Stateful Adaptive Streams with Approximate Computing and Elastic Scaling

João Francisco
INESC-ID Lisboa
Instituto Superior Técnico, ULisboa
Lisboa, Portugal

Miguel E. Coimbra
INESC-ID Lisboa
Instituto Superior Técnico, ULisboa
Lisboa, Portugal

Felix Freitag
Universitat Politècnica de Catalunya
BarcelonaTech (UPC)
Barcelona, Spain

Pedro F. Neto
INESC-ID Lisboa
Univ. do Estado do Rio Grande do Norte
Lisboa, Portugal and Mossoró, Brazil

Luís Veiga
INESC-ID Lisboa
Instituto Superior Técnico, ULisboa
Lisboa, Portugal

ABSTRACT
The model of approximate computing can be used to increase performance or optimize resource usage in stream and graph processing. It can be used to satisfy performance requirements (e.g., throughput, lag) in stream processing by reducing the effort that applications need to process datasets. There are currently multiple stream processing platforms, and most of them do not natively support approximate results. A recent one, Stateful Functions, is an API that uses Flink to enable developers to easily build stream and graph processing applications. It also retains Flink’s features like stateful computations, fault-tolerance, scalability, control events and its graph processing library Gelly. Herein we present ApproXate, an extension over this platform to support approximate results. It can also support more efficient stream and graph processing by allocating available resources adaptively, driven by user-defined requirements on throughput, lag, and latency. This extension enables flexibility in computational trade-offs such as trading accuracy for performance. The user can choose which metrics should be guaranteed at the cost of others, and/or the accuracy. ApproXate incorporates approximate computing (using load shedding) with adaptive accuracy and resource management in state-of-the-art stream processing platforms, which are not targeted in other relevant related work. It does not require significant modifications to application code, and minimizes imbalance in data source representation when dropping events.

KEYWORDS
Adaptive Stream Processing, Approximate Computation, Stateful Functions, Apache Flink

1 INTRODUCTION
More data is being generated than ever, and stream processing platforms play an important role in handling it. Stream processing consists in processing data items (e.g., tuples, events) that continuously arrive at an application. These events can be processed as soon as they are created or they can be stored and later processed as batches [1]. Data processed from the stream can generate results and insight to be used at a later time, or it can be used to generate new streams of data for other purposes. The data arriving from a stream may represent diverse events such as the creation of connections in social networks or bank transactions. These events are related to different areas of human activity, entailing specific approaches to optimize their processing. Some cases may benefit from prioritizing lower latency instead of maximizing accuracy, while others gain from a focus on lower resource consumption.

Since the input data rate of a stream can fluctuate, the scalability of the stream processing platform is relevant. If the incoming data rate is lower than expected, the resources used by the platform may go underutilized. If the incoming data rate is higher than what the platform is able to compute, this may result in a growing queue of events that remain unprocessed. By allowing the system to scale up or scale down when necessary, resource efficiency will increase and the amount of wasted resources will diminish. However, scaling should be efficient: slight variations in the input data rate should not trigger a scaling process. If the scaling is not done in an optimal way it can also lead to wasted resources.

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 [2]. Those can be affected by many factors, 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, and in such a situation, the
other operator must share its data. If the operator that is sharing its state writes it to 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 [3]. For state sharing to be an efficient operation, it cannot utilize persistent storage. Access to persistent storage increases operation latency and decreases throughput. Performance may also be increased by using approximate computing, a characteristic that not many platforms support. It is a computation model that reduces the result precision to decrease system load and/or increase performance. This can be achieved at both the hardware and software levels [4–6].

In summary, a stream processing platform should have: efficient scaling; data consistency; fault-tolerance; low latency and high throughput; the ability of operators to share their state efficiently without costly accesses to persistent storage; and the ability to perform approximate computation to improve performance if that is necessary. Herein we present Approxate, designed to extend the Stateful Functions of Flink (that already offers several such properties) with abilities it is still lacking: to use approximate results; and to dynamically adjust the allocation of its resources based on user-defined requirements.

We present in the next sections the architecture of Approxate, following with its evaluation. The remainder of the paper addresses relevant related work and we then conclude with the key insights of this work.

2 ARCHITECTURE

Approxate is an extension of Stateful Functions that manages stream and graph processing applications. The applications use Kafka [7] as the data-broker to get the events and then to send their results. The applications also run in containers using Docker [8]. It allows the user to define requirements for lag (number of produced events that are not yet processed), throughput (number of results being produced per unit of time) and 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, the level of parallelism and the minimum accuracy (as an estimation), 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 necessary it can use approximate computing to improve the performance while maintaining results meaningful.

\[
\text{Accuracy} = \frac{\text{ProcessedEvents}}{\text{TotalEvents}} \quad (1)
\]

Approxate is composed of three components. The Approximate Library (Section 2.1) which is responsible for receiving the events and then deciding based on the accuracy level if they are to be processed or dropped (load shedding). The second component is the Metrics Reporter (Section 2.2) which collects execution metrics, verifies if the application is under load and/or meeting the requirements, and reduces the accuracy if necessary. After that, it sends the metrics to the final component, the Middleware (Section 2.3), which receives the metrics, analyses them extensively and then decides if it should adjust the application’s resources, parallelism level and accuracy (see Equation 2).

\[
\text{NewRate} = \text{Rate[Adjusted]} - \text{Rate[BeforeAdjust]} \quad (2)
\]

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. The restart is usually fast (few seconds to a minute depending on how fast Flink takes and restores snapshots), and for the resource adjustments to be worth undergoing the restart time, it must be less than the time saved by adjusting the requirements. The formula to calculate the saved time is shown in Equation 3. Saved time 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 to restart.

\[
\text{TimeSaved} = \text{Events} \times \text{NewRate} - \text{RestartTime} \quad (3)
\]

In Figure 1 there is a system overview depicting 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 sends 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.

Flink. Stateful Functions [9] uses Flink [10], 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 Java Virtual Machine (JVM) and indicates the CPU load 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 event 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 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.

Recall that the latency value of the application (full amount of time that an event takes to travel from the producer to the consumer inside the application) is not being utilized, because its calculation requires writing custom code for each application.

Kafka [7] 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 (this represents the 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.

Docker [8] is a platform that allows the user to run applications in containers and to limit resources for each. It offers controls of 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.

2.1 **Approximate Library**

This component is used inside Approximate 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. 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 an 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 data source representation in the results. To do this, the Approximate Library registers the number of dropped events and the total number of events (Equation 4) globally and for each data source (this is represented in Algorithm 1).

\[
Rate = \frac{\text{Dropped Events}}{\text{Total Events}}
\]  

**Algorithm 1 Event selector.**

1. function **Invoke**
2. \( event \leftarrow \text{received event} \)
3. \( accuracy \leftarrow \text{get accuracy} \)
4. if \( accuracy < 100 \) then
5. \( \text{if } \text{randomSelect}(accuracy) == \text{true} \text{ then} \)
6. \( \text{if } \text{EventCounter.canSkip()} == \text{true} \text{ then} \)
7. \( \text{skip}(\text{event}) \)
8. \( \text{return} \)
9. end if
10. end if
11. end if
12. forward(\text{event})
13. end function

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 Figure 2-(1). After they arrive some events are randomly selected to be dropped (A2, C2, C3, A3, A4, C4, A5, A8, B2, B4, and C6).

In Figure 2-(2) the load shedding is done without verifying data source representation, 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. Source B has 2/4 (50%) and C has 2/6 (33.3%). This imbalance could affect not only result precision but also break application semantics.

In Figure 2-(3), load shedding is performed with the algorithm. The events that were randomly selected to be dropped must verify the condition of Equation 5.

\[
Rate(\text{DataSource}) - 0.1 \leq Rate(\text{Global})
\]  

The subtraction of 10% of the rate in Equation 5 is necessary to keep the percentage of dropped events closer to the defined percentage. Without that subtraction, the Approximate 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.

The GR is the current global rate Equation 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 XR value, which represents the rate for a specific data source X.

In the second iteration where 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 Figure 2-(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%.

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 the total balance of data source representation for performance.

### 2.2 Metrics Reporter

This component uses the following Flink classes: **AbstractReporter**, which allows the system to aggregate the metrics; and **Scheduled** which allows the system to report in 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 minimum accuracy value immediately after analysing the metrics. This way it 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 application to restart after some of their resources are modified, every time an application is restarted it begins counting the time for the analysis of metrics from zero.

After modifying the accuracy (if 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:        end if
16:      end if
17:   end if
18:   SendMetrics() Send metrics to the Middleware
19: end function
```
2.3 Middleware

The Middleware is responsible for managing the execution of applications 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 Module for analysis. After that, it receives the results from the Metrics Analyser Module and verifies which adjustments are possible to perform (e.g., the results may suggest an increase of parallelism when 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.

It uses the CPU and memory metrics to check if the resources are sufficient, lacking or in excess. The requirements (lag, throughput, and latency) are verified by comparing the values of the metrics with the desired ones. The analysis is done with percentages (e.g., if a requirement is not being met, the gap is verified in terms of percentage), so the Middleware knows the quantity of resources that should be allocated. After all of the metrics are analysed, the Middleware combines their individual results. This combination of results produces a new list of results with the purpose of optimizing decisions to achieve better results. For example, the analysis of the resources could indicate that there is too much reserved memory, but the requirement results could indicate that one of the requirements is not being met, suggesting the need to increase parallelism. With the increase of parallelism, the memory usage will likely increase, and so in this situation, the combination of results avoids unnecessary intermediate actions such as decreasing memory only to find it must be raised immediately after.

The adjustments of available CPU are performed with Docker by limiting the time that containers have to access it. Approximate does not place hard limits on the level of CPU usage that applications can have. Instead, a soft limit is imposed 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 enables the definition of a priority given the conditions of each application. It also allows applications to use more CPU time if they need it and the CPU is not being used.

The Middleware can modify the memory reserved 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.

**Implementation Details.** Following, we describe some of the implemented components’ aspects. All of them are implemented using Java, which is the language used by Flink and Stateful Functions.

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 three fields: a String **Id** which is the event identifier; a String **Ingress** which is the data source identifier of the events; 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 accuracy. This component gets the current accuracy by using the Java Virtual Machine 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 java.util.Random object, between 0 and 100, and if it is greater than the accuracy value, then the event is a candidate to be dropped.

The Metrics Reporter is a Flink plugin. This component uses a class that extends Flink’s AbstractReporter and implements Flink’s Scheduled classes. It is responsible for filtering and analysing the collected metrics by Flink’s Metric System. It uses two 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 J WM 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 resources are almost fully utilized and should be increased. 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 or maintain the accuracy.

Lastly, it will send the metrics to the Middleware through a DatagramPacket, which is a component of Java 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.

The Middleware is comprised of three different modules. It has a Controller Module which contains the main method from the Middleware and also controls the other two modules. 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 the metrics are received by the middleware, they are sent to the Metrics Analyser Module. After this module returns the results, the Controller Module uses the Communication Module to do the adjustments. For that, it needs to save the applications’ state. It uses an OkHttpClient object to send the HTTP requests to Flink’s REST API. This class builds the requests according to the Flink API, and to do that it creates the necessary Java 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.
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-\textit{java} API. This is an API for Java applications that allows them to send requests to the Docker daemon. It offers various commands such as stopping containers, retrieving statistics, changing the priority of the resources and restricting the resources that a container can use. This API uses Docker Engine API, which converts the requests made in Java to requests that the Docker Engine’s API can accept and understand.

The Middleware also modifies the Flink and Docker configuration files that are used in each application to adjust resources and accuracy. Although Flink allows multiple applications to be 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 to globally manage multiple applications running in the same cluster.

3 EVALUATION

Setup. Approxate was evaluated, initially in local (4 vCPUs with 16 GB memory) and later in cloud\footnote{https://cloud.google.com/compute} setups (machines with 4/8/16/64 vCPUs and with 16/32/64/256 GB of memory). We chose the different machines to illustrate how the system behaves in different scenarios where it has access to different resources. The smaller machines used in local and cloud resources show how approximate computation can be used to improve performance in lower-end machines (such as in edge computing or IoT domains), to achieve matching or similar performance of higher-end machines (without approximation), when time and/or cost are at premium, illustrating how nonetheless relevant accuracy can be achieved. The higher-end machines employed in the cloud are used to assess how Approxate is able to scale when more resources are made available, to illustrate that the gains of approximate computation do not erode as more resources become available (still using affordable machines in the typical server ranges).

Metrics. The following list describes the metrics that are used to evaluate the impact and performance of Approxate, when compared to unmodified 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, e.g., for decision making in scenarios where absolute accuracy is not required;
- **Scalability**: Approxate must allow the applications to scale-up and scale-down according to their load and the user-defined requirements regarding resources and accuracy;
- **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 Usage**: Approxate must be able to process the same dataset with fewer resources within 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 in a significant way. In cases where it can improve, the overhead of Approxate should be less than the performance gains.

**Benchmarks.** Approxate was tested with micro-benchmarks to check the added overhead, and with macro-benchmarks to assess behaviour with 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 them. The messages are generated with random user-ids. The ad-processing receives events that indicate if a user clicked on an ad. The application calculates the ratio of users that clicked on each ad and how many times a user clicked on 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, three on stream processing and three on graph processing. The Taxi-Trip Benchmark uses real data from trips in New York from two different companies to calculate various average statistics (frequency, duration, cost) from the trips. Linear Road Benchmark uses synthetic data that is simulating a variable toll system in four highways. It processes information about the vehicles that are travelling through the highways, calculating accidents, adjusting tolls and predicting future traffic. 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 and consist of finding communities in the graphs. The other graph benchmark is Triangle Counting and uses synthetic data to calculate the number of triangles in a graph.

3.1 Results

This section contains the results of tests to assess (given the space constraints) three main aspects: i) limited overhead; ii) adaptability - how variable input rates are handled by Approxate when adjusting the resource allocation; iii) cost-benefit - how benchmark applications use the Approximate Library to increase performance and how that affects result precision. Precision for each application is calculated as the ratio between the obtained results, when dropping a given percentage of events (based on the accuracy parameter), and the original results when processing 100% of the events. Thus, a given selection of accuracy (estimation) that indicates the amount of events to drop may lead to different variations in result precision for different applications and input data.

**Limited Overhead.** The results from the micro-benchmarks are in Table 1, and we can observe that the overhead of Approxate was 2%. These values include the Metrics Reporter collecting and analysing the metrics and also the Approximate Library overhead in the Stateful Functions application. The Middleware is running outside of Stateful Functions but also consumes resources; however, it stays mostly idling and thus it did not have a noticeable impact.
Table 1: Micro-Benchmarks - Overhead (Local)

<table>
<thead>
<tr>
<th>Test</th>
<th>Time(min.)</th>
<th>Time(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greeter w/o Approxate (100% Acc.)</td>
<td>13:55</td>
<td>100</td>
</tr>
<tr>
<td>Greeter w/ Approxate (100% Acc.)</td>
<td>14:16</td>
<td>102</td>
</tr>
<tr>
<td>Greeter w/ Approxate (99% Acc.)</td>
<td>13:21</td>
<td>96</td>
</tr>
<tr>
<td>Ad-proc. w/o Approxate (100% Acc.)</td>
<td>06:22</td>
<td>100</td>
</tr>
<tr>
<td>Ad-proc. w/ Approxate (100% Acc.)</td>
<td>06:29</td>
<td>102</td>
</tr>
<tr>
<td>Ad-proc. w/ Approxate (99% Acc.)</td>
<td>05:55</td>
<td>93</td>
</tr>
</tbody>
</table>

The tests with 99% of accuracy show the performance impact of dropping 1% of the events in two of the most simple stream processing applications, where the necessary time to process each event is one of the lowest. In the more simple test 4% of time was saved (Table 1 row three), while in the more complex was 7% (Table 1 row six). These results demonstrate that the overhead of Approxate is mostly negligible in situations where it cannot improve the performance.

Adaptability. We tested Approxate’s adaptability by introducing variations on the input rate to test how the system adapts (resources and accuracy) to the variations in order to meet the requirements. These were performed with the taxi-trip benchmark and the synthetic benchmark applications. In Figure 3 we highlight the execution of the taxi-trip (local) test that shows how Approxate managed the resources, the scalability and the accuracy variation. Results were similar across all the applications. We explain the timeline of taxi-trip in detail.

We can observe that the system responded to the increase of arriving events by increasing memory, parallelism and lowering the accuracy. In minute 2, the accuracy dropped from 100 to 70 (the minimum value) due to the Metrics Reporter, with the same thing happening in minutes 4 and 10. Between minutes 6 and 9 there was no need to lower the accuracy. Then in minute 10, there was an increase of events arriving at the application, which increased the memory and parallelism and reduced the accuracy. In the minute after, it became 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 decrease 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, and that the different resources were adjusted at appropriate rates.

Cost-Benefit. Globally, the tests performed in the cloud show that lower-end machines can achieve a performance increase that can get close to the performance of higher-end machines. We illustrate this with an example in Table 2. Here, the Resources are the quantity of vCPUs and memory in gigabytes respectively. It shows the results from the taxi-trip benchmark performed in three different types of cloud machines.

Table 2: Taxi-Trip: comparison between machines.

<table>
<thead>
<tr>
<th>vCPUs/GB</th>
<th>Events(%)</th>
<th>Time</th>
<th>Precision(%)</th>
<th>Time(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 / 64</td>
<td>100</td>
<td>06:21</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>8 / 32</td>
<td>70</td>
<td>08:32</td>
<td>74</td>
<td>134</td>
</tr>
<tr>
<td>4 / 16</td>
<td>50</td>
<td>21:05</td>
<td>51</td>
<td>332</td>
</tr>
</tbody>
</table>

We can see that the middle-machine got 74% 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% additional time instead of twice (albeit the precision loss whose adequacy should be decided for each application). The lower-end machine with a quarter of the resources and dropping half of the events completed processing in 332% of the time, which illustrates that with 25% of the resources it needed 232% additional instead of 300% (once again, only relevant if the precision achieved is application adequate).

While naturally being application dependent, Approxate can enable the use of lower-end devices, more common in IoT and service placement at the edge (e.g.,[11]), by achieving gains following a cost-benefit analysis, i.e., the relative improvement in performance surpasses the relative penalty in precision. If precision penalties are kept to acceptable values, regarding a decision making criteria (that is application-specific), smaller, cheaper and more energy efficiency...
devices can be employed and produce acceptable results within desired time constraints.

Figure 4 shows the percentage of events processed (i.e., the accuracy parameter, and consequent resource usage), the result precision (%) obtained, and the relative execution time (%) for different datasets (in its sub-figures). Linear Road (depicted in Figure 4a and 4b) shows that precision can be kept at 70% and over 85% when times are reduced to 50% and 60% (LR-local). Cloud execution in the most powerful machines shows little relative gains as workload is executed much faster (LR-cloud). The Yahoo Messenger benchmark (depicted in Figure 4c and 4d) achieved similar results as the Yahoo Groups when using graphs with medium (YM-56444-local) and high density (YM-1520005-local). However, the high-density graph lost 26% of precision with only 5% of dropped events (illustrating how a given parameter of accuracy may cause significantly different outcomes in result precision, for different types of applications, e.g., arithmetic aggregations vs. this specific type of graph processing). Thus, 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 (that could break application semantics). The Triangle Counting benchmark (depicted in Figure 4e), however, also using a high-density graph (Triangle high density) 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.

Figure 5 presents results obtained with the cloud machines, following the same structure as previously. The results from the Taxi-Trip benchmark (depicted in Figure 5a and 5b) show that, at best, with 10% of the events dropped, the precision achieved was 91%, using only 71% of time. By dropping 50% of the events, it only needed 45% of the time, and retained 52% of the precision. Naturally, the precision exhibited a close relationship with the percentage of dropped events since these tests calculated averages on values of a fixed period (the calculated values were always divided by the same amount). The cost benefit improved with more capable machines. The Yahoo Groups benchmark (depicted in Figure 5c and 5d) achieved a maximum of 61% of saved time with 50% of events dropped 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. Here, cost benefit improvements permeate all types of machine used.

These results show that on average, the percentage of saved time is always greater than the loss of precision. They show that Approximate supports variable accuracy goals, allowing the user to trade-off result precision (application dependent) for performance, which can lead to applications improving the performance in some cases to around 50% while maintaining meaningful results. On average, 36% of time may be saved by reducing result precision to 78%. It is also possible to save 21% of time and only lose 9% of precision. Approximate can scale up and scale down applications based on defined requirements about lag, latency, 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.

4 RELATED WORK

In this section we present an overview of the characteristics and design of relevant stream and graph processing platforms as well as some works on approximate computing and resource efficiency in data processing.

Stream Processing Systems. Spark [12] 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 as the core data abstraction its Resilient Distributed Datasets (RDDs) [13], which are read-only partitioned collections of records. 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. Spark can also work in
The operators also have a state which they can share. This platform can be recovered later in case of a failure. The state is periodically saved at checkpoints (also known as snapshots). These contain the state of the stream in order to avoid processing the same events multiple times. 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, the last correct state that was checkpointed may be retrieved, avoiding the need for re-execution.

**Stateful Functions** [9] is an API that utilizes Flink and simplifies the process of building low-latency distributed stateful applications. It has the benefits of Flink like control events, fault-tolerance, scalability, operator 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 [7] as a data broker to receive the events that the applications process and to send the results. It can thus be regarded (together with Gelly) as a state-of-the-art platform for low-latency data-intensive applications.

None of these widely used and deployed platforms have built-in support for adaptive load shedding. Thus, not allowing applications to provide fresh results even in peak load situations, as they are only designed to scale up (further) resources as needed, never addressing the amount or rate of data ingested.

**Approximate Computation.** Approximate computation is a 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 [4, 15]. One method of approximate computing is load shedding [6], where some of the input events are dropped when the system is overloaded. With load shedding it is possible to lower the accuracy by dropping some events instead of processing them. None of them address platforms such as Stateful Functions and Gelly, and none of them address preserving an eventual balance of data source representation in the results, when dropping events.

Besides dropping events, delaying event (or update) processing can also avoid work, save resources and improve performance, by having newer recent data overwriting or cancelling out older, still unprocessed, data. However, data pending processing introduces staleness and not all data may have the same importance w.r.t the relevance of the updated results. Quality-of-Data models have been proposed to monitor the individual, and cumulative, impact of data pending processing in order to bound errors, while attempting to save resources and improve performance in continuous workflows [16] and graph processing [17]. But these either address only workflows and not actual stream processing systems, or they are unable to adaptively adjust accuracy and resource usage.

There are other techniques of approximate computation [5, 15] like loop perforations, approximation of arithmetic computations, approximation of communication between computational elements and precision scaling, customized half-precision in graphs [18], among others, but these cannot always be transparently applied to any type of arithmetic operator or user-defined functions in stream processing.

**Resource Efficiency.** In big data analytics [19], several works can be found regarding resource efficiency in large scale data processing clusters. Typical approaches include determining ideal configurations for applications, such as application parallelism [20], and

---

**Figure 5:** Performance of Taxi-Trip (left), Groups (right), increasing #CPU cores and RAM.
recommend them to the users, avoiding resource over-estimation, or with automatic parameter tuning [21]. Additionally, there are those employing game theory to drive container allocation that host streaming applications [22], employing Markov models to optimize the triggering of execution steps in workflows [23], or proposing novel intermediate data structures to accelerate big-data workloads such as multi-label classification [24]. However, none of them actually leverages load shedding as a mechanism to attempt to enforce the QoS of applications dynamically and adaptively in situations of overallocation or limited resources.

5 CONCLUSION

This work presents a proposal and implementation of an extension to be used with Stateful Functions for stream and graph processing. Approxate incorporates approximate computing (using load shedding) with adaptive accuracy and resource management in state-of-the-art stream processing platforms, for distributed stateful applications (Stateful Functions) and graph processing (Gelly), that currently lack it, and are not targeted by other related work. It achieves this without requiring significant modifications to application code, adapting automatically, and aiming at preserving balance of data source representation when dropping events.

Specifically, Approxate adds adaptive resource management that will vary resource allocation based on the state of execution and desired user requirements (latency, lag and throughput). Approxate is capable of performing approximate computation, of improving resources allocation, scalability and performance. It can also vary the level of accuracy if necessary to meet requirements while still producing meaningful results. The benchmarks show that with Approxate it is possible to have lower-end machines processing the same dataset in times close to those of higher-end machines.

As future work, to improve stream processing, the Middleware can be adjusted to analyse the metrics of the containers of multiple applications, 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. Finally, the tradeoff between checkpoint/restore delay and time savings due to adjustment could be explored further. This could involve employing machine learning techniques over application execution profiles [25] and estimate how result precision could be improved, by applying modifications to the accuracy parameter or resource quotas, and determine whether that would compensate the performance penalty of each reconfiguration.

Acknowledgement: This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/00521/2020.

DL 60/2018, de 3-08 – Aquisição necessária para a atividade de I&D do INESC-ID, no âmbito do projeto SmartRetail (02/C-0501/2022).

This work was partially supported by the Spanish Government under research contracts PID2019-106774RB-C21 and PCI2019-111850-2 (DiPET CHIST-ERA).

REFERENCES


