SDD4Streaming

Tiago Lopes, 94055
Instituto Superior Tecnico
tiago.mourao@tecnico.ulisboa.pt

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

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 [6], operational monitoring of large infrastructures, and Internet of Things (IoT) [13]. 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.

First, we should explain what the Stream Processing paradigm is. It is equivalent to dataflow programming [12], event stream processing [4], and reactive programming [2] 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.

The rest of the paper is structured as follows. Section 2 describes briefly the fundamental and state of the art works in the Stream Processing, Resource Management and Input/Processing Management. Section 3 presents the architecture and the resource management algorithm that compose SDD4Streaming. Section 4 presents the evaluation to our SDD4Streaming solution checking its performance on applications. Finally, Section 5 wraps up the paper with our main conclusions.

2 Related Work

We present our related work in four parts. First giving insight on what Stream Processing is and how it works. After that explain how one Stream Processing System works, namely Apache Flink which is the system we have developed our solution against. And finally provide a brief explanation of two different solutions that come to solve specific issues inherent to Stream Processing. The first one is for Resource Management where the solution optimizes the client application before its execution and the second is Input and Processing management where the solution through machine learning can infer information from the inputs and with that increase performance.

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. The dimensions are **Distributed Architecture**, **Programming Model**, **Scheduling**, **Monitorability**, **Scalability**, **Real-Time Processing**, **Fault Tolerance**.

For our solution the more important dimensions that we focus on solving is Scalability and Monitorability which we will explain what they involve.

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.

Scalability: For a system that is constantly dealing with data and with clients that are expecting a certain Quality of Service (QoS) [1] 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 [10] (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 [1] and Medusa [3] are stream processing engines that try to be scalable but still have some issues which the article Scalable Distributed Stream Processing [7] explains what the issues are and how they could be solved.

2.2 Apache Flink

Apache Flink [2] [5] 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.

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

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

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.

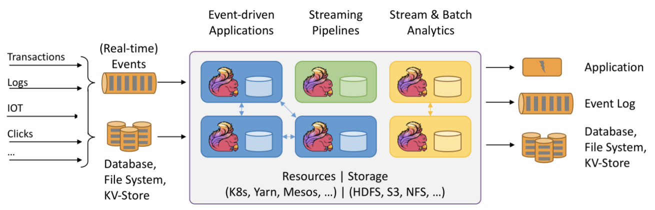

Event-driven Applications
Streaming Pipelines
Stream & Batch Analytics
Transactions
Logs
IOT
Clicks
...
(Real-time) Events
Database, File System, KV-Store
Resources | Storage
(K8s, Yarn, Mesos, ...) | (HDFS, S3, NFS, ...)
Application
Event Log
Database, File System, KV-Store

Figure 1. Apache Flink architecture (https://flink.apache.org/)

2.3 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 [11], 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 [9].

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. 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.

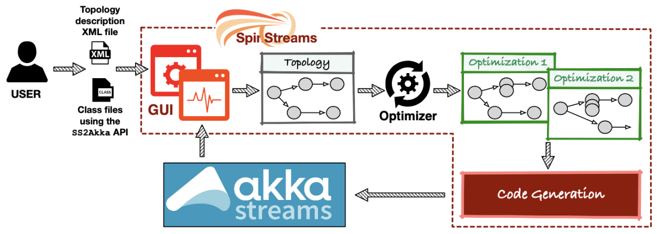


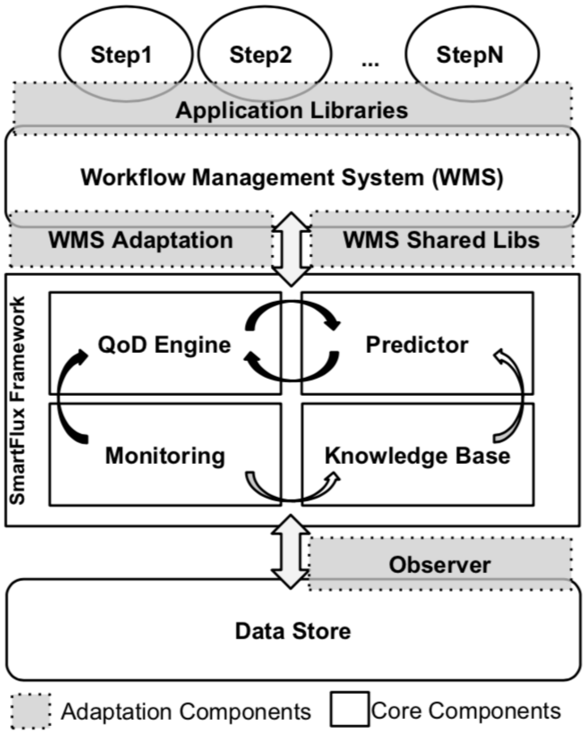
Figure 2. Spin Streams Workflow (from [11])

2.4 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 [8] 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 2.4 we can see the architecture that was designed for the Smart Flux solution.



[8]

3 Architecture

SDD4Streaming [3] 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.

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;

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

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

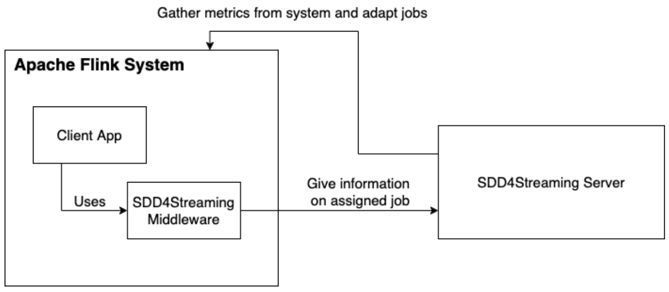


Figure 3. Relation between client application and our components

Before explaining each of these components we will go over the data structures we use.

3.1 Data Structure

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 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 will be further explained in the Implementation chapter

3.3 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 (also often called dataflow graph);
- TaskManager: Supervises all tasks assigned to it and interchanges data between then;
- Task: Node of a Physical Graph. 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.

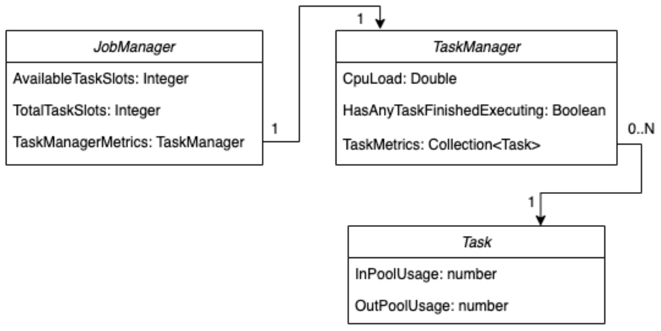

JobManager
AvailableTaskSlots: Integer
TotalTaskSlots: Integer
TaskManagerMetrics: TaskManager
1
1
TaskManager
CpuLoad: Double
HasAnyTaskFinishedExecuting: Boolean
TaskMetrics: Collection<Task>
0..N
1
Task
InPoolUsage: number
OutPoolUsage: number

Figure 4. Metric Data Structure

As shown in Figure 4 above, 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).

3.4 Middleware

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.

3.5 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.

3.6 Resource Manager

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
if !JobGettingRescaled(taskInfo) OR
  !areMetricsAvailable() then
    return true
end
if isJobDegraded(taskInfo) then
    if shouldUpScaleJob() then
        upscaleJob()
        return true
    end
    if shouldSuppressInput() then
        return false
    end
end
if shouldDownScaleJob() then
    downscaleJob()
end
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.

4 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. We are now ready to look into the results of the testing in Section 5.4.

4.1 Workloads

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 [4], an open-source stream-processing software platform developed by the Apache Software Foundation that aims to provide a unified,

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

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.

4.2 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.

4.2.1 Filtering and Data Cleanup

The dataset for the taxi rides/fares is pretty extensive coming in at a total of 116GB 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.

4.3 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, 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.

4.4 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 [5], in specific the one chosen was Prometheus [6].

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.

4.5 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

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

[6]https://prometheus.io/

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.

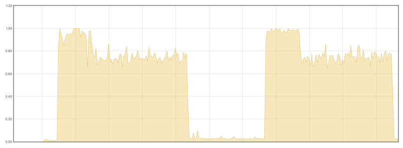


Figure 5. CPU Load on the TaskManager

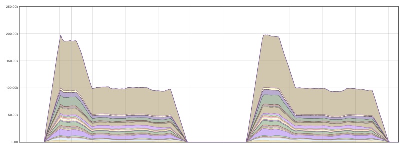


Figure 6. Amount of records getting processed by each sub-task per second (throughput)

We can see the CPU usage in Figure 5 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 6 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.

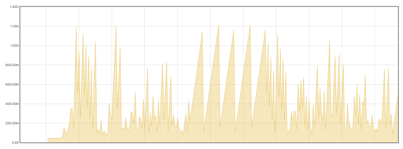


Figure 7. Heap Memory Usage

We can also see memory usage by the TaskManager/-Tasks in the Heap and Non-Heap in Figures 7 and 8 respectively.

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

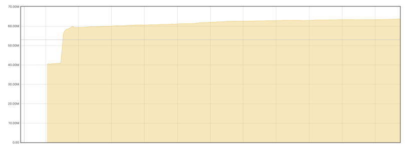


Figure 8. Non-Heap Memory Usage

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.

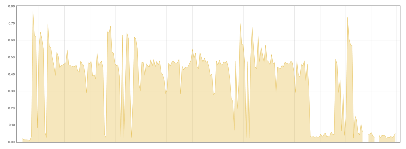


Figure 9. CPU Load on the TaskManager

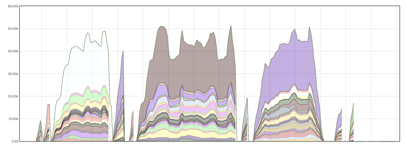


Figure 10. Amount of records getting processed by each sub-task per second (throughput)

Here we have the CPU load and throughput in Figures 9 and 10 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.

In Figures 11 and 12 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.

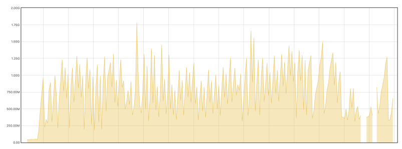


Figure 11. Heap Memory Usage

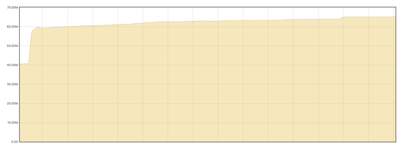


Figure 12. Non-Heap Memory Usage

Finally, specifically for the test with our solution we have in Figure 13 the number of available slots throughout the execution of the job.

5 Conclusion

SDD4Streaming was devised to serve as an extension of the scalability and performance capabilities of a Stream Processing Engine. Meaning that it was created to, in runtime, adapt the system to the current necessities of the system to support the current load its getting. Through the separation of responsibilities between the library and the server we are able to mitigate most of the overhead that our solution causes on the system.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, et al. Aurora: a data stream management system. In *SIGMOD Conference*, page 666. Citeseer, 2003.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):1–34, 2013.

[3] M. Balazinska, H. Balakrishnan, and M. 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*, pages 929–930. ACM, 2004.

[4] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[6] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs. Building a big data platform for smart cities: Experience

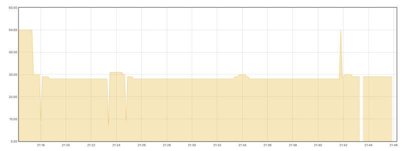


Figure 13. Number of Available Task Slots

and lessons from santander. In *2015 IEEE International Congress on Big Data*, pages 592–599. IEEE, 2015.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[8] S. Esteves, H. Galhardas, and L. Veiga. Adaptive execution of continuous and data-intensive workflows with machine learning. 2018.

[9] M. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[10] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 13–22. ACM, 2014.

[11] G. Mencagli, P. Dazzi, and N. Tonci. Spinstreams: a static optimization tool for data stream processing applications. 2017.

[12] T. B. Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[13] R. Tönjes, P. Barnaghi, M. Ali, A. Mileo, M. Hauswirth, F. Ganz, S. Ganea, B. Kjærgaard, D. Kuemper, S. Nechifor, 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.