SDD4 Streaming

Tiago Mourão Lopes
tiago.mourao@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa

Abstract. With the amount of information nowadays that needs to be processed at a certain time, we need to have resilient systems that are highly scalable and with high throughput. To solve this, stream processing engines were developed. They allow for high throughput in real-time data processing and allow for scaling of operations according to the size of the inputs.
But even with these systems, there can be bottleneck issues where the system isn't able to automatically scale in/out in an elastic way as for example the cloud does, where when we don't have enough power for the amount of work and it creates new machines to handle this requirement.
This work **SSD4 Streaming** tries to solve this by creating an enhanced data structure that besides holding input and intermediate result data, also holds metrics about the current jobs and machines in the cluster. With these enhanced datasets we can make decisions according to the situation and for example, when a bottleneck is detected, tasks can be instructed to split (to engage more resources for a specific operator) or to merge (when the bottleneck situation has ended) to adapt dynamically to workload variations.

Table of Contents

1 Introduction

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

In the Figure 1 we can see a generic representation of how stream processing works from the beginning to the end, from receiving inputs till the generation of outputs from their processing. So stream processing can be characterized by a continuous flow of inputs that can come from various places (e.g. applications, sensors, etc) which will be then processed by an application. This processing operation can involve multiple factors and operators depending on what the user wants to do with the data being handled with. After the data is processed and analyzed it will be outputted to some destination that the user has specified (e.g. data base, server, API, etc).

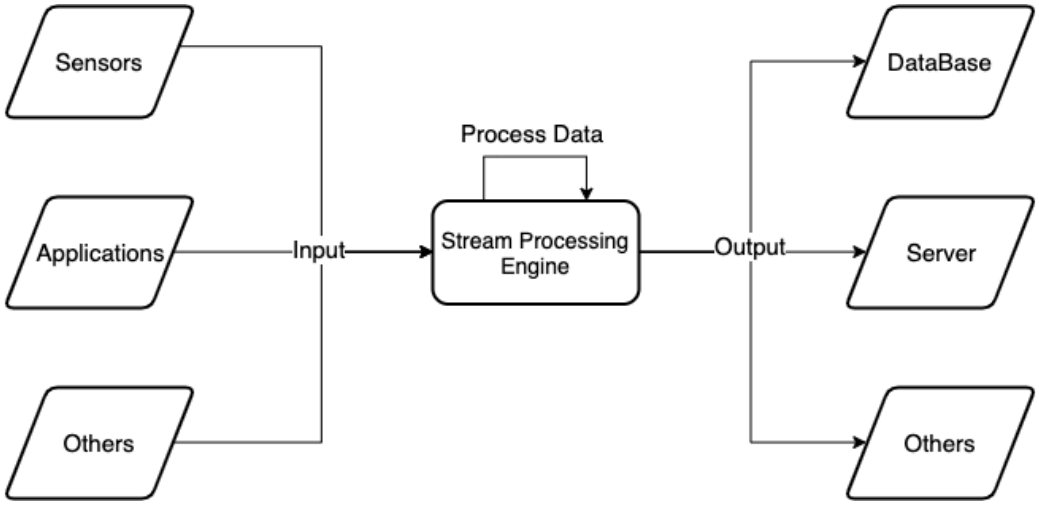

Fig. 1. Generic representation of Stream Processing

1.1 Challenges/Shortcomings of SPEs

Even though stream processing engines come to solve a lot of problems when trying to process large amounts of data, there are still significant challenges that need to be dealt with and solved [1]. To show how old these issues are, we have the system **Borealis** [2] which was designed in 2005 and one of its focus was scalability and flexibility which are still important nowadays. Among the more relevant to our work, there are:

- **Scalability**: Ever since stream processing engines were created, we have had the issue of scalability. During the years, the stream processing engines have been getting new functionalities for handling scalibility. We can nowadays for example have different levels of parallelization of an operator which can be changed dynamically during the runtime.

- **Bottleneck issues**: When having a big influx of inputs and not enough processing power to handle it or the operators are not the most efficient for the workload, the system will suffer a bottleneck which will have an increase in latency and time taken to finish the

processing of the data. We can have bottlenecks in latency and/or throughput. Both are usually due to missing or inefficient scalability and/or resource management components.

- **Resource Management**: Stream Processing Engines are executed in physical/virtual machines and like so, its resources need to be managed. To some degree, they are managed by a component of the engine with that specific responsibility. But this is usually static and just allocates resources enough to execute the user application. So in case, we need more resources since there are more data to process it will do nothing, and performance will decrease. In the Borealis system talked earlier, they had a mechanism that allowed for dynamic resource management [3] where the system can scale up and deal with increasing load or time-varying load spikes, with the addition of new computational resources (this is also related with scalability).

- **Flexibility**: Most of the systems due to its possible complexity, only allow for the static declaration of job graphs (directed graphs describing the high-level logic of a stream processing program). In case the user wants to change some operator since it is not optimized in the way it was configured, for example, it will not be possible to change which in turn reduces performance in the system.

1.2 Roadmap

The rest of the document is organized in the following way: Section 2 describes the main goals of our work. In Section 3, we present an analysis to the related work. Section 4 presents a solution architecture and the main protocols proposed. In Section 5, we describe how evaluate our solution in terms of system metrics, and what workloads will be used to exercise it. Lastly, Section 6 concludes the document and wraps up with the important marks.

2 Goals

Our main goal is to contribute to the development of an extension for a Stream Processing Engine that allows for more context-aware resource management and load balancing. With this, we can process a large number of inputs efficiently and quickly. The individual work goals are:

- Investigate the state of the art and previous researches in SPEs, scalability, elasticity and bottleneck fixes;

- Study how Flink and Spark work, what metrics can be obtained from those systems and how they are obtained;

- Resulting from the previous study, design and create an architecture using middleware and an extended data structure;

- Experimental evaluation of the work to assess how it affects the scalability of the system and how it efficiently addresses bottleneck issues.

3 Related Work

In this section, we present the fundamental and state of art, academical, and commercial, work in the development of Stream Processing in general in Section 3.1, Stream Processing Technologies in Section 3.2 and System Usage Fairness in Section 3.3.

3.1 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

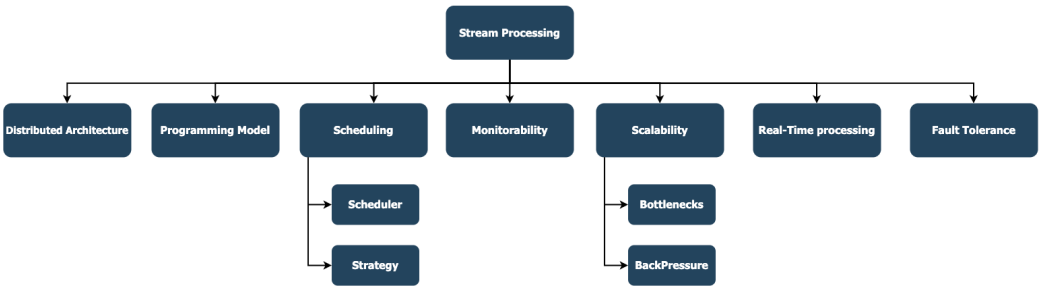

Fig. 2. Stream Processing dimensions

in Figure 2. We have **Distributed Architecture**, **Programming Model Scheduling**, **Monitorability**, **Scalability Real-Time Processing**, **Fault Tolerance**.

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 [4]. 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 3.

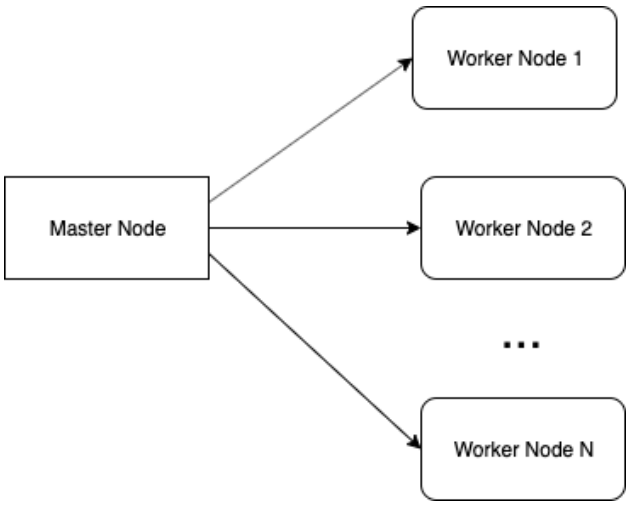

Fig. 3. Generic Distributed Architecture

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 [5] 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 [6] 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 [7] 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 4 we are able to see what a sliding window is and how it can be represented.

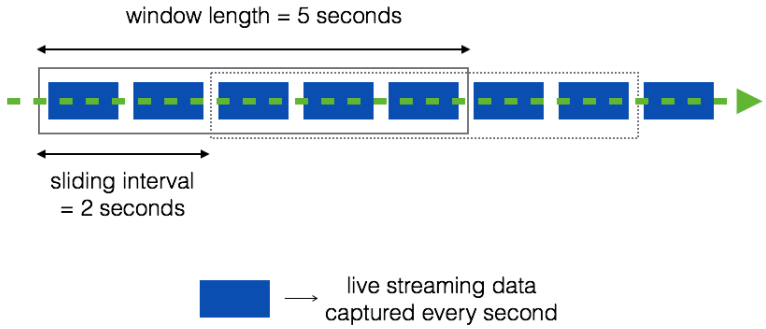

Fig. 4. Sliding Window Example (https://prateekvjoshi.com/2015/12/29/performing-windowed-computations-on-streaming-data-using-spark-in-python/)

There are multiple types of windows [8], 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, [9] 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 [10] 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 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 [11] (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 [12] and Medusa [13] are stream processing engines that try to be scalable but still have some issues which the article Scalable Distributed Stream Processing [14] 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 [15]), they need to be processed and analyzed as fast as possible to get the most value out of them [16]. 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

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/rest_api.html
[2] https://sematext.com/spm/
[3] https://www.networkcomputing.com/networking/basics-qos

computation. To overcome these issues, we need to have a fault-tolerant system [17] and mechanisms that make it possible.

Data losses can happen due to numerous reasons, as for example a machine that has information 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* [18] and *resilient data structures* [19].

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.

3.2 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.

3.2.1 Apache Spark

Apache Spark [4] [20] 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 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.

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

– 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.

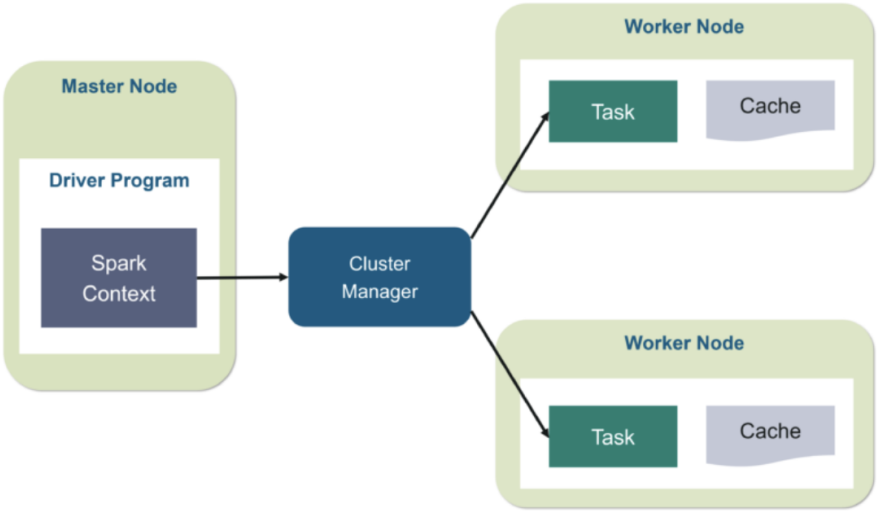

Fig. 5. Apache Spark architecture (https://www.edureka.co/blog/spark-architecture/)

3.2.2 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.

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

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

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.

3.2.3 Apache Flink

Apache Flink [6] [21] 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 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.

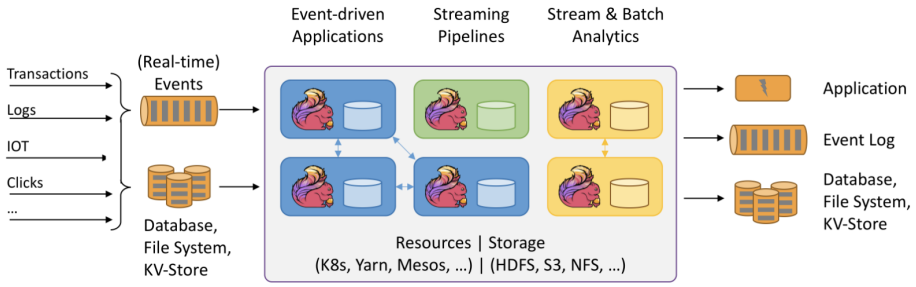

Fig. 6. Apache Flink architecture (https://flink.apache.org/)

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

3.2.4 Google MillWheel

Google MillWheel [22] 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.

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

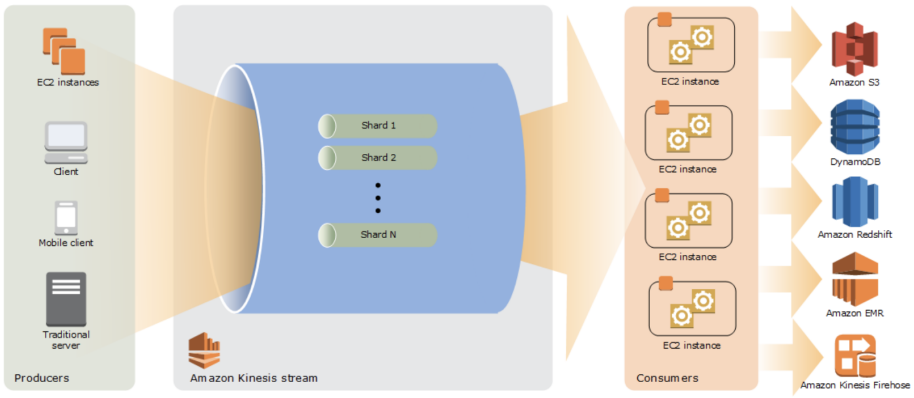

Fig. 7. Amazon Kinesis architecture (https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html)

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

As shown in Figure 7, 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.

3.2.6 Apache Storm

Apache Storm [8] [23] 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 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 8, 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.

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.

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

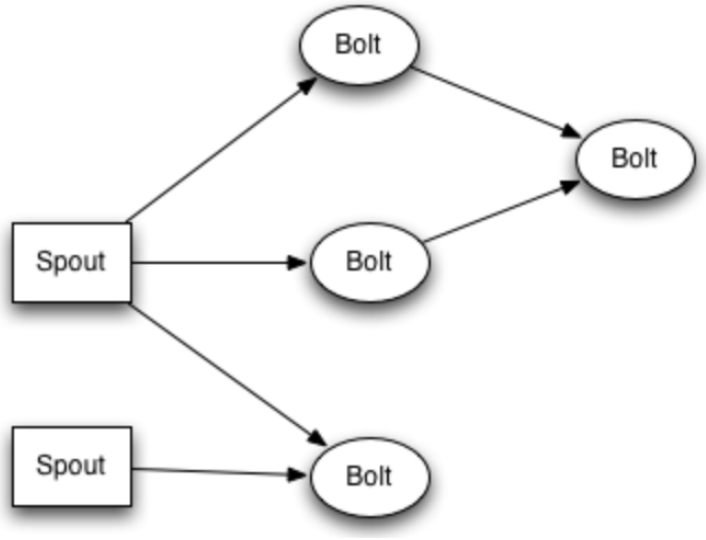

Fig. 8. Apache Storm architecture

3.2.7 Apache Heron

While maintaining API compatibility with Apache Storm, Apache Heron [9] [24] 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.

3.3 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.

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

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.

3.3.1 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 [25], a static optimization tool able to leverage cost models that programmers can use to detect and understand the inefficiencies of an initial application design. 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 [26].

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 9. 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.

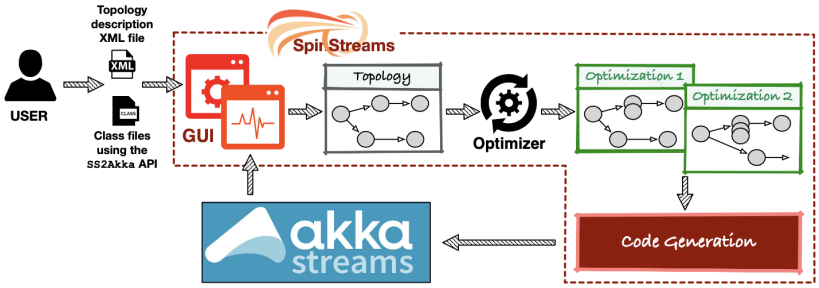

Fig. 9. Spin Streams Workflow (from [25])

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.

3.3.2 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 [27] 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 10 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

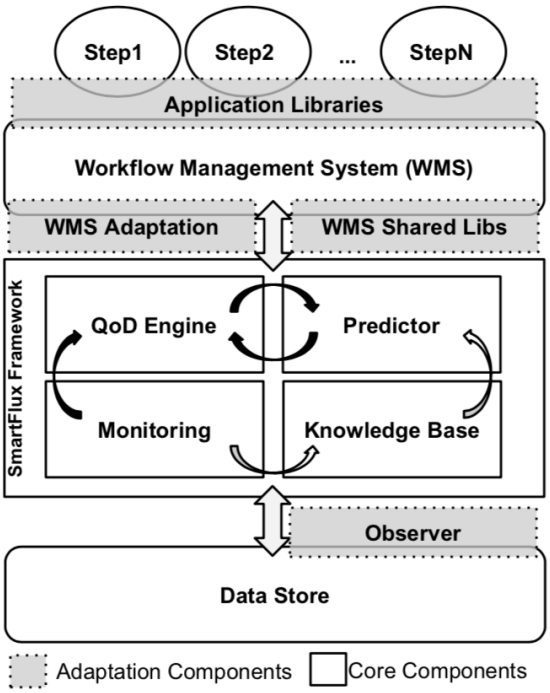

Fig. 10. SmartFlux Middleware Architecture (from [27])

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 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.

3.4 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 needs to be one that fulfills the requisites.

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 modify 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 [28]

4 Solution Proposition

In this Section, we present the architecture of SDD4 Streaming, an extension to Flink embodied as a library that enables dynamic improvements to an Apache Flink application. This library is based on extracting metrics, specific information about a system and, through that makes decisions and acts upon Flink according to a set of pre-defined rules.

4.1 Architecture

Before explaining in detail each component of our solution, we will go through a simple data flow from the input coming from a data source to the output going into a data sink. This flow is shown in Figure 11.

Most of the data flow will remain the same as in a normal Flink application. The data will be received by the system, processed with the operators declared by the client application and then collected as output and sent into a data sink where they will be stored for later use.

Because of the nature of stream processing where the volume of input is ever-changing, the system needs to be prepared for the situation where there is more data than the current resources can handle. This will eventually lead to a degradation of performance.

Our solution comes to solve some of these issues by acting behind the scenes in the system and making changes wherever needed, to make the system run the smoothest possible at any time. Our actions are constrained by a set of rules that represent the

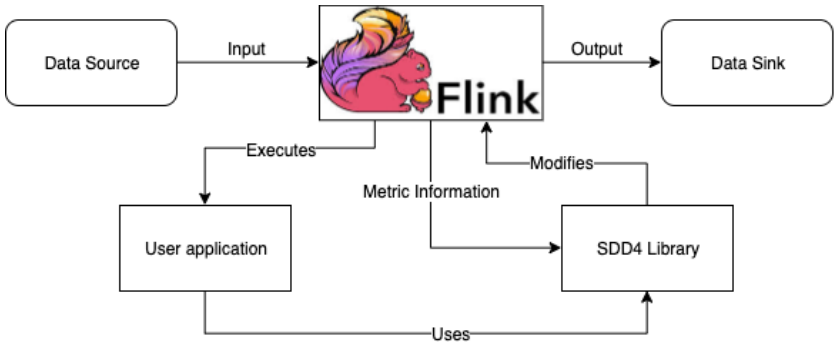

Fig. 11. Data Flow Diagram

Service Level Agreement (SLA) which is defined by the user based on the options we provide.

We can divide our solution into the three following components which will be explained further ahead:

- **Metric Handler:** This component handles all metric related information, from fetching it from Flink to storing it in our data structure for later use;
- **Resource Manager**: This component is responsible for making decisions based on the state of system through the use of the stored metrics;
- **Action Component**: This component is responsible for acting upon the system when necessary.

The Metric Handler will be explained in Section 4.2 where we talk about the integration with Flink. The Resource Manager is explained in section 4.1.2 where we talk about how we handle resource management as well as how we make decisions based on the system state. The Action Component is a mix between Sections 4.1.2 and 4.2 since we make actions upon Flink based on the analysis made by us.

4.1.1 Model

Before we can explain in more detail our components, we will go through the software interface and how the user interacts with our solution.

SDD4 is available as a JAVA library and like so, can be imported like any other library and start using it right away. Before we can do anything, the user needs to initialize our library with information specific to their system.

For the initialization, we require the user to give us the information on how we can make queries to the Flink Cluster (e.g. to fetch metrics) and the SLA they have defined.

The SLA is composed of goals that the user wants to be achieved during the runtime of a job:

- Latency: What is the maximum amount of latency allowed in the system;
- 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.

After initialization, we will start fetching metrics in the background without the need for explicit user interaction. These metrics will be stored into our data structure so we can later analyze it to make decisions about resource management.

Besides this, we provide various classes which will be explained later which should be used by the user for all operators they want us to intervene in. These classes will have our decision and action components integrated.

4.1.2 Resource Management

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 a current job or by suppressing inputs for the sake of latency at the cost of result accuracy.

In order to have a good Quality of Service (QoS), we will enforce the rules declared in the SLA by the user which we got from initialization. We need to maintain a good balance between all the rules and keep the system running.

In Algorithm 1 we have the pseudo-code responsible for the resource management.

Algorithm 1: Decision Algorithm

```
   Input: taskInfo
   Output: shouldProcessInput
1  if !isTaskDegraded(taskInfo) then
2      if shouldDownScaleJob() then
3          downscaleJob()
4      return true
5  if checkAvailableResources() > 0 then
6      upscaleJob()
7      return true
8  if shouldSuppressInput() then
9      return false
10 else
11     return true
```

To explain a bit how the algorithm works, we from every input we receive, we will analyze the state of the current task. This will check if everything is in order which include checking the latency as well as the CPU load in line 1.

With this we are able to make the decision of simply passing the control back to the user code and processing the output or the need to make an action 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 2-3). 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 and Process Input

If we have enough resources available in the system we are able to simply rescale the job and then process the input (lines 5-7). This will decrease the total load on the task and so help reduce performance degradation. But this can only be done in case the system has the required resources to do so. If not we need to follow a different approach.

Our other approach is simply suppressing the input partially and so not processing it (lines 8-9). This will decrease the load/latency 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 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 (lines 8 and 10-11). Even

though this will make an increase the load in the system, there is nothing more we can do without breaking our SLA with the user.

4.1.3 Data Structure

To make this all possible we need to have a reliable data structure that can give us a fine grained vision of the system at any point in time. In Figure 12 we are able to see the data structures we have which represent every element from the system.

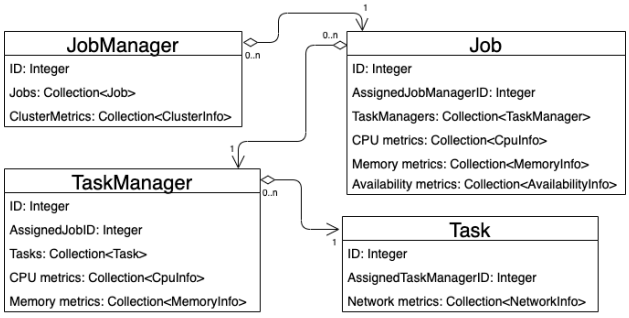

Fig. 12. Data Structure

The Data Structure is all related to each other and we can represent their relation like this JobManager -> Job -> TaskManager -> Task. With this structure, we can, for example, get all tasks executing in the system and check which one is having the worst performance. Then since all data classes are related to each other, we can go to the task's parent to get the ID of the job to rescale.

With the data structured this way, we are able to analyze just what we need and filter data efficiently. For example, we may want to know for a certain TaskManager if there is any bottleneck by checking if any of its tasks is taking a while to process the inputs.

4.2 Integration with Apache Flink

In this section we explain how we fetch metrics from the Flink system, and what we will use from their programming model.

4.2.1 Metric Handler

Apache Flink is able to give us a lot of information at multiple levels and scopes. We have the following system scopes:

- metrics.scope.jm
  - Default: $< host > .jobmanager$
  - Applied to all metrics that were scoped to a job manager.
- metrics.scope.jm.job
  - Default: $< host > .jobmanager. < jobname >$
  - Applied to all metrics that were scoped to a job manager and job.
- metrics.scope.tm
  - Default: $< host > .taskmanager. < tm_id >$
  - Applied to all metrics that were scoped to a task manager.
- metrics.scope.tm.job
  - Default: $< host > .taskmanager. < tm_id > . < job_name >$

  - Applied to all metrics that were scoped to a task manager and job.
- metrics.scope.task
  - Default: $< host >.taskmanager.< tm_id >.< job_name >.< task_name >.< subtask_index >$
  - Applied to all metrics that were scoped to a task.
- metrics.scope.operator
  - Default: $< host >.taskmanager.< tm_id >.< job_name >.< operator_name >.< subtask_index >$
  - Applied to all metrics that were scoped to an operator.

Through all these scopes we are able to easily identify to where a certain metric belongs to and consequently where and what is going wrong in the system. These metrics that Flink disponibilises goes from the CPU and Memory usage to the Network like the number of inputs (buffers) waiting to be processed and the amount of output processed.

Flink provides various types of reporters that can be used to send metrics to an external system/application. For our case, we only care about the Java Management Extensions (JMX) reporter. This is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks.

To use this reporter, the user needs to enable it in the configuration for the application as shown in Listing 1.1. The metrics will be available through the port defined (if not defined by the user, the default Flink port will be used).

```
metrics.reporter.jmx.factory.class:
    ↪ org.apache.flink.metrics.jmx.JMXReporterFactory

metrics.reporter.jmx.port: 8789
```

Listing 1.1. JMX Reporter in Flink

Metrics exposed through JMX are identified by a domain and a list of key-properties, which together form the object name.

The domain always begins with org.apache.flink followed by a generalized metric identifier. In contrast to the usual identifier it is not affected by scope-formats, does not contain any variables and is constant across jobs. An example for such a domain would be org.apache.flink.job.task.numBytesOut.

The key-property list contains the values for all variables, regardless of configured scope formats, that are associated with a given metric. An example for such a list would be host=localhost,job_name=MyJob,task_name=MyTask.

The domain thus identifies a metric class, while the key-property list identifies one (or multiple) instances of that metric.

4.2.2 Programming model

Since we want to act as an intermediary between the incoming data and the operators, we need a way to intercept the processing functions from Flink that the user is using. For this case we will need the user to change his code a bit and instead of using directly the types given by Flink, they will use our types.

So SDD4 provides various classes that the user can instantiate and use for processing data. These classes implement the Flink processing functions [10], which are the ones that process the inputs gotten from a certain operator.

As an example of an operator in Flink, we have the following listing 1.2 that represents the use of a *Map* operator.

[10] https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/

```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
});
```

Listing 1.2. Map operator in Flink without SDD4

In the example above we would have a similar type as **MapFunction** that would have our implementation. Our class would receive as an argument the function that the user would use in that operator so we can call it if accepted by our *Resource Manager* component.

In the following listing 1.3 is represented how the new usage of the operator would look like:

```
DataStream<Integer> dataStream = //...
dataStream.map(new SDDMapFunction<Integer, Integer>(new MapFunction<Integer,
    ↪ Integer>() {
            @Override
            public Integer map(Integer value) throws Exception {
                return 2 * value;
            }
}));
```

Listing 1.3. Map operator in Flink using SDD4

Since we will "override" all the processing functions, we can optimize if needed for any type of operator we deem necessary. This is good to have because each operator has a different impact on the performance since they all do different data operations which are inherently heavier in performance.

Some examples of processing functions that we will override with the schema **Original** $->$ **SDD4** are:

- ReduceFunction $->$ SDDReduceFunction
- WindowFunction $->$ SDDWindowFunction
- JoinFunction $->$ SDDJoinFunction

5 Evaluation Methodology

In this section a proposed evaluation method for validating our solution at a practical level is presented. In sub-section 5.1 we explain the system metrics, and the workloads, in Section 5.2, that we intend to use during the evaluation phase of our prototype.

5.1 System Metrics

For metrics, we will have two categories. One will be to check the performance of the system when using our solution while the other, the overhead our solution gives to the system.

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, to assess fulfillment of quality of service.
- 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.

5.2 Workload/DataSet

We will have two types of workloads:

Synthetic Benchmark: This will be for Flink applications that are specifically made to test our solution. Its purpose is to test how our solution behaves in specific scenarios made to force the system into a performance heavy state. We will be using a few examples from an already existing project on GitHub [11] which has specific benchmarks for Flink.

Example Applications: This will be for testing our solution against example applications that more resemble a real one. Like this we are to check how SDD4 fairs with a real application. For this we will use two different applications.

One will be from a GitHub repository from Yahoo [12] which is an application that manipulates ad event data received into usable data about ads from certain campaigns. They had as inspiration a paper that explains in more detail how benchmarking in stream processing should be approached, instead of using simple workloads, they base the workloads in real-life scenarios [28].

The other application will involve a dataset for taxi trips (116GB of data) and fare data (75GB of data) for the year 2010 to 2013 in New York [13]. From analysing these data [29], we will get the spatial, temporal and cost hotspots in New York for each available year.

6 Conclusion

Our work presented SDD4, a proposal for a JAVA library that allows for the scaling of an Apache Flink application to avoid bottlenecks in the system. We started by describing at a high level how Stream Processing works and its shortcomings. Next, we have settled the goals of our work. After our research throughout the current stream processing systems and scientific literature, we presented a taxonomy/dimensions that a Stream Processing Engine abides by and two other solutions that increase the scalability of a system. The following section introduced SDD4 with the respective architecture. We conclude with the evaluation methodology to assess the future implementation of our solution.

[11] https://github.com/dataArtisans/flink-benchmarks
[12] https://github.com/yahoo/streaming-benchmarks
[13] https://uofi.app.box.com/v/NYCtaxidata

7 Timeline

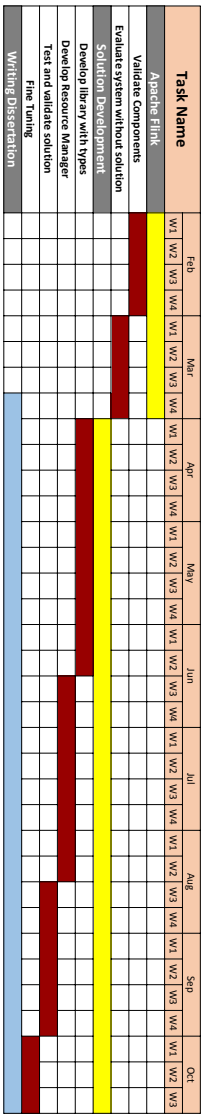

Fig. 13. Gantt chart for the dissertation planing

References

1. Talhaoui, M.A.: Real-time data stream processing - challenges and perspectives. vol. 14 (08 2018). https://doi.org/10.20943/01201705.612
2. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: Cidr. vol. 5, pp. 277–289 (2005)
3. Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., et al.: Distributed operation in the borealis stream processing engine. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. pp. 882–884. ACM (2005)
4. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: 2010 IEEE International Conference on Data Mining Workshops. pp. 170–177. IEEE (2010)
5. Carvalho, O., Roloff, E., Navaux, P.O.: A distributed stream processing based architecture for iot smart grids monitoring. In: Companion Proceedings of the10th International Conference on Utility and Cloud Computing. pp. 9–14. ACM (2017)
6. Haloi, S.: Apache zookeeper essentials. Packt Publishing Ltd (2015)
7. Carbone, P., Gévay, G.E., Hermann, G., Katsifodimos, A., Soto, J., Markl, V., Haridi, S.: Large-scale data stream processing systems. In: Handbook of Big Data Technologies, pp. 219–260. Springer (2017)
8. Gedik, B.: Generic windowing support for extensible stream processing systems. Software: Practice and Experience **44**(9), 1105–1128 (2014)
9. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: 22nd International Conference on Data Engineering (ICDE'06). pp. 49–49. IEEE (2006)
10. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive control of extreme-scale stream processing systems. In: 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06). pp. 71–71. IEEE (2006)
11. Heinze, T., Jerzak, Z., Hackenbroich, G., Fetzer, C.: Latency-aware elastic scaling for distributed data stream processing systems. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. pp. 13–22. ACM (2014)
12. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., et al.: Aurora: a data stream management system. In: SIGMOD Conference. p. 666. Citeseer (2003)
13. Balazinska, M., Balakrishnan, H., Stonebraker, M.: 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. pp. 929–930. ACM (2004)
14. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. In: CIDR. vol. 3, pp. 257–268 (2003)
15. Tönjes, R., Barnaghi, P., Ali, M., Mileo, A., Hauswirth, M., Ganz, F., Ganea, S., Kjærgaard, B., Kuemper, D., Nechifor, S., 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)
16. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM Sigmod Record **34**(4), 42–47 (2005)
17. Kwon, Y., Balazinska, M., Greenberg, A.: Fault-tolerant stream processing using a distributed, replicated file system. Proceedings of the VLDB Endowment **1**(1), 574–585 (2008)
18. Carbone, P., Fóra, G., Ewen, S., Haridi, S., Tzoumas, K.: Lightweight asynchronous snapshots for distributed dataflows. arXiv preprint arXiv:1506.08603 (2015)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: 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. pp. 2–2. USENIX Association (2012)
20. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. Communications of the ACM **59**(11), 56–65 (2016)

21. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)
22. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. In: Very Large Data Bases. pp. 734–746 (2013)
23. Evans, R.: Apache storm, a hands on tutorial. In: 2015 IEEE International Conference on Cloud Engineering. pp. 2–2. IEEE (2015)
24. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 239–250. ACM (2015)
25. Mencagli, G., Dazzi, P., Tonci, N.: Spinstreams: a static optimization tool for data stream processing applications (2017)
26. Gupta, M.: Akka essentials. Packt Publishing Ltd (2012)
27. Esteves, S., Galhardas, H., Veiga, L.: Adaptive execution of continuous and data-intensive workflows with machine learning (2018)
28. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J., et al.: Benchmarking streaming computation engines: Storm, flink and spark streaming. In: 2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW). pp. 1789–1792. IEEE (2016)
29. Patel, U., Chandan, A.: Nyc taxi trip and fare data analytics using bigdata. Retrieved June **9**, 2017 (2010)