Smart Distributed DataSets for Stream Processing

Tiago Lopes[1,2], Miguel Coimbra[1,2], and Luís Veiga[1,2(✉)]

[1] INESC-ID Lisboa, Lisbon, Portugal
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{tiago.mourao,miguel.e.coimbra}@tecnico.ulisboa.pt,
luis.veiga@inesc-id.pt

Abstract. There is an ever-increasing amount of devices getting connected to the internet, and so is the volume of data that needs to be processed - the Internet-of-Things (IoT) is a good example of this. Stream processing was created for the sole purpose of dealing with high volumes of data, and it has proven itself time and time again as a successful approach. However, there is still a necessity to further improve scalability and performance on this type of system. This work presents SDD4Streaming, a solution aimed at solving these specific issues of stream processing engines. Although current engines already implement scalability solutions, time has shown those are not enough and that further improvements are needed. SDD4Streaming employs an extension of a system to improve resource usage, so that applications use the resources they need to process data in a timely manner, thus increasing performance and helping other applications that are running in parallel in the same system.

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 the 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. To enable this, the concept of stream processing and its solutions were 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 simplifying software and hardware parallelization by restricting the parallel computation that can be performed. For a given sequence of data elements (a stream), this is achieved by applying a series of operators to each element in the stream.

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

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 249–265, 2021.
https://doi.org/10.1007/978-3-030-85665-6_16

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 that no data is lost while any stream is being processed.

We present SDD4Streaming as an extension to `Flink` to improve resource usage efficiency, so that applications use only the resources needed to process data in a timely manner, therefore increasing overall performance by freeing resources for other applications running in parallel in the same system.

The rest of this document is structured as follows. Section 2 briefly describes the fundamentals and state-of-the-art works on stream processing, resource management and input/processing management. Section 3 describes the architecture and the resource management algorithm that compose SDD4Streaming. Section 4 presents the evaluation of our SDD4Streaming solution, showing its performance on applications. Finally, Sect. 5 wraps up the paper with our main conclusions.

2 Related Work

We present related work first giving insight on what stream processing is and how it works. After that we explain how one relevant stream processing system works, namely `Apache Flink` which is the system we have developed our solution against. We then provide a brief explanation of two different solutions that solve specific issues inherent to stream processing.

Stream Processing. Taking into account that stream processing systems are parallel and distributed data processing systems, stream processing can be decomposed in various dimensions/aspects which need to be addressed to create a functional system offering good Quality-of-Service (QoS). For our solution, the most important dimension that we focus on solving is scalability, which we address next.

For a system that is constantly dealing with data and with clients that are expecting a certain Quality-of-Service[1] from the system, we need to have a degree of scalability to be prepared for any type of situation that might happen. Scalability is thus an important property that a system must have - to be elastic [10] (the ability to adapt) to accommodate its requirements and the ever-changing amount of work it receives. This involves a change in the number of resources available, consisting of 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 that internally has a load-balancer that redirects the requests to the worker machines which will then process them. Suppose such a system supports 1000 requests per second at a certain point in time. Three situations can then occur: *a)* receiving fewer requests than the limit supported by the system, and thus wasting resources (e.g. incurring unnecessary

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

billing, etc.); *b)* having the exact supported amount of requests, which would be the perfect situation for the system (although it is not a real scenario that we should take into account as it usually only happens for a really small amount of time); *c)* receiving a number of requests exceeding the limit of what the system supports and so a bottleneck shall occur and the QoS will decrease while latency increases.

For examples of other systems, we note `Aurora` [1] and `Medusa` [3] which, despite aiming for scalability as stream processing engines, still have some issues for which there are explanations and solution proposals in the literature [7].

2.1 Apache Flink

`Apache Flink`[2] [5] offers a common runtime for data streaming and batch processing applications. Applications are structured as arbitrary directed acyclic graph 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 dataflow that starts with one or more sources, comprises transformation operators, and ends with one or multiple sinks (entities in the dataflow which represent outputs). 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.

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.

SpinStreams for Resource Management. When developing a stream processing application/job, the programmer will define a DAG with all the operations that will be performed on received inputs. 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. Due to this, the literature has seen proposals such as `SpinStreams` [11], a static optimization tool able to leverage cost models

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

that programmers can use to detect and understand the potential inefficiencies of an initial application design. `SpinStreams` suggests optimizations for restructuring applications by generating code to be run on a stream processing system. For the testing purposes of `SpinStreams`, the streaming processing system `Akka` [9] was used.

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 is equal to the throughput of its slowest operator, which represents the bottleneck. A technique to eliminate bottlenecks applies the so-called pipelined fission, which creates 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 *back pressure* and recursively propagates to upstream operators up to the sources.

The `SpinStreams` workflow [11] is summarized as follows. The first step is to start the GUI by providing the application topology as input. 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.

SmartFlux for Input and Processing Management. A stream processing application will usually 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 on previous training (machine learning), it decides whether or not it should process them or just simply return the last results. For certain applications where the workflow output changes slowly and without great significance in a short time

window, resources are inefficiently used, making the whole process take a lot longer than it needs to while the output remains moderately accurate.

To overcome these inefficiencies, `SmartFlux` [8] presents a solution that involves looking at the input the system receives, training a model using machine learning and using the model to check and analyze (with a good confidence level) if the received input needs to be processed all over or not. This is done through a middleware framework called `SmartFlux` which affects a component (the workflow management system) of a stream processing engine in order to intercept the way the workflows are being processed.

3 Architecture

SDD4Streaming[3] was designed as a stream processing engine extension, focused on improving scalability and overall performance. We seek to accomplish these improvements while trying to minimize the loss of output accuracy which is usually inherent to adaption during the run-time of complex stateful systems.

Stream processing involves processing a variable volume of time-sensitive data and the system needs to be prepared to handle the issues stemming 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), so we do not need to further address these specific aspects.

Since `Flink` already handles these issues, it is not necessary to focus on low-level aspects related to resource usage and system alterations, which would otherwise be needed. Overall, we adapt the execution of jobs by changing the used level of parallelism. On these systems, multiple jobs can be executed at the same time, each taking a percentage of the total resources and the number of resources needed to process the input changes through time. We propose two ways to handle these two aspects.

When creating, a job we can define the level of parallelism we want, changing 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. However, this also creates an overhead on overall memory usage due to more data needing to be stored for savepoints/checkpoints.

Before acting on the system, it is necessary for the client application to provide information to base decisions on, and this is defined in our Service Level Agreement (SLA). The SLA is comprised of:

- **Maximum number of task slots**. What is the maximum amount of parallelism or number of 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?

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

This SLA will allow us to make decisions according to the type of performance the client needs from the system. This is essential because every client has an idea of how the system should behave, which our proposed extension, as an external element, is not aware 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, a decision was made 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;

With these, the system where the client application runs will be able to use its resources in a more efficient manner, focusing solely on the computation it was designed to do. Most of our contribution is structured on the server, where our major features are located.

Regarding the two parts explained above, they can further be divided 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 system state through the use of stored metrics and altering the system based on them;

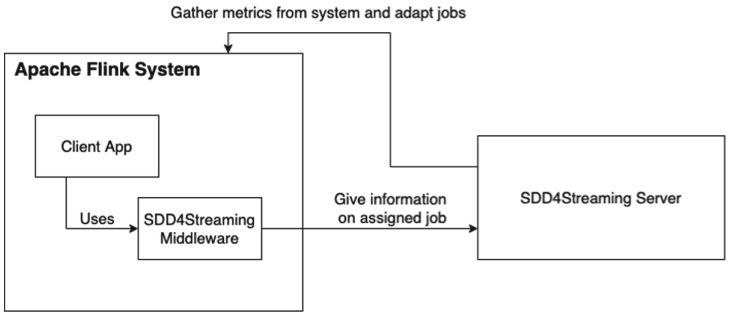

Fig. 1. Relation between client application and our components.

Figure 1 depicts the relation between the client application and the two components above. Explaining each of these components entails going over the employed data structures.

Data Structure. SDD4Streaming has two major sets of data structures necessary for its execution. These consist of data that the client application provides about the overall system where execution will take place, as well as metrics fetched from said system.

Initialization Data. To be able to achieve anything at all in the system, some initialization data is first necessary, and it comes directly from the client application using our solution. These will provide the means to query the system about its resources as well as enable dynamically changing it. This structure has the following elements:

- **Service Level Agreement (SLA)**. Details the optimal performance the clients want the system to have;
- **Job name**. Used to identify a running job;
- **Server base URL**. The base URL for where our web server is running;
- **Client base URL**. The base URL for the `JobManager` where the job shall be executed;
- **JAR archive name**. Name of the JAR used to create the job;

System Metrics. `Apache Flink` provides an extensive REST API that can be used 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, it is necessary to map the necessary endpoints from the API, according to the data that must be sent and received for each one. On some elements this is not so simple and so, instead of directly determining the relation between components, it is necessary 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;

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, they are connected in

our data structure to enable easily checking all elements required of the job in order to make decisions.

Regarding the kinds of data obtainable from these elements, the `Flink` API provides a valuable degree of information with different representations. Information can be gathered either on a collection of items (e.g. the metrics for all task managers in the system) or for a specific element, which may then be used to store in our internal structure.

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

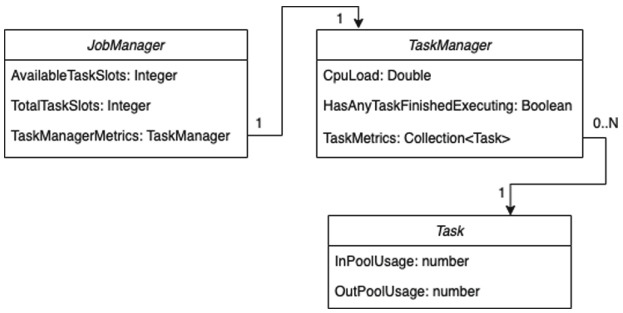

Fig. 2. SDD4Streaming metric data structure.

As shown in Fig. 2, each structure is visible, as well as relations between 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 a job can be scaled 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 required level.

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 those where tasks are being executing.

Finally we have the `Task`, which the `TaskManager` will store a collection of, with each one having the buffer usage for input and output, which are then used to identify back pressure issues (e.g. possible bottlenecks).

Middleware. To work as expected, our solution requires that the user specifies (intended to be simple) modifications to its application. These will involve using our components instead of the ones `Flink` offers, as well as providing initialization data with information about the overall system the client created.

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 to seamlessly handle resource management without our solution having to make any extra communication with the client application.

Besides this, SDD4Streaming also has internal metric management with the information obtained from an API `Flink` provides. This API not only provides metrics about each component running in the system (e.g. `Jobs`, `Tasks` and others), it also allows the system to be adapted. This allows for the creation of an internal data structure that supports decision-making, and then it is also possible to act on the system with the same API.

Metric Manager. SDD4Streaming's decisions and actions are based on information gathered from the system through our `Metric Manager`. This component is responsible for getting metrics on the current system from `Flink` and organizing them according to our data structure.

`Flink` allows its clients to fetch system metrics through various means. For SDD4Streaming, we find that the provided REST API is the best way to achieve this. This API supplies details on every level and component of the system without requiring 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 it is necessary to know the port for each running component, and there is no easy way to find out programmatically, so this solution was not adopted.

For the API, each of the endpoints deemed to be useful were internally mapped to obtain the needed information. 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 (the job itself) to the lowest one (the sub-tasks generated by `Flink` from the operators the client is using).

Resource Manager. For SDD4Streaming, the `Resource Manager` component is the most important one and where all the decisions on the system will happen. This component will make use of the metrics received from the `Metric Manager` and make decisions depending on the system's compliance with the SLA.

To avoid performance issues, the system must make the best of the resources it has available. Whenever needed, the system will undergo changes to accommodate the ever-changing needs for processing the incoming data. The system can be affected in two different ways. Either by re-scaling the assigned job or by suppressing inputs for the sake of performance at the cost of result accuracy.

Algorithm 1 presents the pseudo-code for checking the job-related metrics and adapting the system accordingly. For every input received, the state of the job will be analyzed. First, the `Metric Manager` is checked to find what are the current metrics for the job. If there are no metrics available or the job is getting re-scaled, `true` is returned (line 1) since there is nothing that can be done at that point. If there are metrics currently available and the job is not getting re-scaled, then our solution checks the job state and also if it is compliant with the defined SLA.

This allows SDD4Streaming to make the decision of simply passing the control back to the user code and processing the input or to satisfy the need to

Algorithm 1. Decision Algorithm

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

act upon the system first. If the system is running smoothly, before control is passed to the user code, it checks if the job can be scaled down (lines 9–10). If the system is running abnormally, it means changes must be made, but before a decision may be made in that direction, the appropriate corrections must be determined. Three possible actions may take place at this point, depending on the decision undertaken:

- Process Input
- Suppress Input
- Re-scale Job

If enough resources are available in the system, it is possible to simply re-scale the job (lines 4–6). This will help solve bottlenecks and decrease the load on each task, thus helping to reduce performance degradation. But the current job is re-scaled only if no other re-scaling operation is happening on the job. If not enough resources are available for this operation, then a different approach must take place. The other approach is suppressing the input partially and 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 there is a rule for it the SLA, where the user may 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 the other approaches are not possible, then the control will have to be passed to the user code and allow it to process the input as normal (line 12). Even though this will increase the load in the system, there is nothing more that can be done without breaking the SLA with the user.

4 Evaluation

To evaluate SDD4Streaming 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 are based on how applications behave with our solution, comparing them with the scenario of exclusively using what `Flink` provides to see how much improvement is achieved.

We start by looking into the used workloads, then we document the dataset we used as well as the transformations necessary to make the data viable. We then move on to an analysis of the metrics we intended to gather, and present and analyse the results.

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

The producer of data will be `Kafka` [14], an open-source stream-processing software platform developed by the Apache Software Foundation that aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds.

Initially, `Kafka` is prepared with a big volume of data which the application will read from upon starting. Since the volume of data is so high, after the application finishes processing it, there is a waiting period after which we check if our solution down-scales the job since its load at the time will be very low. Finally after the waiting period, another large volume of data will be sent over to `Kafka` to see if our solution is able to adapt the system to support the new load.

Dataset. For the explained workload, we will use a dataset provided by the University of Illinois System. This dataset represents the taxi trips (116 GB of data) and fare data (75 GB of data) from the year 2010 to 2013 in New York.

Filtering and Data Cleanup. The dataset of the taxi rides/fares is extensive, totalling 116 GB, of which we need not 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. Before the data can be used by the application, it requires cleanup.

The data must be analyzed to remove rows that are missing essential information since these will provide nothing for our results, and to map the columns which are necessary for our execution.

Metrics. For each execution, we look to extract two key groups of data: system performance and overhead caused by our solution. The following lists (performance and overhead) describe 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**. Observes variation of application accuracy over time to assess Quality-of-Service fulfillment.
- **CPU usage**. Checks percentage of CPU being used by tasks in the cluster as well as CPU that is reserved but not used (assesses 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.

Testbed Configuration. We designed the 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 to perform the necessary tests.

The 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[4], having chosen `Prometheus`[5]. `Prometheus` is a time series database (TSDB) combined with a monitoring and alerting toolkit. 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 a 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 was executed in the authors' personal computers to avoid using more credits on GCP than were needed.

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

[5] https://prometheus.io/.

Results. For both tests `Apache Flink` was configured to have one `JobManager` and one `TaskManager`. The `JobManager` was configured to 1024 MiB memory 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 are started with a parallel level of 20, therefore using 20 of the 50 total slots.

First we go over the metrics resulting from the test where the application is running without using SDD4Streaming. All figures below for this test belong to the same time interval and have a duration of 23 min.

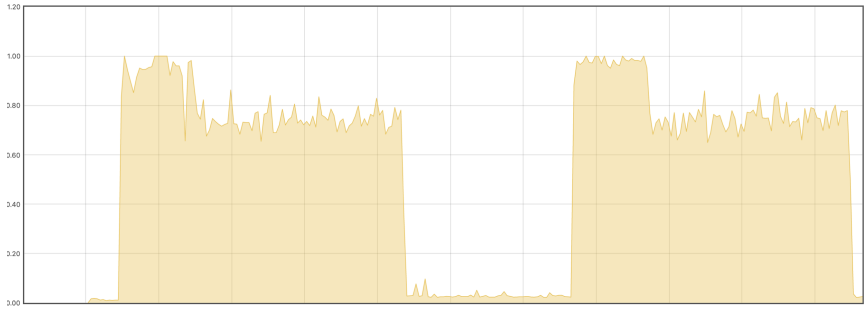

Fig. 3. CPU load on the `TaskManager`.

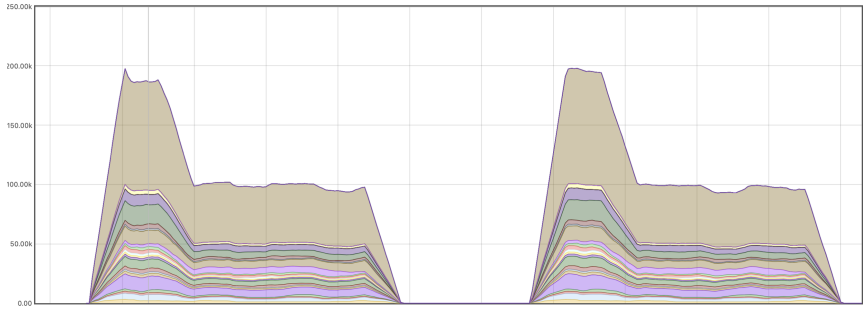

Fig. 4. Amount of records getting processed by each sub-task per second (throughput).

Figure 3 showcases high CPU usage for the tested workload. Besides the first minutes where data is fetched from `Kafka`, the load is mostly constant overall, with occasional fluctuations. For throughput in Fig. 4, behavior similar to the CPU load graph is shown, which is to be expected because higher throughput means that more data is being processed, and for that to happen, higher CPU load is expected. These figures depict an initial very high throughput which then decreases after a few minutes but remains constant. Also when comparing the first volume of data sent and the second one, it can be observed that the CPU load and throughput are fairly similar.

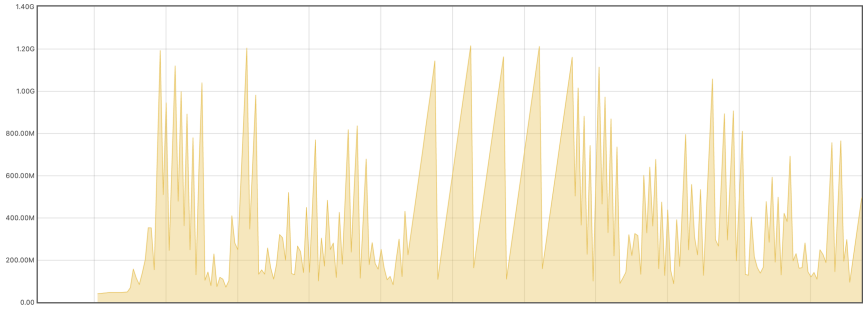

Fig. 5. Heap memory usage.

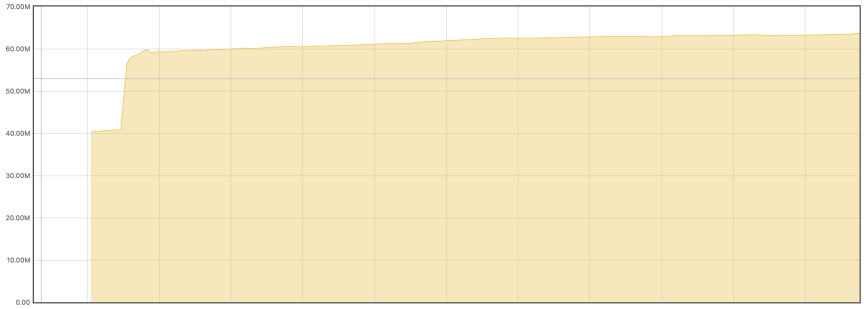

Fig. 6. Non-heap memory usage.

Memory usage by the `TaskManager/Tasks` for heap and non-heap memory is shown in Figs. 5 and 6 respectively.

For the test where the application runs incorporated with our solution, the configurations are the same as the other test but there is an extra configuration of SDD4Streaming. The important part needed before showing the results is the SLA used for this test:

- Max number of task slots: 22;
- Resource usage: 50%
- Input coverage: 80%;

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

CPU load and throughput are shown in Figs. 7 and 8 respectively. The graphs show that the execution was very different from the one without our solution. For example, there are visible drops in both of them that mostly represent when the job was getting re-scaled, 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.

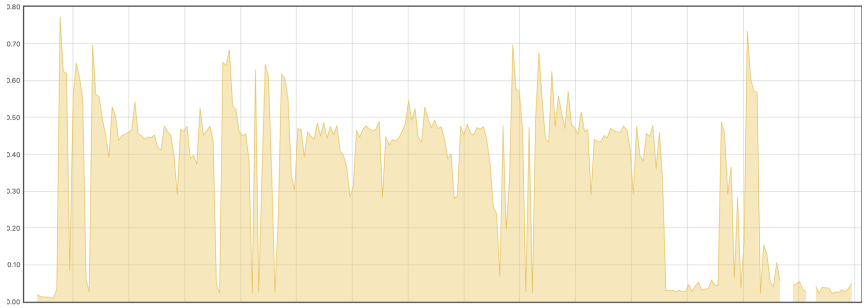

Fig. 7. CPU load on the `TaskManager`.

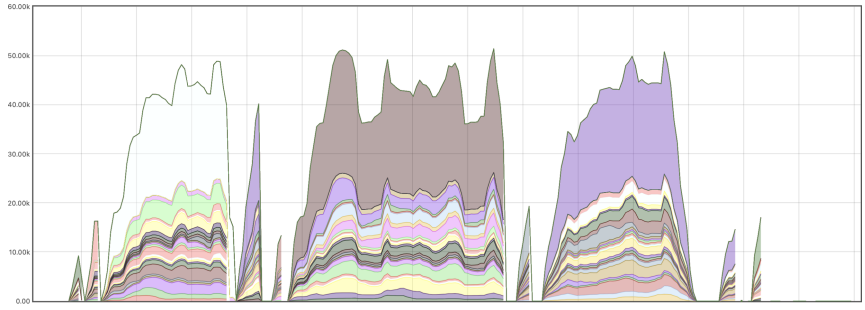

Fig. 8. Amount of records getting processed by each sub-task per second (throughput).

Figures 9 and 10 show the memory usage for the heap and non-heap respectively. From these it can be seen that the non-heap memory is very similar to the previous test, but for the heap memory quite different results are obtained. Since our solution will adapt the system in runtime, the `TaskManager` will need to use more memory in order to do the re-scaling of the jobs. And due to this

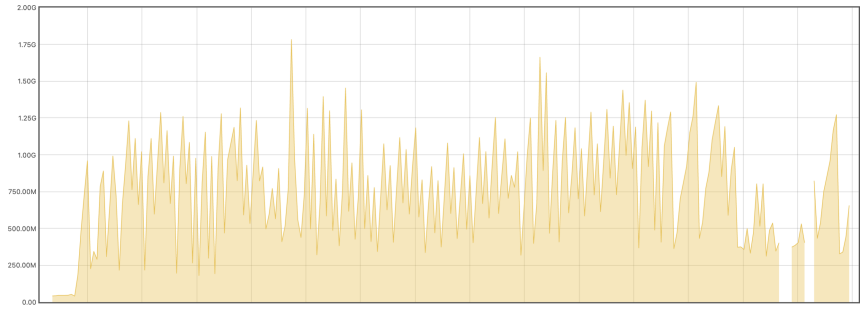

Fig. 9. Heap memory usage.

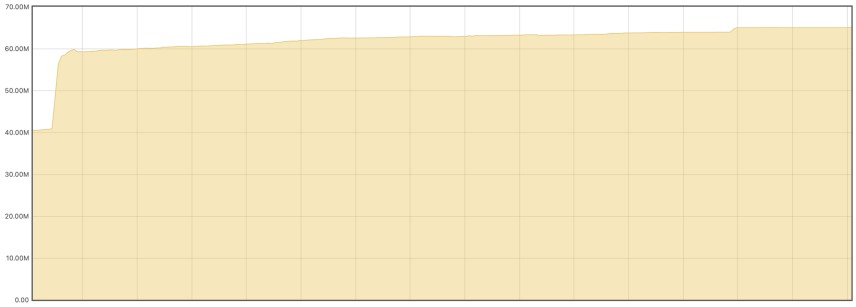

Fig. 10. Non-heap memory usage.

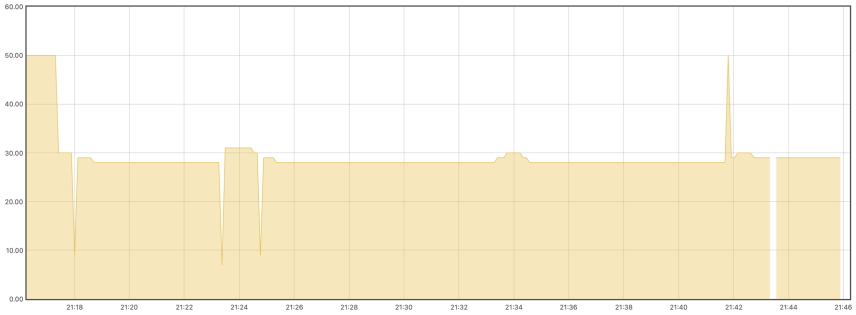

Fig. 11. Number of available task slots.

we observe a higher average use of memory as well as the maximum amount of memory used overall.

Finally, specifically for the test with our solution we have in Fig. 11 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 such as `Flink`. Results show it is able to adapt, in run-time, to current application requirements, supporting current load and improving efficiency. 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. In future work, we intend to address similar issues in the `Gelly` graph library and also in the `Spark` [15] engine.

Acknowledgements. This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020 and PTDC/EEI-COM/30644/2017.

References

1. Abadi, D., et al.: Aurora: a data stream management system. In: SIGMOD Conference, p. 666. Citeseer (2003)
2. Bainomugisha, E., Carreton, A.L., van Cutsem, T., Mostinckx, S., de Meuter, W.: A survey on reactive programming. ACM Comput. Surv. (CSUR) **45**(4), 1–34 (2013)
3. 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)
4. Barga, R.S., Goldstein, J., Ali, M., Hong, M.: Consistent streaming through time: a vision for event stream processing. arXiv preprint cs/0612115 (2006)
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Tech. Committee Data Eng. **36**(4), 28–38 (2015)
6. Cheng, B., Longo, S., Cirillo, F., Bauer, M., Kovacs, E.: Building a big data platform for smart cities: experience and lessons from santander. In: 2015 IEEE International Congress on Big Data, pp. 592–599. IEEE (2015)
7. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. CIDR **3**, 257–268 (2003)
8. Esteves, S., Galhardas, H., Veiga, L.: Adaptive execution of continuous and data-intensive workflows with machine learning (2018)
9. Gupta, M.: Akka Essentials. Packt Publishing Ltd., Birmingham (2012)
10. 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)
11. Mencagli, G., Dazzi, P., Tonci, N.: SpinStreams: a static optimization tool for data stream processing applications (2017)
12. Sousa, T.B.: Dataflow programming concept, languages and applications. In: Doctoral Symposium on Informatics Engineering, vol. 130 (2012)
13. Tönjes, R., 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 (2014)
14. Wang, G., et al.: Building a replicated logging system with Apache Kafka. Proc. VLDB Endow. **8**(12), 1654–1655 (2015)
15. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)