Approxate: Stateful Functions for Approximate Stream Processing - Extended Abstract

João Francisco
Instituto Superior Técnico, Universidade de Lisboa
joao.silva.francisco@tecnico.ulisboa.pt

Abstract

Approximate Computing is a computing model that can be used to increase performance or use fewer resources in stream and graph processing. It can be used to achieve performance requirements (e.g. throughput, lag) about stream processing by lowering the amount of effort that the applications need to process the datasets. This is achieved by lowering the results' precision (i.e. approximate results). Currently, there are multiple stream processing platforms, most of them do not support approximate results natively. Stateful Functions is a platform that allows to easily build stream and graph processing applications. It is an API that uses Flink and allows the functions to exchange their state arbitrarily. It also retains Flink's characteristics, like stateful computations, the fault-tolerance, the ability to scale, the graph processing library Gelly and the control events. This document proposes the design and implementation of an extension to be used with Stateful Functions that supports approximate results. It can also support more efficient stream and graph processing by allocating the available resources intelligently and variably using user-defined requirements about throughput, lag, and latency. This extension allows flexibility in trade-offs (e.g. the user can trade accuracy for performance). The user can choose which metrics should be guaranteed at the costs of the others, and/or the accuracy.

Keywords: Stateful Functions, Apache Flink, Stream Processing, Approximate Computation

1 Introduction

More data is being generated than ever, and it can be processed by stream processing platforms. Stream processing consists in processing items of data (tuples, events) that are continuously arriving at an application. These events can be processed as soon as they are created or they can be stored and later processed as a batch [10]. The processing of the stream can originate results that can later be used again or can originate a new stream.

These events range from connections on social media to bank transactions, so different types of events can have different approaches that are optimal, in some cases low latency is better than maximizing accuracy, in other cases consuming fewer resources is the better approach, just to nominate some examples.

Since the input data rate can fluctuate the scalability of the platform is important. If a platform is not scalable it can lead to a waste of resources when the data rate is lower than expected since the allocated resources will not be fully utilized. It can also lead to the opposite situation where the input rate is greater than what the system can handle without leaving a high amount of lagging events waiting to be processed.

By allowing the system to scale up or scale down when necessary the resource efficiency will increase and the amount of wasted resources are reduced. However, the scaling should be efficient, slight variations in the input rate should not trigger a scaling process. If the scaling is not done in an optimal way it can also lead to wasted resources.

Another important characteristic is the fault-tolerance, if a system does not contain some sort of mechanism to recover from fails, then the data will not be consistent if a failure occurs. If the data loses consistency then the results will not be precise [15]. The data consistency can also be affected by other factors like stream partitioning [6] or synchronization in distributed systems.

Usually, these systems need to have high throughput and low latency since they can receive a lot of events constantly to be processed as soon as possible [8]. Those can be affected by many characteristics, but one that can have a major negative impact is the state sharing between operators. One operator might need data from another one to process some event, if that is the case, then the other operator must share the data.

If the operator that is sharing its state writes it into persistent storage and then the other operator needs to read the data from there, it introduces a costly operation in the dataflow which will decrease the performance [15]. For state sharing to be an efficient operation it cannot utilize persistent storage. The access to the persistent storage increases the latency of the operations and also decreases the throughput.

Another feature that not many platforms support is approximate computation, which is a computation model that reduces the precision of the results to decrease the load on the system and/or increase the performance. This can be done utilizing multiple ways at both, hardware and software level [1, 9, 12].

In summary, a stream processing platform should support an efficient scaling, should have data consistency and fault-tolerance, should operate with low latency and high throughput, it should allow the operators to share their state efficiently without costly operations that access persistent storage and should support approximate computation to improve the performance if that is necessary.

Currently, there are not many mainstream stream processing platforms that support dynamic resources allocation, that

allow to easily build applications, that can auto-scale efficiently, that can use state sharing between the operators without using a storage component, and that allows the user to define the requirements that must be met by trading other characteristics. The support for using less precise results in popular stream processing platforms is also not common. The loss of precision can be used to do the trade-offs to achieve the user-defined processing requirements. The purpose of this work is to give Stateful Functions the ability to use approximate results and to dynamically adjust the allocation of its resources based on user-defined requirements.

2 Related Work

In this section is an overview of some stream and graph processing platforms' characteristics and design. It also explains approximate computation in more detail.

Flink [3] is a framework and distributed processing engine for stateful computations over bound and unbounded streams. Flink can run any kind of application on unbounded streams because it has precise control of time and state. That control allows it to treat an unbounded stream as if it was multiple bounded streams. It is also a scalable engine that allows parallel processing by having multiple instances of the same operator at the same time. It also includes Gelly [3], a library to be used in graph processing [4].

The operators can also have a state and share it. This platform tries to keep the state only in memory so the access can be efficient. Flink contains control events to save the state that can be recovered later in case of a failure. The state is periodically saved in checkpoints. A checkpoint save is called a snapshot and also contains the state of the stream, to avoid process the already processed elements again. This model provides data consistency and fault-tolerance.

Flink also guarantees exactly-once state consistency in case of failures by periodically and asynchronously checkpointing the local state to durable storage. In case an application fails, it can retrieve the last correct state that is checkpointed.

The Flink jobs that are part of the applications are in the form of a dataflow, where there is a chain of operators that receive an event, apply some operation to it, and then send it to the next ones. The job is submitted to the Job manager or the Master that then distributes the job to one or more tasks or Workers. One of the components in the Flink Master is the Resource Manager that allocates or deallocates the available task slots to the operators, the number of which will depend on the parallelism level.

Stateful Functions [2] is an API that utilizes Flink and that simplifies the process of building distributed stateful applications. It has the benefits of Flink like the control events, the fault-tolerance, the scalability, the operators' state, and the operations that Flink supports.

It also allows the operators to message others in a decoupled (the communication does not need to occur in the dataflow's order) and efficient way without using persistent storage. This platform currently supports Kafka [16] as a data broker to receive the events that the applications process and then to send the results.

Spark [14] is a scalable framework that is used for processing large-scale data. It offers functionalities like memory management, job scheduling, data shuffling, and fault recovery. Spark uses Resilient Distributed Dataset (RDD) [18], which are read-only partitioned collections of records, as the data core abstraction. They are fault-tolerant and can be used to share data between users. They can be used to generate new RDDs that can be the result of transformations or operations applied to their data.

This platform supports linear scalability, fault-tolerance, and also in-memory processing, where multiple operators can process some data, and only after that, the results are stored in storage. Spark can also work in a distributed way with the operators spread across multiple clusters.

Spark has cluster managers and like the name implies they are responsible for acquiring and releasing cluster resources depending on the jobs that are being executed. The cluster manager also has the job of managing the resource sharing between Spark applications.

Storm [17] is a real-time fault-tolerant distributed and scalable stream processing platform.

The data processing architecture consists of streams of tuples flowing through topologies, where a topology is a directed graph. The vertices of the graph are the operators that process the events and the edges are the relationship between the operators. The edges represent the data flow. The vertices are divided into two categories, the spouts which are the sources for the topologies, and the bolts which are the vertices that receive data from other vertices and then pass it to the next. Each topology can define its own partition strategy for distributed processing.

The Storm distributed cluster has master nodes that receive topologies from the clients. Each master is responsible for distributing and coordinating the execution of the topology, which is executed by workers. If the system has enough memory, Storm can keep all the data and state from the operators in memory, instead of using storage to get more efficient access to the data and improve the performance.

2.1 Approximate Computation

Approximate Computation is a computing model where the results are not completely accurate. It can be used in scenarios where the applications or systems can tolerate some loss of accuracy [9, 11].

One method is through load shedding [12], where some of the input events are dropped when the system is overloaded. With load shedding we can lower the accuracy by dropping some events instead of processing them, if we drop 10% of the events randomly we would probably get 90% of accuracy. However, one concern with just randomly dropping some

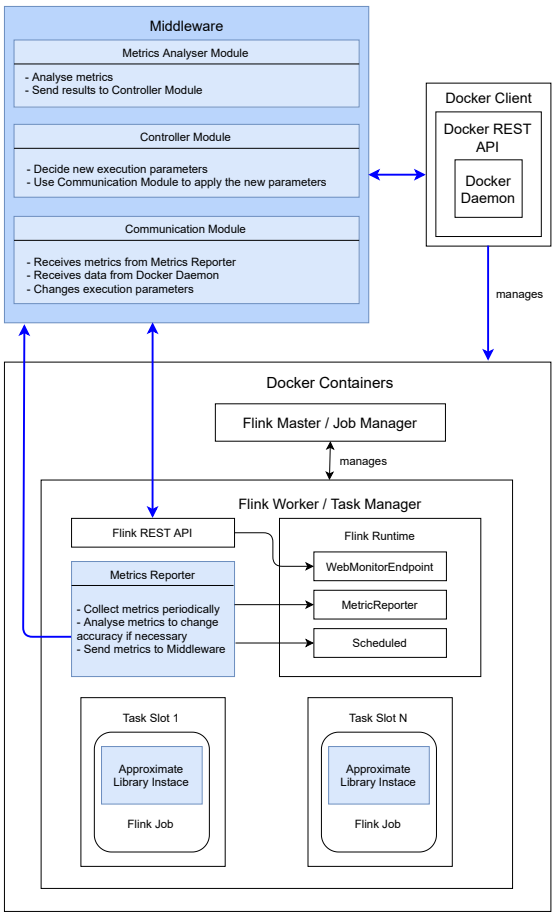

Figure 1. System Overview

events is the fact that the events can come from different data sources. This can lead to some data sources being less represented than the others, so it is also necessary to have attention to the source of each event [12]. If the data source of the events is not considered in the dropping decision, the approximate results will not reflect the events from all of the data sources and will, possibly, be meaningless.

There are other techniques of approximate computation [1, 11] like loop perforations, approximation of arithmetic computations, approximation of communication between computational elements, precision scaling, among others.

Results can also be approximated by carefully delaying the re-execution of workloads (e.g., Map-Reduce workflows) when new input or updated data arrives, providing previous results in response. This way, execution is avoided until the amount and/or significance of the data pending processing reaches application-defined criteria for *Quality-of-Data*. This can be useful to save resources in shared or multi-tenant environments [7] and can be further fine-tuned with machine learning [5].

3 Solution

The solution is **Approxate**, it is an extension of Stateful Functions that manages stream and graph processing applications. The applications use Kafka [16] as the data-broker to get the events and then to send their results. The applications also run in containers using Docker [13], which is the official way of deployment of Stateful Functions applications.

Approxate allows the user to define requirements for the **lag** (number of produced events that are not yet processed), the **throughput** (number of results being produced per unit of time), and the **latency** of the producers (time that is necessary for a produced result to be sent to Kafka). The user can also define the maximum and minimum values for the amount of memory that can be used, for the level of parallelism, and the minimum accuracy which is defined as the percentage of processed events (Equation 1). To achieve this Approxate will scale the system according to the load and the defined requirements, and if it is necessary it can use approximate computing to improve the performance and still keep the results meaningful.

$$Accuracy = \frac{ProcessedEvents}{TotalEvents} \tag{1}$$

Approxate is composed of three components, the Approximate Library (Section 3.4) which is responsible for receiving the events and then deciding based on the accuracy level if they are processed or dropped (load shedding). The second component is the Metrics Reporter (Section 3.5) which collects the execution's metrics, verifies if the application is under load and/or meeting the requirements and if necessary reduces the accuracy. After that, it sends the metrics to the final component, the Middleware (Section 3.6). The Middleware receives the metrics and does a more extensive analysis of them and then it decides if it should adjust the application's resources, the parallelism level, and the accuracy.

When the Middleware changes the memory that the application is allowed to use or the parallelism level it is necessary to restart the application so the changes take effect. Usually, the restart is fast (few seconds to a minute depending on how fast Flink takes and restores snapshots), and for the adjustments in the resources to be worth the time that the restart takes must be less than the time saved by adjusting the requirements. The formula to calculate the time saved is in Equation 3, the time saved is given by multiplying the number of events that are to be processed by the difference between the rate of processing with the adjusted resources and the rate before the adjustment, and then subtracting the time it takes for the restart to happen.

$$NewRate = Rate[Adjusted] - Rate[BeforeAdjust] \tag{2}$$

$$TimeSaved = Events * NewRate - RestartTime \tag{3}$$

In Figure 1 there is a system overview where we can see how the different components will interact with each other. The Approximate Library is instantiated in each Flink job and it does not communicate with the other components directly; the Metrics Reporter runs inside the Flink Worker container, it collects, uses and send the metrics to the Middleware; the Middleware can adjust the resources of the applications and containers, and restart the applications. It interacts with Docker through the Docker Client and with Flink through Flink's REST API which uses the *WebMonitorEndpoint* class.

3.1 Flink

Stateful Functions [2] uses Flink [3], so it has access to its components. One of those is Flink's Metric System. This system can be used to gather various metrics and can be extended to create custom metrics reporters. This system also allows the reporters to perform their actions on a schedule. This way it can collect the metrics at a fixed time rate. Approxate uses this system to collect the following metrics:

- **Recent CPU Load**: This metric is produced by the JVM and indicates the load of the CPU for a short period of time;
- **Memory Heap Used** and **Memory Non-Heap Used**: These metrics are produced by the JVM and they indicate the amount of heap and non-heap memory in use;
- **Memory Heap Committed** and **Memory Non-Heap Committed**: These metrics are produced by the JVM and they indicate the amount of heap and non-heap memory that is committed;
- **Records Lag Max**: This metric is produced by Kafka and indicates the maximum value of events lag, it indicates the number of events that a consumer has not yet consumed in a partition;
- **Request Latency Max**: This metric is produced by the Kafka producers and it measures the time between the sending of the message by the producer and the message being received, the latency of the applications are not being measured since that requires custom code for each application;
- **Record Send Rate**: This metric is produced by the Kafka producers and indicates the number of events that are being sent by the producer per unit of time.

The latency value of the application (amount of time that an event takes to travel from the consumer to the producer inside the application) is not being utilized nor calculated because there is no way of calculating it in a general way. To calculate that value is necessary custom code for each application and so is not being used.

3.2 Kafka

Kafka [16] is a distributed, partitioned, and replicated publish-subscribe messaging system. It can be used to route messages (events) through different applications. The Kafka integration with Flink is done with Kafka Connectors (Kafka-Consumers and Kafka-Producers). The consumers and producers are executed inside the Flink applications. The consumers can consume events and will keep the offset value to know how many events they consumed and how many are they behind from the latest (lag value). The producers are used to write events. The Flink internal metric system receives the Kafka metrics by using the Kafka Connector which allows it to receive the metrics periodically.

3.3 Docker

Docker [13] is a platform that allows the user to run applications in containers. It offers controls to the resources that an application can access, including a priority system for the CPU time, so one container can have priority over the others. However, if the containers with higher priorities are not using the CPU, the containers with less priority can use it. Approxate uses Docker to limit the resources that each application has and, if necessary, also restart the containers where the applications are running.

3.4 Approximate Library

This component is used inside the Stateful Functions applications. It is instantiated in each instance of the operators and is used to perform approximate computation based on the accuracy values defined by the Metrics Reporter or Middleware.

$$Rate = \frac{DroppedEvents}{TotalEvents} \tag{4}$$

It is invoked for each event and uses a random selector, based on the current accuracy level, to decide if it should be considered to be dropped. Before a event is dropped, this component will verify if it can drop that event by checking its origin, the data source. This is done to drop events in the same percentage across the data sources. This leads to an eventual balance of the data sources' representation in the results. To do this the Library registers the number of dropped events and the total number of events (4) globally and for each data source. This is represented in Algorithm 1.

$$Rate(DataSource) - 0.1 \leq Rate(Global) \tag{5}$$

Algorithm 1 Event Selector

```
1: function Invoke
2:     event ← received event
3:     accuracy ← get accuracy
4:     if accuracy < 100 then
5:         if randomSelect(accuracy) == true then
6:             if EventCounter.canSkip() == true then
7:                 skip(event)
8:                 return
9:     forward(event)
```

Figure 2 contains an example of two ways of performing load shedding on the same dataset. The dataset consists of 8 events from source A, 4 events from source B, and 6 events from source C. They arrive with the order that is in Part **(1)** of the figure. After they arrive some events are randomly selected to be dropped (A2, C2, C3, A3, A4, C4, A5, A8, B2, B4, and C6).

In Part **(2)** the load shedding is done without verification for the data sources representations, so the selected events are simply dropped. We can see that the events that were not dropped do not represent all data sources equally, the source A has 3/8 (37,5%) of events represented. The B has 2/4 (50%) and the C has 2/6 (33.3%). This could affect the accuracy of the results.

In Part **(3)** the load shedding is done with the algorithm. The events that were randomly selected to be dropped must verify the condition of Equation (5). The **GR** is the current global rate (4) value in each iteration before deciding if that event is dropped or not. That value is used to decide if the event is dropped, together with the ***X*R** value, which represents the rate for a specific data source ***X***.

In the second iteration where the event A2 is selected both of the dropped rates are 0, so the event can be dropped. In iteration 5 (event C2) the C data source rate is 0 and thus, the event can be dropped. In the next iteration, the Global rate is 0.4, since that 2 events were dropped out of the 5 so far, and the C rate is 0.5 because there were 2 events from data source C and only one had been dropped, so this event can be dropped since 0.5 - 0.1 ≤ 0.4.

The iterations continue and in iteration 11, which corresponds to event A5, we have the first example of the algorithm not dropping a selected event. The A data source rate was 0.75 and the global rate was 0.6, and since (0.75 - 0.1 = 0.65) is not less or equal to 0.6 the event was not dropped.

When the results from both parts are compared, we can notice that in Part **(3)** all data sources got an equal representation of 50%, contrary to what happened before where each data source got a different representation. However, the percentage of dropped events was 50% instead of 61%.

The subtraction of 10% of the rate in (5) is necessary to keep the percentage of dropped events closer to the defined percentage. Without that subtraction, the Library will not drop most of the events, unless they are chosen in ideal conditions (same percentage for each data source). With less 10% the amount of dropped events is closer to the targeted one and the different representations between the data sources are still maintained.

This component uses a model for approximate computation that employs an eventual balance of the data sources' representation. This component could have used a precise balance, however, that would increase the processing cost of each arriving event when deciding if it would be dropped or forwarded to the functions. It trades a total balance of the data sources representation for performance.

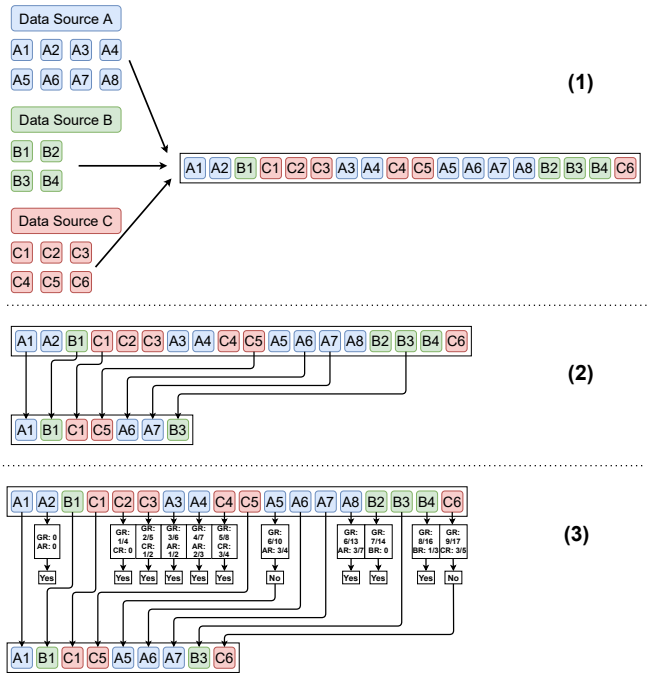

Figure 2. Load Shedding

3.5 Metrics Reporter

This component uses Flink's classes **AbstractReporter**, which allows the system to aggregate the metrics, and **Scheduled** which allows the system to perform the reporter actions with constant intervals, to collect the execution metrics periodically. Both of these classes are part of Flink's Metric System.

By using the Metric System, the Reporter can collect the JVM (resources) and Kafka (requirements) metrics that are used to evaluate the processing. After collecting the metrics it analyses them by comparing their values with the minimum desired values in the requirements if they exist (latency, lag, throughput). It will also verify if the CPU usage or memory utilization is adequate to the quantity of allocated resources. The logic is detailed in Algorithm 2.

The Reporter can vary the execution's minimum accuracy value immediately after analysing the metrics, this way is not necessary to wait for the Middleware decision if the requirements are not being met. This component does not wait for the Middleware because they both analyse the metrics periodically. Even if the period is the same on both, they will likely be desynchronized. This may happen because of the period that it takes for the Flink applications to restart after some of their resources being modified, every time an application is restarted it starts to count the time for the metrics' analysis from zero.

After modifying the accuracy if it was necessary, the Metrics Reporter will send the metrics to the Middleware that will do a more in-depth analysis of the metrics and decide what

the execution resources should be. Depending on the metrics, the accuracy can be maintained, increased, or decreased.

Algorithm 2 Metrics Reporter Pseudo-Code

```
1:  function ReportMetrics
2:      resMetrics ← get resource metrics
3:      reqMetrics ← get requirements metrics
4:      requirements ← get requirements
5:      resRes = AnalyseResources(resMetrics)
6:      if ResourceUsageIsNotOk(resRes) then
7:          LowerAccuracy()
8:      else
9:          reqRes = AnalyseReq(reqMetrics, requirements)
10:         if RequirementsAreNotMet(reqRes) then
11:             LowerAccuracy()
12:         else
13:             if CanIncreaseAccuracy(resRes, reqRes) then
14:                 IncreaseAccuracy()
15:     SendMetrics() Send metrics to the Middleware
```

3.6 Middleware

The Middleware is responsible for managing the applications' executions by deciding the resource allocation and the minimum accuracy. This component has three modules, the Controller Module; the Metrics Analyser Module; and the Communication Module. The Controller Module is responsible for controlling the other two modules. It uses the Communication Module to receive the metrics from Flink and then it sends them to the Metrics Analyser where they are analysed. After that, it receives the results from the Metrics Analyser and it verifies which adjustments are possible to perform (e.g. the results can indicate to increase the parallelism, but it is already at the maximum value). When an adjustment can be done it uses the Communication Module to perform it. Lastly, it will use again the Communication Module to restart the application if that is necessary.

This component analyses the metrics that are described in Section 3.1. It uses the CPU and memory metrics to check if the resources are good, lacking, or are more than necessary. The requirements (lag, throughput, and latency) are verified by comparing the metrics' values with the desired ones.

The analysis is done with percentages (e.g. if a requirement is not being met, it is verified by how much in terms of percentage), so the Middleware knows the quantity of resources that should be allocated.

After all of the metrics are analyzed the Middleware combines their individual results. This combination of results produces a new list of results. The combination of results has the purpose of produce better results, e.g. the analysis of the resources could indicate that there is too much memory reserved, but the requirement results could indicate that one of the requirements is not being met, so it is necessary to increase the parallelism. With the increase of parallelism, the memory usage will likely increase, so in this situation, the combination of results does not decrease the memory.

The adjustments of the available CPU are done with Docker by limiting the time that the containers have access to it. Approxate does not put hard limits on the level of CPU usage that the application can have, we use a soft limit by using Docker's CPU shares. The number of shares that a container has is the level of priority over other containers. This approach was chosen because it allows defining a priority given the conditions of each application. It also allows them to use more CPU time if they need it and it is not being used.

The Middleware can modify the reserved memory for the application. It also changes the amount of memory that the container can access through Docker. Defining a hard limit of memory when using Docker is important because if it is not defined then Docker will continue to use the memory until the system crashes. This component can also adjust the parallelism level of the Flink operators, and the accuracy level of the approximate results.

4 Implementation

This section describes some of the implemented components' aspects. All of them are implemented using Java, which is the language used by Flink and Stateful Functions.

4.1 Approximate Library: Implementation

The Approximate Library contains one class that is used as a wrapper for the events that arrive at the applications. It is a generic class, so it can work with any type of event. This class represents an object and it stores the necessary information for the Approximate Library to use about an event. This class contains 3 fields: a *String* **Id** which is the event identifier; a *String* **Ingress** which is the events' data source identifier; and a generic type **Message** which is the event.

Another class is used to keep track of the percentage of global and data sources dropped events. It does this by keeping a counter of the total number of events and of dropped events for each data source. The other classes are used to select the events that might be dropped randomly based on the accuracy.

This component gets the current accuracy by using the Java Virtual Machines Properties (when the application is started Flink loads the system property that contains the accuracy). After that, it checks if its value is below 100. If the accuracy is below 100 then it will generate a random value, with a **Random** Java util object, between 0 and 100, and if it is greater than the accuracy value, then the event is a candidate to be dropped.

4.2 Metrics Reporter: Implementation

The Metrics Reporter is a Flink plugin. This component uses a class that extends Flink's **AsbstractReporter** and implements Flink's **Scheduled** classes. It is responsible for filtering and analysing the collected metrics by Flink's Metric System.

It uses 2 analysers, one for the metrics generated by the **JVM** that checks the utilization of the available resources, and another for the metrics generated by **Kafka**. The latter loads the user-defined requirements about throughput, latency, and lag through the JVM properties system (the values are loaded by Flink when the application starts) and uses the metrics to check if they are being met or not. The analysers return a value between -1 and 2 for each metric. If it is -1 it means that a requirement is not being met or the system's resources are almost fully utilized and should be increased.

Listing 1. Calculate Result

```
1  private static int calculateResult(String
       property, int curr) {
2      String propertyValue = System.getProperty
           (property);
3      if (propertyValue == null) {
4          return 0;
5      }
6      int max = Integer.parseInt(propertyValue)
           ;
7      int difference = max - curr;
8      if (difference <= 0) {
9          return -1;
10     }
11     return difference > max / 2 ? 2 : 1;
12 }
```

If the value is 1 or 2 it means that the system may have more allocated resources than those that are necessary. If the value is 0 it means that the requirement was not analysed because it was not defined. This component does not increase or decrease the resources but it can increase/decrease/maintain the accuracy. An example of one of the metrics being analysed is in Listing 1 where the metric value is compared to the desired one.

Lastly, it will send the metrics to the Middleware through a **DatagramPacket**, which is a component of Java that is used to represent datagram packets. The datagrams are used to route messages between machines through the network. This is a fast way to send messages without the need of having to establish a connection between the machines, however, there is no delivery guarantee, the packet can get lost in the network.

4.3 Middleware: Implementation

The Middleware is comprised of 3 different modules, the Controller Module which contains the *main* method from the Middleware, this module controls the other two. Next is the Metrics Analyser Module which is responsible for analysing the metrics, and the last module is the Communication Module which is used to communicate with the outside world.

After Middleware starts it reads its configuration file that contains several parameters. They are used to know the Flink's address, docker address, the requirements, among others. After that, it creates a Web Socket that is kept running in a thread in the background. It is used to receive the datagrams sent by the Metrics Reporter.

After the metrics are received, they are sent to the Metrics Analyser Module. After this module returns the results, the Controller Modules uses the Communication Model to do the adjustments. For that, it needs to save the applications' state. It uses a *OkHttpClient*[1] object to send the HTTP requests to Flink's REST API. This class builds the requests according to the Flink API, to do that it creates the necessary JSON objects. To save the job it needs to know the job ID, so to save the state it needs to send a request to receive the job ID, and then it uses the ID to build and send the request to save the state.

Listing 2. Get Container Statistics method

```
1  private Statistics getContainerStats(String
       containerId) throws
       DockerRequestException {
2      Statistics stats;
3      InvocationBuilder.AsyncResultCallback<
           Statistics> callback = new
           InvocationBuilder.AsyncResultCallback
           <>();
4      dockerClient.statsCmd(containerId).exec(
           callback);
5      try {
6          stats = callback.awaitResult();
7          callback.close();
8          return stats;
9      } catch (IOException e) {
10         throw new DockerRequestException("
               retrieve container statistics");
11     }
12 }
```

To get and modify the Docker containers (priority of the application's containers and information to identify which containers belong to the application), the Middleware uses an instance of *docker-java*[2] API. This is an API for Java applications that allows them to send requests to the Docker daemon. They allow performing various commands such as stopping containers, retrieving statistics, change the priority of the resources, and restricting the resources that a container can use. This API uses Docker Engine API, it converts the requests made in Java to requests that the Docker Engine's API can accept and understand. In Listing 2 is an example of one request that is made with the **DockerClient** to retrieve the statistics about one container.

[1] https://square.github.io/okhttp/4.x/okhttp/okhttp3/-ok-http-client/
[2] https://github.com/docker-java/docker-java

Another way of communicating with Docker that is used to restart the containers is through the **Docker Compose**. This is done with the Java **Runtime** class which interacts with the operating system to call the Docker Compose tool. The Docker Compose is used because the characteristics of the tested applications' containers are defined in this format.

The Middleware also modifies the Flink and Docker configuration files that are used in each application to adjust the resources and accuracy. Although Flink allows multiple applications running in the same cluster, the Middleware can only analyse the resources and requirements of one application at a time, so it cannot be used with clusters that support multiple applications.

5 Evaluation

5.1 Setup

Approxate was evaluated in local and in cloud setups[3]. The cloud machines had different resources (16/8/4 vCpus with 64/32/16 GB of memory). We choose the different machines so we could illustrate how the system behaves in different scenarios where it has access to different resources. They also show how approximate computation can be used to improve the performance level in lower-end machines to get near, or match, the performance of higher-end machines.

5.2 Metrics

In the following list are described the metrics that are used to evaluate the impact and performance of Approxate when compared to vanilla Stateful Functions:

- **Accuracy**: Approxate must be able to utilize approximate computing to lower the accuracy of the results in exchange for a performance improvement however, the results should still be acceptable;
- **Scalability**: Approxate must allow the applications to scale-up and scale-down according to their load and the user-defined requirements;
- **Processing Time**: Approxate must take less time to process the same dataset with the same resources;
- **Throughput**: Approxate must be able to process more data in the same time with the same resources;
- **Resource Utilization**: Approxate must be able to process the same dataset with fewer resources with the same time;
- **Resources' Overhead**: Approxate's overhead should not have a significant impact on the amount of used resources;
- **Cost-Benefit**: In cases where it is not possible to improve any of the metrics above, Approxate should not impact them negatively significantly. In cases where it can improve the overhead of Approxate should be less than the performance gains.

[3]https://cloud.google.com/compute

5.3 Benchmarks

We tested Approxate with micro-benchmarks (simple Flink tests) to check the added overhead and with macro-benchmarks (realistic applications workloads).

There are two micro-benchmarks, the **greeter** and the **ad-processing**. The greeter counts the number of messages that are sent by each user and replies to it. The messages are generated with random user-ids. The ad-processing receives events that indicate if a user clicked in an ad. The application calculates the ratio of users that clicked in each ad and how many times a user clicked in each ad. These are some of the more simple stream processing applications that can be done and their purpose is to check if the solution's components affect the performance negatively.

There are 6 macro-benchmarks, 3 of stream processing and the other 3 of graph processing. The **Taxi-Trip Benchmark** uses real data from trips in New York[4] from 2 different companies and calculates various averages values from the trips like the number of trips per weekday, the money earned per day for each month, the average distance of the trips per weekday, among others.

The **Linear Road Benchmark** uses synthetic data[5] that is simulating a variable toll system in four highways. It processes information about the vehicles that are traveling through the highways and it calculates the accidents that happened, the tolls that each vehicle passed by, and it also uses historical information to predict how long it will take to travel through the segments of the highway based on the weekday and the hour.

The **Synthetic Benchmark** uses randomly generated synthetic data and simply applies a load for each received event that consists of creating and shuffling a list.

Two of the graph benchmarks (the **Yahoo Groups and Messenger**) use real data from Yahoo Groups and Messenger[6] and consist of finding communities in the graphs. The other graph benchmark is **Triangle Counting** and uses synthetic data[7] to calculate the number of triangles in a graph.

5.4 Results

This section contains the results of 2 types of tests. The first consists of having the application use the Approximate Library to see how much performance increases it is possible to achieve and how that affects the results' precision. The second type of test has a variable input rate to test how Approxate handles that by adjusting the resource allocation.

The **micro-benchmarks** results showed that the Approximate Library with 100% of accuracy increased the processing time by 2% which is mostly negligible. However, just dropping 1% of events led to at least 4% of performance improvement.

[4]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[5]https://www.cs.brandeis.edu/ linearroad/index.html
[6]https://webscope.sandbox.yahoo.com/catalog.php?datatype=g
[7]http://graphchallenge.mit.edu/data-sets

Table 1. Taxi-Trip: Comparison between machines

Resources	Events (%)	Time	Precision(%)	Time (%)
16 / 64	100	06:21	100	100
8 / 32	70	08:32	74	134
4 / 16	50	21:05	51	332

The results from the **taxi-trip benchmark** with 10% of the events dropped achieved 91% of precision using only 71% of time. By dropping 50% of the events, it only needed 45% of the time, and retained 52% of precision. The precision got a close relationship with the percentage of dropped events since these tests were calculating averages about values of a fixed period (the calculated values were always divided by the same amount). Due to the nature of this benchmark, we can get a result with a precision of around 96%, by using (6). The results from the **linear-road benchmark** demonstrate that Approxate could save 27% of the time and remain with 88% of precision when dropping 30% of the events.

$$Value = \frac{ObtainedValue * 100}{Accuracy} \tag{6}$$

The tests performed in the cloud show that lower-end machines can get an increase in performance that can get close to the performance of higher-end machines. Table 1, where the *Resources* are the quantity of vCpus and memory in gigabytes respectively, contains the results from the taxi-trip benchmark performed in 3 cloud machines. We can see that the middle-machine got 74% of precision and needed 134% of the time when compared to the better machine, however, this machine has half of the resources of the higher-end machine, so with 50% of the resources it only needed 34% more time instead of 50% more. The lower-end machine with a quarter of the resources and dropping half of the events processed in 332% of the time which demonstrates that with 25% of the resources it needed more 232% of time instead of 300% more.

The second type of test with variations on the input rate tests how the system adapts the resources and accuracy to the variations to meet the requirements. These were performed with the **taxi-trip benchmark** and the **synthetic benchmark** applications. In Figure 3 we have a running of the taxi-trip local test that shows how Approxate managed the resources, the scalability, and the accuracy variation. Those results were similar in the different applications.

We can observe that the system responded to the increase of events arriving by increasing the memory, the parallelism, and lowering the accuracy. In minute 2 the accuracy dropped from 100 to 70 (the minimum value) and it was caused by the Metrics Reporter, the same thing happened in minute 4, and in minute 10.

Between minutes 6 and 9 there was no need of lowering the accuracy. Then in minute 10, there was an increase of events arriving at the application, which increased the memory, the

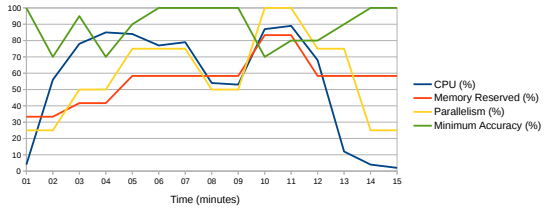

Figure 3. Taxi-Trip Benchmark: Resources Variation

parallelism, and reduced the accuracy. In the minute after, it got stable, so the accuracy started to increase. In minute 12 the parallelism was reduced, and then in minute 14 it was reduced to the minimum value, while the accuracy increased to the maximum. The memory stayed stable in the last 4 minutes. In the end, when no more events were arriving, the memory utilization did not fell enough to lower the reserved memory. This happened because of the historical data that this application keeps. This result shows that Approxate can identify the necessary resources to adjust, the different resources were adjusted at different rates.

The **Yahoo Groups benchmark** got a maximum of 61% of time saved with 50% of dropped events while keeping a minimum precision of 75%. With 30% of dropped events, it achieved a minimum precision of 87% and needed a maximum of 70% of the time depending on the used graph and machine. With the cloud tests, the middle cloud machine got a similar processing time using 90% of the events as the higher-end machine processing all of the events, and still kept a precision of 97%.

The **Yahoo Messenger benchmark** got similar results as the Yahoo Groups when using graphs with medium/high density. However, one case with a high-density graph lost 26% of precision with only 5% of dropped events. Although using approximate computation could lower the necessary time, this type of test (community counting) with the tested high-density graph has too much loss of precision.

The **Triangle Counting benchmark** using a very low-density graph got 100% of precision in 86% of the time. This benchmark with a high-density graph got 78% of precision in 47% of the time with 30% of the events dropped. Dropping 10% led to a time saving of 5% but with 98% of precision.

5.4.1 Summary. Figure 4 shows the effect of using approximate results in all tested applications. These results are the averages. We can see that the percentage of saved time is always greater than the loss of precision. On average, we can get 78% of precision and save 36% of time. It is also possible to save 21% of time and only lose 9% of precision.

The results show that Approxate allows a variable accuracy in the results, the user can choose to trade-off accuracy for performance, which can allow the applications to improve the performance in some cases to around 50% and still get meaningful results. Approxate can scale up and scale down

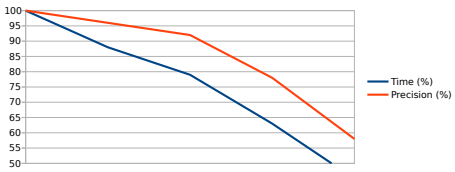

Figure 4. Time and Performance Relationship

the applications based on the defined requirements about lag, latency, and throughput, and also based on the current load. It can reduce the time that is needed to process the data. They also show that the performance gains are greater than the overhead. They show that all of the evaluation's metrics were achieved.

6 Conclusion

This work contains a proposal and implementation of an extension to be used with Stateful Functions for stream and graph processing. It adds an intelligent and variable resource management that will vary the allocation of the resources based on the state of the execution and the desired user requirements (latency, lag, throughout). Approxate is capable of approximate computation, of improving the resources' allocation, scalability, and performance. It can also vary the level of accuracy if that is necessary to meet the requirements, while keeping the results meaningful.

The benchmarks show that with Approxate it is possible to have lower-end machines processing the same dataset in times close to the ones of higher-end machines. They also show that Approxate can save up to 50% of the processing time and remain with acceptable results.

6.1 Future Work

To improve stream processing, the Middleware can be adjusted to analyse the metrics of multiple applications' containers, including where Kafka is running. Another way to improve Approxate is to convert the Approximate Library to work directly in the Kafka Broker in situations where it knows the data sources. That would avoid the events being transferred through the network to the application where they are dropped.

Another way is to adapt the Library for specific types of events, instead of a generic type. By adapting for a specific event type (and its expected value distribution) we could extract and process some information of the event instead of discarding it completely. Lastly, Approxate could be adapted to manage multiple applications.

References

[1] Ankur Agrawal, Jungwook Choi, Kailash Gopalakrishnan, Suyog Gupta, Ravi Nair, Jinwook Oh, Daniel A Prener, Sunil Shukla, Vijayalakshmi Srinivasan, and Zehra Sura. 2016. Approximate computing: Challenges and opportunities. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 1–8.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1890–1893.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[4] Miguel E. Coimbra, Alexandre P. Francisco, and Luís Veiga. 2021. An analysis of the graph processing landscape. *J. Big Data* 8, 1 (2021), 55. https://doi.org/10.1186/s40537-021-00443-9

[5] Sérgio Esteves, Helena Galhardas, and Luís Veiga. 2018. Adaptive Execution of Continuous and Data-intensive Workflows with Machine Learning. In *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, Paulo Ferreira and Liuba Shrira (Eds.). ACM, 239–252. https://doi.org/10.1145/3274808.3274827

[6] Sergi Esteves, Nico Janssens, Bart Theeten, and Luis Veiga. 2017. Empowering stream processing through edge clouds. *ACM SIGMOD Record* 46, 3 (2017), 23–28.

[7] Sérgio Esteves and Luís Veiga. 2016. WaaS: Workflow-as-a-Service for the Cloud with Scheduling of Continuous and Data-Intensive Workflows. *Comput. J.* 59, 3 (2016), 371–383. https://doi.org/10.1093/comjnl/bxu158

[8] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2013. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1447–1463.

[9] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 1–6.

[10] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316.

[11] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–33.

[12] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. Streamapprox: Approximate computing for stream analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 185–197.

[13] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. 2017. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)* 17, 3 (2017), 228.

[14] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1, 3-4 (2016), 145–164.

[15] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *ACM Sigmod Record* 34, 4 (2005), 42–47.

[16] Khin Me Me Thein. 2014. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research* 3, 47 (2014), 9478–9483.

[17] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.