes sense because higher throughput means that we're processing more data and for that to happen higher CPU load is expected. So we see an initial very high throughput that then decreases after a few minutes but remains constant.

Also when comparing the first volume of data sent and the second one, we can see that the CPU load and throughput are fairly similar.

We can also see memory usage by the TaskManager/Tasks in the Heap and Non-Heap in Figures 5.3 and 5.4 respectively.

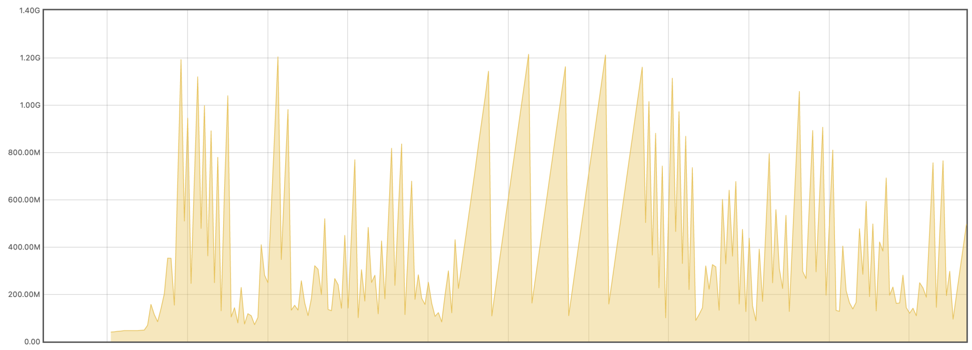


Figure 5.3: Amount of used Heap Memory

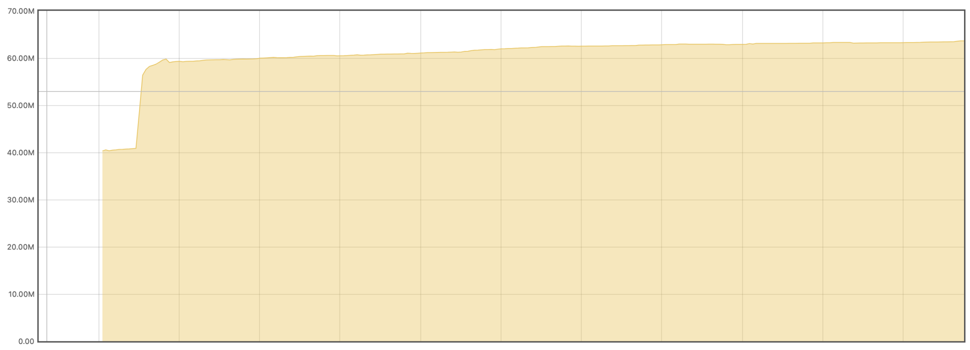


Figure 5.4: Amount of used Non-Heap Memory

Now we will show the results from the test where we have the application running incorporated with our solution. The configurations are the same as the other test but we have the extra configuration of SDD4Streaming. The important part needed before showing the results is the SLA used for this test:

- Max Number of Task Slots: 22;
- Resource Usage: 50%
- Input Coverage: 80%;

All figures below for this test belong to the same time interval and have a duration of 30 minutes.

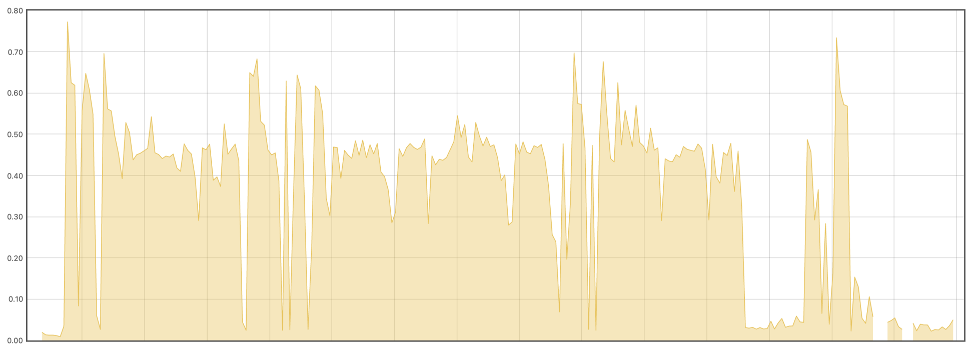


Figure 5.5: CPU Load on the TaskManager

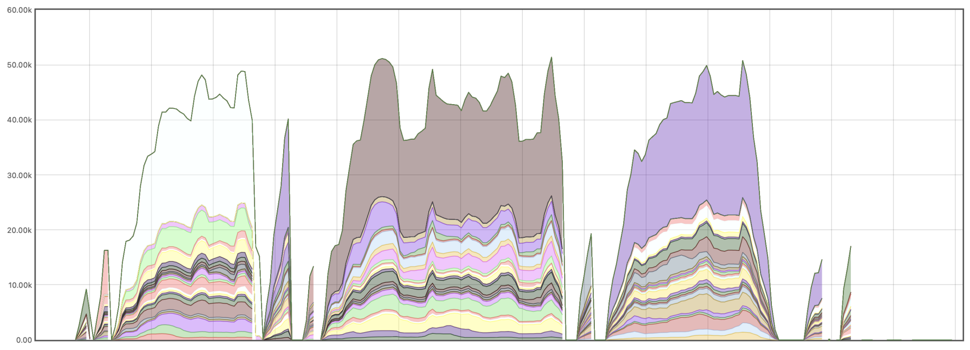


Figure 5.6: Amount of records getting processed by each sub-task per second (throughput)

Here we have the CPU load and throughput in Figures 5.5 and 5.6 respectively. By looking at these graphs we can see that the execution was very different from the one without our solution. For example we can see drops in both of them that mostly represent when the job was getting rescaled since at that point no input will be processed and so throughput will drop to 0 and CPU will be mostly used by the TaskManager that is adapting the job.

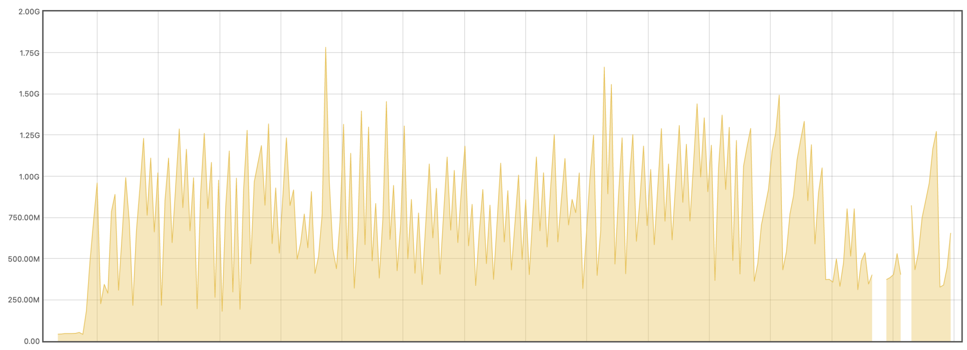


Figure 5.7: Amount of used Heap Memory

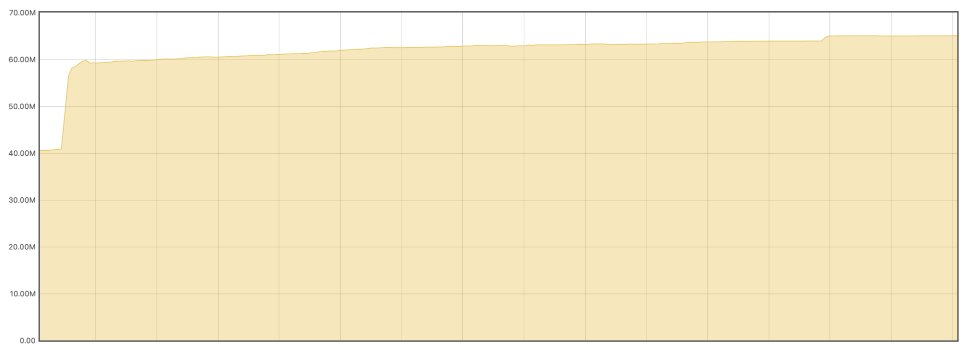


Figure 5.8: Amount of used Non-Heap Memory

In Figures 5.7 and 5.8 we have the memory usage for the Heap and Non-Heap respectively. From this we see that the Non-Heap is very similar to the previous test but for the Heap we are getting quite a difference. Since our solution will adapt the system in runtime, the TaskManager will need to use more memory in order to do the rescaling of the jobs. And due to this we see a higher average use of memory as well as the max amount of memory used overall.

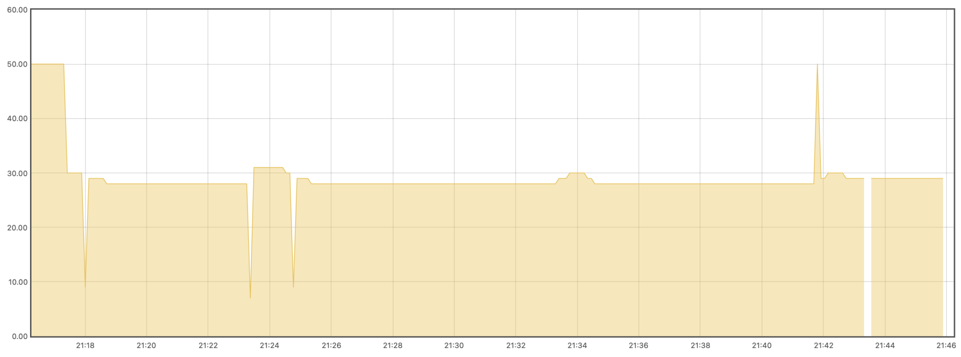


Figure 5.9: Number of Available Task Slots

Finally, specifically for the test with our solution we have in Figure 5.9 the number of available slots throughout the execution of the job.

Chapter 6

Conclusions

We started our work by first analyzing the Streaming Processing paradigm and what defines it. We then outlined its most important components creating a taxonomy and provided some examples of major technologies that implement this paradigm such as Apache Flink, Spark, Google Millwheel as an example. After this, we showed how two solutions tried to fix some of the inherent issues to Streaming while focusing specifically on what they try to solve.

Following the related work, we explained the architecture of our purposed solution, SDD4Streaming. We gave a brief overview of what we are seeking to accomplish while also explaining some of our architectural decisions. We looked at what we expect from the clients such as the SLA for example and how they will use our extensions of the Flink model. We covered our two major components, the MetricHandler and the ResourceManager while the middleware is considered minor since all it does is communicate with the others. We also explained how our data is structured.

Our next step was to cover the actual implementation of the architecture we detailed. We started by talking about the library itself and the details on the language, versions, and dependencies used by it. We then did a deep-dive to go over each of our components, how they were implemented and the decisions we made for each of them. Finally, we showed an example of an application using our solution to show how many changes are needed specifically to use our solution.

Finally, it was time to test our system. To do so, we created an application of our own using a dataset of taxi rides and fares in NewYork in the year 2013. The objective of this application is to test how our solution handles variable volumes of data being sent to the system. In order to accomplish this we used a Cloud Service called Google Cloud Platform where we created three VMs, one to host the Flink cluster, other for hosting Kafka and providing the data the system is gonna use and finally a third VM that hosts our web server.

We did two tests, one with the application alone without our solution and another with our solution incorporated into it. This application would get initially a high volume of data in order to increase the load on the system and see how it behaves. After it finished processing this data we wait for a few minutes and then send over again a high volume of data.

We concluded that our system provides can become an alternative to the current solutions by interacting with the system during its execution. We see that it can identify bottlenecks on the system and adapt accordingly and so increase the overall quality of the job. Unfortunately due to limitations by part of Flink, we had to use savepoints to achieve job scaling which can cause performance issues by itself that in turn decreases the overall throughput of a job.

6.1 Future Work

There is, of course, room to do a lot more, not only features but also technical improvements. Here we provide a list of suggestions for improvements and extensions to SDD4Streaming:

- Provide support for other Streaming Engines besides Flink so more applications can use this solution by creating a library specific to that system, as well as create mappings in the server for getting metrics and adapting these other engines;
- Increase Scalability of the web server for when it needs to handle with large amount of jobs;
- Enhance the Resource Management to allow it to instead of just doing rescaling at the Job level, also be able to allocate machines if for example the system is being hosted on the cloud;
- Extend the SLA and the resource management mechanism to allow to check output accuracy besides only looking at input coverage so it can have a better understanding of how much we're affecting the output for the client;

Bibliography

[1] Bin Cheng et al. “Building a big data platform for smart cities: Experience and lessons from santander”. In: *2015 IEEE International Congress on Big Data*. IEEE. 2015, pp. 592–599.

[2] Ralf Tönjes et al. “Real time iot stream processing and large-scale data analytics for smart city applications”. In: *poster session, European Conference on Networks and Communications*. sn. 2014.

[3] Tiago Boldt Sousa. “Dataflow programming concept, languages and applications”. In: *Doctoral Symposium on Informatics Engineering*. Vol. 130. 2012.

[4] Roger S Barga et al. “Consistent streaming through time: A vision for event stream processing”. In: *arXiv preprint cs/0612115* (2006).

[5] Engineer Bainomugisha et al. “A survey on reactive programming”. In: *ACM Computing Surveys (CSUR)* 45.4 (2013), pp. 1–34.

[6] Leonardo Neumeyer et al. “S4: Distributed stream computing platform”. In: *2010 IEEE International Conference on Data Mining Workshops*. IEEE. 2010, pp. 170–177.

[7] Otávio Carvalho, Eduardo Roloff, and Philippe OA Navaux. “A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring”. In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. ACM. 2017, pp. 9–14.

[8] Saurav Haloi. *Apache zookeeper essentials*. Packt Publishing Ltd, 2015.

[9] Paris Carbone et al. “Large-scale data stream processing systems”. In: *Handbook of Big Data Technologies*. Springer, 2017, pp. 219–260.

[10] Buğra Gedik. “Generic windowing support for extensible stream processing systems”. In: *Software: Practice and Experience* 44.9 (2014), pp. 1105–1128.

[11] Peter Pietzuch et al. “Network-aware operator placement for stream-processing systems”. In: *22nd International Conference on Data Engineering (ICDE’06)*. IEEE. 2006, pp. 49–49.

[12] Lisa Amini et al. “Adaptive control of extreme-scale stream processing systems”. In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS’06)*. IEEE. 2006, pp. 71–71.

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

[14] Daniel Abadi et al. “Aurora: a data stream management system”. In: *SIGMOD Conference*. Citeseer. 2003, p. 666.

[15] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. “Load management and high availability in the medusa distributed stream processing system”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 929–930.

[16] Mitch Cherniack et al. “Scalable Distributed Stream Processing”. In: *CIDR*. Vol. 3. 2003, pp. 257–268.

[17] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. “The 8 requirements of real-time stream processing”. In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47.

[18] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. “Fault-tolerant stream processing using a distributed, replicated file system”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 574–585.

[19] Paris Carbone et al. “Lightweight asynchronous snapshots for distributed dataflows”. In: *arXiv preprint arXiv:1506.08603* (2015).

[20] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[21] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[22] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[23] Tyler Akidau et al. “MillWheel: Fault-Tolerant Stream Processing at Internet Scale”. In: *Very Large Data Bases*. 2013, pp. 734–746.

[24] Robert Evans. “Apache storm, a hands on tutorial”. In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 2–2.

[25] Sanjeev Kulkarni et al. “Twitter heron: Stream processing at scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250.

[26] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. “SpinStreams: a Static Optimization Tool for Data Stream Processing Applications”. In: (Washington DC, USA). 2017.

[27] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[28] Sérgio Esteves, Helena Galhardas, and Luís Veiga. “Adaptive Execution of Continuous and Data-intensive Workflows with Machine Learning”. In: (Rennes, France, Dec. 10–14, 2018). 2018.

[29] Sanket Chintapalli 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.