TÉCNICO LISBOA

Approxate: Stateful Functions for Approximate Stream Processing

João Paulo da Silva Francisco

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Helena Isabel De Jesus Galhardas

November 2021

Abstract

Approximate computing is a computing model that can be used to increase performance or use fewer resources in stream and graph processing. 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. There are applications where a function (e.g. executing as a part of a dataflow, or data pipeline) needs data that is a result of an operation made by another function, and it does not have a way of receiving the data directly from the other. Instead, the other function needs to write that data into persistent storage like a database, and then the data needs to be read from there. This causes an issue with performance (high latency and/or low throughput) and resource efficiency (wasted resources to write and read the data from disk) due to the I/O bottleneck, and also increases the cost for the user because the application needs to do more requests to the storage. Stateful Functions is a platform that uses Flink, and that addresses this by allowing message exchange between functions. This document proposes the design and implementation of an extension to be used with Stateful Functions. It can support more efficient stream processing by allocating the available resources intelligently and variably and also using approximate results. To validate the results, the performance of the extension was evaluated using benchmarks with real and synthetic data, running locally and in cloud machines with typical stream processing applications.

Keywords

Stateful Functions; Flink; Stream processing; Approximate computation.

Resumo

A computação aproximada é um modelo de computação que pode ser utilizado para melhorar o desempenho, ou utilizar menos recursos no processamento de *streams* e grafos. Isto é feito ao diminuir a precisão dos resultados (i.e. resultados aproximados). Atualmente existem múltiplas plataformas para processar *streams*, a maioria não suporta nativamente resultados aproximados. Existem aplicações onde uma função (e.g. que está a ser executada como parte de um fluxo de dados) precisa de dados que são o resultado de uma operação que foi efetuada por outra função, e não tem uma forma de receber esses dados diretamente. Em vez disso, a outra função precisa de escrever os dados num componente de armazenamento, e depois têm que ser lidos a partir daí. Isto causa problemas com o desempenho (aumento de latência) e com a eficiência dos recursos (desperdício de recursos para aceder ao armazenamento) por causa do *bottleneck* no *I/O* e também aumenta o custo para o utilizador pois a aplicação necessita de realizar mais pedidos para o armazenamento. O Stateful Functions é uma plataforma que aborda este problema ao permitir que as funções troquem messagens. Este documento propõe o desenho e implementação de uma extensão para ser utilizada com o Stateful Functions. Ela permite processar *streams* mais eficientemente ao alocar os recursos disponíveis de uma maneira inteligente e variável e também utilizar computação aproximada. Para validar os resultados, a extensão foi avaliada com *benchmarks* de dados reais e sintéticos, executados localmente e na nuvem com aplicações típicas de processamento de *streams*.

Palavras Chave

Stateful Functions; Flink; Processamento de *streams*; Computação aproximada.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Cloud computing is becoming more popular every day, as it is a computing model that offers the user resources like processing power, data storage, among others, without the user having to manage those directly [1]. This model offers advantages such as the possibility of building applications that can scale well on-demand and where it only charges the users for the resources they consume [2]. These platforms usually are distributed and can employ servers in data centers that are spread across different regions of the planet, which can cause some issues with latency and consistency.

Serverless computing [2, 3] is one of the service models of cloud computing in which the cloud provider manages the virtual machines where the functions are being executed, the number of instances, the fault-tolerance, and the communication between servers. Such a model is a good approach to use with deterministic functions that only need the input to calculate the result. However, if they need to get data from other functions, a bottleneck might be introduced in the application. This might happen because of the approach used for state sharing: if that operation needs to write the data on persistent storage, such as a database, and then the other function needs to read the state from there, this will cause an I/O bottleneck. Also, it will increase the cost of operation for the client, as it needs to do more requests to the storage and it is being charged for the number of requests.

Stream processing [1, 4] is a form of programming that consists in processing items of data that are continuously arriving at an application. Besides owned or rented clusters, it is also increasingly used in serverless models. Stream processing applications are built using chains of functions (also called operators) that are used to process the elements of the stream.

Graph processing [5–7] can also be combined with stream processing. The main specific aspect is that the events come in the form of graph components. A graph is a set of edges and vertices where the edges represent the relationships between the vertices. Graph processing can have more challenges addressed than stream processing, like dynamic graphs and their partitioning.

Approximate computation [8–10] is a computing model that allows trading some accuracy for per-

formance in systems that do not need precise results. There are multiple ways to achieve this, with one being through load shedding [11, 12] which consists of dropping some of the input events to reduce the load on the system. The results' accuracy can be lower, however, the results must be accurate enough to be acceptable.

One of the most popular frameworks that is used to do stream processing (and can also be used for graph processing) is Apache Flink [13], which offers many features, and it is used by Stateful Functions [14]. Stateful Functions allows the functions to share state between them in an decoupled way without relying on persistent storage.

1.1 Motivation

The motivation for this work is the fact that currently there are not many mainstream stream processing platforms that cumulatively support dynamic resources allocation, allow to easily build applications, can auto-scale efficiently, and can use state sharing between the operators without using a storage component. Moreover there is a lack of platforms that allow the user to define the requirements that must be meet by trading other characteristics (e.g. trade accuracy for throughput). Stateful Functions allow elastic scaling and efficient state sharing, however, it does not allow trading characteristics to meet defined execution requirements, nor a dynamically resource management.

The support for using less precise results in popular stream processing platforms is also not common [15]. The loss of precision can be used to do the necessary trade-offs to achieve the user-defined processing requirements. In summary there is not a platform that allows stateful stream processing with approximate results and adaptive resource management driven by accuracy and performance tradeoffs.

1.2 Current Shortcomings

Some of the current shortcomings of stream processing are the state sharing between functions and the data distribution that can cause issues with consistency and performance. Most platforms do not allow an arbitrary communication between their functions (or operators), usually, the functions are organized in a DAG (Directed Acyclic Graph) way and one function can only message the functions that have a common edge (i.e. downstream the DAG). Some platforms also use storage to exchange state, which is a costly operation that has a major impact on the performance and throughput.

One aspect related to the data distribution is the partitioning of the stream; if the relation between the elements and the geographic location is not considered, then it can result in low performance [16].

Another important aspect related to the data distribution, the state sharing and the fault-tolerance is the data consistency. If any of these is done in an incorrect way, the data will be inconsistent (e.g. if a platform does not recover correctly from fails, the data loses the consistency).

The scalability is also important and can also cause issues if not addressed adequately. If it is done in a non-optimal way, it can lead to wasted resources by using more than necessary, or it can lead the system into an inconsistent state (e.g. some operators might not support multiple instances, others might need synchronization mechanisms). The auto-scaling is also an issue, some platforms do not support variable resource management or auto-adjustment of the parallelization level. The fault-tolerance can also impact the resource usage, if it is done in a non efficient way (e.g. costly save states, synchronization issues with different operators), it can lead to wasted resources. Another problem that some platforms, like Akka [17], may also have is the management of the operators' lifecycle; when it is not done correctly, the system may have instantiated operators that are not needed.

Another limitation is the lack of flexibility in choosing trade-offs. Most models will not allow the user to configure trade-offs between the resources, accuracy, and performance. These trade-offs can be done using approximate computation, however, it is lacking in commonly used platforms like Flink [13], Spark [18] and Storm [19].

1.3 Proposed Solution

The proposed solution is **Approxate**: it is an extension that can be used with Stateful Functions, that extends and executes over Flink. Stateful Functions already has an efficient state sharing between the functions, and is capable of scaling horizontally with multiple instances of the same operators. It also has data consistency, fault-tolerance, and can process graphs. Stateful Functions gets these characteristics from Flink, but it simplifies the building and deployment process of applications by using a model of decoupled operators that can have a state and interact with each other, instead of using a dataflow model.

Approxate's aims are to add adaptable resource management, to scale the application when it is necessary, and to lower the precision of the results to improve performance if that is required. The management of the resources and results' precision is based on requirements that can be defined by the applications' users. Those requirements are: **lag** (number of input events that are yet to be processed), **throughput** (number of events that are processed in a certain period), and the results' **latency** (necessary time to return the results after their production).

The proposed solution contains the following components: i) a Stateful Functions library, that is embedded in, and runs within, the context of Stateful Functions, therefore extending it (it is loaded by applications and intercepts the input tuples before they are processed to perform load shedding and it is used to achieve the approximate results); ii) a metrics reporter that is loaded in the Flink workers and uses the Flink Metric System to collect the metrics and do a short analysis of the execution; and iii) a middleware that receives the metrics, performs a more extensive analysis and manages the execution resources, the scaling, and the trade-offs. It also interacts with Docker [20] to manage the applications'

containers. It receives the events and return the results to Kafka [21].

1.4 Contributions and Goals

This work has the following contributions and goals:

- Presenting the current state of serverless computing, stream and graph processing, and the platforms used for that;
- Design and development of a library that can be used with Stateful Functions to give the applications the ability to use approximate results;
- Design and development of a metrics reporter for Flink, that can collect the metrics periodically and send them to be analysed;
- Design and development of a middleware that can communicate with Flink and Docker, receive and analyse the metrics to manage the resources based on the user-defined requirements;
- Design and development of a model and API that uses those components in order to achieve increased scalability and performance, with adaptable resource management that allows the users to balance the performance, results' accuracy, and used resources in their applications;
- Evaluate the solution and analyse the differences in performance and used resources.

1.5 Document Structure

This work contains the states of the art of serverless computing, stream processing, graph processing, approximate computation, and an overview and analysis of the relevant stream and graph processing platforms in Chapter 2. In Chapter 3 we describe the design of the Approxate and its components. There is also an overview of the already existing components with which our solution interacts.

In Chapter 4 we present some details concerning how our solution was implemented. There is an overview of the code structure and some examples of used methods, APIs and libraries. The reasons why they were used are also stated.

Chapter 5 contains the used methods to evaluate the implemented solution. We describe the used setups, the used workloads, the metrics that should be achieved and the results. Chapter 6 is the conclusion of this work and it contains a summary and the future work that can be done with Approxate, and also the aspects that can be improved.

Chapter 2

Related Work

In this section, there is a brief overview of serverless computing (Section 2.1), since that is usually the used cloud computing model of stream and graph processing applications and its shortcomings, that we explain in Section 2.1.2, can also be found in the processing platforms.

After that, there is the state of the art of stream processing (Section 2.2) that explains its benefits, the current issues, and also the desirable properties that stream platforms should have. Next is the same but for graph processing (Section 2.3). Following that, there is the state of the art of approximate computation (Section 2.4), and how this computation model can be used in stream/graph processing platforms.

Next, there is an overview of some relevant platforms used for stream and graph processing (Section 2.5). Lastly, there is an analysis and comparison between the design, properties, and performance of the platforms (Section 2.6).

2.1 Serverless Computing

Serverless computing is a model of cloud computing, in other words, the resources that are needed to execute a determined job don't need to be in a local or user-owned device, instead, they are in a server that receives the input data, do the operations it must do, and then it can return the result. These resources can be processing power, cloud storage, among others. Despite the name, servers are still needed. The difference between this and other cloud models is that developers do not need to manage the servers directly. The number of servers used and their capacity is managed by the cloud provider. This can give the client an appearance of having infinite resources on-demand [2].

2.1.1 Key Features

These models abstract away most of the operational concerns from the clients [1], like the managing of low-level infrastructure, servers, resource allocation, and scalability. The client does not need to invest in building an infrastructure, which eliminates the up-front commitment [2]. It also allows the cloud provider the chance of controlling the entire development stack which can reduce the operations costs because of the optimizations that can be performed. The cloud provider can also offer other platforms to work together, so the client can have a more flexible and agile ecosystem to work with [1].

Usually, the functions that are run in the cloud are small pieces of code that are used in several micro-services that comprise the application. These micro-services can be chain together and thus building an application by making a composition of functions [3, 22].

One of the advantages of having a partitioned application is the possibility of changing one micro-service without needing to change the others. This approach can lead to faster application deployment, update, and correction of bugs, without having the risk of doing some change that can break the entire application, since a micro-service is a modular component. A micro-service can also scale individually, thus leading to better efficiency of resource usage, as they can be applied only where they are needed [22].

2.1.2 Shortcomings

The communication and coordination between functions can be done in a non-optimal way with a non-efficient state sharing, like using storage to write and read the data which can lead to lower performance [1–3]. One way of reducing the latency in the communication consists of having the application stored in memory so that the functions, and their state, would be in memory and that way it could be shared with almost zero latency [2].

Another disadvantage of this model is that the clients need to build the application around the platform and know how it works [1].

2.2 Stream Processing

Stream processing is a way of programming that is used to process continuously arriving items of data (e.g. tuples, strings), named events. This processing is done in a parallel way, where clusters can have multiple nodes with different purposes in a composition. The events can be processed as soon as they are created or can be stored for later processing [4].

For example, a user can have a sensor to know when a room temperature achieved a determined value and the user wants to be warned when that happens. One way of doing this is to have the temperatures collected by the sensor being streamed to some service that will analyze the values of

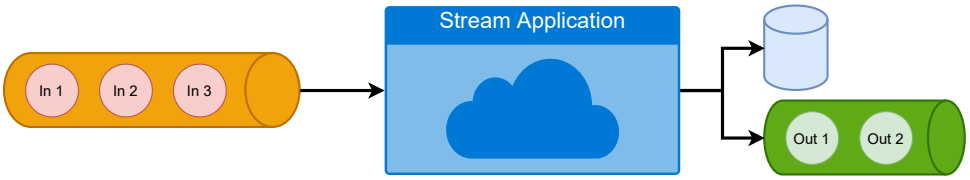

Figure 2.1: Example of a stream transformation

that stream, in this case, each one of these temperatures (sensor data samples) that are streamed is considered an event. The result of the processing can be a new stream of events or can be values that are going to be stored in a database or can be a combination of them, as illustrated in Figure 2.1, where there is a stream of events that are arriving into a stream application to be processed, and the results can then be stored in a database or can be returned in a form of 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.

This type of processing can be done using a serverless model and it is useful to systems where there are lots of changes on the data and those changes need to be constantly processed with low latency and high throughput [23]. Another way of processing these events is to store them until they reach a certain size and then they are all processed at once, this way is called batch processing and when compared to stream processing it has a bigger delay since the data is stored until it is processed. It can also be worse to process large quantities of data because the data can occupy more memory than the machines have, in that cases, it cannot be processed all at once. Stream processing models can avoid problems of data storing management since they do not necessarily need to store the data to process it, however, they can store historical data that later can be used with the processing of the new elements [24].

There are two types of stream, bounded and unbounded, the difference between these two is that bounded streams have a defined start and end, while with unbounded we only know the beginning and not the end, it may even not have one. The unbounded type is common in cases where the data sources constantly produce new data that needs to be processed in real-time.

Stream processing systems usually are distributed systems, so they need to address questions like distributed memory management, the geographic distribution of the data, the latency in the communication between different data centers, and data consistency. The load balancing between different instances should be done in a way where each instance gets a similar load [25].

The applications that use stream processing usually consist of multiple functions (or operators) chained together where each one performs some transformation in each element that arrives at the application, these functions can either store some state or be stateless. These chains of functions can originate a dataflow.

Besides state sharing and storing, stream processing can have more difficulties [26]. Some of them are the fact that the stream is not all available, the order of each tuple cannot be controlled, the input rate can suffer fluctuations, sometimes each tuple can only be processed once, and also storing all the data in most cases is impossible, the data may never stop arriving. The data storing should be handled in a way that minimizes the impact of the processing of the other elements of the stream [24]. The stream processing platform should be capable of fault-tolerance and at the same time be able to keep the deadlines which can be achieved by having efficient resource scheduling strategies [26].

Stream processing can allow the processing of events with low latency and high throughput with accurate results and in a distributed and scalable way. However, this type of model can have issues with the communication between functions, with data sharing, and with efficient scaling. It can also have issues with the stream (e.g. elements arriving out of order).

Some aspects must be considered when we are doing data stream processing. These aspects can also be applied to graph processing and other types, but are presented in the context of stream processing in the next sections, and there are also some examples of how some of the stream processing platforms solve the issues that result from these aspects.

2.2.1 Data Sharing

Some stream processing functions need data from other functions to process some event. In that case one function will need the result from another and what is done for a lot of stream processing frameworks is to simply write the result to persistent storage and then the other function will read the data from there. However, the stream processing platforms must keep the dataflow without having a costly storage operation in the middle, the storage access would increase the latency, so it must be able to do the critical operations without it [14, 24].

This can be solved by allowing the functions to have an internal state and to share it without using storage [2, 13, 14, 17, 27, 28], it can be done by using the network. The state exchange can also be improved by allowing arbitrary communication between the functions without the need of using the functions' order in the dataflow. The state should be kept in memory or in efficient disk-structures to load fast and not cause a major decrease in performance. If the data sharing is done in an efficient way it can have positive effects on the response time of the application.

The functions that receive the state can receive it instantly if they are instantiated or if are virtual instances, or they can receive the state in the next iteration if the platform works in supersteps [14, 17, 27, 29, 30].

2.2.2 Scalability And Performance

Some platforms focus on being efficient, fault-tolerant, and having low latency, even when working with streams that are very large, this can be done by scaling up the system [18, 30].

With large streams, and the need of scaling up the systems, come more difficulties related to job scheduling, data consistency, memory management, race conditions, and other problems that must be solved. The platform being capable of parallelism is a requirement to be able of scaling up.

Usually, latency is an important performance aspect in stream processing. It is affected by all of the components of the platform being used and the architectural decisions. The latency should be low even with large streams, and the scalability of the platform should be capable of keeping the performance and the latency close to the ones it had on small streams. Some systems will focus on having the lowest possible latency, even if it comes at the expense of other aspects. Other systems, the majority, will have a more balanced approach. High latency in some components can lead to a huge bottleneck in the entire system, so critical components should always have low latency.

The performance of the applications can be improved with parallelism by taking advantage of the mapping of the different parts of the stream (the same part of the stream can be replicated in multiple machines). By dividing the stream to multiple instances of the operators, or multiple machines it can be processed in a parallel way.

However parallelization does not necessarily equate to better performance: it is necessary to have the right amount of it, so there is a need for some mechanism that must be able to know when there is a need of scaling up or down certain parts of the application [23].

To achieve auto-parallelization it is necessary to know which part of the application can be replicated and which is necessary to be. The latter can be achieved through monitoring the resources that each part of the application is using and also identifying where the bottlenecks are. Another important aspect to consider when replication is used is the dataflow order, in a general way the operators need to be in the same order after being replicated [23].

Scaling up can improve the performance but can also cause issues, one of the issues in distributed models is the replication of state, which leads to higher periods of time being needed to scale up. However, the performance can also be increased by using a cache system to keep data from the stream with fast access when needed. Elastic scaling can also be used however, multiple factors need to be considered, e.g. using parallelization in every part of the stream can lead to worse performance than not using it [23, 27].

2.2.3 Stream Partitioning

As mentioned in the previous section, partitioning the stream and distributing it to be processed in multiple machines in a parallel way can improve the performance. However, there are some things to be

considered like the distribution of operators with different demands of resources and also the distribution of the stream after it is partitioned.

The performance can be increased by distributing the most demanding operators across multiple machines. This way if a machine is busy with one operator, another machine can be used by other operators. The processing can also be distributed through multiple clusters, so if a cluster is busy, another one can process the events [31].

The data streams can also be dynamically redirected to the computing resources [32]. This feature is useful in order to improve latency and throughput. A framework should be able to know when and how data streams should be redirected based on the requirements of the application and the complexity of the elements. The complexity of an element can be determined by the functions it implements, and it is an important aspect when deciding the allocation of its instances and resources.

2.2.4 Operators

There are multiple optimizations that can improve the performance by adjusting the operators [31]. They are the following:

- Operator reordering: There are cases when one value might be dropped by some operator because that operator knows that it will not change the final result. This value may already have been processed by a previous operator, so if it is a situation where the relative order of those two operators does not matter they can switch places;

- Redundancy elimination: There are cases where multiple instances of the same operator are being used on the same data, achieving the same result, in different paths of the dataflow. This can be avoided by having one instance of the operator sending the data to all the following operators;

- Operator Separation: There are cases where one operator is doing multiple tasks on the data. If those tasks are decoupled into different operators they can have different resources reserved for them. This can lead to better resource management;

- Fusion: There are cases when multiple operators are chained together in a flow. If one of these operators is doing a very light-weighted task, the state sharing can be more demanding than the task. If they are converted in a single operator we can have less latency;

- Fission: This one consists in splitting the data so it is processed in multiple instances of the same operator and then the results are merged to get the final result.

2.2.5 Data Consistency and Result Accuracy

Another important aspect of stream processing frameworks is their capacity of keeping the data consistent, and correspondingly, keeping results accurate, i.e., completely reflecting all of the input. The data consistency is related to the fault-tolerance because if a server (or machine) fails and does not recover correctly, that server will have data that is not correct and that can affect the rest of the application. One strategy to avoid this is to have replication, however, it comes at a cost of the need for more servers and the replication of requests. Depending on the type of application that is being executed, the decision of having total consistency might not be worth it, it will depend on the application's criteria. The application can get fewer consistency guarantees but have a lower replication factor or better performance. The balance between data consistency, replication, latency, and others, can be different for each application.

The servers' failures are not the only thing that can cause inconsistency in data, the distributed memory systems also have to deal with this problem. The memory can be distributed across machines that are in different data centers, and can also have some parts replicated, which can cause the need for having some sort of mechanism to manage the writes and reads. If a machine changes some part of the memory and the memory is shared, then all machines that have access must see that change if they are going to use data from that part. This consistency can be done in multiple ways, e.g. the memory can have locks, there might be race conditions, quorums can be used, just to name a few. The chosen approach will have an impact on the performance of the system.

Another aspect is that the results should be predictable, if a stream gets a result after being processed and if an equal stream is processed in the same conditions it should produce the same result [24], however, that is not the case in systems that allow losing accuracy and get approximate results to improve performance, this is called approximate computation/computing.

Eventual consistency can also be used [17, 27, 33], in the same cluster different nodes can write and read from the same memory at the same time, and then the results are converged. This allows only eventual consistency to be upheld, and thus produce an eventually consistent state.

The opposite can also happen to guarantee consistency, a platform can have only one function accessing the same memory at a determined time. The consistency can also be achieved without explicit coordination between the functions that are using the data by using a clock system so the system knows the modifications and their order. Functions that access critical memory to perform some actions must only access the memory in a way that guarantee consistency.

2.2.6 Fault-Tolerance

Systems that process streams must have some sort of fault-tolerance to guarantee that they lose no data in case of a failure. There are several ways of achieving this, some frameworks will store the stream's partition that is being processed until the results are in persistent memory, others will do that and also

store the flow of operations that were executed on them. Others will save checkpoints from time to time and there are multiple other ways. If the data is lost because of some failure the results would not be consistent.

The method used for fault-tolerance will have a great impact on the time that a system needs to recover from a failure, and can also affect the latency of processing, the costs of operation, the memory used, and the time that is necessary for the system to scale up or scale down. Therefore, it should be a method that is efficient for each scenario.

On average, current systems use a nominal value for checkpoint interval assuming roughly one failure every nineteen days [34]. When the parallelism in a stream processing system increases, the mean time to failure (MTTF) decreases. Using checkpoints to save the state that later can be used for recovery is more efficient than state replication, some platforms [13, 19] store the messages at the source to avoid local checkpoints (i.e. buffer the messages at every operator).

The platforms should do checkpoints where they maximize the total utilization time [34]. The utilization time is the fraction of time of the system where its resources are available to do useful work, it does not count the overhead to maintain the system running. A system can have different types of checkpoints (e.g. store the state in persistent storage), some can be more costly than others and so they can be performed less frequently than the less costly ones. In summary, the checkpoint optimal interval only depends on checkpoint cost and failure rate.

In platforms that uses systems of operators with parental relationships in a tree way, if an operator fails because of an internal error, then the situation is handled by the parent of that operator. That can continue to the operator at the top of the tree. The parent operator can decide to stop the children or restart them [17, 28].

2.3 Graph Processing

A graph consists of a set of vertices and edges. The relationship between two vertices is represented by the edge (or edges) connecting them. If a vertex has no parent (a previous vertex) then it is a source, and if does not have a child it is a sink. In the graph stream context, each vertex represents an event. The workflows usually are modeled as Directed Acyclic Graphs (DAGs).

Graph streaming consists of a series of events that are being streamed in the form of a graph to an entry-point of some application that is going to process them. Usually, these applications apply one function or more for each vertex, and because these events can be connected the result of the processing of one can have consequences on the result of another one if they are related [35].

These graphs can suffer changes after started being processed, they can have a not known end (unbounded streams), they can be related to other graphs that can be stored in data-centers in other regions, among other situations, so it is only normal to exist multiple graph processing frameworks that

have different purposes and advantages.

Graph processing has the same advantages and disadvantages as stream processing (Section 2.2). It also has different issues and aspects that must be considered. Those are detailed in Section 2.3.1 (Graph Partitioning) and Section Section 2.3.2 (Dynamic Graphs).

2.3.1 Graph Partitioning

Some graphs have terabytes of data and when they need to be processed they must be distributed across different machines because one machine often does not have enough memory to load the entire graph. In these cases, or for another reason like improving performance, the graph will be partitioned into a distributed graph. These machines usually are part of clusters, and a graph can be spread across multiple clusters [18, 35, 36].

The way this distribution is made will take some factors into account, for example, an even distribution can be ideal for a lot of cases, but not for all. There are situations where one cluster should get some data the fastest it can, so in this particular situation, the data should be the closest possible to the cluster's physical location. One approach like this can result in some clusters having a larger part of the graph than others.

However most of them use a hash function which generally results in an even distribution [28, 29], but it also does not take into account the relationships between the vertices. Other approaches use strategies like edge cutting to try to keep the vertices that are more related in the same cluster or nearby clusters [36]. Comparing to the hash function, this approach takes more time to divide the graph but it can have better results in the processing phase.

The partition strategy will influence the final result, one good strategy can lead to faster results, better accuracy, and also reduce data transfers.

The partitioning can be done using online or offline methods [36]. Online methods are executed in real-time when the graph is being streamed to the application, while the offline ones are executed before the stream is done. The online has to be fast and is executed on one pass through the graph, while the offline can be more complex and pass the graph multiple times.

The most common approach to edge cutting is to use a multilevel method, most commonly in an offline way [36]. This algorithm consists of three phases: i) in the first, the coarsening phase, the graph is transformed into a sequence of coarser graphs; ii) then in the second phase, the partitioning phase, each of the coarse graphs is partitioned; iii) after that in the third phase, the uncoarsening phase, a refinement algorithm is applied at each level to improve the cut size that was previously obtained in the previous levels. This can be applied in an online way and due to the multiple levels, some vertices that weren't put in the ideal part will eventually get corrected, the algorithm can correct poor decisions that were made in earlier iterations. In streaming graphs where the data arrives in batches or windows, the

algorithm can be applied at each one that arrives, and then when newer batches arrive it can use the location of the previous batches to do a better job.

2.3.2 Dynamic Graphs

Some graphs are static in the sense that their vertices and edges will remain the same. However, others evolve with time, so at one moment some vertex can have a connection to another one, and in the next moment, that connection might not exist anymore. These graphs are called dynamic or temporal and can register the development of relationships between the vertices and edges of a graph throughout time [37].

Dynamic graphs contain all the vertices history [37], those are the vertices that were added, changed, or deleted. These changes occur through a stream of events, where each change is an action. Because the stream can be unbounded, stream ingestion is a problem: before any action is done it must be verified first if the pre-requisites are valid, for example, a vertex cannot be deleted if it does not exist. This verification is necessary in order to not let the graph reach an inconsistent state. An update can arrive out-of-order and the pre-requisites for it may still not exist. However, that update cannot be discarded and should be kept until it can be performed, which can introduce a bottleneck in the system and can also lead to the graph representation not reflecting the latest updates. Since normally the systems are distributed and events can arrive out-of-order, keeping the data consistency is a bigger challenge in dynamic graphs than it is in static graphs, but even with these conditions the graph should be in a state where it can be processed at all times and it also should be able to perform computations at any historical point.

The dynamic graphs can be processed by using snapshots of the different versions of the graph [38]. This way the snapshots can be treated as if they were a series of static graphs. The snapshots are processed by their watermark time. This allows the platforms to keep a consistent state because the platforms cannot perform updates with the wrong order and keep the consistency. The events (graph's modifications) can be aggregated by their timestamp and then be processed as a batch [39].

Another way of processing these graphs, and the more costly in terms of resources, is to maintain the dynamic graphs over a distributed set of partitions and do the ingestion and updates in real-time. This can be done by maintaining the entire graph in-memory and processing the events as they arrive, instead of waiting for a certain number to reach a batch. This way is not necessary to use a snapshot model [37].

2.4 Approximate Computation

Approximate computation [8–10, 40, 41] is a computing model where the results are not completely accurate. Such a model can be used in scenarios where the applications or systems can tolerate some loss of accuracy.

There are multiple ways of implementing approximate computation, it can be done at the software level, but it can also be done on the hardware level, i.e. a hardware component that must add two values can be accurate on the most important bits and can generate the less important bits in a random way [40]. There are also multiple ways of dealing with the data, we can compute only through a set of samples instead of the entire data set, or we can process the entire input but with operations that calculate an approximate result.

There are multiple advantages of using approximate computing, by lowering the results' accuracy we can increase the throughput and so it is possible to process data faster with the same resources, or at the same time using fewer resources, and can also be more energy-efficient [41].

One way of using approximate computation is through load shedding [11, 15], where some of the 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. One concern with just randomly dropping some 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 [15]. 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 [8, 41, 42] like loop perforations, approximation of arithmetic computations, approximation of communication between computational elements, precision scaling, among others.

The loop perforation method consists of skipping iterations in hot loops (a loop that heavily uses the resources for a long time). This technique can skip approximately half of the loops, which cuts around half of the processing time and still maintain acceptable results.

Another technique consists of a width reduction in the data, by reducing it to 10-16 bits instead of the 32/64 that are common is still possible to get acceptable results. This is called reduced precision computation. By using fewer bits the processing can be done in cheaper hardware and with less costs in power consumption and time.

Relaxation of synchronization can also be used in parallel applications. It allows processing the same amount of data in half of the time that is necessary when performing the same operations with precise computing. This is done by not synchronize certain parts of the applications that would not lead to a failure of the system. Another method to achieve approximate results is to store the results of some

functions and then use those results when the input is similar and/or the function is similar, this is called memoization.

One method that involves the hardware is voltage scaling. By reducing the voltage of some components like the memory some errors will possibly be introduced, however, the same dataset can be processed with less power, and the results will lose some accuracy due to the errors.

It is also possible to combine multiple techniques of approximate computing and, by tunning the accuracy of each component is possible to produce acceptable results in the end.

StreamApprox [15] is an algorithm that was made to achieve approximate results in stream processing environments. It is a generic algorithm that can be applied to Apache Spark or Flink. This algorithm receives the available budget and uses approximate computing to process the stream without exceeding it. This algorithm can be used in batch and pipeline models. It uses a stream aggregator to receive multiple streams from different sources (sub-streams) that will form the input stream.

It uses two different techniques: Reservoir Sampling and Stratified Sampling. The Reservoir Sampling works by aggregating a certain amount of events in the stream and then select which ones will be processed with a certain probability. The Stratified Sampling does the same thing as the Reservoir but instead of applying the randomness of selection in the input stream, it does that in each of the sub-streams in order to guarantee that the samples chosen are fairly distributed among each data source, however, this sampling method only works in sub-streams where the size is already known in advance, because if the size is unknown then the number of samples that should be chosen from each data source to make a fair distribution is also unknown.

The algorithm uses both techniques to achieve a sampling method that can select a number of samples from each sub-stream by calculating the total number of events that the sub-stream has, and then uses a reservoir to store a certain number of samples. The next step is to decide, using a probabilistic function which samples will be processed. Then each of the chosen samples gets a weight attribute that depends on which sub-stream that each sample comes from. The purpose of this weight is to maintain the statistical importance of each sample as they were in the original sub-streams, i.e. if the reservoir has a size of three and one sub-stream has six in size, while the other has two, only half of the elements of the first sub-stream could be chosen, but all of the elements of the second could be chosen. If the samples did not have a weight, the impact of the samples from the second sub-stream would be greater than the impact of the ones from the first when all of them should have the same impact.

Results can also be approximated by carefully delaying the re-execution of workloads (e.g., MapReduce 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 [43] and can be further fine-tuned with machine learning [44].

2.5 Relevant Related Systems

Multiple platforms are used for stream and/or graph processing, each one has its strong and weak points, and some of them are very close in terms of features. In the sections ahead there is a summary overview of some of them, so it is visible how they compare with each other and also the current state of stream and graph processing. There are different platforms and not all have the same focus point or are optimized for the same scenarios. Some of them allow some flexibility in choosing trade-offs. They are detailed in the next sections, the first ones will be the most common and the systems that were derived, based, or are related to them. After that are the other stream processing platforms and in the end, are the ones that are specific for graph processing.

2.5.1 Apache Flink

Flink [13] is a framework and distributed processing engine for stateful computations over streams.

Flink is able to perform the same actions on both, bound and unbounded streams so it is not necessary to wait for several elements to form a batch. The reason why Flink can run any kind of application on unbounded streams is 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. Alternatively, it can just process each element when it arrives.

Flink applications can run at any scale since this platform support multiple instances of the same operators. Those applications are dataflow programs that contain stateful operators connected with each other in a DAG way through data streams. The Flink operators can also be used in an iterative model. When this model is used the stream has heads and tails to mark when a step has occurred. This model can also be used to perform an iteration (also called superstep) when new data is added to the stream.

The operators can have a state, which can be something like a sum of the values of each element in the stream. Flink deals with the state by maintaining it on memory, or in access-efficient on-disk structures in the case it exceeds the memory amount. The state can be shared among the operators through intermediate data streams. An intermediate data stream is a logical handle to some data, it is produced by some operator and can be consumed by other operators.

The data exchange between consumers and producers can be done in a pipelined way. This allows them to exchange data in a continuous way, and Flink tries to avoid the materialization of data. The other way of exchanging data is by using a blocking data stream, which keeps the results until it is full and then delivers them. This approach needs more memory than the other but is useful for cases where it is necessary to isolate successive operators.

Flink's approach to balance latency and throughput is to serialize the data after being produced by a producer and then divide it through one or more buffers. The buffer is sent when it is full or when it

reaches a timeout, this way Flink can have a high throughput by defining a large size for the buffers and also maintain low latency by setting the timeout to a small value.

Another important aspect of Flink is the control events. The streams can contain events that are injected by the operators. They can be used to create checkpoint barriers that divide the stream into different parts that are saved into persistent storage. They can also be Watermarks that have the purpose of signaling the progress state. Another use for events is to mark an iteration barrier.

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 snapshots taken by Flink contain the state of the operators and also the state of the stream. This snapshot is consistent in regards to logical time among all operators. This is done by injecting a checkpoint event into the stream. Then, the operators will process the stream and when they see the event they will record a snapshot of their state, and then the stream moves to the next operator who will do the same thing and so on. When all operators have done this, it is achieved a global snapshot. When a failure occurs, the state is recovered from the snapshot.

2.5.1.1 Stateful Functions

Stateful Functions [14] is an API that uses Flink. This API has the purpose of making it easier to build distributed stateful applications that have high-throughput and low latency and that can scale efficiently by using Flink and adding arbitrary communication between the instances of the operators that are processing the data. Stateful Functions retains Flink's abilities to graph streaming, fault-tolerance with its snapshot model, and others.

The functions (operators) of this platform are virtual, so when a function is not active it does not consume resources, which allows the applications to scale horizontally. They are also invokable through messages, so a function can invoke another one. Stateful Functions allows arbitrary communication between functions, so the communication does not need to be done in the functions' flow order, and a function can message another or itself with exactly-once guarantees. The state exchange does not need to use storage.

Also, there is no need for service discovery, the message exchange is done by logical addresses. Each function's instance is associated with a function type and an ID, both of them when combined form the function's unique Address. When a function is invoked it will be provided with context about the call, this context can be something like the invoker's function address, or its own address or some state that will be used to update the function's state. There are two types of functions, embedded and remote, but despite that, remote functions can message embedded and the other way around.

The functions can also communicate with the outside world using ingresses and egresses. With ingresses, the functions can be invoked by the outside world, and they can also send state to the outside world with egresses using a data broker like Apache Kafka [21]. Stateful Functions uses a Flink cluster

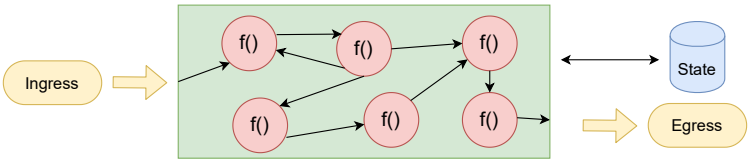

Figure 2.2: Stateful Functions' Overview

to handle the messaging routing and state management, the functions can be deployed in a different service.

In Figure 2.2 there is an overview of the way that Stateful Functions deals with the events: they are received through the ingresses, then they are processed by the functions that can communicate and save snapshots to persistent storage, and then the result can be returned to the outside world using egresses.

2.5.2 Apache Spark

Spark [18] is a scalable framework that is used for processing large-scale data, it can process streams and graphs. It offers functionalities like memory management, job scheduling, data shuffling, and fault recovery. Spark uses Resilient Distributed Dataset (RDD) [45], 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.

GraphX [46] is part of Spark and is a library for large-scale graph analysis. It has features like the abstraction of graph-oriented data, graph transformations, graph algorithms, and graph builders. It also has an extension for the RDD API called Resilient Distributed Graph (RDG), which has a property graph as the core data structure. A property graph is a directed multigraph. RDG has RDD's immutability, distributed data, and fault-tolerance. GraphX also has Spark's graph operators for property graphs.

2.5.2.1 GraphTau

GraphTau [38] is a time-evolving graph processing framework that has the purpose of being efficient to process dynamic graphs. GraphTau is built on top of Apache Spark.

This platform treats time-evolving graphs as if they were a series of consistent graph snapshots, and in the case of dynamic graph computations, they are treated as if they were a series of deterministic batch computations.

Those graph snapshots can be processed by two different computational models. One of these models is the Pause-Shift Resume, and its purpose is to be able to process graph snapshots that can change to another snapshot. In other words, the algorithm that is responsible for processing the graph will start to work as soon as the snapshot is available and then if another element arrives, the computation is paused on the old snapshot and it is started on the new one, it is not mandatory to process the elements that already have been processed.

The other offered model is the Online Rectification, and it consists of rectifying errors that are caused by a different snapshot. If a value of the result of the algorithm computation on one vertex is affected by the result of the previous and the new snapshot causes the previous one to be modified or deleted, then it is necessary to re-calculate the result of the vertex on the new snapshot. The vertices that are not affected by the new snapshot do not need to be re-calculated. Consistency is achieved by processing the batch using timestamps, this way when the snapshots of an interval are processed, all changes made in the period of that interval have been processed in the correct order. This leads to consistent snapshots.

2.5.3 Apache Storm

Storm [19] 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.

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.

2.5.3.1 Twitter Heron

Heron [47] is an engine that is capable of real-time streaming for events like calculating the number of active users on Twitter. Before using this engine Twitter used Apache Storm [19]. For Twitter's use case Storm had some issues related to scalability, debug-ability, manageability, and efficient sharing of cluster

resources with other data services.

These issues came from different places, for example, a worker can have different tasks, and all of them output logs to the same files, so when performance issues occur, it is hard to determine which task is causing them, the usual solution is to restart all of them, even the ones that were good. Another problem is the resource division, it was assumed by Storm that each worker will need an equivalent share of resources, so this leads to a non-optimal resource allocation. Another one is that workers from different topologies that are running in the same machine can interfere with each other, which leads to not being possible, or being very difficult, to trace what is causing the performance issues.

Heron improves the communication between processes by using the *protobuf* protocol, also there is one topology master for each topology, so the topology only has one single point of contact. It has a Stream Manager to route the tuples efficiently. It also has a mechanism to dynamically adjust the rate at which data flows through the topology, and a metric manager to collect system metrics. Those metrics will then go into a monitoring system.

2.5.4 Cloudburst

Cloudburst [27] is a platform to be used with serverless computing models that allows stateful computations. The goal of this platform is to preserve the benefits of a serverless model, like auto-scaling, and also add a shared state between the different functions.

This platform considers three types of state sharing, which are Function Composition, Direct Communication, and Low-Latency Access to Shared Mutable State.

Function Composition is when the functions (operators) share state by invoking each other and pass the state as an argument and then receive the return value, this is done in a DAG way by the platform, however, the user can do an arbitrary function composition. Direct Communication is the case where a function can message another and send it a value, usually, this state exchange is realized using storage, which causes a high latency. The Low-Latency Access to Shared Mutable State is when two or more functions or clusters are trying to access the same persistent memory and it is a concurrent access that can lead to the need of using locks or race conditions.

The storage engine used by Cloudburst is Anna [33], and it provides consistency without the need of having explicit coordination among the various requests that are made to the storage, this is achieved with the help of a clock system that keeps track of when the last change was made in a determined version of some data. The Anna KVS (Key-Value-Store) is used by the functions so they can manage their state.

Regarding fault-tolerance, this system has it, but in two different ways. One of them is at the storage level and it is a responsibility of Anna and its replication scheme. The other is at the computation level and in the case of failure by a machine, the DAG that was being processed is re-processed from the

beginning on another machine.

2.5.5 Akka

Akka [17] is a toolkit that can be used to process streams and graphs, it allows the building of concurrent, distributed, message-driven, and scalable applications. The operators that process the data have a parental relationship that is done in a tree way. The top-level operator receives all messages that are sent to the system. After that, it can route them to other operators or process them itself. The operators have an internal state and can communicate with each other by sending messages.

The operators are virtual instances, however, they need to be explicitly started, stopped, and destroyed when they are not needed anymore. When an operator stops it will stop its children.

This platform offers solutions to problems like the distribution of work among the operators that are part of the same cluster, recovering from internal errors, the scaling, the migration of entities from a crashed system without losing their state, and also guarantee that an entity only exists in one system, thus maintaining the system state consistent. The data sharing between nodes in a cluster supports multiple operators writing at the same time and then the result is achieved by converging the writes, leading to an eventual consistent state.

This platform also has fault-tolerance, the operators check their children for failures and if that occurs the parent can restart the children or have another one process the data. Another method used for fault-tolerance is persistent memory where the state of the operators and the stream can be saved at a certain point. Then in case of a failure or a service crash, the operations can be resumed from that point. It also supports having a single entity managing the same task distributed among different clusters, which improves the fault-tolerance but limits the scaling abilities.

2.5.6 Orleans

Orleans [28] is a platform where the operators can have and manipulate state. The operators are virtual instances, if they are not being used then they are not consuming resources.

Each operator has its state and memory, which is not shared, so the only way for two operators to exchange data between them is through messages, with an exactly-once guarantee. These messages are asynchronous. This allows the operators to keep working instead of being waiting in a blocked state without explicit thread management. Also, an operator is always addressable because, since they are virtual instances, they always exist. The runtime instantiates and deactivates the operators as they are needed.

This platform uses a single-threaded style approach per activation, meaning if an operator is activated then it is the only one who will be reading its state, so there is no need for locks or race conditions.

Orleans supports a scalable approach where an operator that has a state can only have one physical instance, where a stateless operator can have multiple.

Another feature is the management of the reliability aspects, so if a server fails the state can be recovered, however, this platform does not enforce a checkpointing state saving model; the user is the one to choose how often the state is saved, so there is no guarantee of how recent is the latest saved data. The runtime also check if an operator fails, and when that happens it will re-instantiate that operator on another server, so there is no concern about an operator crashing, or to take care of checking if the operator is alive, that responsibility is not being put on the users, the platform will do it.

2.5.7 Pregel

Pregel [29] is a distributed programming framework for graph processing. In this framework, applications are expressed as a sequence of iterations or steps.

In each iteration, a vertex (operator) can receive messages that were sent to it in the previous iterations. It can also send messages to other vertices or modify its own state. This framework allows the vertices to retain a state that can be shared between them. It also allows a vertex to deactivate itself if it does not have more work to do, although it can later be activated by receiving a message from another vertex.

This framework divides the graph into different partitions. One copy of the program is the Master, and it is responsible for knowing the state of the other copies (Workers), this is done by pinging the worker machines periodically, if the worker does not answer then it is considered that it failed. When a worker needs to contact another worker it sends a message that is stored in a buffer until the buffer is full and then flushed. However, the worker can skip this step and send it directly to the worker's message queue, but, in most cases, it will have a higher network cost when compared to waiting for the buffer to reach a certain size.

The Pregel programs iterate through supersteps, a superstep is when the framework invokes a user-defined function for each vertex in a parallel way. Pregel also uses a checkpointing system to allow fault-tolerance, the state of each partition is stored at the beginning of a new superstep, and the Master can also ask a worker to store the state of its partitions.

2.5.8 Apache Giraph

Giraph [30] is an iterative graph processing framework, built on top of Apache Hadoop [48]. Hadoop is a framework that is intended to be used for processing a large set of data across multiple different clusters. It offers the ability to scale up and use a local state.

Giraph does computation over graphs, a graph being composed of vertices and their directed edges. Each vertex and each edge store a value. The iterations are done over supersteps, in each one, a

function provided by the user that is associated with each vertex is invoked. All vertices start in an active state. The user-defined functions can message other vertices. In each superstep, the vertices receive the messages that were sent to them in the previous superstep. For a vertex to know the value of another vertex or of one of the edges it needs to ask them, in other words, they have to communicate to share the state.

2.6 Analysis and Discussion

This section contains some performance comparisons (Section 2.6.1) between some of the previously mentioned platforms and a summary of their design characteristics and properties (Section 2.6.2).

2.6.1 Performance Comparison

In this work by Inoubli et al. [49] there is a comparison made between some of the most popular data processing frameworks. The tests were performed in similar conditions across all the platforms with batching and stream models when possible. Some of the results achieved are the following:

- When comparing Flink, Hadoop, and Spark with small datasets, Hadoop was the slowest of all. Until 1250 MB of data were used Flink was the fastest, after that point Spark was the best. This test was done with a Wordcount processing using real data from Twitter in batching mode;
- Hadoop was slower because it was affected by the I/O operations that were necessary to process the data;
- Spark deals with large datasets better than Hadoop and Flink, but it is not suitable for tasks that have an intense memory utilization, because of the RDD creation process, Spark uses more memory than the other two;
- Hadoop performed well on all tested situations, but suffers from I/O access to intermediate states;
- Flink maximizes the CPU utilization which does not happen in the other two, and it also minimizes the periods of idle;
- Flink has a high networking utilization because the communication between workers is done using the network;
- In the stream scenario tests, Flink and Storm achieved similar results;
- For stream Flink, Spark, and Storm are all good alternatives;
- Spark and Flink are also good for batch processing;

- Hadoop is not optimal for graph processing.

This work by Karimov et al. [50] compared the performance of Flink, Storm, and Spark. The performance metrics used were latency, in this case, the time that it took from the data production at a source until the engine produced an output, and throughput, in this case, the number of ingested and processed records for a given time. There are two types of latency, event-time which is the time when an event is captured, and processing time which is the time when an operator process the tuple.

To measure the latency it was created a mechanism that takes into consideration real-world scenarios and also the maximum sustainable throughput. This mechanism was made to evaluate the performance of stateful operators. The data was generated on-the-fly instead of reading it from a broker because in a lot of cases this reading was the bottleneck and not the processing of the engines. In a general way, Flink achieved the best results, except when the data was skewed, in this situation Spark performed better. The results obtained by Karimov et al. [50] are similar to the ones from Inoubli et al. [49], where both, Flink and Spark achieved similar results in performance.

2.6.2 Summary

In Table 2.1 are the most important design characteristics and properties of each of the platforms that were mentioned in detail.

The operators/functions relationship model occurs in 4 different ways in the mentioned platforms:

- **Decoupled Model**: Any operator can send their results and data to any other operator or external system. In this model any operator can also receive data from an external system;
- **Directed Acyclic Graph**: The operators communicate in the order of the graph (the operators are the vertices and the edges the relationships), and the graph cannot have any cycle;
- **Directed Graph**: The same as the previous but with the difference that the graph can have cycles;
- **Tree Structure**: There is one operator on the top of the tree that receives all events and then passes them to the others. Every other operator is a child of the top-level operator.

The decoupled model has the advantage of arbitrary communication between all operators and external systems. The operators can receive the data directly instead of having it routed to other operators. The directed graphs have the advantage of knowing the order in which each operation is performed. The tree structure has the advantage of the top-level operator deciding where each event should go, but has the disadvantage of the events having to pass to all parent operators to get to an operator.

In Akka, the operators are resources that need to be explicitly started and stopped. They also need to be destroyed when they are no longer needed. This is not a common approach, in the majority of the platforms, that will be handled without the need for a user's intervention. When an operator stops it will

Table 2.1: Principal Design Characteristics and Properties of each platform

Platform	Design Characteristics and Properties
Flink	Stream and graph processing; Bounded and unbounded streams; Dataflow model; Scalable; Control events; Fault-tolerance; Distributed;
Stateful Functions	Same as Flink plus support to shared state in an arbitrary way without the use of storage;
Spark	Stream and graph processing; RDDs; State sharing with the RDDs; Cluster managers; Resource sharing between applications; Fault-tolerance; Distributed;
GraphTau	Same as Spark plus dynamic graphs;
Storm	Stream processing; Dataflow model; Dataflow can have cycles; Fault-tolerance; Distributed;
Heron	Same as Storm plus better debugging, fault-tolerance and scalability; Dynamic and adjustable data rate through the dataflow;
Cloudburst	Stream processing; Anna KVS for managing, manipulating and sharing state, also used for fault-tolerance;
Orleans	Stream processing; Fault-tolerance; Load-balacing; State sharing;
Akka	Stream processing; Fault-tolerance; Can use eventual consistency; Tree structure; Load-balacing; State sharing; Distributed;
Pregel	Graph processing; Superstep model; Graph partitioning; Fault-tolerance; Distributed;
Giraph	Graph processing; Superstep model; Fault-tolerance; Distributed;

recursively stop all of its children operators and so on. This does not work the other way around, a child operator cannot stop its parent even if it was stopped. Another feature that operators have is the ability to watch other operators, this way when an operator is stopped, if another operator was observing it, it will receive an event. Orleans also uses an operator system with virtual instances. Orleans simplifies the management of operators' lifecycle and race conditions to state updates in a shared database. Orleans' operators also work similarly to the functions in Stateful Functions and Pregel, which are also virtual instances.

Pregel has similar functionalities as the ones in Stateful Functions. This framework allows the user-defined functions to share state with the other vertices' functions in each step, which Stateful Functions also allows. They both also provide fault-tolerance with a checkpoint system. In both of them, the functions are virtual instances that can be paused when not in use to not consume resources. In Pregel, a vertex can stop itself when its work is finished, so unlike Akka, they do not need to be explicitly started and stopped. Apache Giraph is another system where the operators communicate using a superstep model. It allows message exchanging in each step.

Apache Spark uses RDDs to represent the data. Spark and Flink both serve the purpose of batch and stream processing. They are both distributed platforms. Flink can be faster than Spark in a lot of situations because it uses a streaming model while Spark uses a micro-batch processing model.

GraphTau also uses RDDs, because it uses Spark. In GraphTau one graph is made of two RDDs, one for the vertices and one for the edges. Since GraphTau uses GraphX, it has two recovery mechanisms, one of them is a parallel recovery of lost state and the other is speculative execution. When

the system detects some failure it launches tasks to recompute the state from the last saved point, this reconstruction is applied in the nodes that failed and it can recover the missing edges and vertices of the graph.

Cloudburst uses functions in a DAG way, the same as Stateful Functions. It also allows these functions to have a state and share it with other functions. Cloudburst's functionalities are very similar to the ones of Stateful Functions.

Heron is a stream processing platform that allows to dynamically adjust the rate at which data flows through the functions, which is a functionality that the other platforms that were mentioned do not have.

All of the platforms that were previously mentioned support some sort of fault-tolerance and state recovery, and in Table 2.1 we have a comparison between the principal properties of each platform. We can see that Akka, Stateful Functions, and Cloudburst support arbitrary communication between their operators/functions, although, Cloudburst is the less efficient one because it usually uses persistent storage to exchange the state. Giraph and Pregel communicate with a superstep model, so the operators have to wait until the end of the step to receive the messages. Spark can exchange the state using RDDs, which are immutable, so it is necessary to create a new RDD to share different data. Storm and Heron process directed graphs and do not support arbitrary communication between the operators, only between those who are linked. GraphTau supports dynamic graphs. All of the mentioned platforms are scalable.

Although the mentioned platforms have many features, none of them support approximate computation out of the box to improve the performance, however, that can be added with plugins/extensions to the platforms that allow being extended.

Another important lacking feature in mainstream platforms is auto-parallelization, where the platform should be able to change its parallelism level depending on the current load or on other metrics. This feature was introduced in Flink in its latest release with the Reactive Mode. This mode adjusts the parallelization based on the current load, however, the same resources are divided into the parallel instances of the workers. Flink does not support a dynamic allocation of the resources, it only assigns the resources to different operators, it can not use more than the ones that are allocated, the lack of dynamic resource allocation is common in the mainstream platforms. Some platforms like Spark can share resources from one application to another when they are available and running in the same cluster. As previously mentioned, the mainstream platforms are lacking cumulatively support for dynamic resource allocation, elastic scaling, efficient state sharing and approximate computing.

Chapter 3

Solution

In this chapter we present and explain Approxate, which is the proposed solution. We start with an overview in Section 3.1, after that we explain the architecture in Section 3.2. Lastly there is a summary of this chapter in Section 3.3.

3.1 Overview

The proposed solution is Approxate, it is a system that manages stream and graph processing applications that are made with Stateful Functions [14], that internally uses Flink [13]. Approxate controls the resource management and allows the applications to use approximate computing when necessary. The applications use Kafka [21] to get the events and then to send their results after the processing is done. The applications run in containers through Docker [51], since it is the official and simplest way of deploying applications that use Stafeful Functions, so it is expected to be the most common approach when using it. Approxate is designed primarily to work with stream processing, however, it can also be applied to graph processing by using Gelly [13]. Gelly contains a library of algorithms that are used to process graphs and can be used with Stateful Functions.

Approxate allows the user to define requirements for the **lag** (number of produced events pending, i.e. that are not yet processed), the **throughput** (number of results being produced per unit of time), and the **latency** of the producers (complete time that is necessary for a produced result to be sent to Kafka). Approxate aims at fulfilling these requirements whenever is possible by increasing the necessary resources and decreasing the accuracy of the results with approximate computing. 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 theoretical minimum accuracy which is defined as the percentage of processed events (3.1). Naturally, the actual accuracy value can be different for several reasons since not all applications use the events in the same way (e.g., one application where only extrema, i.e., maximum or

minimum values are relevant, may suffer more than other applications where, for instance, only average results are required).

$$Accuracy = \frac{ProcessedEvents}{TotalEvents} \tag{3.1}$$

3.2 Architecture

In this chapter, we describe the components that are part of Approxate. It is composed of three components, the Approximate Library (Section 3.2.5) 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 3.2.6) 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 3.2.7). 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 (3.2). 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.

$$TimeSaved = Events * (Rate[AfterAdjustment] - Rate[BeforeAdjustment]) - RestartTime \tag{3.2}$$

In Figure 3.1 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. 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.

The components and their advantages that Approxate will use that already exist are described in the next sections. Following that, we explain the architectures of the individual components that compose Approxate. We also explain the responsibility of each component and their interactions with the other

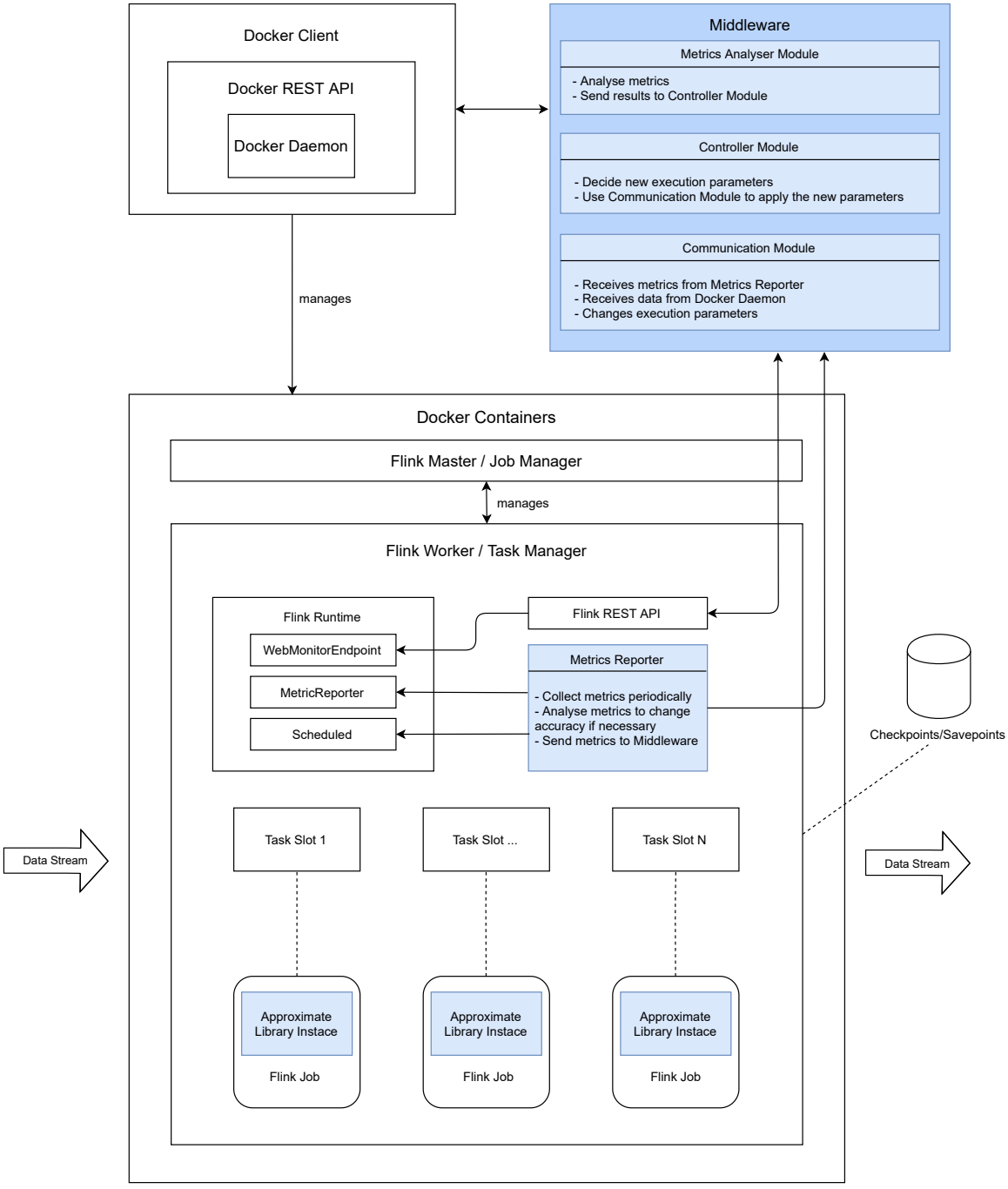

Figure 3.1: System Overview

components.

3.2.1 Flink

Flink [13] is a processing engine for stateful computations over streams. As previously mentioned it is scalable, distributed, can process data streams in a parallel way and the operators can maintain and share the state between them. It has a snapshot model for fault-tolerance and control events that are used to retrieve execution metrics.

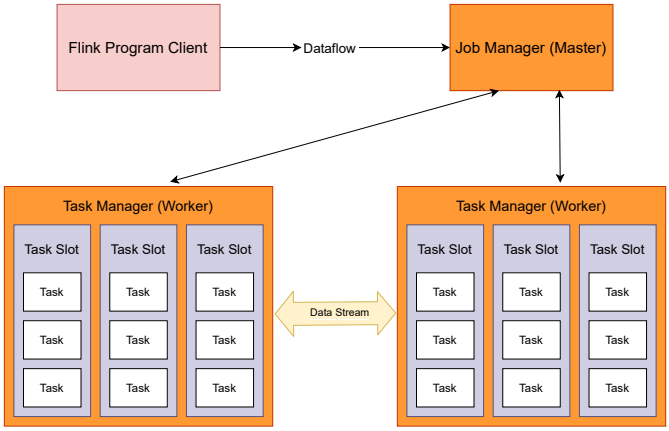

Figure 3.2: Flink's Architecture

Figure 3.2 shows Flink's architecture (i.e. of Flink Jobs workers) in more detail where we can see how a Flink program is executed. The client, which is not part of Flink's runtime, submits a job, in the form of a dataflow, to the Job Manager. Then the Job Manager has to assign that dataflow to one or more tasks. For this, the client needs to connect to a running Flink cluster that can accept job submissions. There are different types of clusters:

- i) they can be session clusters whose session will be started and stopped manually, which are not tied with the job's lifetime;
- ii) another type is job clusters where a cluster is initiated for each submitted job and it is only used for that one;
- iii) the last type is application clusters where the clusters only execute jobs that come from the same application and the *main()* method runs on the cluster instead of running in the client. In this approach, there is no need of having to start manually the cluster, the cluster entry-point will start it and this type is bound with the application's lifetime.

The Job Manager needs to coordinate the execution of the various Flink components. One of them is the Resource Manager and its role is to allocate or deallocate the available resources among the

clusters. This is done by managing task slots, which are used to execute the tasks from the dataflows. They are also used to buffer and exchange data streams. Flink can chain several subtasks together into a single task to reduce the overhead of changing threads and buffering. Each operator has a defined number of task slots available and it assigns them to perform the dataflow tasks (the tasks are the operators/functions).

Another component is the Dispatcher and it is used for submitting Flink applications and then it starts a new Job Manager for each job that is part of the application. The last component that is part of the Job Manager is the Job Master and it performs the management of the execution of a JobGraph, it will check if the instances are running, perform checkpoints among others (a JobGraph is necessary for each job that is being executed). When a job is saved the operators will save their state into persistent storage so they can resume the job when it is restarted.

3.2.1.1 Metric System

Flink contains a Metric System that allows various execution metrics to be gathered. It also allows the creation and use of custom metrics reporters, such as the one proposed in Approxate. This system also collects the JVM metrics and the Kafka metrics by using the Kafka Connector. This system is used in Approxate to collect the metrics periodically, which is done by using the **AbstractReporter** and **Scheduled** classes of Flink.

The **AbstractReporter** gathers all metrics and separates them by their type. This class can be extended to build custom reporters, by using this class the reporters already have the metrics gathered, they just need to filter the necessary ones to use them. The **Scheduled** class is used to perform the **AbstractReporter** and the custom reporters operations periodically.

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 consumed in a Kafka 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;

- **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 a 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.

3.2.2 Stateful Functions

Approxate is designed to work with Stateful Functions [14] because it supports arbitrary communication between the functions (the communication does not need to happen in the operators' order in the dataflow), and any functions can share their state with another without the need of using persistent storage. It also facilitates building streaming processing applications because it generates the dataflows that are executed in Flink. Since Stateful Functions uses Flink it has the same advantages, like the snapshot model and the ability to process graphs with the library Gelly.

In Figure 3.3 we can see how Flink gets the data that is sent to the Stateful Functions cluster through the Ingress Connectors from Kafka, resulting in function instances being invoked by the Function Dispatcher to process the data, and after that, the results can be stored or/and sent to Kafka by the Egress Connectors. We can also see that Flink uses the Metrics Reporter to communicate with the Middleware. The Flink Master is responsible to manage the Flink components (Job Manager, Resource Manager, Task Manager, and Dispatcher).

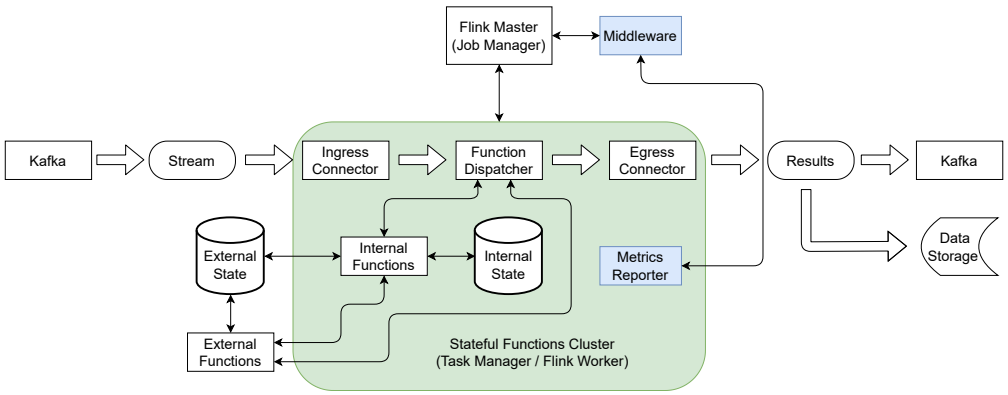

Figure 3.3: Flink's Execution Flow with Stateful Functions

The Stateful Functions cluster is responsible for routing the messages and also maintaining the state. Unlike Flink, the computations do not need to happen in the cluster partitions, they can happen in the functions' services. Flink is also responsible for invoking the stateful functions.

3.2.3 Kafka

Apache Kafka [21] is a distributed, partitioned, and replicated publish-subscribe messaging system. It is used to route messages (events) through different applications. It has strong message durability and also ordering of messages. Kafka uses Zookeeper [52] which is a logically centralized distributed coordination service for distributed systems. Kafka uses it to store its metadata like the location of each partition.

Kafka divides the messages into categories called Topics, which are part of Brokers, the Kafka architecture is in Figure 3.4. Each broker can store a number of topics and each topic can be stored in a number of brokers. The brokers divide the topics into partitions. Each partition can have consumers consuming the messages and producers sending messages to the partition.

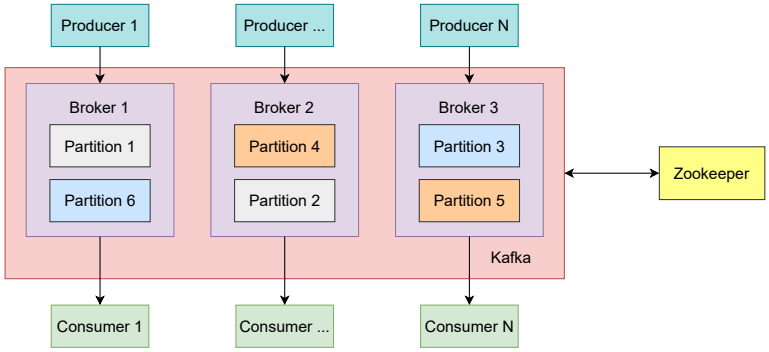

Figure 3.4: Kafka's Architecture

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 from one topic or more and will keep the offset value to know how many events they consumed and how many are they behind from the latest (lag value), they usually consume events in batches. Similar to the consumers, the producers can also write events to more than one partition. After a restart happens to the applications, if the job was saved, the consumers and producers will know their offset and they can continue to consume/produce events from where they left. The Flink internal metric system receives the Kafka metrics by using the Kafka Connector which allows it to receive the metrics periodically.

3.2.4 Docker

Docker [51] is a platform that allows the user to run applications in containers. It offers controls to the resources that an application can have access to, including a priority system for the CPU time, so one container can have priority over the other, but if the containers with higher priorities are not using the CPU, the containers with less priority can use it.

The running of applications in containers also has other advantages like the fact that the entire code

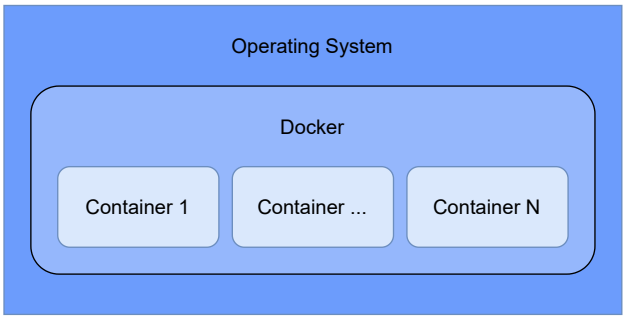

Figure 3.5: Docker

and dependencies are all packaged together so it is easy to change from one computational environment to another. As previously mentioned Docker is the official way of deploying Stateful Functions applications, so it is used in Approxate.

Approxate utilizes Docker to limit the resources that each application can have and also restart the containers where the applications are running, if necessary. In Figure 3.5 is the representation of the Docker containers that run through Docker. The Docker Engine is responsible to limit the resources of the machine that the containers can use.

3.2.5 Approximate Library

This component is used inside the Stateful Functions applications. It intercepts the events that the applications are receiving before they are routed to the functions that will process them. It is used to perform load shedding based on the current accuracy value.

$$Rate = \frac{DroppedEvents}{TotalEvents} \tag{3.3}$$

This component, whose logic is represented in Figure 3.6, is invoked for each event. Then it will use a random selector to decide if that event should be dropped or not. This decision will be based on the current minimum accuracy level in the application. 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 so the Approximate Library can drop events with the same percentage between the different 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 skipped/dropped events and the total number of events (i.e. the rate (3.3)) globally and for each data source. If there is only one data source that verification will not occur.

If that verification did not occur when using multiple data sources, some data sources could be under-represented while others would be over-represented. Figure 3.7 contains an example of two ways of 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

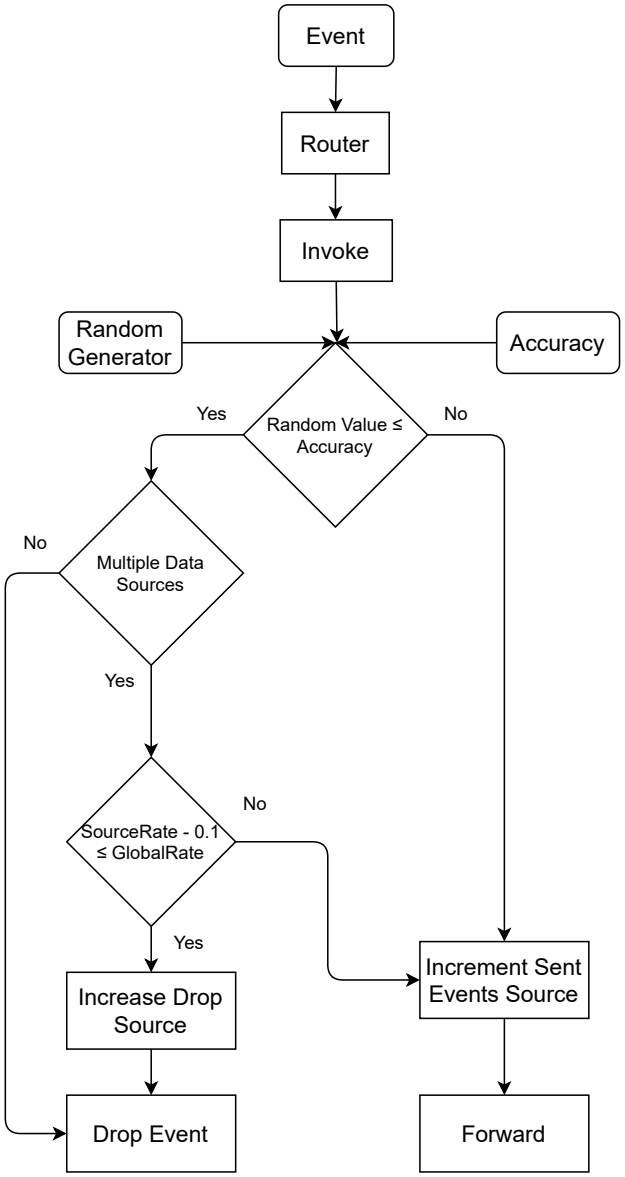

Figure 3.6: Approximate Library

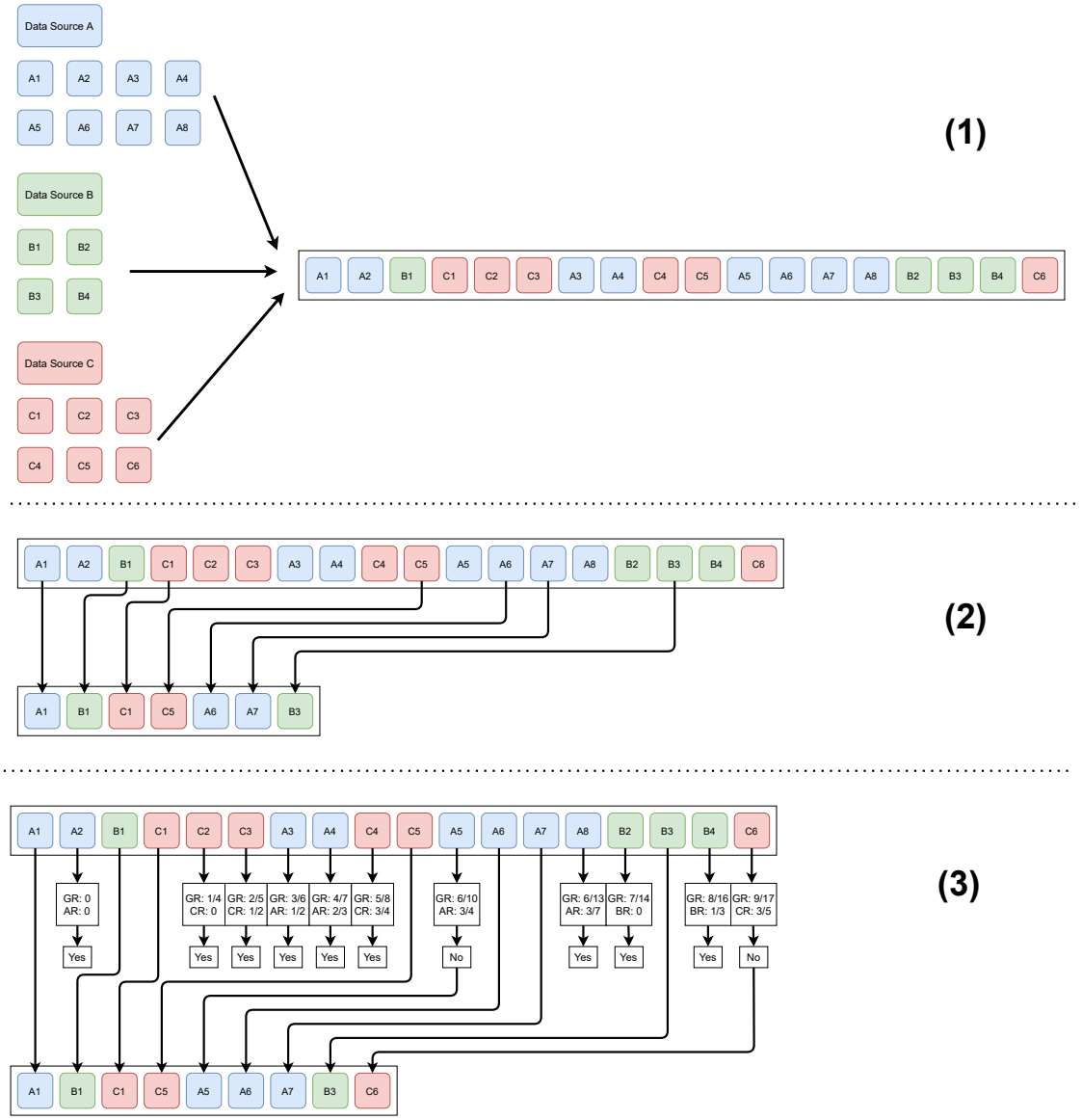

Figure 3.7: Load Shedding

some events are randomly selected to be dropped. They are the A2, C2, C3, A3, A4, C4, A5, A8, B2, B4, and C6. This part is equal for the two examples.

Part **(2)** represents the example where the load shedding is done without verification for the data sources representations, so what happens is that 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.

$$Rate(DataSource) - 0.1 \leq Rate(Global) \tag{3.4}$$

In Part **(3)** of the figure the load shedding is done with the algorithm. The events that were randomly selected to be dropped must verify the condition of Equation (3.4). The **GR** is the current global rate 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 $\leq$ 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.

After all iterations, we can compare the result with the result from Part **(2)**, and with the algorithm of the Approximate Library, 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%.

Even when the accuracy value changes, the library still uses the same formula (3.4) to decide if it can skip an event, which means that the percentage of dropped events for each data source are close to each other. Before that, the Approximate Library checks if the rate (3.3) from the data source with 10% less is equal or lower than the global rate, and the event is only dropped if that condition is verified.

The value for dropped events that this component is targeting is the theoretic value, and in ideal conditions (e.g. all of the events from different data sources arrive in a sequence where they have equal representation among them) it can be achieved. However, the actual percentage of dropped events will most likely be less than the targeted, because sometimes when an event is selected to be dropped, it cannot be because of its data source rate and the global rate.

When that happens the Approximate Library will not select another event instead of the one that

was not dropped, it will continue to select the events by chance, this way the overhead of the library is smaller, but the actual percentage of dropped events in most cases will be lower than the theoretical value.

The subtraction of 10%[1] of the rate is necessary to keep the percentage of dropped events closer to the one that is defined. Without that subtraction, the Library will not drop most of the events, unless the events arrive in the ideal conditions described previously. With less 10%, the amount of dropped events is closer to the targeted one and the representations between the different data sources are still maintained.

The Equation (3.5) gives us the expected results' accuracy, the accuracy is calculated by summing the percentage of the number of processed events for each accuracy ($Accuracy \subseteq [1, 100]$) multiplied by the accuracy value.

$$FinalAccuracy = \sum_{Acc=1}^{100} \frac{Events(Acc) * Acc}{TotalEvents} \tag{3.5}$$

Algorithm 1 Approximate Library Pseudo-Code

```
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)
10: function CANSKIP
11:     globalRate ← global rate
12:     sourceRate ← data source rate
13:     if sourceRate - 0.1 ≤ globalRate == true then
14:         return true
15:     return false
```

In Algorithm 1 is the pseudo-code of the Approximate Library. As we can see after the library is invoked it verifies the accuracy and uses it to select or not the event randomly to be skipped, and then verifies if it can skip that event. If it is not skipped then it is forwarded to the operator that will process it.

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 of each arriving event when deciding if it would be dropped or forwarded to the functions. The Approximate Library trades a total balance of the data sources representation for performance.

[1] This is not a precise value, it was decided by observing how the Library behaved when different values were chosen.

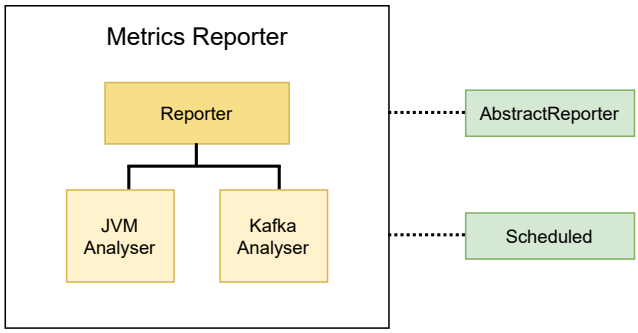

Figure 3.8: Metrics Reporter Diagram

3.2.6 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. As previously mentioned both of these classes are part of Flink's Metric System.

By using the Metric System (Section 3.2.1.1), the Reporter can collect the JVM and Kafka 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. It will also verify if the CPU usage or memory utilization is adequate to the quantity of allocated resources.

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.

In Figure 3.8 we can see the two modules of this component. It uses an analyser for the metrics produced by the Java Virtual Machine, which is where Flink is running, and they are used to verify the processor and memory utilization. It uses a different analyser for the metrics produced by Kafka which give us the necessary information about the events that are coming in and going out of the application, e.g. how many events are waiting to be processed, and also how long it takes for a result to get to Kafka after being generated.

Algorithm 2 Metrics Reporter Pseudo-Code

```
1: function REPORTMETRICS
2:     resourceMetrics ← get resource metrics
3:     requirementMetrics ← get requirements metrics
4:     requirements ← get requirements
5:     resourcesRes = AnalyseResources(resourceMetrics)
6:     if ResourceUsageIsNotOk(resourcesRes) then
7:         LowerAccuracy()
8:     else
9:         requirementsRes = AnalyseRequirements(requirementMetrics, requirements)
10:        if RequirementsAreNotMet(requirementRes) then
11:            LowerAccuracy()
12:        else
13:            if CanIncreaseAccuracy(resourcesRes, requirementsRes) then
14:                IncreaseAccuracy()
15:    SendMetrics() Send metrics to the Middleware
```

In Algorithm 2 we show the pseudo-code of the Metrics Reporter. As we can see the Reporter gets the metrics and the desired requirements and check if it should increase, decrease or maintain the current accuracy. After that, it sends the metrics to the Middleware.

3.2.7 Middleware

In this section we explain the Middleware logic and components. We start with an overview and the pseudo-code, and then we explain each component that is part of the Middleware.

The Middleware is responsible for managing the applications' executions by deciding the resource allocation and the minimum accuracy. It receives the metrics that are sent by the Metrics Reporter and after that, it analyses them. The analysis done by the Middleware take into consideration more factors than the Metrics Reporter's analysis, the Middleware has access to more information (quantity of available resources, parallelism level) and can also do more adjustments than the Metrics Reporter, which can only modify the accuracy level.

In Figure 3.9 we show a diagram of the Middleware. 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 where they are analysed. After that, it receives the results from the Metrics Analyser Module and it verifies which adjustments are possible to perform. 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.

In Algorithm 3 we show the pseudo-code of the Middleware. As we can see it starts by getting the metrics and loading the configurations. After that, it will analyse the metrics using the user-defined execution requirements, if they exist. Then the Middleware generates a result for each metric. When

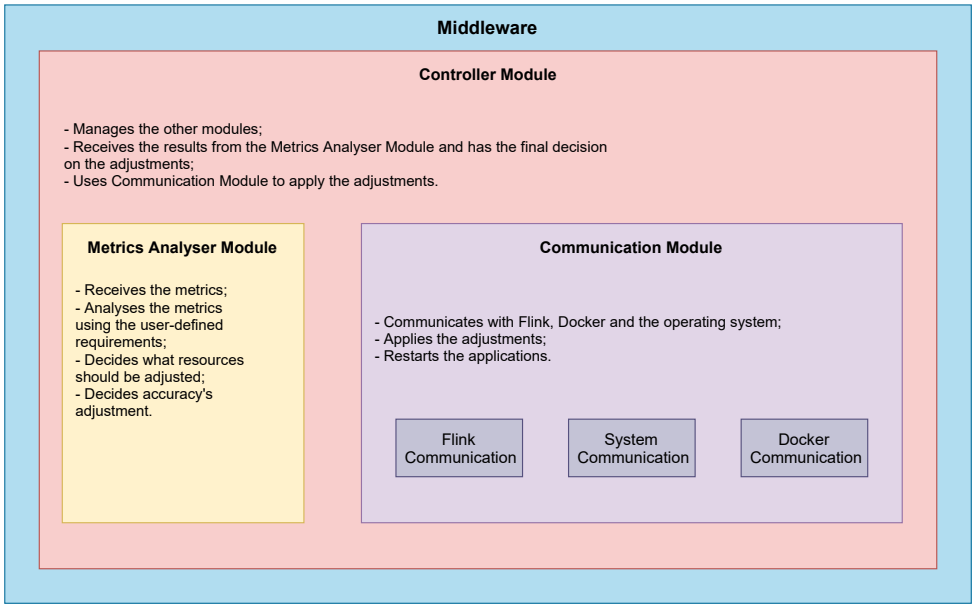

Figure 3.9: Middleware

Algorithm 3 Middleware Pseudo-Code

```
1:  function MAIN
2:      configs ← load configs
3:      metrics ← get metrics
4:      results = AnalyseMetrics(metrics, configs) ← get the new configurations
5:      ApplyResults(results) ← apply the new configuration if necessary
6:  function ANALYSEMETRICS(metrics, configs)
7:      resultsList ← initialize empty list
8:      for metric in metrics do
9:          minDesiredValue = GetTarget(metric) ← get desired value for the specific metric
10:         result = AnalyseMetric(metric, minDesiredValue)
11:         resultsList.Add(result)
12:     finalResults = AnalyseAllResultsCombined(resultsList)
13:     return finalResults
14: function APPLYRESULTS(results)
15:     for result in results do
16:         ApplyResult(result) ← modify the configuration files and/or the Docker containers
17:     if RestartIsNecesseray() == true then
18:         Restart()
```

all metrics have been analysed, the results are used to modify the execution resources and accuracy, if necessary, and also the Docker containers resources. After the results have been applied it will restart the application depending on the parameters that have been modified.

3.2.7.1 Controller Module

This module starts by loading the configurations of Middleware. It then uses the Communication Module to receive the metrics from Flink. After receiving the metrics it uses the Metrics Analyser Module to analyse them and decide what adjustments it will perform on the resources and accuracy.

After receiving the results this module verifies if the adjustments decided by the Metrics Analyser Module can be performed (i.e. if it decided that the memory should increase it will verify if it is already at the maximum). After that, it uses the Communication Module to perform the adjustments.

3.2.7.2 Metrics Analyser Module

This module analyses the metrics that are described in Section 3.2.1.1. It uses 2 levels for increasing or decreasing the resources and the accuracy. We chose this approach because it allows a more precise control than just using an increase or decrease as a result of an analysis. The maximum/minimum levels are used to adjust the **parallelism** and **accuracy** to the maximum/minimum possible levels. The other levels adjust them in steps, the parallelism step can be defined in the options and the accuracy is adjusted 10% at a time. We chose this value because it is enough to have an impact on the performance, but not a major one.

The **memory** is always adjusted in percentages that are relative to the increase/decrease level. It can be increased by 25% or 50%. When it is decreasing it is by 25% or 40%. We chose these values because they allow the system to increase without using all the memory, which could lead to instability (i.e. if the memory increased to the maximum amount that can be used, it could lead to wasted memory and so Approxate would have to decrease the memory afterward). When decreasing, it is used 40% instead of 50% due to observations of how the different tested scenarios behaved, using 40% leads to more stability than 50%.

The **CPU** increases are done with Docker shares, when the result is to increase to the maximum it will increases the shares to double of the default value, so it has twice the default priority. Otherwise, it increases a quarter of the default value at a time. When is to decrease it also decreases a quarter of the default at a time. When is to decrease to the minimum it decreases to one-eighth of the default value.

The levels of adjustment are decided based on percentages. If a requirement is not being met by a margin of at least 15% then it is considered that the resources should be increased to the maximum level. This module uses a similar model in the analysis of the resources utilization (i.e. if the CPU load is 25% or less then CPU priority should be decreased to the minimum). We decided the values for defining

each level of adjustment to get a progressive adjustment of the resources whenever is possible to avoid over-increasing or decreasing. This way we can achieve a more stable system.

This module uses the **CPU load** to check if the system is under load or not. Depending on the load the Middleware can increase or decrease the level of parallelism, the accuracy, and the priority of the container where the application is running in Docker. 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, which means that if the host machine is not under load any container can use the CPU for any given time, but if the machine is under load some containers will have priority over others. 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 **memory** values are used to verify if the current reserved memory for that application is enough, or if it is too much, given the memory in use and the memory committed. 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.

The **lag** value is used to verify if the lag requirement is being met. If the requirement is not being met, the Middleware will increase the available resources and decrease the accuracy until the application is meeting the requirement. If the lag value is high it means that the Kafka consumers that are running in the application are not being able to keep with the rate of the events' production.

The **latency** value is used in the same way as the lag one. A high latency value means that the events produced in the Flink application that are sent to Kafka are taking a long time before being received. This can mean that Kafka is under load and cannot keep with the rate of the producers or that the application is under load and cannot send the events in a short amount of time after they have been produced. Since Approxate is not analyzing Kafka, we assume that if the latency is high is because of a lack of resources and the Middleware increases them and lowers the accuracy.

The record send rate is the **throughput** and it is used to check if the requirement is being met, if it is not being met it would mean that the application can no longer keep the desired processing rate which can be caused by having all of the resources being used and none available. This value is also used to increase or decrease the resource allocation and accuracy.

After all of the metrics are analyzed this module combines their individual results to decide what modification it will do. This combination starts by checking if the requirements are being met or not. If any of the requirements is not being met by a large margin (15% of the target value), then it will increase the parallelism level to the maximum and decrease the accuracy level to the minimum. After that it verifies if any of the resources' results about the processor and the memory is to decrease, and if that is the case, then it changes the results to maintain the allocated resources. However, if those results are

to decrease at the maximum level (which indicates that it has too many resources), they are modified to just decrease. If those results are to increase then they are changed to increase to the maximum.

Another case that can happen is when some requirement is not being met by a smaller margin than the previous case (less than 15%). When this happens, the Metrics Analyser decides that the accuracy is decreased and the parallelism increased, but not to their maximum/minimum levels. The parallelism increases by a amount that can be defined in the options and the accuracy decreases 10%. It also checks the resources' results and if they are to decrease at the maximum level they are modified to just decrease, otherwise they stay the same.

If the requirements are being met then the focus is in checking the resources' results and not generate contradictory results (i.e. if the parallelism level increases it is expected that the CPU and memory utilization will increase, so it does not make sense reducing those to the minimum level).

To verify the resources' results, the Metrics Analyser adjusts the parallelism level according to the CPU result (i.e. if the CPU increases the parallelism also increases, if the CPU increases to the maximum then the parallelism also increases to the maximum). The accuracy level is also adjusted in an inverse proportion of the CPU level (i.e. if the CPU decreases to the minimum then the accuracy increases to the maximum). The memory result can also be modified in the cases where the CPU result indicates to increase it to the maximum and the memory result indicates to decrease it to the minimum, in this case the memory just decreases since it is expected to get more usage after the CPU and parallelism adjustment.

With this analysis, the Middleware can identify what resources are needed to increase or decrease (e.g. it can increase the amount of memory while decreasing the CPU priority). To conclude, this component can adjust the CPU priority, the memory limit, the level of parallelism, and the level of accuracy based on the current metrics.

3.2.7.3 Communication Module

The Communication Module has three sub-modules. The first is the Flink Communication that is used to receive the Flink metrics in an HTTP web socket and to trigger the savepoints in Flink's webpoint by using its REST API. The second is the System Communication which is used to change the Flink application's configuration files and to restart the execution by using the system to build new Docker images with the new Flink configurations. The last one is Docker Communication that is responsible to retrieve the containers' information, changing the resources that the applications can use, and adjusting the applications CPU priority over the others.

3.3 Summary

This chapter describes Approxate's architecture. It describes how Approxate interacts with Stateful Functions' applications, and how it interacts with Flink and Docker.

Approxate collects and analyses the metrics periodically. Based on the current load, and in the user-defined values for lag, latency, and throughput, it adjusts the resource allocation and also utilizes approximate computation to lower the results' accuracy and increase the performance.

The approximate computation is done using a technique called load shedding which consists of dropping some of the events instead of processing them. Approxate takes into account the data source of the events before they being dropped, and so the results will eventually represent all of the data sources in the same percentage as the received events. This way no data source is under-represented in the results.

This chapter also shows how the theoretic value of the accuracy is calculated. It also shows the formulas that can be used to calculate the final theoretic accuracy of the results and to verify if it compensates, regarding the time, to restart the application after adjusting the resource allocation.

Chapter 4

Implementation

We describe the implemented components in this chapter. We explain the used APIs/libraries and their purpose. All of the components are implemented in Java, since it is the language that Flink and Stateful Functions use.

4.1 Approximate Library: Implementation

The Approximate Library is comprised of four classes. The first class is the **ApproximateMessage** class and it is used as a wrapper for the input events of 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:

- **Id**: An object of String type and it is the event identifier, which is used to route the event in the applications;
- **Ingress**: An object of String type and it is the identifier of the data source of the event;
- **Message**: An object of generic type and it is the event.

The next classes are the **ApproximateSingleIngressResultsFunction** and **ApproximateMultipleIngressResultsFunction**. They are used to select and drop the events according to the accuracy value. The main difference between those is that one of them is used when there is only one data source, so it does not keep a data source or global rate, it just drops the events that are randomly selected.

A code snippet of the creation of the **ApproximateMultipleIngressResultsFunction** is in Listing 4.1. The instance can be created in two ways, in the first it receives a **Map** that contains the relationship between each input event data source and the functions that can process that event. In the second way, it receives the identifier of the function that can process the events and receives a list of the data sources

that can produce the events. Although it only has a single function type, it is still necessary to create a map, since the methods that forward the events search the map using the data sources to obtain the functions types.

These 2 ways exist because an application can have one or multiple functions to process the events when they arrive. The **FunctionType** is a Stateful Functions object type and is used as part of the functions' identifiers. It is used when the events are routed to the functions that will do the processing.

After the instance of the Library is created it receives the events and performs the load shedding. The method used to perform the load shedding 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.

When an event is not dropped it is forwarded to the functions using its **FunctionType** and a **Downstream** object, which is a Stateful Functions' object used to route events through the application.

The last class that is part of the Approximate Library is the **RecordCount** and it is used to keep track of the percentage of global and data sources dropped events. This object contains a map to store the information about each data source, that information is stored as an object of type **CountInfo** which is an inner class of **RecordCount**. The **CountInfo** just stores the number of total events and how many were dropped, and can also return the rate.

Listing 4.1: Library Instance Creation

```
1  public ApproximateMultipleIngressResultsFunction(Map<String, List<
       FunctionType>> ingressMap){
2       this.ingressMap = ingressMap;
3  }
4
5  public ApproximateMultipleIngressResultsFunction(FunctionType functionType,
       List<String> ingresses) {
6       ingressMap = new HashMap<>();
7       List<FunctionType> list = new ArrayList<>();
8       list.add(functionType);
9       for (String ingress : ingresses) {
10          ingressMap.put(ingress, list);
11      }
12 }
```

4.2 Metrics Reporter: Implementation

The Metrics Reporter is a Flink plugin that is comprised of 5 classes. One is the **Commom** that is used by the others to perform computation or extract values from Strings. Another class is the **CustomReporterFactory** and it is used by Flink to create an instance of the Metrics Reporter.

Two of the classes are the **JvmAnalyser** which analyses the metrics generated by the JVM to check the utilization of the available resources, and the **KafkaAnalyser** which is used to verify the metrics generated by Kafka. It loads the user-defined requirements about throughput, latency, and lag through the JVM properties system (the requirements 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. 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 it was not possible to be evaluated due to the user not having defined a requirement for that metric. An example of one of the metrics being analysed is in Listing 4.2 where the metric value is compared to the desired one.

Listing 4.2: 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 }
```

The **CustomReporter** class extends Flink's **AsbstractReporter** and implements Flink's **Scheduled** classes. It is responsible for filtering the metrics collected by Flink's Metric System. After it filters the desired metrics it analyses them using the previously described analysers. The analysers return an `Integer` value: if it is negative then the accuracy needs to be lowered, if it is positive and enough metrics returned a positive value then the accuracy will be increased. It can also let the accuracy stay

the same. The accuracy adjustment is done by modifying the JVM's accuracy system property. This component does not increase or decrease the resources, only the accuracy.

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.

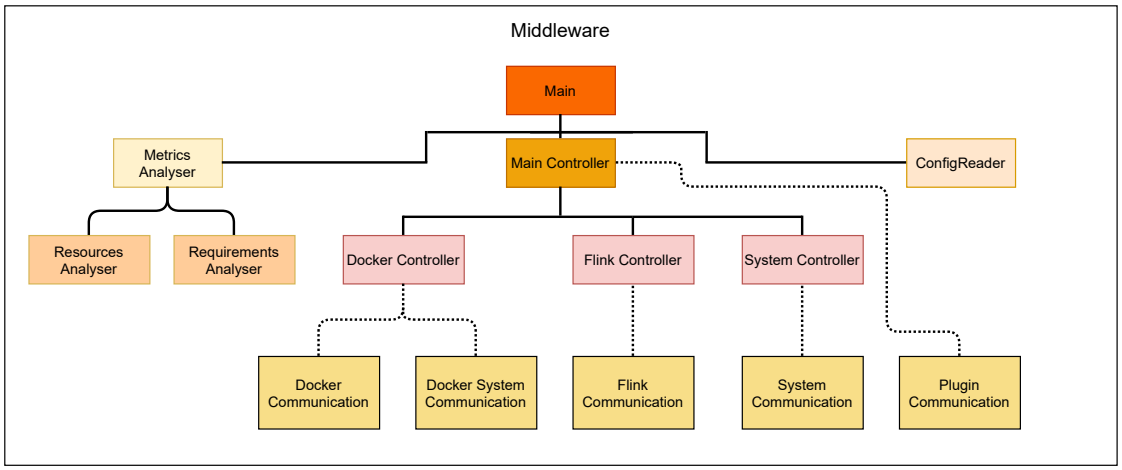

Figure 4.1: Middleware Classes

Some of the most important classes are represented in Figure 4.1. The less relevant classes like utility or exception classes are not represented in the diagram.

The **Main** class starts and controls the Middleware. It starts by creating a **ConfigReader** object and uses it to read the Middleware's configuration file, which location must be passed as an argument when starting the Middleware. The Configuration Reader is responsible for reading the configuration parameters and loading them up. These parameters allow the user to define some options, the most relevant are the following:

- The maximum number of processors that the application can use;

- The minimum and maximum memory that the application can use;
- The minimum and maximum application's parallelism level;
- The minimum value for the accuracy;
- The maximum value for lag;
- The maximum value for latency;
- The minimum value for throughput;
- How often the Middleware will check if it has new information and perform a metrics analysis;
- The time to wait for the application to save its state before assuming that an error has occurred.

After the configurations are loaded it creates the **Main Controller** and the **Metrics Analyser**. The **Main Controller** creates the Web Socket that receives the metrics from the Metrics Reporter with the **Plugin Communication** class. The Web Socket is kept running in a thread in the background. It receives the metrics through datagrams and stores them in a concurrent linked queue. This class also keeps a flag to know if the metrics were already used or not, so the Middleware does not analyse the same metrics twice.

The **Main Controller** is also responsible for executing the commands sent by **Main** after getting the results of the metrics analysis that are done by the **Metrics Analyser**. This component uses the other three controllers to execute the commands.

The **Docker Controller** is used to retrieve statistics about Docker (priority of the application's containers and information to identify which containers belong to the application) and to change the configurations (CPU time priority and maximum allowed memory) about the containers that are being used to run the applications. It uses the **Docker Communication** class to send the commands to the Docker daemon.

The communication with the Docker daemon is done through a **DockerClient** instance which is part of *docker-java*.[1] This is an API for Java applications that allows them to send requests to the Docker daemon to perform 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 4.3 there is an example of one request that is made with the **DockerClient** to retrieve the statistics about one container.

Listing 4.3: Get Container Statistics Method

[1] `https://github.com/docker-java/docker-java`

```
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(" to retrieve container statistics")
               ;
11     }
12 }
```

The **Docker System Communication** is also used to communicate with Docker, but instead of using the Docker Engine API, it uses the Docker Compose as a way of managing the containers' life-cycle. 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. This is one of the ways of deploying Stateful Functions through Docker and was chosen to be used in this work because the applications use multiple containers, and multiple containers can be created and started using Docker Compose configuration files. The **Docker System Communication** is only used to build the container images when it is necessary to add a Flink's saved state location so the application can resume the work from where it was left, and then restart the containers.

The **Flink Controller** is used to communicate with the Stateful Functions' applications. It is only used to send the necessary requests to save the state. It sends the requests using the **Flink Communication** class and then it returns the saved state location. It also keeps a flag to know if a save is occurring so it does not send two save requests at the same time.

The **Flink Communication** utilizes a *OkHttpClient*[2] 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.

After that, it receives the response almost instantaneously, but the response just acknowledges that the save was started. For the application to be restarted from the same state it needs to know the save

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

location, so the **Flink Communication** will send another request about the save that was triggered and Flink will respond that the save is in progress or will respond with the saved state location. This last request will be retried a fixed amount of times (that can be changed in Middleware's configuration file) with a pause of 1 second between each try, or until the response includes the location.

The **System Controller** is used to read and modify the configuration files of Flink and Docker. It also keeps an internal state of some configurations to avoid to re-read the files. This class receives the desired resources' adjustments (results from the **Metrics Analyser**) from the **Main** and modify the configuration files using the **System Communication** class. It calculates the new values for the resources based on the results, e.g. if it receives a result that indicates that the memory should be increased it will calculate the new value based on the current reserved memory value.

It also verifies if the desired adjustments are possible, i.e. if an application is already with the maximum parallelism level and the analyse of the metrics returns a result that indicates to increase the parallelism, the **System Controller** will not try to increase more. In the same way, it also verifies the minimum values when it receives commands to lower the resources.

The **System Communication** class is used to read the configuration files, parse them, and then modify the necessary lines. It contains methods to open Flink applications configuration files and read and/or update the values for parallelism, accuracy, and memory. This class also reads the Docker Compose files to insert the Flink saved state location that is necessary when restarting the applications.

The **Metrics Analyser** class is used to analyse the metrics and returns results that represent the desired adjustments. It starts by analysing the utilization of the resources with the **Resources Analyser** class. After that, it uses the **Requirements Analyser** class to check if the requirements that the user can define for the processing are being met.

The **Resources Analyser** simply check if the application's CPU usage is very low, low, high, or very high, it does the same for the memory. This class also takes into account how many processors the application can use because Docker can restrict that. The collected metrics indicate the CPU usage in terms of the total host machine CPU utilization, so if the container where the application is running only has access to 2 processors and the machine has 4, the reported value in the metric would be half of the real utilization. A 50% utilization would mean that the application used 100% of the CPU that it can access. After that, it does the same thing for memory utilization.

The **Requirements Analyser** will compare the metrics values about lag, throughput, and latency with the user-defined requirements. It returns, for each requirement, if it is being met or if the resources should be raised.

The Middleware uses two levels of increase/decrease to adjust the resources, parallelism, and accuracy. For any of those properties there is the *X_UP_MAX*, *X_UP*, *X_DOWN_MAX* and *X_DOWN*, where *X* is the property identifier. An adjustment with an *MAX* variant means that the property will have a greater increase/decrease than the non-*MAX* variant. For example, if a requirement value is not being met by a

difference of 10% the returned result is that the resources should be increased, but not by much, so it is the *RESOURCES_UP* result.

Listing 4.4: Get Latency Result Method

```
protected static Result getLatencyResult(List<Metric> requestLatencyMax)
    throws ConfigLoadException {
    if (requestLatencyMax == null) {
        return null;
    }
    double maximumLatency = configs.getMaximumLatency();
    double latency = greaterMetricValue(requestLatencyMax);
    if (maximumLatency == -1) {
        return OK;
    }
    if (latency > maximumLatency) {
        double percentage = latency / maximumLatency;
        if (percentage > 1.15) {
            return RESOURCES_UP_MAX;
        }
        return RESOURCES_UP;
    }
    return OK;
}
```

In Listing 4.4 an example is shown of how the **Requirements Analyser** checks the latency requirement; the other requirements, and resources utilization are checked similarly.

The *getLatencyResults* method returns an object of type **Result**, which is a class of Middleware that is an Java **Enum** (the **Result** class is used to represent the results from the **Metrics Analyser**). This method starts by verifying if the latency list is null, a list is used because the latency of each KafkaProducer produces a metric, and the list contains the metrics of all producers.

After checking if the list is null, it checks if the user defined a value for the desired maximum latency, if it is not defined the *configs* object, which is an instance of a **Configurations Reader**, returns -1. If that is the case then it returns the *OK* result, which is a result that is neutral in the analysis of the combined results.

If that requirement is defined, then it is compared with the greater latency value among the producers, which is the maximum latency value of the application. If the producer latency is equal to or smaller than the required value this method returns the *OK* result. However, if the latency is greater than the desired

one, this method will calculate the percentage of how much greater it is. If the producer's latency is 15% or more than the required value it is returned the *RESOURCES_UP_MAX* result. If it is smaller than 15% it is returned the *RESOURCES_UP* result. The thresholds where we decide which result should be returned were defined to allow a balanced increase in resources allocation when the requirements are not being met without allocating more than necessary.

After the resources' utilization and the requirements are analysed, the **Metrics Analyser** uses all of those results and combines them using Algorithm A.1 to decide the final results.

The function starts by creating a list where it will store the results. The next step is to check if any of the requirement results is equal to *RESOURCES_UP_MAX*. This means that a requirement is not being met by a large margin. If that is the case it adds the results that indicate to decrease the accuracy to the minimum and to increase the parallelism to the maximum.

Following that it verifies if the CPU/memory utilization's result is to increase and if it is, then it adds to the list the result to do the maximum increase of that resource. If the result is to decrease to the minimum level it changes that to only decrease slightly. This is done because the parallelism level is going to increase, and thus it is excepted that the application will use more resources.

If no requirement result is *RESOURCES_UP_MAX*, and instead there is one *RESOURCES_UP* result, the function will add the results to lower the accuracy and to increase the parallelism. However, it will not increase the result value from the CPU or memory, if they are to increase or maintain they will stay the same, but if they are to lower at the minimum level they will be changed to just lower.

If none of the results from the requirements' analysis is to increase the resources, then this function adds the results from the CPU analysis. Next, if that result is the maximum level increase, it will check if the memory result is to decrease at the maximum level, or if it is to just decrease. In the first case it is changed to just decrease, in the latter it is changed to maintain the same memory. In any other case (other results), the memory result is not changed. The accuracy also decreases to the minimum and the parallelism increases to the maximum.

This is done because is expected an increase in memory utilization. If the result is to decrease the memory at the greater level it means that there is much memory that is not being utilized. However, with the increase in the CPU and parallelism, the memory may increase, so it is not decreased at the maximum level. If the memory decreases it still should be enough. Also if the function gets here, all requirements are being met and they are expected to continue that way, thus the memory can decrease.

If the CPU result is any other it is checked to adjust the accuracy and parallel level accordingly. The memory result stays the same.

4.4 Summary

All components[3] are implemented in Java because it is the used programming language for Flink and Stateful Functions, although Scala is also officially supported.

The Approximate Library uses a data structure to keep track of the percentage of dropped events for each event data source. The used algorithm allows for an eventual balance of data-sources representation in the approximate results. This approach is faster than using a total balance but is slower than not doing any effort to represent all data sources equally. However, the latter could affect the results' precision and they would not be representative of all the generated events. The Approximate Library gets the accuracy value through the JVM properties, where Flink inserts the value from the configuration file on start-up.

The Metrics Reporter utilizes Flink's classes and Metric System to collect the metrics periodically, it then filters them and does a basic analysis. Depending on the results it can adjust the accuracy value through the JVM properties. It also creates a datagram to send the metrics to Middleware.

The Middleware receives the metrics in a Web Socket (Datagram Socket). It communicates with Docker using *docker-java* which is a Java client for performing Docker commands using Docker Engine API. It also interacts with Docker through the Docker Compose, which is used to manage multi-containers applications. The interaction with Docker Compose is done through Java Runtime instances where the operating system is used to call the Docker Compose. The Middleware also uses the *OkHttp-Client* to send HTTP requests to Fink's REST API.

Although Flink and thus, Stateful Functions, support different cluster deployments, the currently implemented solution only supports the **application cluster** type. Approxate supports clusters dedicated to a single application in multicore machines (on to 16 cores as found in medium range cloud server instances). It currently analyses the metrics from, and applies modifications to resource allocation to, a single application cluster at a time. An extra layer of coordination to aggregate the metrics received from several applications, and to manage each one, is left as future work.

[3]The project code can be found in the following repository: `https://github.com/joao-francisco/Approxate`; currently the repository is private, but it will be switched to a public repository in the future.

Chapter 5

Evaluation

This chapter describes the solution's evaluation. We describe the used setups (Section 5.1), the metrics (Section 5.2), and the workloads and benchmarks (Section 5.3) that are used to evaluate Approxate.

After that, in Section 5.4 are the results obtained when Approxate is used in applications that are typical use cases of stream and graph processing. In this section, we describe how various types of processing cases are affected by the approximate results and how Approxate can improve the scalability and resource allocation. We also show the improvements that can be achieved in the time it takes to process the data or how many resources are necessary to do it. Lastly there is a summary of this chapter in Section 5.5.

5.1 Setup

All of the values presented in the subsequent sections are averages of multiple runs of each test to reduce random performance variations that can occur due to external motives by other processes. The local test setup consisted of a quad-core I7-6700HQ (2.60 GHz) with 16GB of RAM, and all of the different tests had runs performed on the local machine. Some of them were also performed in up to three different cloud machines[1] (16/8/4 vCpus with 64/32/16 GB of memory). We used 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 improve the performance level in lower-end machines to get near, or match, the performance of higher-end machines.

Approxate uses Stateful Functions, Flink, Docker, and three new components: the Approximate Library, Metrics Reporter, and Middleware.

[1] `https://cloud.google.com/compute`

5.2 Metrics

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

- **Accuracy**: Approxate must be able to utilize approximate computing to lower the results' accuracy 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 in 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.

5.3 Workloads and Benchmarks

We used micro-benchmarks (simple stream processing tests) to test the overhead in the communication and processing of the new components and to observe the impact of running the components with Stateful Functions. We also performed macro-benchmarks (realistic applications workloads) to test how Approxate manages the resources and accuracy. These benchmarks were also used to test how using approximate computing can affect the precision of the results in different stream and graph processing applications. Some benchmarks used real data and others used synthetic data.

They are described in the following list:

- **Greeter**: This test counts the number of messages that each user sent and then replies to it, it was used as a micro-benchmark and uses randomly generated synthetic data;

- **Ad Processing**: This test calculates the ratio of users that clicked an ad and how many times a user has clicked in each ad, it was used as a micro-benchmark and uses randomly generated synthetic data;
- **Taxi Trip:**[2] This benchmark uses real data from trips of taxis in New York and it was used to calculate various averages from the trips, like the number of trips per weekday, the money earned for each day of each month, among others. This was used to test the dynamic resource allocation and the effects in the results precision;
- **Linear Road:**[3] This benchmark uses synthetic data 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. This was used to test the effects in the results precision;
- **Synthetic Benchmark**: This benchmark uses randomly generated synthetic data and simply applies a load for each received event that consists of creating and shuffling a list, we created this benchmark to test situations where each input event can cause a high load on the CPU without affecting much the memory usage. This was used to test the dynamic resource allocation;
- **Yahoo! Groups:**[4] This benchmark uses real data from Yahoo Groups and is used to find communities between the users that are part of the groups, a community is a set of nodes that is densely connected between each other. This was used to test the effects in the results precision;
- **Yahoo! Messenger:**[5] This benchmark uses real data from Yahoo Messenger and is used to find communities between the users based on the friendship relationships between them. This was used to test the effects in the results precision;
- **Triangle Counting:**[6] This benchmark uses synthetic data and is used to calculate the number of triangles in a graph, a triangle is formed when three vertices are all connected between them. This was used to test the effects in the results precision.

5.4 Results

This section contains the obtained results and their analysis. We start with Approxate's overhead (Section 5.4.1), after that there are the results that are related to the stream processing applications (Section

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

Table 5.1: Micro-Benchmarks' Results (Local)

Test	Time (Minutes)	Relative Time
Greeter without Approxate (100% Accuracy)	13:55	100
Greeter with Approxate (100% Accuracy)	14:16	102
Greeter with Approxate (99% Accuracy)	13:21	96
Ad-processing without Approxate (100% Accuracy)	06:22	100
Ad-processing with Approxate (100% Accuracy)	06:29	102
Ad-processing with Approxate (99% Accuracy)	05:55	93

5.4.2), which contains the results related to the dynamic resource allocation (Section 5.4.2.1) and the performance comparisons (Section 5.4.2.2). Following that, are the results from the graph processing applications (Section 5.4.3). After that there are the results' analysis (Section 5.4.4).

5.4.1 Overhead

We tested the overhead of Approxate with the micro-benchmarks. The tests that were used as micro-benchmarks are simple applications where the processing of each event that the application receives is very low, so the impact of Approxate's components can be the most noticeable possible. The tests were run without Approxate (vanilla Stateful Functions) and with Approxate's components running. The tests performed with the components used 100% of accuracy (where the library simply forwarded all events) and with 99% accuracy (where the library need to perform the calculations but only 1% of the events are dropped).

All runs of the same benchmark used the same datasets. Those datasets were randomly generated. There are 2 micro-benchmarks, the first is the **Greeter**. It is an application that receives a message that is associated with a user, and just counts the number of messages that were sent by that user. This is one of the simplest and less resource-demanding stream processing applications that can be done.

The second, the **Ad-processing**, is an application that is processing advertisement information. Each event received contains a user identifier, an advertisement identifier, and if the user clicked in that ad or not. The application calculates the ratio for each ad (the percentage of users that clicked in that ad). It also keeps a record of the times that each user clicked in each ad. This test also differs from the last one in the number of data sources. The first one only had one data source, so the Approximate Library did not need to know how many events it had dropped from each source. This test was made with three data sources.

The results from the micro-benchmarks are in Table 5.1, 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. 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, while in the more complex was 7%.

These results demonstrate that the overhead of Approxate is mostly negligible in situations where it cannot improve the performance.

5.4.2 Stream Results

The stream results were used to evaluate how Approxate manages the resources and accuracy with variations on the applications' input rate (Section 5.4.2.1). They were also used to test the effects on performance and results' precision when using approximate computing (Section 5.4.2.2).

5.4.2.1 Dynamic Resource Allocation

In Figure 5.1 we can observe how Approxate managed the resources and the accuracy in **Taxi-Trip Benchmark**. This test was performed with a minimum accuracy of 70% and with the following requirements: 30000 maximum lag events, 15000 milliseconds of maximum latency and a minimum throughput of 15000 results per second. We can notice that the accuracy dropped when the CPU load increased (minute 2). Then it incremented again when the load on the system decreased (minute 6). The memory utilization increased throughout the test period and then decreased in the end when the input rate dropped (minute 11). The CPU load also lowered in the end. This test also shows Approxate's auto-parallelization, during which the parallelism increased at a constant level and then stabilized at 75% (minute 5).

After that, when the load decreased, the parallelism level also decreased. Then the rate of the input events increased (minute 10), and so the requirements were not being met. Due to this, Approxate raised the parallelism level raised to 100% and stayed at that level until the load lowered. At around the same time the accuracy also decreased to 70%, however, it was not necessary to increase the memory. This test shows how Approxate reacts to the increase or decrease in the input rate by adjusting the resources and the accuracy as necessary. We can observe that it did not adjust all the resources at the same rate, it only increase/decrease some parameters when necessary.

The **Synthetic Benchmark** was tested with a variable load, minimum accuracy of 70%, and with the following requirements: 150000 maximum of lag events, 10000 milliseconds of latency, and a minimum throughput of 1000 records per second. The results are in Table 5.2, and, they also can be seen in Figures 5.2, 5.3 and 5.4. We can observe that the system could identify the times when it was necessary to engage more resources to keep up with the execution requirements, and then it scaled according to that. Due to the application being very CPU-bounded, when the system receives a continuous load it can't keep the throughput and lag requirements.

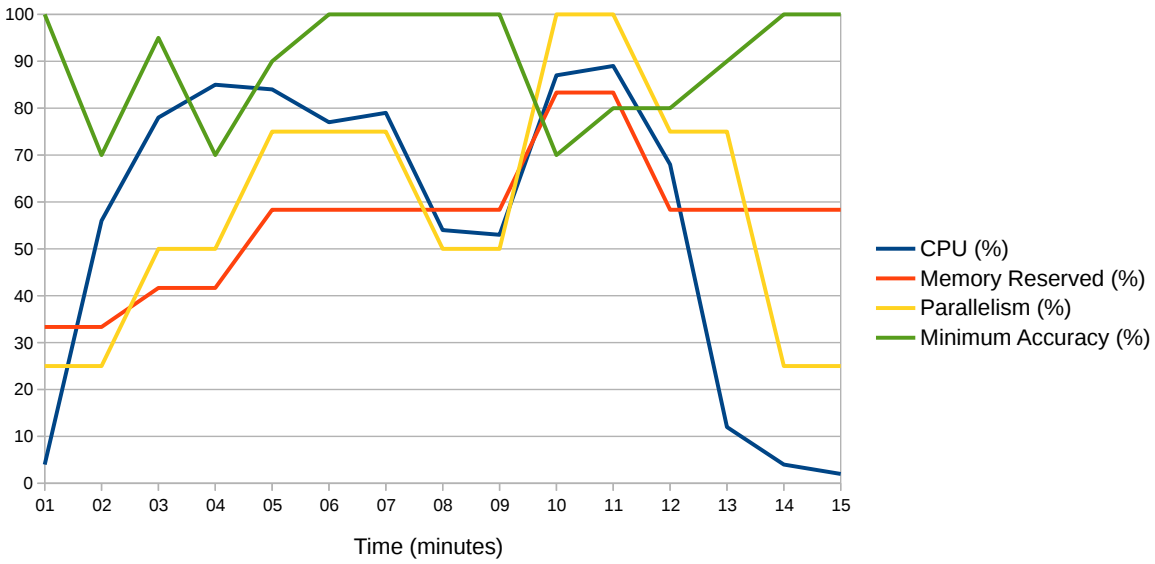

Figure 5.1: Taxi-Trip Benchmark (Local): Resources Variation

Table 5.2: Synthetic Benchmark: Cloud (16 vCpus/64GB)

Time (Minutes)	CPU (%)	Memory Reserved (GB)	Parallelism	Minimum Accuracy (%)
01	1	4	1	100
02	98	4	16	70
03	97	4	16	70
04	98	4	16	70
05	98	4	16	70
06	98	4	16	70
07	98	4	16	70
08	98	4	16	70
09	97	4	16	70
10	98	4	16	70
11	98	4	16	70
12	76	4	16	70
13	1	4	16	80
14	1	4	1	100
15	6	4	5	70
16	31	4	5	80
17	30	4	5	90
18	1	4	1	100
19	1	4	1	100
20	1	4	1	100

We can see in the beginning that the system was running with the defined minimum resources, and then when it started receiving the first events it scaled up, the parallelism went to the maximum level and the accuracy to the minimum. The system stayed overloaded (the requirements were not being met, so the accuracy never increased) until the received load was processed (minutes 12 to 13). After that in minute 13 we see that the accuracy increased, this was caused by the Metrics Reporter that verified that the application had enough resources to increase the accuracy based on the requirements and on the load of the CPU which was 76%.

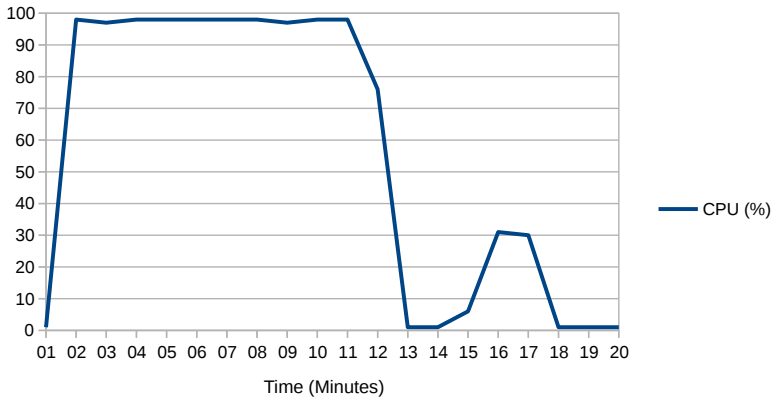

Figure 5.2: Synthetic Benchmark: CPU utilization

In the next minute (minute 14), we can see that the accuracy went to 100%. The Middleware made this decision when it analysed the metrics sent by the Metrics Reporter. Next, in minute 15, the system received a load of events. The amount of received events were more than what it could process with the parallelism set to 1. To the application achieve the requirements, the Middleware set the accuracy to 90% and increased the parallelism by 4 (from 1 to 5). However, during the same minute, the application restarted and the Metrics Reporter analysed the metrics and since the requirements were still not being met, it decreased the accuracy to the minimum (70%).

Then the system stabilized for two more minutes (this load had some idle after some events, so it was not continuous), and the Metrics Reporter gradually increased the accuracy. After that, when the application stopped receiving events the Middleware set the accuracy to the maximum and the resources to the defined minimum values. We can see that the memory was not changed during the entire execution time. This benchmark is heavily CPU-bounded, and 4GB was enough memory for this workload in this CPU. These results also showed how the system verifies by how much the resources are not being met and increases them according to that.

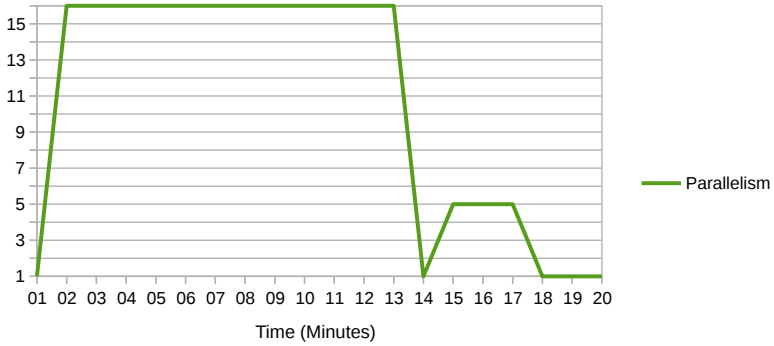

Figure 5.3: Synthetic Benchmark: Parallelism variation

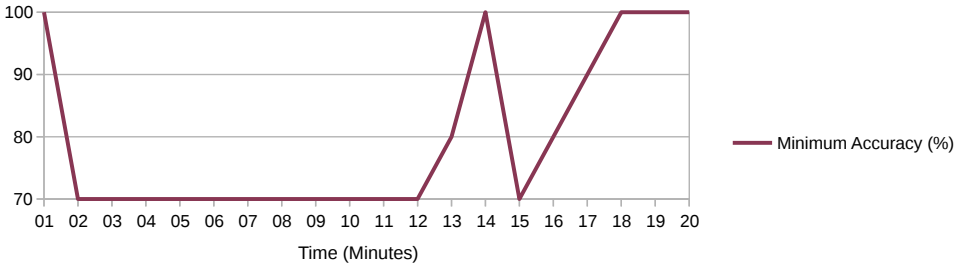

Figure 5.4: Synthetic Benchmark: Accuracy variation

5.4.2.2 Performance Comparison

The **Taxi-Trip Benchmark** was also used to compare the Approxate's performance improvements. This test calculates some averages related to the taxi trips (number of trips/passengers per weekday, number of trips per hour, money earned for each day of the month, money earned for each month). Those averages are calculated by dividing the values from the events for a fixed number of days/weekdays/months.

Table 5.3: Taxi-Trip (Local): General Results

Events (%)	Time (Minutes)	Precision (%)	Relative Time (%)
100	13:03	100	100
95	12:20	96	95
90	09:19	91	71
70	08:34	73	66
50	05:54	52	45

The results are in Table 5.3 and we can observe that the final precision (the precision is the ratio between the obtained values and the values when using 100% of the events) of the results is proportional to the amount of dropped events, however, the saved time is not. Dropping 10% of the events can save 29% of time. Due to the nature of the test and how the results are calculated, by using (5.1) is possible to calculate an approximation of the final result with a margin of error of 4%.

Table 5.4: Taxi-Trip (Cloud - Best Machine): General Results

Events (%)	Time (Minutes)	Precision (%)	Relative Time (%)
100	06:21	100	100
95	05:12	95	82
90	05:00	91	79
70	04:17	74	67
50	03:23	52	53

Table 5.5: Taxi-Trip (Cloud - Middle Machine): General Results

Events (%)	Time (Minutes)	Precision (%)	Relative Time (%)
100	12:40	100	100
95	11:13	95	89
90	10:55	91	86
70	08:52	74	70
50	05:47	52	46

$$Value = \frac{ObtainedValue * 100}{Accuracy} \tag{5.1}$$

The Taxi-Trip was also used to compare the performance in different machines with approximate results. In Table 5.4 are the results of the cloud machine with 16 vCpus and on Table 5.5 the ones of the 8 vCpus machine. We can see that the time saved when dropping 5% and 10% were better than the ones in the local machine, however, the rest of the results do not differ by much. These results show us that we can save close to 50% of the time and get results with around 96% of precision using (5.1).

We performed the **Linear Road Benchmark** application with data from four different highways for three hours of traffic. It was tested with 100, 70, and 50% of accuracy. In Table 5.6 we can see that dropping 30% of the events kept the system with a high precision level (88%) while saving 27% of the processing time. With 50% of dropping the time saved was 44%, however, the results lost 34% of precision.

Table 5.6: Linear Road (Local): 4 Highways - 3 Hours

Events Used (%)	Time (Minutes)	Precision (%)	Relative Time (%)
100	05:46	100	100
70	04:11	88	73
50	03:15	66	56

In Figure 5.5 there is a graph that demonstrates how lowering the precision can save time, the data is the average of all of the tested datasets in all stream processing benchmarks in all machines. It shows that the percentage of precision in the stream processing applications is always greater than the necessary time to process. At around 90% of results precision, we can save around 20% of the time.

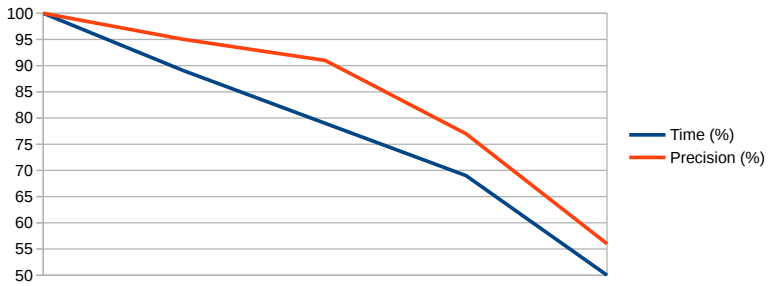

Figure 5.5: Stream Results: Precision and Time Relation

5.4.3 Graph Results

The graph results were only used to evaluate how the approximate results affect the precision and performance of graph processing applications.

The **Yahoo Groups Benchmark** was performed in different machines to test the impact of the performance and results' precision when using approximate results when different resources are available. It also used different datasets with different densities to see how they would affect the results.

In Table 5.7 we show the results of this benchmark performed with the dataset that contained 500000 groups (events) performed in the machine with 8 vCpus and 32GB of memory. With 5% of dropped events the performance only improved 1%, however with 10% of dropped events it increased 15%. For the rest of the results in the table the performance gains were similar to the percentage of dropped events, however, the precision of the results was much higher, even when dropping half of the events the results are 78% precise.

Table 5.7: Community Counting (Cloud - 8 vCpus / 32GB): Groups (500000 Groups)

Events Used (%)	Time (Minutes)	Communities	Precision (%)	Relative Time (%)
100	18:36	1497084	100	100
95	18:22	1475620	99	99
90	15:54	1447914	97	85
70	12:30	1325154	89	67
50	08:49	1173440	78	47

The same benchmark with the same dataset was also performed in the machine with 16 vCpus and 64GB of memory, the results are in Table 5.8. The performance increases were smaller than the previous ones, with the most difference where the dropped events were 10%, the performance only increased 3% respectively, instead of the 15% of the previous tests. The rest of the results were similar.

The difference of the test with 10% of dropped events can be explained due to the test requiring much memory: the machine with more memory got a more linear relationship between dropped events and performance increases, while the machine with less memory got a great improvement at 10% of dropped

Table 5.8: Community Counting (Cloud - 16 vCpus / 64GB): Groups (500000 Groups)

Events Used (%)	Time (Minutes)	Communities	Precision (%)	Relative Time (%)
100	16:30	1497084	100	100
95	16:10	1475575	99	98
90	16:01	1448409	97	97
70	11:32	1327682	89	70
50	08:14	1172374	78	50

Table 5.9: Community Counting (Local): Groups (250000 Groups)

Events Used (%)	Time (Minutes)	Communities	Precision (%)	Relative Time (%)
100	05:46	1165250	100	100
95	05:24	1147292	98	94
90	05:05	1123086	96	88
70	03:45	1015351	87	65
50	02:46	875315	75	48

events because with the previous tests with more processed events, the CPU was bottle-necked by the memory. When the dropped events increased the system had enough memory to process all events with the CPU being fully utilized, thus we got a linear scaling between dropped events and gained performance.

This test was also performed with a smaller and more dense dataset (250000 groups), and thus did not need as much memory. Because of that, the relation between dropped events and improved performance is almost linear. The results are in Table 5.9. The precision was also similar to the more dense graph dataset used in the previous tests.

The last test compares the processing times in two machines, one has double the resources of the other, but is processing 100% of the events, while the one with fewer resources is processing 75% of events. In Table 5.10 we can see that the machine with half of the resources took only 13% more time and still kept a precision of 91%. This is a relevant result, showing that when resources are scarce or machines are more constrained (e.g. edge devices, spot instances), the savings in resources do not necessarily entail significant loss in performance or accuracy.

The **Yahoo Messenger Benchmark** was also performed with different datasets. The graph with less density got good results as we can see in Table 5.11, where even with 50% of dropped events the precision was 83%, and it only took 32% of the time.

However, this test performed with a graph with more density got bad results, we can see in Table 5.12 that even dropping 5% of the events made the results lose 26% of precision. This is explained by the fact

Table 5.10: Community Counting (Cloud): Comparison between different machines

Machine	Events (%)	Time (Minutes)	Communities	Precision (%)	Time (%)
8 vCpus / 32GB	100	18:36	1497084	100	100
4 vCpus / 16GB	75	21:01	1359348	91	113

Table 5.11: Community Counting (Local): Messenger (560444 Users)

Events Used (%)	Time (Minutes)	Communities	Precision (%)	Relative Time (%)
100	05:48	38067	100	100
95	03:35	39101	97	62
90	03:25	40294	95	59
70	02:28	43925	87	43
50	01:50	46028	83	32

Table 5.12: Community Counting (Local): Messenger (1520005 Users)

Events Used (%)	Time (Minutes)	Communities	Precision (%)	Relative Time (%)
100	17:47	3316	100	100
95	16:16	4500	74	92
90	13:38	6351	52	77
70	07:51	15596	21	44
50	05:21	28222	12	30

that the graph dataset is very dense, and so, dropping just a few events has a major impact when doing community counting, since each event likely has many connections, by dropping it we are creating more communities than the real ones. This test was also performed in a graph with an intermediate density between the previous two, and the precision of the results is better than the higher-density graph and worse than the lower-density graph.

The **Triangle Counting Benchmark** was performed with different datasets. The results from the test with a higher-density graph are in Table 5.13, and they show that the precision of the results was almost the same until 30% of the events were dropped. Even with 30% of the events dropped the results still are 78% precise and more than half of the time was saved. In comparison with the results from the test with the lower-density graph, which are in Table 5.14, the precision is similar, but the time saved is worst until 50% of the events are dropped.

In Figure 5.6 we can see how dropping the results' precision can save time, the used data comes from averaging all of the graph benchmarks results. The graphic shows that the graph processing applications got a better relationship between the time that can be saved and the results' precision than the stream processing applications. On average, we can save 25% of time and still get 90% of precision, or even get 80% of precision in 60% of time.

Table 5.13: Triangle Counting (Local): High Density

Events Used (%)	Time (Minutes)	Triangles	Precision (%)	Relative Time (%)
100	02:27	160000	100	100
95	02:23	159731	100	98
90	02:19	157199	98	95
70	01:09	125220	78	47
50	01:00	73267	46	41

Table 5.14: Triangle Counting (Local): Low Density

Events Used (%)	Time (Minutes)	Triangles	Precision (%)	Relative Time (%)
100	24:41	7	100	100
95	20:17	7	100	82
90	17:32	7	100	71
70	16:06	6	86	65
50	12:33	3	43	51

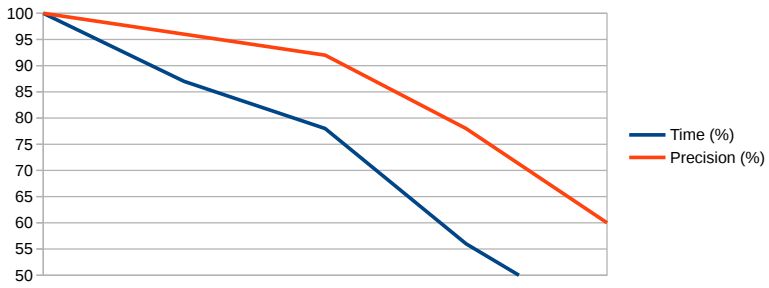

Figure 5.6: Graph Results: Precision and Time Relation

5.4.4 Results Analysis

The Micro-Benchmarks (Section 5.4.1) shows that Approxate's overhead has a small impact on the performance (only decreased 2%) when Approxate is not using approximate results, which is mostly negligible. When Approxate uses approximate results the impact is always positive. Even dropping 1% of events increased the performance.

Then we have the Stream Processing Benchmarks (Section 5.4.2). The first tests demonstrate how Approxate can identify when it is necessary to increase the applications' resources allocation and it can also identify when the resources are being wasted. This was tested multiple times and Approxate had a similar behavior where it adjusted the resources and accuracy as necessary.

One example can be observed in Figure 5.1 where the different resources and accuracy are being adjusted at different rates. We can observe that Approxate correctly identified the situations where the application could not meet the requirements with the allocated resources and, in order to respond to that, the system increased the resources and lowered the accuracy until the application's state got stable.

Another example is the Synthetic Benchmark, where we can see that the system adjusted the execution's resources accordingly to what was necessary; after the beginning of the test, where it received an intense load, it scaled up and decreased the accuracy. After it processed the events it scaled down, and then for a brief moment it scaled up a little and stayed stable for some time, then it scaled down again. We can also see that the application had enough memory and it stayed constant throughout the entire test.

Following that there are the performance tests, where the applications just had to process the datasets the fastest they could. This was done to observe how much time could be saved by using approximate results and how they would affect the final results' precision.

The benchmarks from the Taxi-Trip Benchmarks shows us that in types of tests where the results are an average of values obtained from the input, but the division is made with a fixed number (number of days/weeks/months of the year), the produced value for the used accuracy can be converted to an approximation of the result that has total accuracy by using Equation (5.1). By doing that we can save more than half of the necessary time than when not using approximate computation. This benchmark was also executed in cloud machines and the results show that by using approximate computing we can get approximate results in weaker machines (fewer resources) in around the same time as more powerful machines and keep an acceptable precision.

With Linear Road Benchmarks we can observe that Approxate increased the application's performance by 27% and the results still retained 88% of precision, which again shows that the performance can be greatly increased and the results still are acceptable, in scenarios where decision making needs to be fast and perfect accuracy is not a mandatory requirement.

Following the Stream Processing Benchmarks, we have the Graph Processing Benchmarks (Section 5.4.3). These tests were only used to observe how much performance can be gained by using approximate results in graph processing applications and how much precision is lost.

The first benchmark from the Graph Benchmarks is the Yahoo! Groups Benchmark and it calculates the number of communities based on the relationships between users and the groups that they belong to. We can notice that the precision was at least 75% even with using only half of the events. Using 70% of the events the application achieved at least 87% of precision with only 70% of the time. The results were similar across the different datasets.

The next benchmark is the Yahoo! Messenger Benchmark. The test that used the largest dataset was the one with worse results. Even using 95% of the events led to losing 16% of the precision. The other runs with a lower number of users got better results, the one with 1021120 users got 91% precision with only 70% of the time by dropping 5% of the events. The run with the dataset that contains the lowest number of users got a result 83% precise in 32% of the time. These results are explained because each user usually got a small number of friends, so by dropping even a low amount of events we cut connections inside a community which leads to a community appearing as multiple communities in the results.

The increase in precision in the datasets with fewer users is explained by the fact that with fewer users there are no chains of users that can be formed as large as the ones with more users. This is because the users are less connected between them, so there are more communities. By dropping some events more communities are split into multiple, but since there were already a large number of communities the end result is less affected. The large data set had only 3316 communities in total while

the smallest data set had 38067.

The last benchmarks are the Triangle Counting Benchmarks which count the number of triangles in a graph. We can see that the results from the graph with a high density were very accurate while dropping 95% and 90% of the events, but the time saved was small, 5% at most. However the run with 70% of the events processed got 78% of precision but only used 47% of the time, so there is a point in terms of the percentage of events being dropped where we can have a large amount of time saved with a small loss of precision.

The second benchmark from Triangle Counting was using a graph with low density. By dropping 10% of the events we got 100% of precision and 29% of time saved. Using 70% of the events we got 86% of precision in 62% of the time, so we can save a large amount of time without losing much precision with low-density graphs in this use case scenario.

Both the stream and graph benchmarks were executed in cloud machines, and so we can compare results across machines with different resources. One of the comparisons is shown in Table 5.10 where the machine with fewer resources could process the data in not much more time than a machine with double the resources by using Approxate. The other benchmarks also show similar results where the lower-end machines can sometimes get close or match the performance of the higher-end machines without losing too much precision.

All of the results show that the number of events that can be dropped while keeping the results meaningful and the time saved by doing that is different for each application, but in a general way, we can save 5% to 50% of the time while keeping levels of precision close to 80%.

By analysing the results we can conclude that the system allows variable accuracy in order to improve the performance or to meet one or multiple execution requirements (lag, latency, or throughput). The system also improved the performance, i.e. it can process the same datasets quicker than vanilla Stateful Functions, which leads to the system needing fewer resources to process the same amount of data. The system also improved the scalability since it allows Stateful Functions to change the parallelism and the amount of allocated resources as it needs. The results also indicate that the eventual balance of the dropped events between the different data sources is enough to represent all of them in a meaningful way. The resources overhead are also minimal and they do not have a significant impact on the amount of resources used. The cloud tests also demonstrate that it is possible to process the same amount of data with less powerful machines using approximate computation.

5.5 Summary

This chapter explains how we carried out the evaluation, it contains the necessary information about the used setups, the evaluated metrics, and the used workloads and benchmarks. The analyses of the benchmarks' results allow us to make some conclusions. They show us 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 up to around 50%. All of the tested cases got acceptable results, except for one which has a graph with a high density of nodes. However, other tests which also contained high-density graphs still got good results.

The results also show that Approxate can scale up and scale down the applications based on the defined requirements about lag, latency, throughput, and the current load. By comparing the cloud tests with each other and with the local tests we see that Approxate with approximate computation can reduce the time that is needed to process the data, it can also process the same data in roughly the same time with less resources, and it can also process more data in the same amount of time than vanilla Stateful Functions would need.

The micro-benchmarks also show us that Approxate does not have a negative impact on the resources utilization, and since it reduced the time needed to process the data in all tests, we can conclude that the performance gains are greater than the overhead.

All of this confirms that all of the metrics used to evaluate the solution were achieved in almost all of the tested cases.

Chapter 6

Conclusion

In this work, we developed an extension to be used with Stateful Functions to add dynamic resource allocation, to improve the scalability, the performance, and to allow the user to chose trade-offs between performance and results accuracy.

6.1 Summary

This work contains an overview of cloud computing and its shortcomings usually, it is used in stream and graph processing platforms. Its shortcomings also affect the platforms. This work also addresses the state of the art of serverless computing, a cloud computing model that stream and graph processing platforms use. This work presents and explains the key features those platforms should have. It also presents some relevant platforms.

There is a comparison between the relevant stream and graph processing platforms, and also the features of each one. That brings us to the motivation for this work, it is noticeable that generally, they lack support for dynamic resource allocation, auto-scaling (elastic scaling), and to use approximate computation that would allow them to trade some accuracy to improve performance based on preferential user-defined requirements/metrics.

There are some difficulties associated with bringing those features, like controlling the quantity of resources that are allocated depending on the load and on the requirements that must be met (i.e. if the load spikes up it does not make sense to just allocate the maximum available resources, it is necessary to know what resources should be allocated and in what quantity). The use of approximate results also brings difficulties, because the accuracy can not be reduced to the point where the results are not acceptable. Also, when using load shedding to decrease the accuracy, all data sources must be equitably represented in the results.

This work contains a proposal and implementation of an extension to be used with Stateful Functions

for stream processing, a platform that allows to easily build applications, fast state sharing between functions, it is scalable, distributed, and contains fault-tolerance.

The proposed and implemented solution is named Approxate and is capable of improving the resource allocation, scalability, and performance. It also adds an intelligent and variable resource management that will vary the allocation of the resources, and the level of parallelism, based on the state of the execution and the desired user requirements (latency, lag, throughout). It can also use approximate computation to vary the level of accuracy whenever it is necessary to meet the requirements while keeping the results meaningful. The extension can also be used in graph processing since those operations are supported by Gelly in Flink and so they are supported in Stateful Functions.

Approxate allows the use of lower-end machines without a major degradation of performance and results' precision. This can be used in serverless computing models to increase the resources' utilization, machines that have less available resources can still be used to process loads that otherwise they couldn't (i.e. the processing may have requirements that would not be met without approximate computing), instead of not being used.

Approxate also improves the resources' management since they are adapted to the processing requirements and the load at a given time. Since the management is improved, the amount of wasted resources is also improved. When the resources are not necessary they are not allocated, so they are not being wasted. The scalability is also improved because the systems can better adapt to variations in the input load with an elastic scaling.

Chapter 3 describes the design of the proposed solution and its components (Approximate Library, Metrics Reporter, and Middleware). The Approximate Library is the component that has the responsibility to decrease the results' accuracy through load shedding in the events that are arriving on the applications. The Metrics Reporter takes advantage of Flink's metric system to collect the relevant metrics periodically and can also adjust the accuracy level that the Approximate Library is targeting, it also sends the metrics to the Middleware. The Middleware is responsible for analysing the metrics, and make sure that the system is meeting the requirements about latency, throughput, and lag that the user can define. If necessary, the Middleware changes the resources' allocation, the parallelism level, and the accuracy of the results to achieve the requirements.

In summary, it is described how Approxate gets the execution metrics, analyses them taking into account the user-defined requirements, and then, based on those results, it can decide to adjust the accuracy level of the results and/or modify the resource allocation. Although Flink supports multiple applications in the same cluster, Approxate was not developed to work with multiple applications at the same time.

Chapter 4 explains how Approxate was implemented, there is an overview of the code structure, and some examples. Some of the technical aspects are also explained.

This work also contains the methods used for evaluating the solution. It presents the setups, the met-

rics, the workloads, and benchmarks used to perform the evaluation. It also shows the effects of using approximate results in some typical stream and graph processing use scenarios, wherein the majority of cases the performance could increase in amounts up to 50% and the results still are acceptable. Those results also show how Approxate improved the scalability and resource allocation. The evaluation also shows the overhead of Approxate, which can be considered negligible.

6.2 Future Work

To improve stream processing with Stateful Functions, the Middleware (which already can modify the state of Docker containers) could be extended to analyse the metrics of the containers of multiple applications, possibly the containers where Kafka is running, to manage multiple applications and even the Kafka resources. This way it could take resources from any container and give them to another one depending on the situation (e.g. if the Middleware knows that one application that is using one Kafka container will not be able to process the amount of events that already arrived in a short period of time, the Middleware could take resources from the Kafka container, which would delay a bit the receiving of new events, and give those resources to a container where they can be better used).

Approxate can also be adapted to manage multiple applications that are running in the same cluster, it can be adapted to work with the other Flink cluster types (i.e. job and session clusters).

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. This also involves adapting the Middleware to communicate with Kafka.

Another way that stream processing could be improved is to develop extensions for the Approximate Library adapted for specific types of events, instead of the one that is used for all types. 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. If a system is processing two types of information that are contained in one event and the processing of one is preventing the application from achieving the requirements, we could use approximate computing to process that information and increase the performance of the application.

Bibliography

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[4] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[5] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB Journal*, vol. 29, no. 2, pp. 595–618, 2020.

[6] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz *et al.*, "The future is big graphs: a community view on graph processing systems," *Communications of the ACM*, vol. 64, no. 9, pp. 62–71, 2021.

[7] M. E. Coimbra, A. P. Francisco, and L. Veiga, "An analysis of the graph processing landscape," *J. Big Data*, vol. 8, no. 1, p. 55, 2021. [Online]. Available: https://doi.org/10.1186/s40537-021-00443-9

[8] G. Rodrigues, F. Lima Kastensmidt, and A. Bosio, "Survey on approximate computing and its intrinsic fault tolerance," *Electronics*, vol. 9, no. 4, p. 557, 2020.

[9] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[10] H. B. Barua and K. C. Mondal, “Approximate computing: A survey of recent trends—bringing greenness to computing and communication,” *Journal of The Institution of Engineers (India): Series B*, vol. 100, no. 6, pp. 619–626, 2019.

[11] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, “Load shedding on data streams,” 01 2003.

[12] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, “Load shedding in a data stream manager,” in *Proceedings 2003 vldb conference*. Elsevier, 2003, pp. 309–320.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[14] A. Akhter, M. Fragkoulis, and A. Katsifodimos, “Stateful functions as a service in action,” *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1890–1893, 2019.

[15] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe, “Streamapprox: Approximate computing for stream analytics,” in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 185–197.

[16] S. Esteves, N. Janssens, B. Theeten, and L. Veiga, “Empowering stream processing through edge clouds,” *ACM SIGMOD Record*, vol. 46, no. 3, pp. 23–28, 2017.

[17] A. Rosà, L. Y. Chen, and W. Binder, “Profiling actor utilization and communication in akka,” in *Proceedings of the 15th International Workshop on Erlang*, 2016, pp. 24–32.

[18] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, “Big data analytics on apache spark,” *International Journal of Data Science and Analytics*, vol. 1, no. 3-4, pp. 145–164, 2016.

[19] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, “Storm@ twitter,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[20] D. Organization, “Docker,” https://www.docker.com/.

[21] K. M. M. Thein, “Apache kafka: Next generation distributed messaging system,” *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[22] G. McGrath and P. R. Brenner, “Serverless computing: Design, implementation, and performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[23] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, “Elastic scaling for data stream processing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2013.

[24] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[25] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, “Parallel stream processing against workload skewness and variance,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 15–26.

[26] D. Sun, G. Zhang, C. Wu, K. Li, and W. Zheng, “Building a fault tolerant framework with deadline guarantee in big data stream computing environments,” *Journal of Computer and System Sciences*, vol. 89, pp. 4–23, 2017.

[27] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, “Cloudburst,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, p. 2438–2452, Aug 2020. [Online]. Available: http://dx.doi.org/10.14778/3407790.3407836

[28] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, “Orleans: Distributed virtual actors for programmability and scalability,” *MSR-TR-2014–41*, 2014.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[30] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[31] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, “A catalog of stream processing optimizations,” *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–34, 2014.

[32] D. Sun, S. Gao, X. Liu, X. You, and R. Buyya, “Dynamic redirection of real-time data streams for elastic stream computing,” *Future Generation Computer Systems*, 2020.

[33] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, “Anna: A kvs for any scale,” *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[34] S. Jayasekara, A. Harwood, and S. Karunasekera, “A utilization model for optimization of checkpoint intervals in distributed stream processing systems,” *Future Generation Computer Systems*, 2020.

[35] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, “Cusp: A customizable streaming edge partitioner for distributed graph analytics,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 439–450.

[36] N. Jafari, O. Selvitopi, and C. Aykanat, “Fast shared-memory streaming multilevel graph partitioning,” *Journal of Parallel and Distributed Computing*, 2020.

[37] B. Steer, F. Cuadrado, and R. Clegg, “Raphtory: Streaming analysis of distributed temporal graphs,” *Future Generation Computer Systems*, vol. 102, pp. 453–464, 2020.

[38] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, “Time-evolving graph processing at scale,” in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.

[39] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[40] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 2013, pp. 1–6.

[41] S. Mittal, “A survey of techniques for approximate computing,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[42] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura, “Approximate computing: Challenges and opportunities,” in *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 2016, pp. 1–8.

[43] S. Esteves and L. Veiga, “Waas: Workflow-as-a-service for the cloud with scheduling of continuous and data-intensive workflows,” *Comput. J.*, vol. 59, no. 3, pp. 371–383, 2016. [Online]. Available: https://doi.org/10.1093/comjnl/bxu158

[44] S. Esteves, H. Galhardas, and L. Veiga, “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*, P. Ferreira and L. Shrira, Eds. ACM, 2018, pp. 239–252. [Online]. Available: https://doi.org/10.1145/3274808.3274827

[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “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)*, 2012, pp. 15–28.

[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework,” in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[47] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 239–250.

[48] M. Olson, "Hadoop: Scalable, flexible data storage and analysis," *IQT Quart*, vol. 1, no. 3, pp. 14–18, 2010.

[49] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An experimental survey on big data frameworks," *Future Generation Computer Systems*, vol. 86, pp. 546–564, 2018.

[50] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1507–1518.

[51] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[52] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

Appendix A

Sample Code of the Project

Listing A.1: Combine Results Method

```
1  private List<Result> getCombinedResults(Result cpuRes, Result throughputRes,
       Result lagRes, Result memRes, Result latencyRes) throws
       ConfigLoadException {
2      ArrayList<Result> results = new ArrayList<>();
3      if (throughputRes == RESOURCES_UP_MAX || lagRes == RESOURCES_UP_MAX ||
           latencyRes == RESOURCES_UP_MAX) {
4          results.add(ACCURACY_DOWN_MAX);
5          results.add(PARALLEL_UP_MAX);
6          if (cpuRes == CPU_UP || cpuRes == CPU_UP_MAX) {
7              results.add(CPU_UP_MAX);
8          } else if (cpuRes == CPU_DOWN_MAX) {
9              results.add(CPU_DOWN);
10         }
11         if (memRes == MEMORY_UP || memRes == MEMORY_UP_MAX) {
12             results.add(MEMORY_UP_MAX);
13         } else if (memRes == MEMORY_DOWN_MAX) {
14             results.add(MEMORY_DOWN);
15         }
16         return results;
17     }
18     if (throughputRes == RESOURCES_UP || lagRes == RESOURCES_UP || latencyRes
           == RESOURCES_UP) {
19         results.add(ACCURACY_DOWN);
```

```
20          if (cpuRes == CPU_DOWN_MAX) {
21              results.add(CPU_DOWN);
22          } else {
23              results.add(cpuRes);
24          }
25          if (memRes == MEMORY_DOWN_MAX) {
26              results.add(MEMORY_DOWN);
27          } else {
28              results.add(memRes);
29          }
30          results.add(PARALLEL_UP);
31          return results;
32      }
33      if (cpuRes == CPU_UP_MAX) {
34          results.add(cpuRes);
35          results.add(PARALLEL_UP_MAX);
36          results.add(ACCURACY_DOWN_MAX);
37          if (memRes == MEMORY_DOWN_MAX) {
38              results.add(MEMORY_DOWN);
39          }
40          else if (memRes != MEMORY_DOWN) {
41              results.add(memRes);
42          }
43          return results;
44      }
45      results.add(cpuRes);
46      results.add(memRes);
47      if (cpuRes == CPU_DOWN_MAX) {
48          results.add(PARALLEL_DOWN_MAX);
49          results.add(ACCURACY_UP_MAX);
50          return results;
51      }
52      if (cpuRes == CPU_DOWN) {
53          results.add(PARALLEL_DOWN);
54          results.add(ACCURACY_UP);
55          return results;
56      }
57      if (cpuRes == CPU_UP) {
```

```
58            results.add(PARALLEL_UP);
59            return results;
60        }
61        return results;
62    }
```