TÉCNICO LISBOA

Stream Economics

Resource Efficiency in Flink Stream Processing with Accuracy Awareness and Load-Shedding

Luís Manuel Tavares Jordão Alves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Pável Pereira Calado

May 2018

Acknowledgments

I would like to thank both my parents and my sister for their support and encouragement over all these years and throughout the development of this work. Without them this thesis would not be possible.

I would also like to acknowledge my dissertation supervisor Prof. Luís Manuel Antunes Veiga for his support, feedback and for the time he invested on making this dissertation possible, as well as for the opportunity to let me explore a subject of my interest during this thesis.

Finally, I would like to thank all my friends, colleagues and co-workers for helping me grow as a person and as a professional.

To each and every one of you – Thank you.

Abstract

Distributed stream processing systems appear as a solution for processing huge volumes of data in real-time. One of the key problems that these systems need to deal with is associated with managing all the computational resources that are required to execute their applications, so that no resources are wasted and unnecessary costs are reduced. This, while at the same time, guaranteeing that all tasks are well provisioned with the resources that they need to keep up with their incoming workload.

This work explores precisely this problem by proposing a novel resource management model, that targets resource efficiency in a stream processing environment where several workload dynamic and resource heterogeneous applications execute in parallel. A resource aware task scheduler is proposed, that follows what we call an optimistic approach, as it doesn't limit the amount of tasks that can be executed by a single machine, instead, it adapts tasks' allocation / placement based on their runtime metrics. At the same time, load shedding in stream processing applications is explored as a solution to overcome situations of tasks' overallocation to resources that may appear due to previous wrong decisions performed by the task scheduler, due to its optimistic approach.

Additionally, a proof-of-concept of such system is implemented, targeting the widely used Apache Flink system. This proof-of-concept is evaluated against different workloads, to show how the different aspects of the algorithm and its mechanisms operate and behave.

Keywords

Stream Processing; Task Scheduling; Resource Efficiency; Load Shedding; Apache Flink.

Resumo

Os sistemas de processamento distribuído de *streams* aparecem como uma solução para o processamento de grandes volumes de dados em tempo real e com baixas latências. Um dos problemas chave destes sistemas prende-se com a gestão dos recursos computacionais disponíveis, de modo a reduzir custos e a evitar desperdícios de recursos. Sendo também necessário garantir que cada tarefa tem os recursos que precisa para poder acompanhar o ritmo a que recebe novos eventos para processar.

Este trabalho explora precisamente este problema ao apresentar um novo modelo de gestão de recursos computacionais, com o objectivo de optimizar o uso destes recursos em ambientes de processamento em *stream*, onde são executadas aplicações que processam *workloads* dinâmicos e que têm diferentes requisitos em termos de recursos computacionais. Um novo escalonador de tarefas é proposto, que toma decisões optimistas no sentido em que deixa de limitar o número de tarefas que podem ser executadas em cada máquina. Em vez disso, a alocação de tarefas aos recursos disponíveis é adaptada em tempo real, com base nas métricas de execução das mesmas. A utilização de *Load Shedding* em processamento de *streams* é também explorada como uma ferramenta para resolver problemas de sobre-alocação de tarefas, que pode ocorrer devido às decisões optimistas tomadas pelo escalonador.

Ao longo deste trabalho, foi implementada uma prova de conceito deste sistema no Apache Flink, um sistema de processamento de *streams* moderno. O sistema proposto é avaliado utilizando diversos *workloads*, de modo a mostrar os diferentes componentes e algoritmos usados em acção.

Palavras Chave

Processamento em Stream; Escalonamento de Tarefas; Optimização de Recursos; Load Shedding; Apache Flink.

Contents

List of Figures

List of Algorithms

Listings

Acronyms

QoD	Quality-of-Data
QoS	Quality-of-Service
DSMS	Data Stream Management System
SLA	Service Level Agreement
DAG	Direct Acyclic Graph
DSL	Domain-Specific Language
TM	Task Manager
DSPS	Distributed Stream Processing System
JM	Job Manager
FIFO	First-In First-Out
TD	Taxi Drives
KNN	K-Nearest Neighbors
JMX	Java Management Extensions

1

Introduction

Contents

Every day large amounts of data are produced from diverse data sources. This data contains insightful and valuable information, and as such there is a need not only to process it, but also to do it in a near real time manner. Thus, allowing users to get results as fresh as possible. Distributed stream processing systems appear as a tool that allow to easily build large scale real time applications that can process these huge amounts of data and keep up with the incoming rate at which the data is generated. Excused to say that large computational clusters are required to run these large scale applications.

1.1 Motivation

Efficiently managing these large amounts of resources is a challenging problem, particularly in environments where several heterogeneous applications can be executing in parallel [1–3]. Usually an allocation based approach is followed, where the application owners can reserve specific amounts of resources for the execution of the applications' tasks. In such environments, it is common to face *resource underutilization* situations. This not only yields a pain for the application developers, since they pay for more resources than the ones they need at a specific instant in time, but it also shows an opportunity for the resource owners to increase their profit. In fact, this is also an observed issue in large scale datacenters, where reports can be found claiming a resource usage below 60% [4] of the full capacity.

One of the main reasons for this to happen is because resources tend to be allocated to the applications' peak workload, while on average workload most of these resources will not be used [5]. Note that it is usual for stream processing applications to have very unstable and dynamic workloads, where most of the time the workload is fairly low but suddenly some unpredictable bursts occur. This makes this overallocation understandable, since applications still need to respect their Service Level Agreements (SLAs) when facing these worst case scenarios. Additionally, estimating the resource needs for specific applications is not a trivial task, and it's easy for clients to simply overestimate the amount of required resources. As such, although these resources are available, no application will be able to use them, since they are already allocated for the execution of other applications.

Apache Flink[1], is a modern and state of the art stream processing system. Nonetheless, it suffers from the same problems when it comes to resource management. In fact, Flink's resource management model is even more rudimentary, since the resources used for each task execution are not specified by the application owner. Instead, each machine provides a specific amount of slots that can be used for tasks' execution. Machines' resources are equally distributed over these slots. Needless to say that this amount of tasks is defined by the Flink cluster administrator, thus it has to assume that all application tasks have the same resource requirements. As such, if tasks are expected to consume

[1] "Apache Flink", https://flink.apache.org, (June 22, 2018)

few resources, then machines should be configured to provide high amounts of fine granular slots, avoiding underutilization situations. On the other hand, if tasks are expected to be resource heavy, then machines should provide coarse grained slots, avoiding resource starvation situations. Clearly this rather inflexible one-size-fits-all approach doesn't work in scenarios where the applications' tasks are highly heterogeneous, as it is more and more expectable in multi-tenant scenarios / deployments.

The opposite problem of resource underutilization is usually named as *task overallocation* [6], which consists in having too many tasks allocated to a specific machine. It leads to poor applications' performance and eventually disrupting the applications' SLAs, where applications fail to cope with their incoming workload because no computational resources are available for them to use. This is clearly an undesirable situation in such real time critical systems.

1.1.1 Shortcomings of Current Solutions

As said, current stream processing systems solutions follow an allocation based resource management approach [7]. As such, they can't handle resource underutilization situations, since they still require clients to have a fine control on the resources that are allocated to their applications. Other stream processing solutions such as Apache Storm[2], follow a more naive approach, distributing applications' tasks over the available resources in a round robin way. While this approach reduces the probability of resource underutilization, it increases the probability of resource starvation situations.

Overallocation of tasks to machines is a commonly used solution to avoid resource underutilization [8], being commonly used in datacenter environments running several virtual machines. Nonetheless if the applications' workload increases, their SLAs may become compromised. This leads to the need of constantly monitor the applications' runtime in order to react to possible bottleneck situations and guarantee that their SLAs keeps being respected; as well as the necessity to have some compensation mechanism to perform eventual corrections in the system. A common compensation mechanism consists in auto-scaling the application. This may not be ideal, first because it may not react fast enough for the applications to keep coping with their workload. Second, because it implies the system having to perform runtime modifications to the applications' topology (i.e. increasing / decreasing the parallelism), without the user being in control of them.

Additionally, Flink provides an almost similar method to overallocation, named *slot sharing* [9], that allows to use a single slot to execute multiple tasks of the same application. Nonetheless, it is still up for the user to decide whether to use it or not.

In resource management systems such as Apache Mesos[3] similar solutions to overallocation can be found. For instance, *Oversubscription* [10] is used in Mesos to take advantage of temporarily unused

[2]"Apache Storm", http://storm.apache.org, (June 22, 2018)
[3]"Apache Mesos", http://mesos.apache.org, (June 22, 2018)

resources to run best-effort, lower priority tasks. This way, these tasks can be assigned to revocable resources, meaning they can be taken away once they are needed by other higher priority task, in order to meet his target Quality-of-Service (QoS). This is done either by killing or throttling the lower priority tasks. Nonetheless, such approach doesn't take advantage of the domain knowledge on stream processing systems, where the applications' accuracy can be used as a trade-off for the freshness of their results.

Additionally, resource usage estimation mechanisms for the applications' tasks have also been proposed [5], allowing to correct the resource over-estimation performed by the user. This estimation is then taken into account by the task scheduler. Unfortunately, these solutions usually require some history from the applications' execution in order to perform good estimations.

1.2 Proposed Solution

This work proposes a novel approach for resource management in stream processing systems that targets resource efficiency. It does so, by following a different mechanism from current resource reservation based solutions. In fact, the proposed solution completely drops the concept of resource allocation to applications' tasks. The rationale behind this solution consists in using a task scheduling policy that given a new task, schedules it to the machine with more available resources, in terms of CPU usage. Therefore, avoiding situations of resource underutilization, due to having unused but reserved resources that can't be used by other applications.

By following this policy, it is clear that it can lead to task overallocation situations, since it may allocate multiple resource heavy applications to the same machine, and because the applications' workload and resource requirements can dynamically change over time. For that reason, two mechanisms are proposed, in order to guarantee that applications keep coping with their incoming workload. The first, consists in dynamically re-scheduling tasks to other machines. While the second consists in using load shedding as a way to reduce the workload of specific applications, allowing applications to keep producing fresh results, with the trade-off of reducing the applications' accuracy.

The proposed solution is responsible for deciding when each of these mechanisms should be used. This is performed based on runtime metrics and on SLA requirements provided by the users, namely, the priority of the applications' queries and their acceptable accuracy threshold.

1.3 Goals and Contributions

The main goal of this work consists in exploring the applicability of *Load Shedding* solutions to the problem of resource management in stream processing systems. The proposed solution accomplishes

this goal by leveraging load shedding as a compensation mechanism to the task overallocation problem. Thus, allowing applications to keep up with their incoming workload and improving resource efficiency without compromising their SLAs, while reducing costs both for the applications' and cluster's owners.

Several key contributions from this work can be outlined. The first consists in a model (and underlying algorithms) that allows to govern all the required decisions, providing useful and meaningful semantics for the end line users, as well as a simple and understandable interface for them to specify their applications' requirements. The second contribution consists in a more sophisticated load shedding mechanism for stream processing applications, that takes into consideration accuracy requirements from the different queries within the applications. A detailed description is provided, regarding how such a mechanism can be implemented in Apache Flink; as well as how the solution can take into consideration bottlenecks in source tasks of the applications. A resource usage aware task scheduler is also developed, that performs scheduling decisions based on runtime metrics of the tasks' execution.

As will be seen, these mechanisms are detailed in such a way that the different components end up by being decoupled. Allowing to easily replace the semantics followed by each of them, without compromising the objective of the overall solution. This is also considered to be a relevant contribution, as it not only enables customization, but also allows future work to be performed on top of the proposed framework.

1.4 Document Roadmap

The remaining content of this document is organized as follows. Chapter 2 presents other work that is considered to be relevant and in line with the established goals and context of this work. Three dimensions are explored in this chapter: *Stream processing*, *Resource Management* and *Approximate Computing*. Besides exploring state of the art academic and commercial solutions, this chapter also provides a good background knowledge for the reader to better understand the rest of the work. Chapter 3 focuses on describing the architecture and on detailing the proposed model, as well as the algorithms that will govern the decisions taken by each component of the solution. Follows Chapter 4, that presents a more implementation wise oriented description of the proposed solution in Apache Flink, as well as some setbacks discovered during this phase of the work. Chapter 5 covers the evaluation of the developed solution, providing a description of performed benchmarks and the obtained results, together with their analysis and discussion. This document finishes with Section 6, that summaries the main points of this work and discusses possible future work.

2

Related Work

Contents

2.1 Overview

This work covers three main fields of study: *Stream Processing Systems*, since the solution is specifically design to take advantage of these type of systems; *Resource Management*, as it aims at improving resource efficiency, by handling underutilization and resource starvation situations; and *Approximate Computing*, since it leverages *Load Shedding* as a mechanism that will be used to guarantee that applications will keep up with their incoming workload. As such, this chapter focus on providing a historical and state of the art background regarding these three dimensions of this work. Sections 2.2, 2.3 and 2.4 cover the above mentioned topics, in the respective order. The chapter finishes with a section dedicated to Apache Flink, that contains relevant and more detailed information, that will establish some concepts that will be used throughout the rest of the document.

2.2 Stream Processing

Historically, data stream processing started being explored within the Databases research field. At that time, stream processing systems where called Data Stream Management Systems (DSMSs). DSMSs allowed users to submit queries over data streams, that are continuously executed until the user removes them. A good example of such systems is the Aurora Project [11]. These systems were end-to-end solutions, that provided a storage and computation unified solution. They often did not scale; they were often SQL oriented, which makes them less expressive, though some allowed to specify queries in a graphical way; and they only supported structured data. Eventually these systems evolved to DSMSs such as Aurora* [12], Borealis [13] and STREAM [14], that target distributed, high available and scalable stream processing [15].

Recently, with the ”Big Data explosion” and the increasing need to process huge amounts of data in real time, a new class of stream processing systems emerged. These systems are called Distributed Stream Processing Systems (DSPSs), and examples of them are the Apache Flink, Apache Storm and Apache Spark Streaming[1]. A key difference between DSPSs and DSMSs, is that they only provide the processing layer. More, they are built to scale to hundreds of nodes. This section focuses on distributed stream processing and on current state of the art solutions. Regarding the terminology that will be used, it follows the one defined by Apache Flink, as it's the DSPS that is targeted by this work.

2.2.1 Distributed Stream Processing

Intuitively, stream processing consists in processing *data streams*. These *data streams* correspond to continuous, ordered and possibly infinite flows of events. In order to process data streams, current

[1] ”Apache Spark”, http://spark.apache.org, (June 22, 2018)

solutions follow one of the following paradigms: *Micro Batching* or *Tuple-at-a-Time* processing.

Micro Batching Incoming events are constantly grouped into small batches that are then processed in a batch processing fashion. It has the advantage that it allows to leverage already existing batch processing tools so that they can support stream processing, therefore most of the problems would be already solved and it is easier for companies to shift from batch to stream processing. But, it has downsides. One is the fact that it adds additional delay due to its batch nature, the other is the difficulty to determine the optimal batch sizes [16].

Tuple-at-a-Time Consists in processing events as they arrive to the system. Though this approach cannot reuse already existing batch processing systems, it does have the advantage that it achieves the lowest latency. Note that this does not mean that stream processing tools that follow this paradigm cannot support batch processing, since batch processing is just a special case of stream processing with bounded data streams.

2.2.2 Stream Processing Applications and Concepts

A stream application is usually represented as a Direct Acyclic Graph (DAG), sometimes also denominated as a *dataflow*. In this DAG, the nodes represent operations on the events, and the edges represent streams that connect the output of one operation to the input of the other, as presented in Figure 2.1.

Given a certain node (e.g. blue nodes in the figure), the set of nodes that precede him in the DAG are named its *upstream* (e.g. yellow nodes in the figure), and the set of nodes that succeed him are named its *downstream* (e.g. green nodes in the figure). Nodes without any *upstream* are named *sources*, while nodes without *downstream* are named *sinks* or *queries*. Source nodes are responsible to inject the events from an external data stream into the application in order to be processed.

The figure also shows that there are two types of DAGs, the logical and the physical one. The logical DAG corresponds to the one that is described by the user, whereas the physical DAG corresponds to a compiled and optimized description of the application, representing how the application will actually be executed and deployed to the machines.

Operators Are abstract computations that consume events from a stream, process them and output zero or more events to its output streams. Operators can be implemented by the user, but usually most DSMS provide some standard operators.

Operators may be stateless, i.e. they do not maintain state across events, or stateful, i.e. they do keep state. In order to deal with failures, operators allow to periodically take snapshots of their state.

These snapshots are then stored in a reliable distributed data store.

In the physical DAG, the operators are presented by *subtasks*, or operator's *instances*. Each operator may be mapped to several subtasks that execute the same logic in parallel.

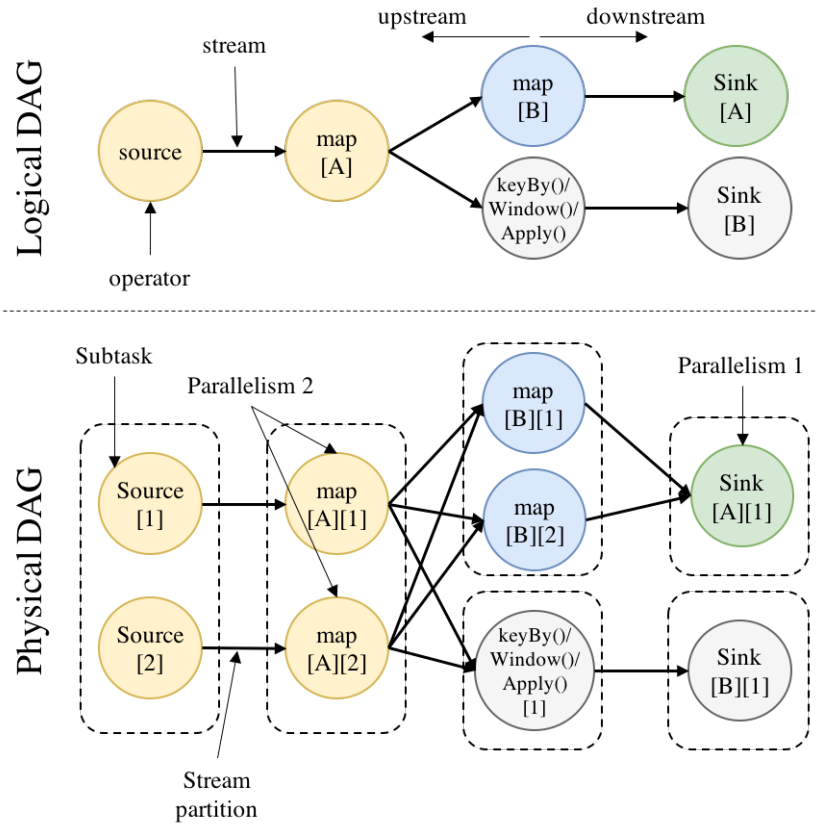

Figure 2.1: Example of a stream application's logical and physical DAGs.

Streams Are abstract channels that allow operators to exchange events. They provide First-In First-Out (FIFO) semantics and allow nodes to communicate asynchronously.

Since sending events between nodes has costs, instead of sending individual events, operators produce buffers of events sent over the streams. Defining the size of the buffer or periodicity at which it must be flushed, has impact on the overall latency and throughput of the system. If the buffer size is higher, the throughput is maximized but the latency increases, since events must hold before being sent to the downstream nodes. If the buffer size is low the latency decreases as well as the throughput. This problem has been explored in [17].

Events in the streams may or may not be persisted. Persisting events leads to faster fail recoveries, but a penalty is paid on latency, introduced by persisting messages before making them available to the downstream. On the other hand, not persisting the events, leads to having to re-compute some of the already processed events [18]. Recently, in [19] an hybrid approach has been proposed, where events

become immediately available for consuming, while at the same time are asynchronously persisted (removing this latency out of the critical path). When a failure occurs, events may or not need to be re-computed.

In the physical DAG streams are divided into *partitions*, each connecting two subtasks. The amount of partitions depends on: the source data streams, on the operators' parallelism and on how they are connected. Operators in a stream may be connected in an all-to-all fashion, i.e. all producer subtasks create tuples to all consumer subtasks; or in a many-to-many, i.e. each producer subtask produces tuples only for a subset of the consumer subtasks. The way a producer instance decides to which partition a tuple is sent depends on the routing policy that can be done based, for instance, on the hash of a tuple field, which ensures load balancing for all subtasks of the same operator.

2.2.3 State of The Art

Regarding examples of DSPSs, the followings are currently the most relevant ones in the stream processing landscape, and as such it is worth to mention them. These systems not only bring interesting ideas to the field, but they are also broadly used in the industry.

Apache Spark Streaming Is a well-known batch dataflow processing framework [20]. To support stream processing, it provides a streaming API named Spark Streaming. The way stream processing works on Apache Spark is based on *Micro Batching*, being its key difference when comparing with other solutions. Thus, it ends up by having all the downsides of this approach. In Spark, these batches are called Resilient Distributed Datasets (RDDs), which are partitioned, fault-tolerant and in memory data structures that can be periodically checkpointed.

Apache Kafka Streams Developed by Confluent, it consists on a stream processing tool built on top of Apache Kafka. Kafka Streams are in all aspects similar to Apache Samza [21], except that it does not rely on a resource manager, since its main focus is on simplicity. Comparing to other solutions, the following are the key features that differentiate Kafka Streams [22]:

- It relies on the duality between streams and tables. Where a stream can be seen as a change-log of a table, and a table as a cache of the latest value associated with each key in the stream. This idea is used in order to store the state of stream processing tasks in Kafka.
- The use of stateful streams and how it leverages Kafka topics as a mean of communications between tasks, while at the same time uses Kafka as a way to provide fault tolerance support.
- The use of standby replicas for fast failure recovery.
- Well suited for micro-services use cases.

Apache Flink Similarly to Spark it supports both batch, iterative and stream processing. At its core, Flink is implemented as a streaming dataflow processing system, on top of which the *Datastream* and *Batch* processing layers are built. Therefore, Flink follows the *Tuple-at-a-Time* approach, enabling true stream processing. Key contributions from Flink are:

- The *Memory Manager* that reduces garbage collection overhead by managing off-heap memory, while at the same time is responsible for moving data to disk whenever the memory is full. It also provides a way to control the amount of memory each operator in the application actually uses. More, it allows to perform operations such as joins on top of the serialized records [18].
- The *Asynchronous Barrier Snapshotting Algorithm* [23] that allows Flink to perform distributed snapshots of the applications' state enabling exactly-once processing semantics.

StreamScope Developed by Microsoft and presented in [19]. Though it is not publicly available, it brings some interesting new ideas such as:

- *Hybrid Streams* (discussed in previous sections) together with a recovery mechanism that provides exactly-once processing.
- Three different task recovery techniques: *Checkpoint-based Recovery*, *Replay-based Recovery* and *Replication-based Recovery*.
- Straggler nodes detection.
- Node maintenance and application scaling without downtime, via replication.

2.3 Resource Management

2.3.1 Resource Management Systems

Resource Management Systems appear as a solution to manage resources and schedule jobs and tasks on a cluster of machines. They expose an interface that allows clients to allocate resources on those machines so that they can be used to run applications. Examples of such applications are any of the DSPSs mentioned above, a web server or even a database.

From the client perspective, he no longer needs to care where the applications are running, he has a well-established abstraction that makes him look at a cluster of machines as a whole. These groups of resources, i.e. CPU, memory and disk, that are reserved for a certain application are often called *containers*.

From the Resource Manager perspective, it only needs to take care of allocating requested containers to the available machines in order to maximize the resource usage. It also needs to solve other problems such as handling failures from the client to avoid having locked resources that are not being used; providing container isolation; handling different scheduling strategies that can be followed depending on the applications nature; and supporting multi-tenancy environments where different users may have different quotas and priorities on the resources. Regarding resource isolation, Resource Managers usually provide this by using their own mechanism (with support from the underlying operative system) or by using Docker[2] containers.

Besides Resource Managers such as Apache Hadoop YARN[3] and Apache Mesos, that will be described below, many others are available, though not usually used in the stream processing context. For instance, Google Kubernetes, Omega and Borg [2, 24].

Apache Hadoop YARN Is the resource management layer for the Apache Hadoop ecosystem. It is decomposed in three main components: the *Resource Manager*, the *Application Master* and the *Node Manager* [25]. Each application has an Application Master associated that will be responsible for negotiating the resource containers that the application requires (by sending requests to the Resource Manager), as well as track their status and monitor them. The Application Master is also responsible by guaranteeing fault tolerance to the application it is responsible for.

The Resource Manager performs the resource allocation and the monitorization of the Node Managers and Applications Masters. It can be sub-divided into two components:

- **Scheduler:** responsible for the scheduling of the containers based on multiple constraints such as capacity, fairness and SLAs. In order to allow for different scheduling policies, the scheduler is a pluggable component. By default, YARN provides a fair scheduler and a capacity scheduler. Both these policies group resources in a *queue*. Queues may be defined per application, user or organization and may have a hierarchical relation between themselves. In the fair scheduler, resources in a queue are distributed in a fair way over the running applications associated with the queue. It also allows to give different weights to the applications so that they get different proportions of the queue. The capacity scheduler allows queues to share resources as needed, giving priority to queues with higher affinity, based on the defined hierarchy.

- **Application Manager:** responsible for accepting job submissions, launching the Applications Masters and by restarting them in case of failure.

The Node Managers are the slaves. They register themselves to the Resource Manager, offer their resources, monitor their resource usage and send heartbeats to the Resource Manager.

[2]"Docker", http://docker.com, (June 22, 2018)
[3]"Apache Hadoop", http://hadoop.apache.org, (June 22, 2018)

Thus, the workflow to schedule an application in YARN, is as follows: deploy the Application Master to the YARN cluster via the Resource Manager; once the Application Master becomes ready, it can negotiate resources for its containers with the Resource Manager, as it requires.

Additionally, in [5], a solution is proposed, that aims at improving the resource efficiency and overcoming the observed resource overallocation issue in YARN, by estimating the resource usage of the worker nodes and taking this estimation into account when performing task scheduling decisions.

Apache Mesos Mesos was initially developed at the University of California, Berkley [3]. It differs from YARN as it introduces two-level scheduling: it delegates the per-application work scheduling to the applications themselves while remains responsible for resource distribution between the applications. This allows different frameworks to use different scheduling strategies adapted to their use case.

It is divided into three main components: the *Mesos Slaves*, that execute the containers; the *Mesos Masters*, that manage the Mesos Slaves daemons; and the *Frameworks*, that perform the per-application work scheduling. Regarding Frameworks, the client may develop its own, for instance, Flink Master works as a Framework when working with Mesos. The communications between these components works as follows: the Mesos Master sends specific resource-offers to each Framework; upon receiving an offer, the Framework can accept it or reject it based on whether the offer matches the application needs; once an offer is accepted, the Mesos Master receives the response with the details of the container and allocates it to the Mesos Slave associated with the offer that was accepted.

The Framework has the power to accept or reject offers, while the Mesos Master has the power to decide which resources to offer. This allows the Mesos Master to optimize resource allocation, and the Framework to decide from a set of offers the ones it prefers, for instance, based on data locality. More, at a given time, an offer to a specific resource can only be sent to one of the Frameworks, avoiding having two Frameworks accepting the same offer. This was called pessimist or exclusive approach. Other systems such as Google Omega follow a more optimistic one, by sending the same offer to multiple Frameworks at the same time.

Mesos' *Dynamic Oversubscription* module [10], allows the execution of best-effort tasks in reserved but unused resources, i.e. revocable resources. By constantly monitoring the cluster resources, the *Resource Estimator* keeps track of oversubscribed resources, while the *QoS Controller* makes sure that the revocable tasks don't interfere with the regular tasks. If they do interfere, these revocable tasks can be killed or throttled in order to correct the QoS. Similar techniques have also been used and proposed in other systems such as in [26].

Another approach that has also been developed on top of Mesos, and that also tackles the resource overallocation problem, is called the *Dominant Resource Fairness* [27]. It consists in a resource scheduling policy that aims at fair resource distribution over heterogeneous tasks and that: incentivizes users to

share resources; guarantees that a user cannot increase his allocation by lying about his requirements; and that no user will want to change his allocated resources with the ones from another.

2.3.2 Task Scheduling in Stream Processing

Multiple strategies can be found for scheduling stream processing application's tasks over the available resources in the cluster.

In Flink, each Task Manager (TM) provides a set of resources and a set of slots [18]. Resources in a machine are fairly shared among slots, e.g. there is no CPU isolation. Flink's scheduling strategy consists in assigning tasks to all slots in a single machine, and only then move to the next one. This strategy works well when working with elastic clusters, or with a resource manager such as Apache Mesos. Additionally, the Flink task scheduler also allows the user to provide hints for task co-location, so that two tasks can be scheduled in the same machine, in order to reduce communications' overhead.

Apache Storm on the other hand, uses a different approach. Its default scheduling strategy consists in a naive round robin distribution of tasks over the available resources [28]. Additionally, a *Resource Aware Scheduler* was proposed in [7], and is supported by Storm. Which allows the user to specify resource related restrictions for each application's task, e.g. amount of CPU and memory. The scheduler then uses a bin packing algorithm to schedule the tasks according with the restrictions, reducing resource wastage and fragmentation. Both schedulers also support multi-tenancy related restrictions, i.e. tenant / application priority and resource isolation.

2.4 Approximate Computing

In stream processing, Approximate Computing techniques appear as a way to deal with overload situations where the application can't keep up with the rate at which data arrives, and therefore as an alternative to scaling up. In such techniques, the accuracy of the application is often penalized whether by discarding data or cutting on processing time and/or space.

2.4.1 State Synopsis and Sampling

State Synopsis is a category of Approximate Computing methods whose objective consist in summarize the information contained in a data stream.

Within this category many methods can be found, such as Sketches, Histograms and Wavelets [29]. Sampling appears as the simplest of these methods. Extensive research has been done on sampling within the field of Statistics. The objective in sampling is to maintain a subset (named *sample*) of the data that captures a good representation of the universe. In stream processing, this subset is maintained

over time and updated whenever it is necessary to reflect new trends or interesting occurrences in the input data stream, due to its dynamic nature.

Bernoulli Sampling is known as the simplest sampling method. Where for each incoming event, the element is included in the sample with a given probability p_i, thus being a uniform Sampling method, otherwise is ignored [30]. Nevertheless, many other methods, such as: Congressional Sampling and Reservoir Sampling, can be found in literature. Though more complex, these methods provide stronger guarantees on the sample that is generated.

Recent work can be found on evaluating these techniques on DSPS such as Flink, on [31] and [30].

2.4.2 Load Shedding

Load Shedding follows a different idea to sampling. Instead of building a sample of the universe, it keeps as many elements as possible until the system becomes stressed. Once this happens, it starts dropping information. A key use case for load shedding is on dealing with bursty workloads.

When performing load shedding in a stream processing application, the following main problems must be considered [32, 33]:

- **When to shed?** The solution passes by creating a system that monitors the load of the application and detects when the application is under stress, according to specific metrics such as throughput.
- **Where in the DAG should the events be dropped?** When applications can have different queries with different accuracy requirements, one needs to find ideal drop locations in order to avoid processing events that will be dropped further down in the application DAG, while assuring the accuracy requirements.
- **Which events to shed?** This leads to having a method for selecting which tuples to drop. Since load shedding is triggered when the system is under stress, usually simple sampling techniques are used, such as random drops. Nevertheless, more advanced methods can be found such as semantic drops, that try to minimize loss of valuable events.
- **How much to shed?** Given the workload of the application, how many tuples should be dropped so that the application can keep up with the incoming data rate.

Studies can be found on applying load shedding techniques in DSMS, for instance in [34] and [35].

2.5 Case Study: Apache Flink

As mentioned previously, Flink is the DSMS that is the focus of this work. As such, this section is dedicated to present not only an overview of its architecture, but also some of its relevant features.

2.5.1 Architecture

Flink follows a master-slave architecture, where the master servers are called Job Managers (JMs) and the slaves, TMs [18]. The JMs are responsible for receiving the requests to deploy stream applications, schedule their tasks and monitor both the applications' tasks and the TMs in order to properly handle failures. At each time, multiple JMs may be alive, though only one of them is the leader. This is the way Flink provides JM high availability. Additionally, they are also responsible for checkpoint coordination and by coordinating application recovery from a previous snapshot, whenever a task fails.

On the other hand, the TMs are responsible for executing the tasks. Each TM offers to the cluster a set of slots that can be used to execute tasks. As such, the amount of slots in a TM limits the amount of tasks the TM can execute in parallel. The system administrator is responsible by defining the amount of slots a TM provides. In Flink stream processing applications, slots do not isolate computational resources such as CPU and memory. Instead, the available CPU is shared fairly among the executing tasks, as required; and memory is either shared fairly among the tasks, as required, or eventually by a third party system such as RocksDB, when dealing with the tasks' state.

Regarding communications, the TMs and the JMs follow the Actor Model [36], where each of these components represent an *actor* (an high level abstraction that is used in concurrent and distributed systems), that holds a state, has a specific behavior and can asynchronously exchange messages with other actors. Flink uses Akka Actors[4], an implementation of the Actor Model. The communication between tasks can be done either in memory, in case they are running within the same JVM; or by using Netty[5], otherwise.

A Flink cluster can either be executed in standalone mode, or be integrated with a resource management system such as YARN or Mesos, in which case, the JM is able to request containers to launch new TMs when needed.

2.5.2 Tasks' Chaining and Pipelining

At runtime, Flink allows to group a chain of subtasks into a single task, that is executed by a single thread, where the subtasks perform computations sequentially on the received events. The advantage of this is that it reduces the overhead of inter task communication, thus reducing latency.

For higher throughput, Flink also allows to pipeline subtasks, i.e. split a chain of subtasks into multiple tasks being executed by separated threads in parallel, that can be executed on different TMs.

Finding the optimal solution on whether to use task pipelining or not, is the responsibility of the user and depends from application to application. In [37] a mechanism has been proposed that allows to discover at runtime when to use pipelining, based on metrics obtained from monitoring the applications.

[4] "Akka", https://akka.io, (June 22, 2018)
[5] "Netty", https://netty.io, (June 22, 2018)

2.5.3 Slot Sharing

Slot Sharing [9] is a way to increase resource efficiency. It does so by changing the semantics of a slot. Instead of a slot executing a single task, it can now execute a pipeline of tasks that belong to the same application. As such, multiple threads may be running within the same slot, sharing the resources. This not only allows to reduce communication overheads, but also to increase resource efficiency whenever an application contains tasks with different resource requirements. A restriction is that slots can only be shared by tasks that belong to the same application and that do not perform the same operations.

2.5.4 Dynamic Scaling Applications

Recently it was added to Flink a dynamic scaling mechanism [38] for applications, that allows users to redefine the task parallelism of running applications. The way this mechanism works is by leveraging Flink's snapshotting mechanism. Whenever an application requires to be scaled, the system takes a snapshot of the application, stops the application and restarts it with the new parallelism degree. For stateless tasks, this is trivial; for stateful ones, the state is redistributed based on the stream key partitions that each task will be responsible for.

Flink does not provide any support for triggering scale up or scale down actions based on metrics such as throughput or backpressure.

2.5.5 Asynchronous I/O

Whenever an application needs to perform some blocking operations, such as a query to a database, the task that executes that operation will be blocked waiting for the response. If the resources allocated to the task are not well dimensioned, resource wastage will occur.

In the past, the solution to this problem was on configuring the TMs with more slots than the amount of CPUs they provide. In recent releases, Flink started providing support for asynchronous calls, thus allowing to perform non-blocking requests to a service and receive the response via a callback. It does so by providing the user an API to define how the request is performed and how to handle the callback response. It also provides mechanisms to preserve order between requests, event time and to handle fault tolerance. A more detailed description of this mechanism can be found in [39].

2.5.6 Window Programming Model

Flink provides an API for users to define their own windows [40], besides the standard ones. The way this is done is by specifying the following functions:

- **Window assigner:** given a new element, it defines to which windows the element is assigned.
- **Window functions:** specify the function to be applied on the windows. These functions can be a *reduceFunction*, a *foldFunction* or a *windowFunction*. The first two types can be computed incrementally as new elements are added to the window. The last one, corresponds to functions that cannot be computed incrementally, therefore are less efficient and consume more resources.
- **Trigger:** define when a window must be triggered, i.e. when the window can be processed by the window function. When the trigger is time based, Flink will take into account the defined time watermark.

Additionally, the user can define *Evictors* that allow to specify which elements to keep after a window is triggered and after or before the window function is applied.

In Flink, windows can be non-keyed or keyed. In the first case, all the windowing logic will be performed within a single task, whereas on the second one this can be performed in parallel.

2.6 Summary

This chapter highlighted related work that contains useful insights that are relevant for the development of the proposed solution.

In terms of stream processing, this chapter provided a good background regarding its main concepts, such as how stream processing applications are represented; and key functionalities like fault-recovery. Several state of the art stream processing solutions were mentioned, together with their most differentiating features. Apache Flink architecture was also explored with more detail, together with key resource efficiency features such as *Slot Sharing* and *Task Co-location* that need to be taken into account by the proposed solution.

In terms of resource management, Mesos *Oversubscription* module provides a good example regarding how resource underutilization and task overallocation problems can be addressed, i.e. a feedback mechanism is required to keep track of the QoS of the applications in order to detect and act upon these situations. As may be noticed in the following chapter, the architecture of our solution is similar to the one used in the Mesos *Oversubscription* module.

Regarding load shedding, the main problems that need to be addressed were identified, which will be taken into consideration when defining the load shedding mechanism that is required by the proposed solution.

3

Architecture of the Solution

Contents

3.1 Overview

As mentioned earlier, the proposed solution targets the problem of resource underutilization in distributed stream processing systems, more precisely in Apache Flink. This chapter focuses on detailing this solution, from its architecture and its components, to the specification of the model and algorithms that will govern the decisions taken by the system.

The rationale behind our solution consists in using a task scheduling policy, that upon a new task scheduling request, simply assigns the task to the cluster machine with the most available resources, in terms of CPU. By following such a policy, the system is able to avoid the problem of resource over-allocation, as the concept of having tasks allocating fixed amounts of resources gets completely dropped.

Nonetheless, by following this policy, the system may lead applications to situations of resource starvation. Such situations can appear due to the optimism of the proposed policy. This may cause applications' performance to degrade and failing to keep-up with their incoming workload.

When such situations are detected by the system, two possible resolution approaches are proposed. The system can either decide to perform *Task Re-scheduling* of specific application tasks to other machines with more available resources, causing some application downtime; or it can decide to use *Load Shedding*, allowing applications to keep up with their incoming workload at the cost of decreasing the accuracy of their results. This second mechanism will be the preferred approach, only using the first one as a last resort.

To guide the system decisions, two restrictions should be specified by the user, for each query in their applications. The first restriction corresponds to the *priority* of the query, which allows users to define some queries as being more important than others. The second restriction corresponds to the *minimum acceptable accuracy* of the query. This last restriction is relevant, since below a given accuracy, the results provided by a query stop being useful or even meaningful to the user.

Both these restrictions are crucial for the system to decide when and how to use *Load Shedding*; and when to switch to *Task Re-scheduling*. As well as on deciding on which tasks these mechanisms should be triggered, clearly being the low priority tasks the first ones to be targeted.

This chapter follows the concepts and nomenclature established by Flink, which have already been described in Sections 2.2 and 2.5.

3.2 Architecture

To achieve the intended behavior, two new components are added to the current Flink architecture: the *Task Scheduler*, that schedules tasks over the available cluster resources, by following the already described policy; and the *Quality-of-Data Controller* (QoD Controller) that continuously monitors all tasks that are executing in the cluster, in order to decide when the *Load Shedding* or *Task Re-scheduling*

mechanisms should be triggered. The *QoD Controller* is mainly guided by runtime metrics from the tasks' execution and by user provided restrictions, i.e. the application queries' *priorities* and *minimum acceptable accuracies*. Decisions from both these components are guided by a model that will be detailed in the following sections.

Figure 3.1 provides a bird's eye view of how these new components are integrated with the current Flink architecture. As shown in the figure, the two components run on the JM (2). The *Task Scheduler* executes for every task scheduling request. Meanwhile, the *QoD Controller* is triggered periodically, possibly sending messages to the TMs, either telling them to fail tasks or tune the tasks' load shedders.

Since at a given instant, Flink's high availability mechanism guarantees that only one of the JMs in the system is the leader, while the others are in standby; only the *Task Scheduler* and *QoD Controller* that are executing in the leader JM will be taking decisions. The remaining instances of these components will only become active when a leadership change occurs, causing their correspondent JM to become the leader.

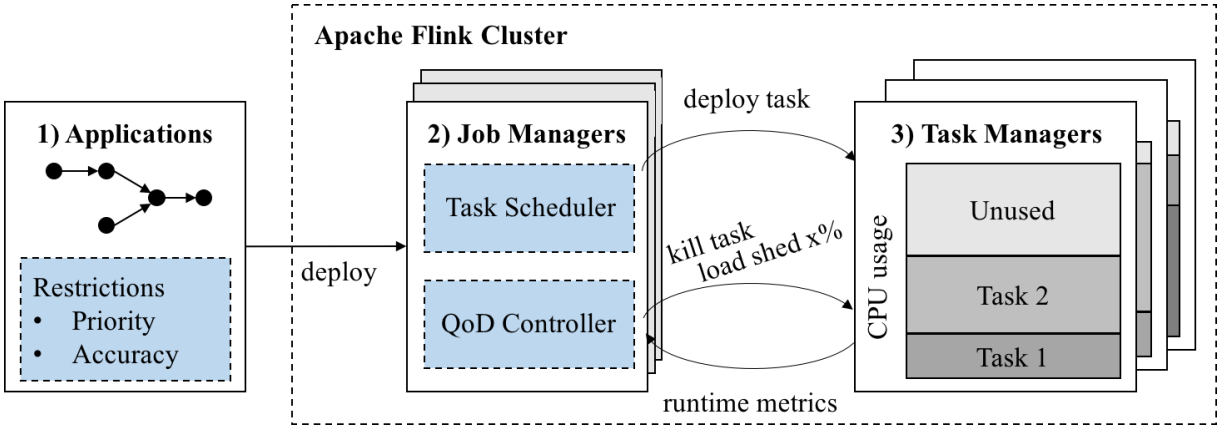

Figure 3.1: High level view of the proposed solution and its components, in a Flink cluster.

All the cluster TMs periodically send runtime metrics to the leader JM such as the CPU usage information and input / output rates from each task they are executing (3). These metrics are exposed to the main components, to help them on guiding their decisions.

To define the restrictions for each application query, Flink's stream application definition Domain-Specific Language (DSL) (1) is also extended. Allowing the user to specify the priority and minimum allowed accuracy for each query. These restrictions are then propagated to the JMs in order for the *QoD Controller* to take them into account.

Additionally, a load shedding mechanism is also introduced at the applications' runtime level. This allows them to eventually drop some of the incoming workload, while taking into account the different accuracy requirements from the application's queries.

3.3 Model

Before going through the specification of how the new components perform their decisions, the used nomenclature and concepts should first be formalized. Given a task instance i, $t(i)$ will be used to refer to the task whose i is instance of; $tm(i)$ will be used to refer to the TM where the task instance i is executing, and $taskManagers$ as the set of all the TMs in the system. Additionally, $queries$ represent the set of all the query tasks in the system's applications, which can be restricted to the queries of a specific task t by using $queries(t)$; and $downstream(t)$ / $upstream(t)$ as the downstream / upstream tasks of a task t, that are directly connected to it.

3.3.1 User Restrictions

The system allows users to specify two restrictions for their applications, that will be taken into consideration during decision making. Users will only need to specify these restrictions for the sink tasks of their applications. The system can then interpolate them to the remaining tasks.

Priority - p(t)

Priorities provide hints on how important certain tasks are compared to others, and as such, each task will have an associated priority, $p(t) \in \mathbb{Z}$, that will allow the system to prioritize access to the computational resources. The higher the value of $p(t)$, higher the priority of the task.

Given the priority that the user specifies for a given query in an application, the system will use Equation 3.1 to compute the priority of the remaining tasks in the application. This is done by simply propagating the priority specified by the user, through the query's upstream tasks. The priority of a task simply corresponds to the maximum priority of its downstream tasks.

$$p(t) = \max_{t' \in queries(t)} p(t') \tag{3.1}$$

The set of all tasks' priorities in the system, sorted in descending order, is represented by $priorities$. Values in this set can be scoped to a specific TM by using $priorities(tm)$. Additionally, $ti(tm, p)$ is used as the set of task instances in a given TM tm, with a given priority p; or $ti(tm)$ for all instances regardless of their priority. The set of all the TMs executing task instances with a given priority p, will be provided by $tm(p)$.

Minimum Accuracy - minAc(t)

The minimum accuracy of a task, $minAc(t) \in [0\%, 100\%]$, is used as a threshold of its accuracy, below which the outcome of the task stops being valuable to the user. This value is defined as the

minimum percentage of input events of the application that must be processed by a given task. These events can be dropped anywhere in the application. Note that this definition is crucial, since it is the definition of accuracy that is followed throughout this work.

Once again, this threshold only needs to be defined for the application queries. The provided value is then propagated through the query's upstream using Equation 3.2. As such, the minimum accuracy of a given task in an application corresponds to the highest minimum accuracy of its downstream tasks. This is logical, because otherwise, the downstream tasks would fail to meet their required accuracy.

$$minAc(t) = \max_{q \in queries(t)} minAc(q) \tag{3.2}$$

3.3.2 Load Shedding

Load shedding allows the system to drop some of the applications' workload for them to cope with it. These workload drops are performed by *load shedders* placed in specific locations of the applications' DAGs. Which are parameterized with the percentage of events that they should keep, named the *non-drop probability*, $d(t, t')$, between a producer task t and a consumer task t'. This parameter provides an interface for the system to tune the load shedders at runtime, as the workload of the applications dynamically changes.

Regarding the way dropped events are selected, the load shedding implementation used in this work simply performs random drops based on the non-drop probabilities that are defined. Other load shedding policies are possible, as long as they can be configured solely using $d(t, t')$.

In terms of placement of the load shedders, each load shedder is associated with a stream connecting two tasks. To avoid wasting network bandwidth, load shedding is performed in the producer task. Source tasks also have load shedders after their first subtask, which is the one responsible for pulling data from the external datasource into the application dataflow, thus, allowing workload to be dropped right after it's fetched by the application. This load shedder is also parameterized by $d(t, t')$, where *t* corresponds to the datasource. Figure 3.2 provides an example of the load shedders' placement in the physical DAG of an application.

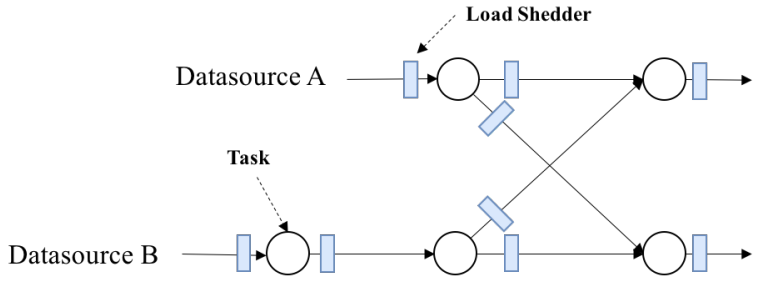

Figure 3.2: Placement of load shedders in the physical DAG of an application.

The percentage of application input events being processed by a task t is given by $ac(t)$, which corresponds to our definition of accuracy of the task. Since tasks in an application may have several instances executing in parallel, load shedding in one of the task's instances and not on the others may lead to biased results. Therefore, because the system has no *a priori* knowledge regarding the deployed applications, all task instances of the same task must respect the following restrictions:

1. They must all have the same accuracy, i.e. $\forall_{i \in instances(t)} ac(i) = ac(t)$. Where $instances(t)$ corresponds to the set of all instances of a task t.

2. For two connected instances $i1$ and $i2$ of different tasks $t1$ and $t2$, respectively, $d(i1, i2) = d(t1, t2)$.

Additionally, given two tasks $t1$ and $t2$, directly connected to a downstream task $t3$, then $ac(t1) = ac(t2)$ and $ac(t3) \leq ac(t1), ac(t2)$. Essentially, this restriction avoids biased results for tasks that consume data from two or more streams.

This yields that, given all the $d(t, t')$ in an application and taking into account the restrictions defined above, it is possible to compute the accuracy for a given task in the application using Equation 3.3. This is done by chaining all the $d(t, t')$ probabilities in a DAG branch (collection of streams) connecting an application source task and the task whose accuracy needs to be computed (including the load shedder between the datasource and the source task of the branch).

$$ac(t) = \begin{cases} 1, & \text{if } upstream(t) = \{\} \\ \frac{\sum_{t' \in upstream(t)} ac(t') \times d(t', t)}{|upstream(t)|}, & \text{otherwise} \end{cases} \tag{3.3}$$

3.3.3 Current Accuracy - currAc(t)

The current accuracy, $currAc(t) \in [0\%, 100\%]$, is used as a guideline that provides an approximation to the runtime accuracy of a given task t, based on the runtime metrics of the application. Thus providing an hint on how much overloaded a task is. The system will always attempt to maximize $currAc(t)$.

This function is defined as presented in Equation 3.4. Where the current accuracy is computed using $locAc(t)$, that represents the local accuracy of a task t, and is defined in Equation 3.5,

$$currAc(t) = \begin{cases} locAc(t), & \text{if } upstream(t) = \{\} \\ locAc(t) \times min_{t' \in upstream(t)} currAc(t'), & \text{otherwise} \end{cases} \tag{3.4}$$

$$locAc(t) = \begin{cases} 100\%, & \text{if } \sum_{t' \in upstream(t)} outRate(t', t) = 0 \\ \frac{\min_{i \in instances(t)} inRate(i)}{\left(\sum_{t' \in upstream(t)} outRate(t', t)\right) / |instances(t)|}, & \text{otherwise} \end{cases} \tag{3.5}$$

It's important to note that, according to our definition, the current accuracy of a task is calculated using the minimum current accuracy of its upstream tasks. Meaning that if a task has two upstream branches with different current accuracies, its current accuracy is determined by the minimum. Because this is the most that is currently being guaranteed. Other definitions of the current accuracy are possible, and eventually the user could even define it for each application in a different way.

Regarding Equation 3.5, the $inRate(i)$ represents the input rate of an instance i of a task t; while $outRate(t', t)$ corresponds to the output rate of an upstream task t' to a task t. Therefore, the local accuracy of a task matches the minimum local accuracy of its instances, with the assumption that the output rate of its upstream is uniformly distributed over its task instances.

Additionally, if the denominator of the Equation 3.5 equals zero then, $locAc(t) = 100\%$. This happens because, if the direct upstream tasks are not emitting any tuples, then their downstream tasks are processing all events that they receive from the upstream, i.e. none. Also note that $locAc(t) = 100\%$ when both upstream and downstream tasks are processing data at the same rhythm.

Using this definition of current accuracy, the objective of our system is therefore to minimize the difference between a task's upstream output rate and its input rate. Or in other words, to maximize the overall application throughput.

Note that the current accuracy of a query, q, is not necessarily the same as the actual accuracy of the query, $ac(q)$. In fact, the actual accuracy corresponds to the upper bound of the current accuracy. This happens because $ac(q)$ is strictly defined by the load shedders' non-drop probability. While the $currAc(q)$ not only embodies the non-drop probabilities, but also the input and output rates differences that appear due to the mismatching input / output throughputs.

3.3.4 Maximum Achievable Accuracy - maxAc(t)

Given the desired accuracy for each downstream query q, represented as $desired(q)$; $maxAc(t)$ returns the maximum accuracy that the upstream task t will have, based on the restrictions imposed by the downstream queries and taking into account that events will be dropped as soon as possible in the DAG. The way this it is computed is described in Equation 3.6.

$$maxAc(t) = \max_{q \in queries(t)} desired(q) \tag{3.6}$$

3.3.5 CPU Load - cpu(t)

The expected CPU load for a task t, $cpu(t)$, is calculated based on the CPU load of its instances. Equation 3.7 shows how $cpu(t)$ is computed, where $cpuLoadMetric(i)$ corresponds to the CPU load percentage obtained from the TM metrics for a task instance i. If a task is using a full CPU virtual core,

then $cpuLoadMetric(t) = 100\%$.

Since the local accuracy, defined in Section 3.3.3, is restricted by the minimum local accuracy of the task's instances; then $cpu(t)$ is also defined by the CPU load of this same instance, i.e. the one with the lowest local accuracy.

$$cpu(t) = cpuLoadMetric\left(argmin_{i \in instances(t)} inRate(i)\right) \tag{3.7}$$

Additionally, it's also defined that $cpu(tm)$ corresponds to the CPU usage of the TM tm, including non-Flink related processes; $tCpu(tm) = \sum_{i \in ti(tm)} cpu(i)$; and $nCores(tm)$ to the amount of CPU virtual cores provided by a given TM, tm.

Note that there is an assumption that all the TM machines have similar computational capacity. This assumption can easily be dropped by using a normalization factor for the computational capacity of each machine, as it is usually done in similar situations.

3.3.6 Minimum Obtained CPU - mObtCpu(i)

The minimum CPU amount a task instance i is guaranteed to get, $mObtCpu(i)$, based on the current available CPU, $aCpu(tm)$, of the respective TM, tm. This is computed as described in Equation 3.8. The available CPU time is evenly distributed over all task instances with the same priority. Note that a task instance may require less CPU resources than the ones it can get, in order to achieve the accuracy that it requires.

$$mObtCpu(i) = \frac{aCpu(tm(i))}{|ti(tm(i), p(t(i)))|} \tag{3.8}$$

3.3.7 Required CPU - rCpu(t, ac)

The CPU required by a task t to achieve $ac(t) = ac$ is represented as $rCpu(t, ac)$. This function is defined in Equation 3.9, and assumes that the CPU load and accuracy are proportional, which is a reasonable assumption for stream processing applications.

$$rCpu(i, ac) = \begin{cases} 0\%, & \text{if } currAc(t) = 0\% \text{ and } ac = 0\% \\ 100\%, & \text{if } currAc(t) = 0\% \text{ and } ac \neq 0\% \\ \min\left(\frac{ac \times cpu(t)}{currAc(t)}, 100\%\right), & \text{otherwise} \end{cases} \tag{3.9}$$

The equation has two special conditions, for which the rationale is the following:

1. If the task's current accuracy and required accuracy, ac, equal 0%. Then, $rCpu(t, ac) = 0\%$, since no CPU is required to achieve 0% accuracy.

2. If the task's current accuracy is 0%, but the required accuracy is not. Then, $rCpu(t, ac) = 100\%$. In this case, the system simply provides its best bet on the CPU that will be required (a full virtual core), since it still doesn't have enough information to provide a safe estimation.

The CPU required by a task t to achieve the minimum, $minAc(t)$, and maximum accuracies, $maxAc(t)$, are also defined in Equations 3.10 and 3.11, respectively.

$$minReqCpu(t) = rCpu(t, minAc(t)) \tag{3.10}$$

$$maxReqCpu(t) = rCpu(t, maxAc(t)) \tag{3.11}$$

3.3.8 Obtained Accuracy - obtAc(t, cpu)

Given a task t and a provided CPU load cpu; $obtAc(t, cpu)$ returns the maximum accuracy the task can provide to its downstream using that CPU amount, as expressed in Equation 3.12. Once again, some special conditions are specified:

1. If the current accuracy is 0% and no CPU is provided, then the task will clearly get 0% accuracy. This is in line with the condition 1. when defining $rCpu(t, ac)$.

2. If on the other hand the current accuracy is 0% and CPU load is provided. Then, an optimistic bet is performed, estimating that the task will be able to achieve a 100% accuracy.

3. If the current CPU load of the task equals 0% and no CPU load is provided, the task will clearly maintain the same accuracy.

4. If the current CPU load of the task equals 0%, but some CPU load is provided. Then the system will once again perform an optimistic bet, estimating that the task will achieve a 100% accuracy.

$$obtAc(i, cpu) = \begin{cases} 0\%, & \text{if } currAc(t) = 0\% \text{ and } cpu = 0\% \\ 100\%, & \text{if } currAc(t) = 0\% \text{ and } cpu \neq 0\% \\ currAc(t), & \text{if } cpu(t) = 0\% \text{ and } cpu = 0\% \\ 100\%, & \text{if } cpu(t) = 0\% \text{ and } cpu \neq 0\% \\ \min\left(\frac{currAc(t) \times cpu}{cpu(t)}, 100\%\right), & \text{otherwise} \end{cases} \tag{3.12}$$

3.3.9 Slack - slack(tm, p)

For a given TM, tm, $slack(tm, p)$ represents the sum of the differences between: the CPU percentage that its tasks with priority p require (in order to achieve the maximum accuracy they can, based on current

restrictions imposed by other tasks and while assuming the minimum accuracy is guaranteed); and the available CPU for the task. The way this is done is presented on Equation 3.14, where Equation 3.13, simply computes the difference between the required CPU to achieve the maximum required accuracy and the one to achieve the minimum accuracy.

$$diffReq(t) = maxReqCpu(t) - minReqCpu(t) \tag{3.13}$$

$$slack(tm, p) = \sum_{i \in ti(tm,p)} (diffReq(t(i)) - mObtCpu(t(i))) \tag{3.14}$$

3.4 Task Scheduler

Given a task instance, the *Task Scheduler* component is responsible for determining to which TM it should be scheduled. This decision is taken by selecting the TM with the most available CPU, reducing the probability of having to re-schedule the task instance later during its execution.

Task's locality preferences should also be taken into account. This is an already existent feature in Flink, that allows to reduce applications' latencies by co-locating some of their tasks in the same TM. Such co-location may not be feasible either because there are no more slots available in the preferred TM; or, in our case, because the preferred TM doesn't have enough resources for the task to achieve its accuracy threshold. In such situations, the locality preference must be disregarded. Therefore, the *Task Scheduler* requires an estimation of the CPU load used by the task instance i that is going to be scheduled, given by $estimatedCpuload(i)$.

If the system has previous CPU load metrics from the task execution, either from the instance that will be scheduled or from the other instances of the same task; then, these metrics are used as an initial estimation of the instance's CPU load. If such execution history doesn't exist, the system performs a pessimistic estimation of the CPU resources required by the task instance that will be scheduled, estimating that the task instance will require a full CPU virtual core, i.e. 100%. Other initial CPU usage estimations are possible. Note that there is a trade-off between having the application's task instance being placed in the same machine, promoting low latencies, but potentially being re-scheduled (if the initial estimation is below the real CPU usage); or having these tasks distributed over the cluster, with low probability of being re-scheduled and making it potentially network I/O bounded instead of CPU bounded (otherwise).

Algorithm 3.1 presents the pseudo-code of the scheduling function that, given a task, returns the TM to which it should be scheduled. Given a task t, it starts by picking all TMs that match its $locality$ preferences, provided by the function $taskManagers(locality)$; and removing the ones that won't be

able to provide enough CPU resources for the task execution (line 2). If none of the task's preferred TMs has conditions for its execution, then the *Task Scheduler* uses all the available TMs in the cluster as candidates for task scheduling (lines 3-5). The algorithm then proceeds by selecting the TM with the maximum estimated available CPU load, from the set obtained in the previous steps (line 6).

Algorithm 3.1 Task Scheduling Algorithm.

```
1:  function SCHEDULETASKINSTANCE(i, locality)
2:      tms ← {tm ∈ taskManagers(locality) : estimatedAvailCpu(tm) − estimatedCpuLoad(i) ≥ 0%}
3:      if tms = {} then
4:          tms ← taskMananger s
5:      end if
6:      tm ← argmax_{tm∈tms}(estimatedAvailCpu(tm))
7:      return tm
8:  end function
9:
10: function ESTIMATEDAVAILCPU(tm)
11:     return 100% × nCores(tm) − cpuLoad(tm) − Σ_{i∈newTaskInstances(tm)} estimatedCpuLoad(i)
12: end function
```

The estimated available CPU of a TM tm, is computed using the $estimatedAvailCpu(tm)$ function (lines 10-12). This function takes into account: the maximum CPU load that can be available in the TM, i.e. the maximum available CPU capacity that it has to execute tasks, without incurring in resource starvation; the CPU load that is already being in use either by tasks that are already running, or by other non-Flink related processes that may be consuming CPU time; and the estimated CPU load for the $newTasksInstances(tm)$, which represents new tasks in the TM for which the system doesn't have yet runtime metrics. The CPU load of these tasks is not yet captured by the $cpuLoad(tm)$.

It is possible, and acceptable, for the $estimatedAvailCpu(tm)$ to take negative values, since often our estimations can be wrong. Even if the estimation happens to be correct and a task is scheduled to a TM where this estimation is negative, the *QoD Controller* will be able to later re-schedule the task instance to another TM.

Note that this CPU estimation is only used for task scheduling purposes, not being taken into account by the *QoD Controller*, described next.

3.5 QoD Controller

The *QoD Controller* is the core component of our architecture, that continuously ensures that all Quality-of-Data (QoD) requirements are met across the cluster. This is done by using Algorithm 3.2, which is periodically executed with a configurable frequency. This frequency is relevant, since it will dictate the reactiveness of the system to QoD changes. If the frequency is too high, then the system will react fast to changes in the QoD. But more resources will be consumed in the JM, without necessarily

being required. If the frequency is too low, it may not be reactive enough. Thus, taking too long to take actions to guarantee that the applications keep up with their incoming workload. Note that the runtime metrics used by the model (CPU load and input / output rates) are also computed periodically, using a moving average to smooth possible fluctuations.

Algorithm 3.2 QoD Controller main cycle.

1: **for all** $tm \in taskManagers$ **do**
2: $\quad aCpu(tm) \leftarrow 100\% \times nCores(tm) - cpu(tm) + tCpu(tm)$
3: **end for**
4: **for all** $q \in queries$ **do**
5: $\quad desired(q) \leftarrow 100\%$
6: **end for**
7: **for all** $tm \in taskManagers$ **do**
8: $\quad reqCpu \leftarrow \sum_{i \in ti(tm)} minReqCpu(t(i))$
9: $\quad released \leftarrow 0$
10: $\quad$ **if** $reqCpu > aCpu(tm)$ **then**
11: $\quad\quad released \leftarrow killTasks(reqCpu - aCpu(tm), tm)$
12: $\quad$ **end if**
13: $\quad aCpu(tm) \leftarrow aCpu(tm) - reqCpu + released$
14: **end for**
15: **for all** $p \in priorities$ **do**
16: $\quad$ **for all** $tm \in tm(p)$ order by $slack(tm, p)$ DESC **do**
17: $\quad\quad distributeEvenly(ti(tm, p), aCpu, desired)$
18: $\quad$ **end for**
19: **end for**
20: $Compute\ d(t, t')\ based\ on\ the\ desired\ accuracy\ for\ the\ queries$
21: $Send\ new\ d(t, t')\ to\ the\ Task\ Managers\ where\ t\ is\ running$

In each iteration, the algorithm starts by initializing: the available CPU of each TM, as the total available CPU it has to execute the Flink tasks (lines 1-3); and the desired accuracy for each query as 100%, since they haven't yet been pruned with load shedding (lines 4-6). At the begin of each iteration, the *QoD Controller* believes that all queries will be able to achieve their maximum possible accuracy.

The algorithm then proceeds to guarantee that each TM has the necessary CPU to keep executing all its task instances with the minimum accuracy they require, regardless of their priorities (lines 7-14). If such is not possible, it immediately releases the required CPU by forcing the re-scheduling of some of the TM's tasks (line 10-12). This happens because, in such cases, load shedding tasks' input workload will not be enough to improve their QoD. The TM available CPU is updated accordingly (line 13), so that it knows how much CPU will be available to further improve the tasks' accuracy.

Once each task is guaranteed to have its minimum accuracy, the algorithm distributes the remaining available CPU over the task instances (lines 15-19), this time, respecting their priorities. It starts by the TMs with highest slack, i.e. the TMs with less CPU available to increase the accuracy of their task instances. This, while avoiding having to backtrack already made decisions, since the TMs with lower slack will be the most restrictive ones in terms of accuracy that they can provide to their task instances.

The system only distributes the available CPU load to lower priority task instances, once all of the higher priority ones, in the same TM, are guaranteed to achieve their maximum accuracy.

The following Sections 3.5.1 and 3.5.2, further detail the two algorithms that are used to decide which tasks should be re-scheduled (line 11) and to fairly distribute the remaining CPU load among tasks with the same priority (line 17), respectively.

Finally, the non-drop probabilities of all streams are computed, using the method that will be defined in Section 3.5.3; and the new values are sent to the TMs to adjust their load shedders (lines 20-21).

3.5.1 Task Re-Scheduling

To select which tasks in a TM, tm, should be re-scheduled in order to release a given amount of CPU load, cpu, the system follows a policy that aims at: 1) Minimizing the amount of tasks that will be re-scheduled to another TM, as well as their priorities; 2) Avoid releasing more CPU load than necessary.

Algorithm 3.3 Selection of tasks to be re-scheduled.

```
1:  function KILLTASKS(cpu, tm)
2:      released ← 0%
3:      I ← {}
4:      for all p ∈ priorities(tm) do
5:          if cpu − released > 0% then
6:              I ← I ∪ ti(tm, p)
7:              released ← released + Σ_{i∈ti(tm,p)} minReqCpu(t(i))
8:          end if
9:      end for
10:     for all i ∈ I order by p(t) DESC, minReqCpu(t(i)) DESC do
11:         if released − minReqCpu(t(i)) ≥ cpu then
12:             released ← released − minReqCpu(t(i))
13:         else
14:             fail(i)
15:             cpu ← cpu − minReqCpu(t(i))
16:         end if
17:     end for
18:     return release
19: end function
```

Algorithm 3.3 is used to achieve these two goals, by performing this task selection in two steps. The first step (lines 3-10), aims at determining the maximum priority of the tasks that may have to be re-scheduled to release at least the required CPU load. It returns a set of candidate task instance to be re-scheduled (I), and the released CPU load if all those task instances are actually re-scheduled ($released$). Since the required CPU to release is computed based on the CPU load that each task requires to achieve its minimum acceptable accuracy, the algorithm considers that the CPU load that is released by re-scheduling a task instance t to another TM, corresponds to the $minReqCpu(t)$.

The second step of the algorithm (lines 10-17) avoids releasing more CPU than necessary, by prun-

ing the candidate tasks set. To do so, it starts by the tasks with higher priority and higher required CPU. If by removing the task from the set, this still allows to release at least the amount of CPU that must be released, then, the task is removed. Otherwise, the task instance is failed on purpose, in order to be re-scheduled to another TM by Flink. Note, once again, that the released CPU by re-scheduling a task instance at this point, corresponds to the one used to achieve its minimum acceptable accuracy.

Regarding this last step, it prefers to re-schedule tasks with low CPU consumption. The reasons why this semantic was selected are: 1) It reduces resource fragmentation; 2) Smaller tasks are easier to re-schedule since they require less resources; 3) Once re-scheduled, these tasks should take less time to recover and start coping again with their incoming workload.

Follows an example, to better understand how the algorithm works: suppose the following tasks in a given TM, represented by a tuple $(minimum\ required\ CPU, priority)$: $(20\%, 4)$, $(10\%, 3)$, $(40\%, 3)$, $(10\%, 2)$, $(10\%, 2)$, $(10\%, 1)$; and that the system needs to release 40% of the CPU load in that TM. The first step of the algorithm (lines 4-9) yields all tasks with priority lower than 4, and a released CPU load of 80%. The second step of the algorithm (lines 10-17) will end up by failing all tasks in the candidate set except $(40\%, 3)$. Therefore, releasing exactly the required CPU load.

The system assumes that there will be another TM with enough available resources to execute the re-scheduled tasks with their required accuracy. This can be achieved by using a separated mechanism for auto-scaling the Flink cluster.

3.5.2 Resource Distribution

To fairly distribute the remaining available CPU of a TM over a set of tasks, Algorithm 3.4 is used. It receives as input the available CPU load to distribute, $aCpu$; the set of tasks over which it should be distributed, it; and the desired accuracy for all tasks in the system, $desired$.

Algorithm 3.4 Fair distribution of CPU load.

1: **function** DISTRIBUTEEVENLY(it, $aCpu$, $desired$)
2: $c \leftarrow 0$
3: $T \leftarrow it$ order by $rCpu(i, maxAc(i)) - minReqCpu(t(i))$ INC
4: **for all** $i \in T$ **do**
5: $maxReq \leftarrow maxReqCpu(t(i)) - minReqCpu(t(i))$
6: $cpu \leftarrow \min\left(maxReq, \frac{aCpu(tm(i))}{|it|-c}\right)$
7: $ac \leftarrow obtAc(i, cpu) + minAc(i)$
8: **for all** $q \in queries(t(i))$ **do**
9: $desired(q) \leftarrow \min\left(desired(q), ac\right)$
10: **end for**
11: $aCpu(tm(i)) \leftarrow aCpu(tm(i)) - cpu$
12: $c \leftarrow c + 1$
13: **end for**
14: **end function**

The algorithm starts by distributing the CPU load by the tasks with lower required CPU to achieve the accuracy they need, avoiding decision backtracking (line 3), since these tasks may require less CPU to achieve their maximum accuracy than the CPU they can get. If such tasks exist, then the remaining CPU from those tasks is fairly distributed across the remaining task instances.

During the traversal, the desired accuracy of the tasks' correspondent queries is updated to match the accuracy the task can provide with its CPU share. The available CPU of the TM is updated accordingly (line 8-11).

3.5.3 Computation of Non-Drop Probabilities

Finally, it only remains to specify how the non-drop probabilities for each application's stream is computed. This is done by using both Equations 3.15 and 3.16, which are responsible by defining how many events should be dropped and where they should be dropped in the DAG, based on the desired accuracy for each query in an application.

The first equation is used to propagate the dropping probabilities upstream. This will allow the system to drop events as early (upstream) as possible in the DAG. Implementation wise, this is done with a sink to source traversal of the DAG. Once done, the second equation is used to compute the $d(t, t')$ values, given the already provided accuracy and the desired accuracy at the downstream task t', $desired(t')$. This second step is done by traversing the DAG from the source nodes to the sink nodes.

Note that in this last equation, the non-drop probability for the load shedder located right after the first subtask of each source task, is also calculated. This is done by the first condition of the equation, that specifies this non-drop probability as the desired accuracy of the source task.

$$desired(t) = \max_{t' \in downstream(t)} desired(t') \tag{3.15}$$

$$d(t, t') = \begin{cases} desired(t'), & \text{if } t \text{ is an external datasource} \\ \frac{desired(t')}{desired(t)}, & \text{otherwise} \end{cases} \tag{3.16}$$

Figure 3.3 provides a visual example of these equations being used to compute the non-drop probabilities of the load shedders in an application with two sinks, A and B. It starts with the desired accuracies for each query of the application (1), which are passed as an argument, $desired(A) = 80\%$ and $desired(B) = 40\%$. The desired accuracies are then propagated upstream (2) using Equation 3.15, that selects the desired accuracy of a task as the maximum desired accuracy of its downstream tasks. The non-drop probabilities are then calculated for each load shedder (3), by using Equation 3.16. The algorithm starts by calculating the non-drop probabilities for the load shedders between the source tasks

and the external datasources, which yields an 80% non-drop probability for both load shedders. This guarantees the desired accuracy for the sink task A. As such, the non-drop probabilities of its upstream load shedders are all 100%. Regarding the sink task B, its desired accuracy is guaranteed by setting a non-drop probability of 50% between the sink task and its direct upstream task, which is the only stream that doesn't belong to the upstream of the task A. This example clearly shows that events are dropped as soon as possible in the application's DAG.

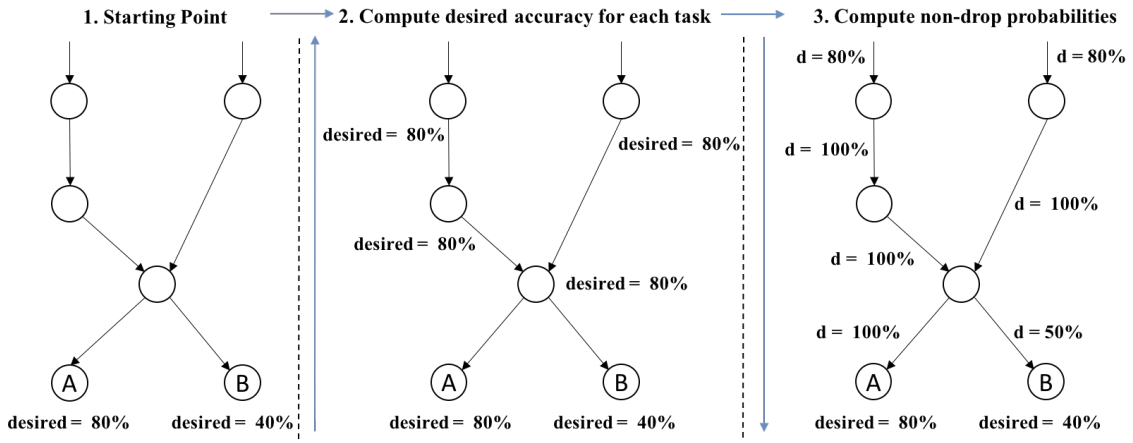

Figure 3.3: Example of the computation of the non-drop probabilities for an application with two sink tasks, A and B, having a desired accuracy of 80% and 40%, respectively.

3.6 Summary

The underlying mechanisms, algorithms and model that constitute the proposed solution were described in this chapter. The solution contains two main components, the *Task Scheduler* and the *QoD Controller*. The *Task Scheduler* schedules tasks to the machines with more CPU load available. The *QoD Controller* periodically monitors the execution of the applications, and eventually applies some corrections to make sure that the user defined restrictions are respected, i.e. priority and minimum accuracy for each query. On each *QoD Controller* cycle, it starts by guaranteeing that all applications' queries have enough CPU load for them to be processing at their minimum accuracy, otherwise, some tasks may be re-scheduled to release the required CPU load. Once this minimum accuracy is guaranteed, the remaining CPU load is distributed over the tasks, this time, by distributing it first to the higher priority tasks and only then to the lower priority ones.

The selection of tasks to be failed, in order to release the required CPU, is performed using a policy that balances a trade-off between not releasing more CPU than required and minimizing the priority of the tasks that are re-scheduled.

Regarding the *Load Shedding* mechanism, it drops events using a random drops semantic, param-

eterized by the *non-drop probability*. Load shedders are placed between every two task instances and right at the beginning of the source tasks' instances.

Several crucial concepts such as *accuracy*, *current accuracy* and *desired accuracy* were also introduced and defined in this chapter.

4

Implementation

Contents

4.1 Overview

This chapter's focus is on the implementation aspects of the algorithms and mechanisms of our solution, described in the previous chapter. As such, it will cover details regarding the scope of the changes performed during this implementation, covering both newly added Flink components and the already existent ones.

Our proof-of-concept (Stream Economics), which is going to be used during the evaluation, was implemented on top of Apache Flink 1.2 release. Thus, all the Flink internals that are mentioned along this chapter are specific to this release.

The final sections of this chapter target specific functionality that is required for the system to achieve its goal, which was not covered in the previous chapter since it's more implementation specific and not relevant for the core functionality of the solution.

4.2 Scope of Changes

To implement the proposed solution, several changes had to be performed in the target Flink release. The purpose for these changes can be break down into several requirements:

1. Users should be able to specify the priority and accuracy for each query in their applications. These restrictions should be available to the components that require them, for decision making.
2. Applications' data streams should have load shedders executing on the data stream's source task side. Source tasks have an additional load sheeder right after their first subtask. These load shedders should be configurable at runtime by the JM.
3. Slots should no longer work as a restriction to the amount of task instances a TM can execute. Instead, they can be considered to be dynamic (avoiding widespread changes on Flink).
4. The TMs should periodically send runtime metrics both related to the TMs machines and to the tasks being executed.
5. The *QoD Controller* should periodically tune the load shedders' parameters and trigger task re-schedules whenever necessary, according to the policy described in Chapter 3.
6. The task scheduling policy that was also proposed in the previous chapter, should be used for all incoming task scheduling requests.

Figures 4.1 and 4.2 will be used to support the description on how each of these requirements were implemented. Both figures, show the Flink components that were added or modified during the implementation, as well as the relations between them.

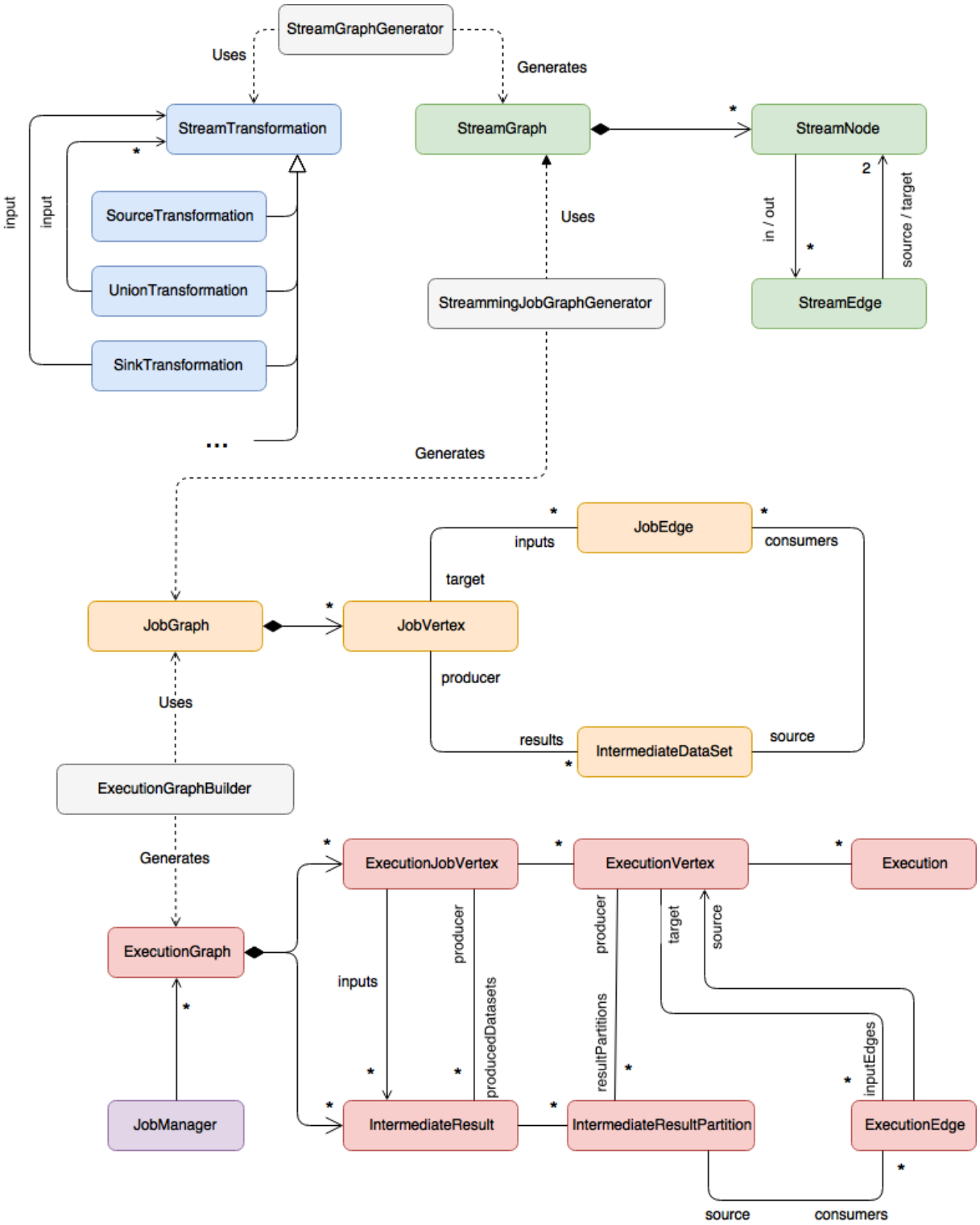

Figure 4.1: Diagram of the key Flink components regarding the representation of a stream processing application on the Job Manager's side.

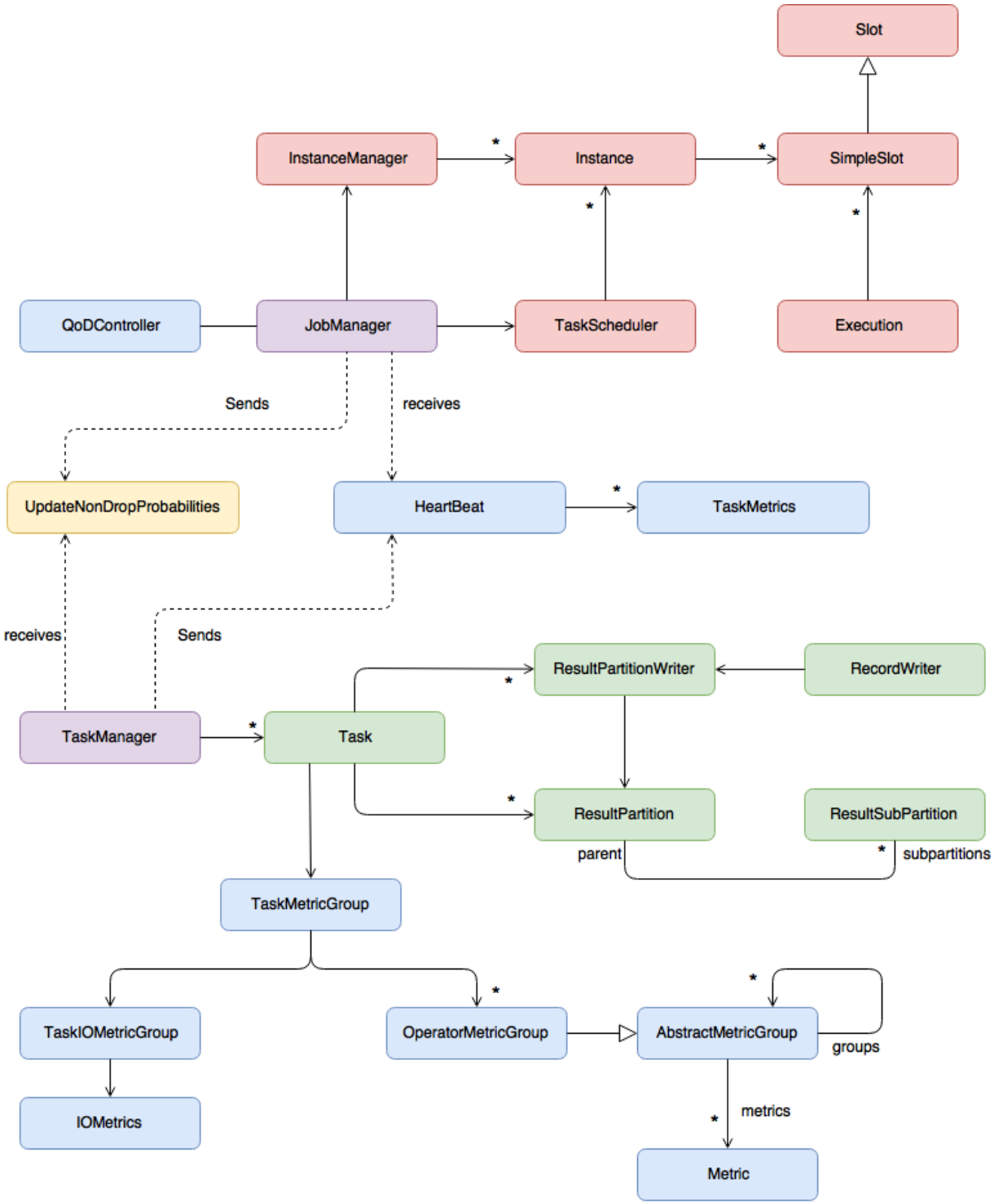

Figure 4.2: Diagram with the Flink components used for metrics reporting, task scheduling and task execution.

4.2.1 Stream Application's Stages

Before a stream processing application in Flink being deployed and actually executed, it goes through several stages, each using different abstractions to represent the application. Figure 4.1 presents these abstractions.

Stream Transformations: Correspond to the main entities used to specify the Flink's stream processing application (blue color components in the figure). A `StreamTransformation` has several specific implementations, each for a different pattern that can be found in a stream application's DAG, e.g. `SourceTranformation`, `UnionTranformation` and `SinkTranformation`. Each of these transformations are further extended into more specialized versions.

Since the system only requires users to specify the application's restrictions for each sink task (*requirement 1.*), the two required restrictions only need to be added to the `SinkTransformation`. Using the newly added `withPriority` and `withMinimumAccuracy` methods, users are able to specify the priority and minimum accuracy for their applications, respectively. Listing 4.1 provides an example where these two methods are being used to specify the restrictions for a word count stream processing application (lines 29-30).

Listing 4.1: Word count stream processing application in Flink with the minimum accuracy (0.7) and priority (5) restrictions being specified for a single query.

```
1  import java.util.Properties
2
3
4  import org.apache.flink.api.scala._
5  import org.apache.flink.streaming.connectors.kafka._
6  import org.apache.flink.streaming.util.serialization.SimpleStringSchema
7
8  object WordCount {
9    def main(args: Array[String]) {
10     val env = StreamExecutionEnvironment.getExecutionEnvironment
11     val props = new Properties {
12       setProperty("bootstrap.servers", "localhost:9092")
13     }
14
15     val source = env.addSource(
16       new FlinkKafkaConsumer010[String](
17         "wordcount",
18         new SimpleStringSchema(),
19         props
20       )
21     )
22
23     val words = source.flatMap(value => value.split("\\s+"))
24     val counts = words.map(value => (value,1))
25       .groupBy(0)
26       .sum(1)
27
28     counts.print()
29       .withMinimumAccuracy(0.7)
30       .withPriority(5)
31   }
32 }
```

Stream Graph: From a collection of `StreamTransformations`, the application is then transformed into a `StreamGraph` (green color components), by the `StreamGraphGenerator`. At this stage, transformations are grouped into `StreamNodes`, representing operator chains. `StreamNodes` are connected via

StreamEdges. The created StreamGraph is then sent to the JM for scheduling.

Job Graph: Represents a Flink dataflow program, conceptually more generic than a StreamGraph since it is also used to represent batch processing jobs. The JobGraph (yellow color components) is created by the JM from a StreamGraph, using the StreammingJobGraphGenerator. Each JobGraph contains several JobVertexes each producing zero or more IntermediateDataSets. Each IntermediateDataSet can be consumed by several JobVertexes, by connecting them using a JobEdge.

Execution Graph: The ExecutionGraph (red color components) contains all the information regarding the application's physical DAG. Meaning it can be fully mapped to the runtime structure of the application.

JobVertexes are mapped into ExecutionJobVertexes, which are then decomposed into several ExecutionVertexes, each representing a task instance that can be executed in a TM. While on the other hand, IntermediateDataSets are mapped into IntermediateResults, which are then decomposed into IntermediateResultPartition. Each IntermediateResultPartition represents a partition of the dataset, that can be consumed by several ExecutionVertexes. This relation is represented by the ExecutionEdges.

Current Flink implementation enforces that each IntermediateResult can only be consumed by a single ExecutionJobVertex. If two ExecutionJobVertexes consume the same events that are produced by the same ExecutionJobVertex, then the producer will have two IntermediateResults, one per consumer ExecutionJobVertex. Each IntermediateResult with a single IntermediateResultPartition. Therefore, each ExecutionVertex will also have the same amount of IntermediateResultPartitions as the amount of ExecutionJobVertexes consuming the IntermediateResults generated by the respective producer ExecutionJobVertex.

Additionally, each ExecutionVertex keeps track of several Executions, one being its current execution in a TM; and the others, being past executions of the ExecutionVertex, that are no longer being executed.

Throughout each of these stages, the priorities and minimum acceptable accuracies of the application's tasks are computed and forwarded to the next stage. Data structures were also added to quickly lookup the downstream sinks of each task in the application.

4.2.2 Task Scheduling

The JM, represented by the JobManager component in Figure 4.2, contains both an InstanceManager and a TaskScheduler. The first, is responsible for managing the TMs that registered to the JM, each represented as an Instance. While the second, corresponds to the component that is responsible

for scheduling a task to the available instances. This component was adapted to use the new task scheduling policy, that was defined in the previous chapter (*requirement 6.*).

Currently, each `Instance` in Flink provides a set of `Slots` for task scheduling. For the current Flink implementation there are two types of slots:

- `SimpleSlot`: slot that can be used for a single `Execution` of an `ExecutionVertex`
- `SharedSlot`: contains a collection of slots, either `SimpleSlots` or `SharedSlots`. This abstraction is used for the *Slot Sharing* feature implementation, which will no longer be required by the system.

To avoid changes to the current Flink implementation that would fall out of the scope of this work, the abstraction of `SimpleSlot` was kept. Nonetheless, there's no restriction to the amount of `SimpleSlots` that an `Instance` can provide (*requirement 3.*). In the new system, they can be simply seen as the representation of an unbounded amount of resources that can change at runtime. For each `ExecutionVertex` scheduling request that is sent to the `TaskScheduler`, it simply returns a `SimpleSlot` that will be used for the `Execution` of the `ExecutionVertex`.

4.2.3 Application Runtime

The TMs, represented by the `TaskManager` component in Figure 4.2, use their own abstractions to execute Flink applications. A `Task`, corresponds to the entity that actually executes the operations defined in an `ExecutionJobVertex` / `ExecutionVertex`, and as such, is responsible by managing the thread that will be used to perform the required work. A `Task`, on the TMs' side, is mapped to an `Execution`, on the JMs' side.

Each `Task` produces several `ResultPartitions`, that map to `IntermediateResultPartitions` on the JMs' side. `ResultPartitions` are decomposed into several `ResultSubPartitions`, each corresponding to an `ExecutionEdge` on the JMs side. Each `ResultSubPartitions` contains information on how to send the partition data to the actual consumer `Task`.

For each `ResultPartition` of a `Task`, there's a `ResultPartitionWriter` that is responsible by writing buffers of emitted records to the `ResultPartition`. A `RecordWriter` is used by the `Task` to write records and events, e.g. snapshot barriers used by the state snapshotting algorithm, to a specific `ResultSubPartition`. This is only performed for the records emitted by the last operator, of the task's operator chain, to its downstream tasks. The `RecordWriter` is also responsible for grouping records into buffers.

To perform load shedding on a per record basis (*requirement 2.*), load shedders were added to each `RecordWriter`. For each record that is emitted by the `Task`, the `RecordWriter` either allows the record to be sent downstream, by adding it to the current buffer that is being filled; or simply drops it.

As previously defined in Section 3.3.2, the non-drop probability for all the stream partitions connecting two Tasks that belong to the same ExecutionJobVertex, should be the same. As such, this probability is stored in the ResultPartition, thus becoming available for the load shedder in the associated RecordWriter.

4.2.4 Metrics System

Flink has its own abstractions to handle system metrics. For the scope of this work, the most relevant abstractions are presented in Figure 4.2. As presented, Flink has the concept of an AbstractMetricGroup that allows to create hierarchies of Metrics, i.e. each AbstractMetricGroup can have several sub-AbstractMetricGroups each containing a set of Metrics at the same level.

Each Task has an associated TaskMetricGroup, an implementation of an AbstractMetricGroup, responsible for centralizing all metrics associated with a single Task. Each TaskMetricGroup has two special metrics groups: the TaskIOMetricGroup, containing metrics such as the input and output record rates of the Task; and the OperatorMetricsGroups, each containing metrics relative to specific operators that are being executed by the Task.

Flink's TaskIOMetricGroup metrics are computed using a sliding time window, allowing their values to be more stable. This was adapted, in order for the sliding window's size to match the frequency at which the *QoD Controller* is triggered.

4.2.5 Communications between Job Managers and Task Managers

As mentioned earlier in Section 2.5.1, both the Flink JMs and TMs are implemented as Akka Actors, and as such they are able to exchange messages among themselves with the properties defined by the *Actor Model*. Meaning that: 1) An actor can send messages to other actors; 2) An actor processes received messages one at a time; 3) State is not shared among actors.

An example of such messages that are exchanged between the JMs and TMs are the Heartbeat messages, presented in Figure 4.2, that are periodically sent from the TMs to the JMs. The system, uses these Heartbeat messages to send runtime metrics of the TMs and their Tasks to the JMs, using a TaskMetrics per Task. Once the JM receives these metrics, it stores each of them in the correspondent ExecutionVertex, so that they become available to all the JM's components that require them (*requirement 4.*).

The UpdateNonDropProbabilites message is also introduced, allowing JMs' QoDController to send the non-drop probabilities to the ResultPartitions in a specific TM (*requirement 5.*). These non-drop probabilities are sent in a batch, i.e. each message can contain several of them.

Additionally, a special message was added, to periodically trigger the QoDController main cycle.

This message is sent from the JM to itself with a configurable frequency.

4.3 Missing Metrics

Although some of the runtime metrics required by the system are already provided by Flink, such as the input and output records rate per (sub)task; the CPU usage both by each task and per TM's process are not yet available in Flink.

As such, these new metrics were added during the implementation of our solution. The CPU load per TM process is fairly easy to obtain using the Java Management Extensions (JMX) provided interface. The CPU load per thread is more complex, since this is not directly provided by JMX, i.e. one must keep track of the CPU time of the thread, in order to compute this load. Just like in Flink's metrics system, these two metrics are computed by our solution also over a sliding time window.

4.4 Overcoming Flink task failing limitations

When a Flink's task instance is failed, for instance by the *QoD Controller*, all the tasks that belong to the same application are also marked as failed. Causing application's tasks to release all their resources and to be re-scheduled once again, not necessarily to the same TM.

This is not the intended behavior by the *QoD Controller*, it simply wants to re-schedule a specific task, while the remaining ones should keep executing on the same TMs, even if some recovery needs to be done. This is a problem, as it affects the optimality of the algorithms that are used to govern the system's decisions, since they depend on the ability of failing individual tasks without affecting others.

Note that there's currently no impediment on the Flink side for this to be possible, it is simply a not implemented feature, which also affects Flink's efficiency when recovering from application failures. At the moment that this document is being written, there's a *Flink Improvement Proposal* that aims at solving this limitation, the *FLIP-1: Fine-grained Recovery from Task Failures*[1].

Nonetheless, to overcome this restriction, every time a task is failed on purpose, the *QoD Controller* flags it. On the *Task Scheduler* side, if a re-scheduling request contains a flagged task instance, then it schedules the task to a different TM from the one it was running previously. Otherwise, it schedules the task to the TM it was previously running on. By following this approach, the optimality of the algorithms used by the proposed solution is guaranteed.

[1] "Flink Fine-grained Recovery", https://issues.apache.org/jira/browse/FLINK-4256, (June 22, 2018)

4.5 Load Shedding on the Source Tasks

As pointed earlier, in Chapter 3, load shedding is also possible right after events are fetched from the external datasource and before being injected into the applications' downstream. This, to drop input events as soon as possible in the application's DAG.

Given a new event, the task instance's thread executes what is called an `OperatorChain`. The `OperatorChain`, abstracts a set of chained operators that are sequentially executed by the thread. Each operator performs the operation on top of the event it receives, and emits the result to the next operator in the chain. In the end of the chain, if the task is not a sink, then the operator will emit its results to the next task, that can either be running in the same TM or in a different one.

Therefore, a load shedder was placed between the first operator and the second one in every source tasks with more than a single operator, since the first operator is the one that performs the integration with the external datasource. This load shedder is similar to the others and as such, its non-drop probability is also configurable at runtime.

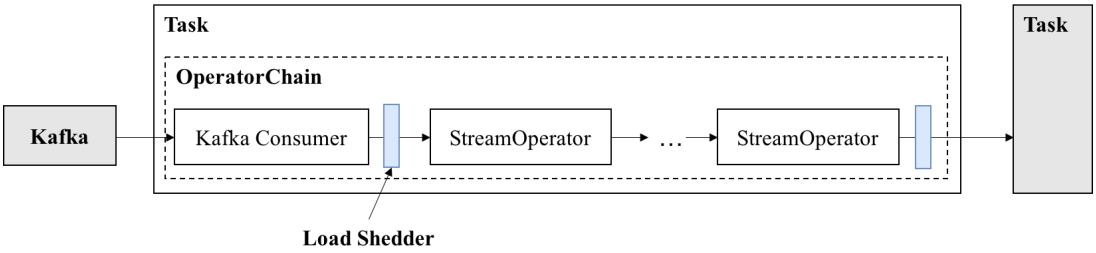

Figure 4.3: Task operator's chain at runtime with a Kafka consumer source operator and a downstream task.

Figure 4.3 shows a diagram with these concepts. In this figure, the source operator corresponds to an Apache Kafka [2] consumer.

4.6 Accuracy of Source Tasks and Integration with Apache Kafka

Computing the accuracy of source tasks in the applications is also a challenging problem. This happens because to compute this accuracy, one needs to know both the throughput at which events are being sent to the external datasources that are consumed by the applications; and the rate at which the source tasks are consuming events. This, while taking into consideration that events may be dropped right after being fetched by the source subtask, as seen in the previous section.

Obtaining these required metrics is not a trivial task. For this reason, the scope of the proof-of-concept was reduced to only support this kind of integration with Apache Kafka. Otherwise, the system

[2] "Apache Kafka", https://kafka.apache.org, (June 22, 2018)

assumes that these source tasks are always coping with the event injection rate of the external data-source, which can be a problem if this is not true.

Apache Kafka Background *Topics* are the main abstraction of Kafka, that represent queues of messages. These queues have both *producers*, that send events to them; and *consumers*, that consume these messages. For high parallelism, Kafka topics are partitioned into several *partitions*, that are distributed and replicated over the available machines in the Kafka cluster, named the *Kafka Brokers*. Consumers can also be grouped into *Consumer Groups*, that together consume a topic in parallel. Each topic's partition is only consumed by a single consumer within each consumer group; while each consumer can be consuming several partitions of the same topic. As such, partitions define the maximum parallelism at which a consumer application can consume data from a given topic. Figure 5.3 presents an overview of Kafka's architecture.

Kafka topic's consumption is pull based, allowing each consumer to consume events at its own rate. For performance reasons, events are fetched from a partition in batches. Additionally, each event has its own *offset*, that identifies the event in the partition it belongs to.

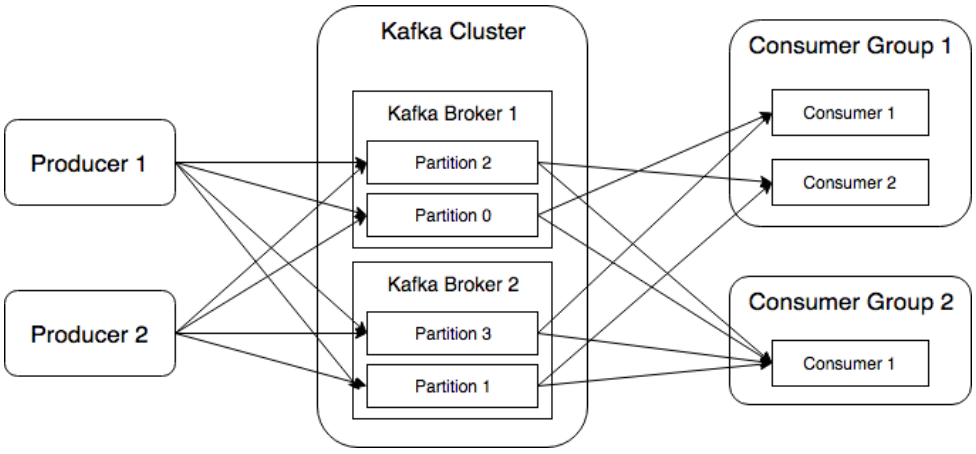

Figure 4.4: Overview of Apache Kafka's architecture.

Kafka Consumer Metrics Kafka consumers expose some internal metrics during runtime. In the specific case of Flink's integration with Kafka datasources, these metrics are already integrated with Flink's metrics system, and therefore accessible from the TMs. These Kafka metrics are updated every time a batch of records is fetched by the Kafka consumer operator in Flink.

Flink's Kafka consumer operator executes in a separated thread, independent from the main one, that is used by the task instance. This operator, continuously fetches batches of events from the partitions it is consuming from. Once a batch is received, the consumer injects it into the task's operators chain. The synchronization between both threads is performed by a size one blocking queue.

This yields an issue, because if this synchronization data structure is full, then the Kafka consumer will be blocked, causing Kafka metrics not to be updated. As such, the Kafka consumers should be configured to fetch records from the data source in batches that are sized to the application's maximum throughput. This can be done by using either the *max.poll.records* or *max.partition.fetch.bytes* property, where the first limits the maximum amount of records fetched in a batch, and the second limits the size of the batch.

Kafka Input Rate Estimation The input rate of the Kafka consumed topics is not provided as a consumer metric. As such, it must be estimated by the system. Which is done using the following metrics:

- ***records-lag-max*:** the lag of a partition corresponds to the difference between the maximum and minimum offsets in a partition. This metric provides the maximum lag among the partitions consumed by the Kafka consumer. Ideally, this value should be zero.
- ***records-consumed-rate*:** consume rate of the Kafka consumer instance.
- ***assigned-partitions*:** given a Kafka consumer instance, corresponds to the amount of partitions it is consuming from.

By computing the variation of the first metric, it is possible to estimate the input and output rates' difference for the Kafka partition with the highest lag. Therefore, given the output rate and the lag variation of this partition, it is possible to compute its input rate. Assuming that the workload is fairly balanced across the topic's partitions, the output rate of the partition can be estimated by dividing the second metric by the third one. Using this result and adding it to the lag variation within the partition, yields the partition's input rate estimation. Similarly, this can be extended to the rest of the topic's partitions.

Source Task's Input Rate Regarding the input rate of the source tasks, it needs to take into consideration that incoming events may be dropped by the load shedders placed between the first and second subtasks of the source task's instances. Thus, if the source task has a second operator, then its input rate is used as the input rate of the source's task instance. If the task has a single operator, its output rate after the load shedder can be used instead. This last situation should be highly unusual.

4.7 Tasks' Warm-up Interval

Every time a task instance is scheduled to a TM, it takes a while for its CPU load to stabilize and for its consumption rate to match the one that would be expected given the resources that are available for the task execution. During this time, any decision performed by the system could be precipitated and

even lead to unnecessary re-scheduling of tasks. As such, whenever a task instance is (re-)scheduled to a TM, the system provides some time for it to stabilize and start coping with its input workload. Which is named the tasks' *warm-up interval*.

The warm-up interval of a task instance, starts when the task is scheduled to a TM; and finishes when the task's current accuracy achieves its minimum acceptable accuracy. Though it is expected for this warm-up period to be short, a configurable timeout is added to avoid situations when the task is infinitely stuck in a warm-up state.

On the *QoD Controller* side, whenever a TM has a task instance in a warm-up state, all the tasks whose instances are being executed by the TM will be processing at their minimum required accuracy, allowing the task in warm-up state to quickly exit this state.

4.8 Summary

This chapter covered the implementation aspects of the proposed solution. The different requirements to fully implement its algorithms and mechanisms were specified. Followed by a walkthrough of all the components that were involved in the implementation (both from Flink and from our solution), and a description of the changes that were performed to fulfill the identified requirements.

The implementation changes covered: the Flink components used to represent the different phases of the stream applications, between being specified by the user and being deployed to the TMs; the Flink's task scheduling components and the ones responsible to keep track of the existent TMs; the runtime representation of Flink tasks in the TMs; Flink's metrics system; and the different communications between JMs and TMs.

Some setbacks were also highlighted, together with an explanation on how the implementation of our solution overcomes them. Namely: missing metrics that are required by the model defined in Chapter 3; Flink limitations to efficiently handle task failures without re-scheduling the whole application; challenges of implementing the *Load Shedding* mechanism, more precisely, on detecting bottlenecks in source tasks and on performing load shedding on these tasks; and the required *warm-up interval* for tasks to start coping with their workload when they are re-scheduled.

5

Evaluation

Contents

5.1 Overview

With both the architecture and implementation of the system specified, it only remains to assess if the proposed solution is actually able to solve the problem that is the subject of this work. This chapter is focused precisely on this evaluation. This is performed by running and analyzing a set of scenarios where stream processing applications are executed in a cluster, while their workload dynamically changes over time. These scenarios are tailored to show situations where current solutions would fall either on the resource underutilization or resource starvation problems.

This chapter also focuses on evaluating the performance impact of the solution both at the application level and at the JM level. This allows to assess whether it's feasible to use the developed solution in large stream processing environments where hundreds or even thousands or tasks execute in parallel.

This chapter starts by describing the goals of this evaluation, the benchmarking environment and stream processing applications that will be used. It will then go through each scenario, presenting the obtained results and a brief analysis of them. It finishes with a critical summary of the gathered results based on the previously established evaluation criteria.

5.2 Evaluation Criteria

Before performing the evaluation, the evaluation criteria that the system must meet, in order to consider that it successfully solves the problem that it was built to address, must be defined:

1. The solution should always respect the restrictions provided by the user. The applications' accuracy should not go below the defined threshold; and the tasks' priority must be respected, accordingly with the semantics established in the previous chapters.

2. It must efficiently react to application' workload changes. For instance, when the application workload increases it should reduce the application accuracy or re-schedule tasks to other TMs, avoiding events to be waiting for the application to process them. When the workload reduces, the applications' accuracy should increase up to its maximum.

3. Its decisions must guide the system towards a state where tasks are optimally distributed to maximize their throughput and increase resource efficiency. Showing the ability to solve both situations of resource underutilization and resource starvation. Obviously, taking into account that the system can reduce the applications' accuracy to delay eventual task re-schedules.

4. The solution must be able to scale for scenarios where several tasks run in parallel.

The benchmarking scenarios that will be presented in the following sections of this chapter are designed to demonstrate the ability of the solution to meet these goals. At the end of this chapter, the

criteria enumerated is used to determine how far the solution is from achieving its objective.

5.3 Benchmarking Environment

The benchmarking environment used to execute the evaluation scenarios consists in four virtual machines running on top of an host machine with a quad-core 2,4 GHz Intel Core i7. Each virtual machine running Ubuntu 16.04 LTS operative system, with a single dedicated core and 3 GB memory. These virtual machines execute in the same host, since the objective is to estimate the solution's overhead without this overhead being masked by the network.

Regarding the roles of each machine, three of them were responsible to run the Flink cluster. One machine executing the JM, and the two others running the two TMs. The Flink setup uses the default configuration settings, with the exception of the configurations required to setup communications between the machines. Additionally, all Flink nodes were configured to use a file metrics reporter. Allowing to gather and later analyze all the metrics reported by the Flink cluster. The *QoD Controller* was configured to be executed with a 5 seconds frequency, allowing a fair response time for workload changes.

The fourth machine runs Apache Kafka and the data injector for the Kafka topics. The Kafka topics are used as input datasources for the applications that are executed in the evaluation scenarios. The Apache Kafka 0.10.2.1 release was used, together with Apache Zookeeper[1] 3.4.11. Zookeeper is mandatory, since the Kafka consumers use it for checkpointing the latest consumed offset from each consumed partition. In terms of configurations, the defaults were used for both systems.

To inject data to the consumed Kafka topics and to coordinate the execution of the evaluation scenarios, a new configurable data injector for Flink applications was developed. This injector receives a JSON specification of the scenario, with information regarding: which applications should be used, when they should be started and stopped, and how the data injection rate changes for each topic that is consumed by the applications. For a given Kafka topic, the injected data is obtained from a given input file, by picking random lines from it. Underlying, the injector uses Gatling [2] to control the actual data injection.

5.4 Test Suite and Datasets

Two applications were created for the evaluation scenarios. Both designed to show specific details of the system that are required to meet the previously specified goals, hence behaving as micro-benchmarks of the metrics critical to the system. While at the same time, being simple enough to make the obtained results easily understandable. Configurations such as the priority and accuracy of each applications' query will be described individually for each scenario, whenever relevant.

[1] "Apache Zookeeper", https://zookeeper.apache.org/, (June 22, 2018)
[2] "Gatling – Load Testing", http://gatling.io, (June 22, 2018)

5.4.1 Taxi Drives Application

The Taxi Drives (TD) application consumes data from two distinct Kafka topics, both providing events regarding taxi drives. These events are generated based on the NY TLC Trip Record dataset[3].

Incoming events are parsed by two application' source tasks, one for each Kafka datasource; and only then, the two parsed streams are blended and consumed by two distinct sink tasks, each computing some useful metrics regarding the taxi trips. The logical DAG of this application is presented in Figure 5.1, DAG (1). Where the task $TDSnk1$ of this application is configured to have a parallelism of 2, while the remaining tasks are executed with a parallelism of 1. Additionally, the $TDSnk2$ task is more CPU intensive than the remaining tasks.

Although simple, this application will allow us to analyze how load shedding is propagated in complex application DAGs, since it not only has a join and fork pattern, but also because one of the sink tasks has two executing instances.

5.4.2 K-Nearest Neighbors Application

The second application consists in an implementation of the K-Nearest Neighbors (KNN) algorithm for a three dimensional problem that classifies incoming events based on a given dataset [41]. These incoming events were randomly generated three dimensional tuples. As presented in the DAG (2) in Figure 5.1, this application has a single task. This allows to test how the system will be able to take into account metrics provided by the Kafka consumer. For instance, whether the system is able to trigger load shedding based on the data injection rate to the input Kafka topic; or not, causing the Kafka topic lag to increase once the application reaches its bottleneck.

Regarding the parallelism of this application, it will be used both with a parallelism of 1 and 2, depending on the evaluation scenario. In both cases, the input Kafka topic will be configured with two partitions, allowing to validate that the system is able to handle situations when a single task instance consumes data from several partitions of the same Kafka topic.

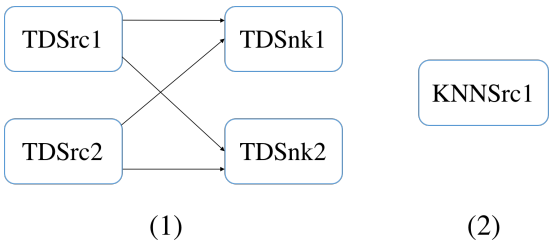

Figure 5.1: Logical DAGs of the Taxi Drives (1) and KNN (2) applications.

[3]"NY TLC – TLC Trip Record Data", http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml, (June 22, 2018)

5.5 Scenario 1: Tasks With Same Priorities

The first scenario consists in the execution of the TD application where both sink tasks are configured to have a similar priority, and a minimum acceptable accuracy of 40% and 60% for $TDSnk1$ and $TDSnk2$, respectively. Figure 5.2 shows the obtained metrics from the execution.

The starting point for this execution consists in a cluster with a single TM, $TM1$, to which all the TD application tasks are scheduled, as it is the only available machine. Once the application is running, a second TM, $TM2$, is added to the cluster (A), without any tasks being immediately scheduled to it.

As the workload of the application increases to levels where the application stops coping with it, the load shedding mechanism gets triggered, causing the application' accuracies to decrease. Instant (A), shows one of those situations, where load shedding workload allows the application to keep up, without having to re-schedule some of the application tasks. As the workload decreases, the application accuracy increases, up to the moment where load shedding is no longer required (B).

The effect of all tasks having equal priority is clear when looking at the application tasks' accuracy, since load shedding is applied to both sinks without any of them reaching their accuracy threshold. Instant (A) is an example of such situations.

In the instants (C) and (D) the $QoDController$ re-schedules some tasks to the later added TM, as load shedding stops being able to release the required resources without compromising the application's restrictions. In (C) the first instance of the $TDSnk1$ task was re-scheduled, while in (D) it was the second instance of the same task. The reason why both $TDSnk1$ instances where re-scheduled rather than $TDSnk2$, is because their re-scheduling was the one that minimized the amount of released CPU, while still releasing enough for the remaining tasks in $TM1$ to keep up with their workload.

After each of these re-schedules, the maximum application throughput increases. Note that after (D), the system achieves an optimal task distribution that maximizes both the throughput and accuracy of the application, and reduces resource wastage. Additionally, the effects of the tasks' warmup period in their accuracy are also visible every time a task is re-scheduled. This warmup phase took no longer than 1 minute in both re-schedules, showing that tasks can quickly exit from this state.

Looking at the CPU usage of $TM1$, it's noticeable that a considerable amount of CPU load is not being tracked by the system, even though only the Flink TM is being executed in the machine. This happens because Flink tasks can create more threads rather than their main one. This is common in several operators, such as window operators or even in Kafka source operators, where a second thread fetches events from Kafka and injects them into the processing pipeline. A better mechanism should be provided by Flink to control all the threads created from a single task instance, as this is a loophole in Flink's resource management model.

During the execution, it was also observed an average difference of 0.6% and 0.5%, between $currAc(TDSrc1)$ and $ac(TDSrc1)$, and between $currAc(TDSrc2)$, respectively.

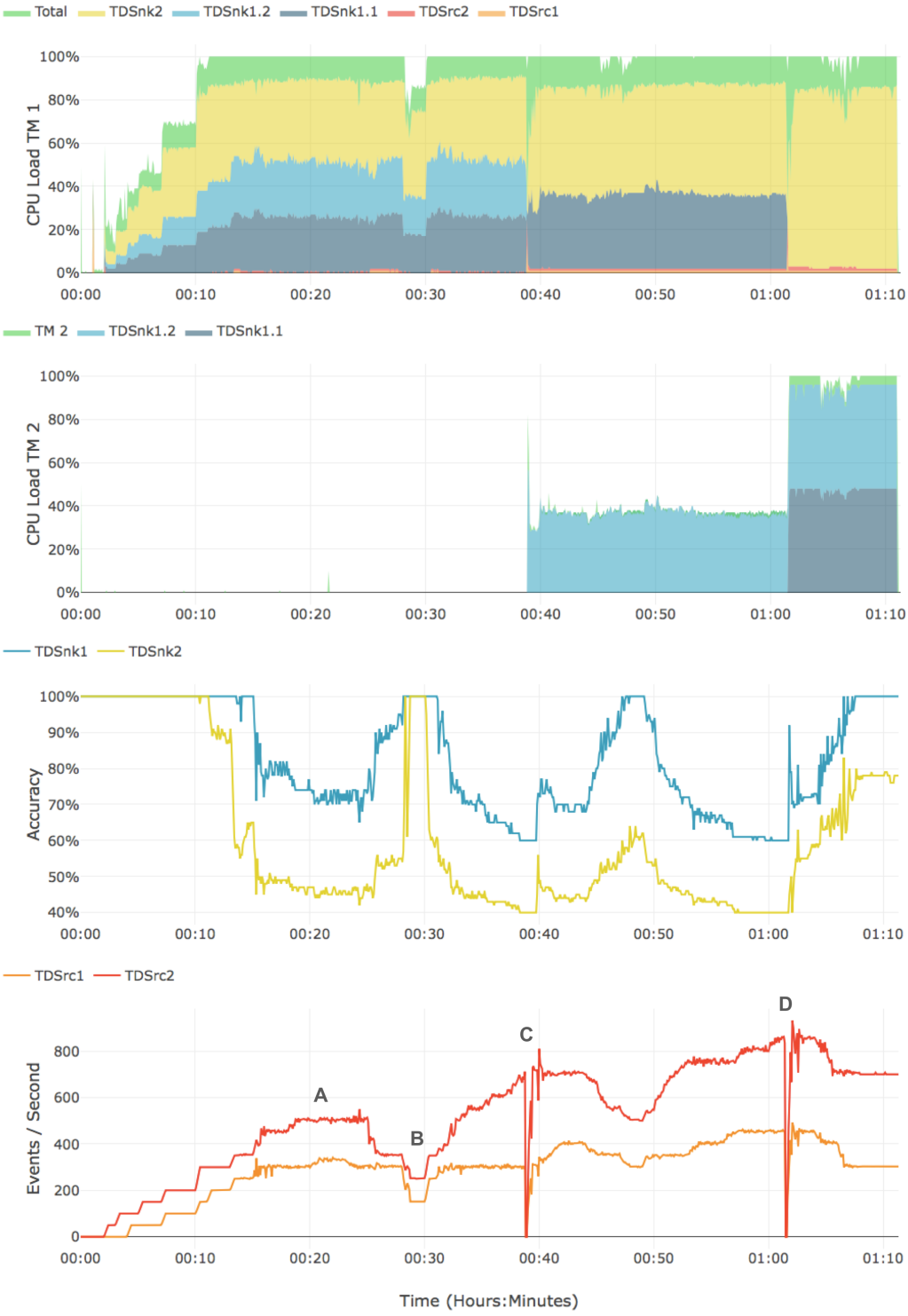

Figure 5.2: Execution of the Taxi Drives application where all sinks have the same priority. First two charts show the CPU load of each TM and their executing tasks. Third and fourth charts show the application sinks' accuracy and the application throughput for each input Kafka topic, respectively.

5.6 Scenario 2: Tasks With Different Priorities

This scenario shows an execution with both TD and KNN applications, where their tasks have different priorities, both between the two applications and between the two queries in the TD application. This way, this scenario allows to evaluate how different priorities are taken into account by the system.

The test starts with both TMs available for executing tasks, and with the TD application' tasks being scheduled among them following the established scheduling policy. The KNN application, which is configured with a parallelism of 1, is only added later to the system (A). Since this scheduling occurs when the TD application is already processing events, the KNN task is scheduled to $TM2$, which is the TM with the lowest CPU usage. Once the KNN application is scheduled, the effect of its warmup state is visible on the accuracy of the other tasks whose instances are also executing in TM (A).

In terms of priorities, it's visible that the $TDSnk1$ task has an higher priority than the $KNNSrc1$ task. For instance, when the KNN application' workload increases (B), its accuracy drops, while the accuracy of the $TDSnk1$ task (which is being executed in the same TM) remains unchanged. A similar behavior appears later, when the workload of the TD application increases (C), causing the accuracy of the KNN application' to decrease in order for $TDSnk1$ to cope with its incoming workload.

On the other hand, *TDSnk1* has a lower priority than the *TDSnk2* task. As the workload of the TD application increases, the *TDSnk1* instances' accuracy drops before load shedding gets triggered on *TDSnk2*, eventually re-scheduling the first *TDSnk1* task instance (C) in order to allow *TDSnk2* to keep up with its workload.

Once again this scenario shows the ability of the system to adapt task scheduling based on the changing runtime requirements of application tasks. By doing so, it was able to improve the applications' throughput, as can be seen in (C), after re-scheduling the task instance *TDSnk1.1*. Flink 1.2 is incapable of performing this type of adaptation, clearly showing a situation where the proposed system performs better in terms of resource efficiency.

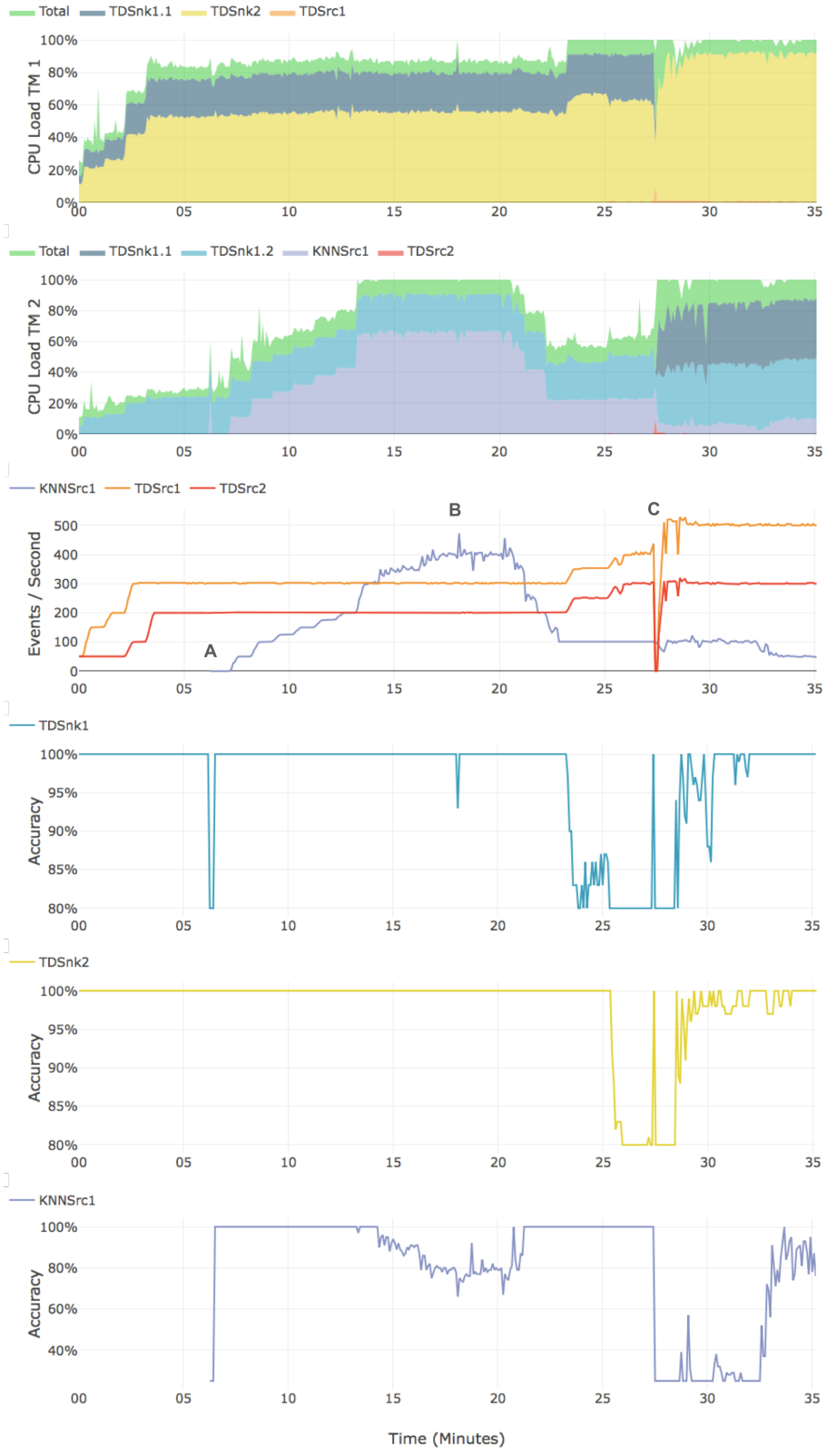

Figure 5.3: Taxi Drives and KNN applications' execution, having tasks with different priorities. First two charts show the CPU load in both TMs. Third chart shows the applications' throughput for each input Kafka topic. The accuracy for each sink task is presented in the remaining charts.

5.7 Scenario 3: Apache Kafka Integration

The third scenario focus on evaluating the integration of the mechanism with the metrics provided by Kafka consumers, as described previously in Section 4.6. In this evaluation scenario, the KNN application is executed with a parallelism of 2, where both task instances consume from different partitions of the input Kafka topic. Both TMs were used, each executing one of the task instances, due to the task scheduling policy. Figure 5.4 presents the obtained results.

In terms of the load shedding mechanism, the results show that the system is able to properly adapt the application' accuracy based on the incoming workload (first and second charts in the figure), either when it increases (A) or decreases (B). While the load shedding mechanism was active, it was measured a Pearson correlation of 0.95% between the accuracy and the estimated Kafka input rate.

The second chart in Figure 5.4, shows a comparison between the actual and estimated injection rate to the consumed Kafka topic, where the estimated injection rate is the one computed based on the method previously described in Section 4.6. Results show that even when load shedding is active, the system is able to estimate the actual injection rate with a mean error of 0.4% $\pm$ 6.2% and 2.8% error in percentile 90%. A slight delay between the estimated value and the actual injection rate, can also be noticed, which explains the high standard deviation. This happens due to delays on updating the Kafka consumer's metrics.

Regarding Kafka topic partitions' lag, some spikes can also be observed, which appear due to the following two reasons:

1. The definition used for *current accuracy*, in Section 4.2, aims at reducing the difference between the producer and consumer throughput, not at minimizing the amount of events that are in queue to be processed by a task. Other definitions for the *current accuracy* could be used to address this problem.

2. A misalignment between the instant when the injection rate suddenly changes and the instant when the next *QoD Controller* cycle is triggered, causing events to wait in queue to be processed, since the system can't react immediately after the workload change.

Once again, our load shedding mechanism proves its value, as it prevents the Kafka topic lag from increasing as the application reaches its bottleneck. The same wouldn't happen in Flink 1.2, where its back-pressure mechanism wouldn't be able to keep Kafka lag near zero. This will be observed in the benchmark presented in the next scenario.

Figure 5.4: Execution of the KNN application as the event injection rate changes. First chart shows its accuracy; second chart a comparison between the injection rate and estimated injection rate to the Kafka topic; and third chart, the lag of each Kafka topic partition.

5.8 Scenario 4: Impact on Applications' Performance

This fourth scenario aims at understanding the impact of the proposed mechanism on the applications' performance. This was done by executing the TD application and monitoring both its latency and throughput as its workload increases. This scenario was executed both in Flink 1.2 and in our modified version of it. To allow a fair comparison, a single TM was used, guaranteeing that tasks are scheduled in the same way (since there is only one TM).

The obtained results are presented in Figure 5.5, which shows the Kafka topics' lag, the application throughput at each input Kafka topic and the application latencies, over time. The measured latency corresponds to the latency inside the application, i.e. it doesn't take into account the time each event spends in queue, waiting to be consumed by the application

As observed, once the application bottleneck is reached (instants A and B), the different behavior of both solutions becomes clear. In Flink 1.2, the application throughput stops keeping up with the increasing injection rate. Causing the Kafka topics' lag to continuously increase, as Flink's back pressure mechanism gets triggered. The same doesn't happen when using the developed solution, thanks to the load shedding mechanism. In this case, the application is able to cope with the injection rate, keeping Kafka topics' lag almost null.

In case of the modified Flink version, it's possible to see interesting patterns regarding the application's latency. As presented, there are several latency spikes aligned with changes in the injection rate. Nonetheless, these latency spikes eventually converge to latencies below the ones observed in Flink 1.2 (C), though still slightly above the ones observed before the application reaches its bottleneck. The reason this happens is the same pointed in Section 5.7, though this time, events are queuing up in the application's internal buffers between the producer and consumer tasks.

Regarding latency in Flink 1.2 execution, results shows that it increases once the application bottleneck is reached (A). This happens because without load shedding, the intermediate application buffers will remain full. Note that the time events spent in the Kafka topic should also be added to the measured latency; thus, the overall latency is actually getting worst as the Kafka topic lag increases.

These results show a clear advantage of the proposed solution compared to Flink 1.2, as it allows applications to maintain higher throughput and lower latencies, even though optimizing the latency is not entirely the focus of the implemented model.

As a side note, the reason for the misalignment between instants A and B happens both due to the different task parallelism, and because the union operator implementation prioritizes the consumption of one of its input streams over the other.

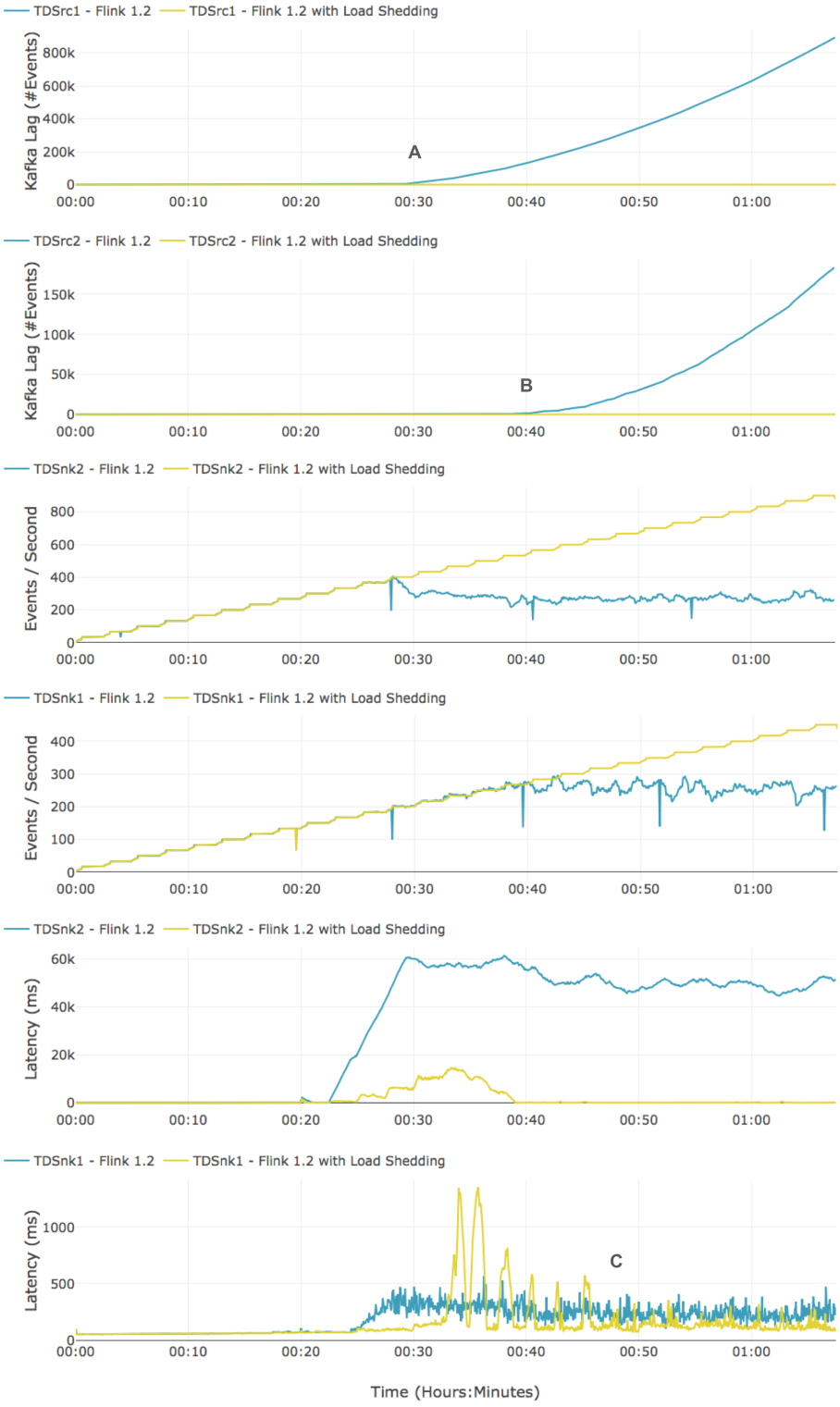

Figure 5.5: Lag, consumption throughput and latency of the Taxi Drives application as the injection rate increases, both when using Flink 1.2 and in using the proposed solution.

5.9 Scenario 5: Scalability and Job Manager Performance Impact

The fifth and final scenario focuses on understanding the scalability of the decision making mechanism as the amount of tasks in the system increases. The results presented in both Figures 5.6 and 5.7 were used to perform this assessment. The first figure shows how the CPU load of the JM increases as the amount of tasks being executed increases. Figure 5.7 shows how the execution time of the *QoD Controller* monitoring cycle increases with the amount of executing tasks. Both benchmarks were performed using two TMs, with the applications' workload dynamically changing. Causing load shedding to be triggered, but without any task re-scheduling.

The obtained results show that even with 2000 executing tasks, which is already a very high amount of tasks, the JM's CPU load is below 15% in the third quartile. Meanwhile, the execution time of the *QoD Controller* is below 115 ms in the second quartile, for the same amount of tasks. Compared with Flink 1.2 (first box plot in Figure 5.6) the solution shows to be more CPU intensive, which was an already expected outcome.

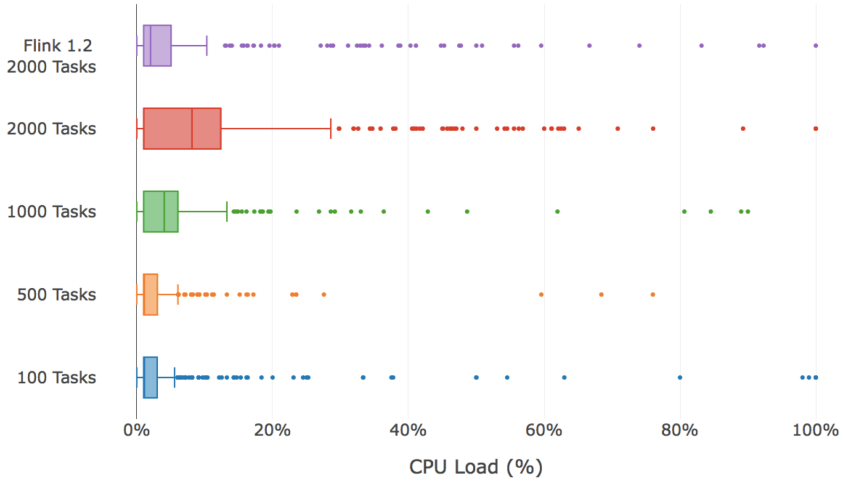

Figure 5.6: CPU Load of the JM for different amounts of executing tasks, with two TMs, and comparison with Flink 1.2 release.

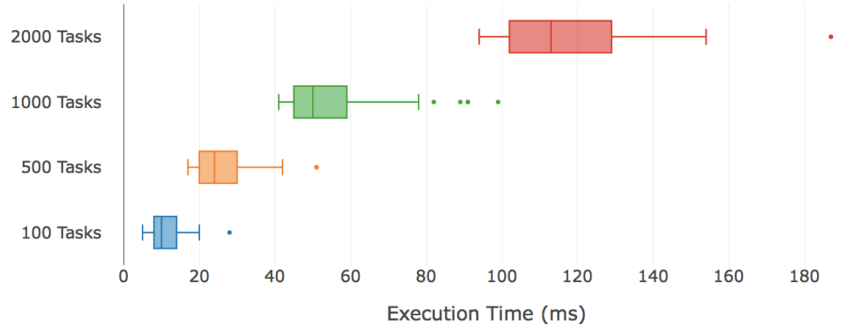

Figure 5.7: Execution time of the *QoD Controller*, given a fixed amount of executing tasks and using two TMs.

5.10 Summary

Overall, the obtained results from the different scenarios were promising. They are able to show that the system is able to adapt both load shedding and task scheduling to the dynamic workload of the applications. Thus, causing the system to eventually converge to a state of optimal task distribution, which can change over time due to the dynamic nature of the applications' workload.

At the same time, it was possible to observe situations where load shedding worked as a buffer, preventing the system from taking precipitated task re-scheduling decisions. Even when tasks were re-scheduled, their warmup interval showed to be acceptable.

The integration with the external datasource metrics, was also validated, proving that the system can effectively detect bottlenecks in the applications' source tasks.

In terms of performance, the system was able not only to improve the applications' throughput, but also their latencies. Thus, showing great improvements when compared with Flink's current approach, obviously at the cost of lowering applications' accuracy. The solution also proved to be scalable, which shows that it can be feasible for real world applications, where the provided semantics are acceptable.

Regarding the previously established evaluation criteria, it's possible to conclude that the evaluation results proved that the system was able to fulfill them. Nonetheless, the obtained results also show room for improvements, both in terms of the defined decision making model; and regarding the way runtime metrics are being obtained / calculated.

6

Conclusion

Contents

6.1 Conclusions and Final Remarks

This dissertation explored the problem of efficient resource management in large scale clusters running several workload dynamic and resource heterogeneous applications, either facing underutilization or resource starvation situations. While at the same time exploring how approximate computing techniques can be used to address this problem.

A novel resource management model was proposed, that specifically targets this problem under the domain of distributed stream processing systems. The solution leverages a resource usage aware task scheduler together with two mechanisms to handle task overallocation situations, i.e. task re-scheduling and load shedding. The system exposes a simple, yet meaningful, interface for the application developers, that allows them to specify the importance of their queries and their acceptable accuracy. The model governs the system decisions at runtime by respecting the user defined application requirements, while following the established semantics for this decision making mechanism. The proposed model was developed and evaluated on top of Apache Flink, a modern and state of the art stream processing system.

Defining the different followed semantics in a way that they are useful for most stream processing use cases, such as identifying a good definition for accuracy and specifying how the different application priorities should be taken into consideration; turned out to be a non-trivial problem. Additionally, guaranteeing the computational efficiency of the algorithms that perform this decision making was also a challenging task. Because in order to avoid decision backtracking, decisions need to be performed by the system in the right order, since decisions taken for one task instance may also affect the others.

The developed load shedding mechanism for Flink should also be highlighted, since it not only takes into consideration bottleneck situations in internal application tasks, but also on source tasks, based on metrics from the external datasources. This mechanism is also able to drop incoming workload as soon as possible in the applications' DAGs, minimizing the processing of events that will be surely dropped by downstream tasks, while taking into account the different desired accuracies of the applications' queries.

Implementation wise, several limitations were identified, such as the lack of ability to re-schedule a single application task in Flink without having to re-schedule the whole application. Or even missing datasource metrics that could be useful to identify source tasks' bottlenecks.

Performed benchmarks show that, compared to current stream processing solutions, the overall system is able to better explore the available resources, as well as improve the throughput and latency of the applications in bottleneck situations. The system is able to dynamically change tasks' allocation to resources, as well as their accuracy, based on the runtime applications' needs. Allowing application queries to keep outputting fresh results, albeit at the cost of accuracy drops. The underlying algorithms also proved to be scalable, allowing the system to manage the execution of hundreds or even thousands of parallel tasks.

6.2 Future Work

Besides overcoming the already identified limitations of the system and making it more production ready, future work should also focus on exploring other dimensions of the proposed solution.

It would be interesting to integrate an auto-scaling mechanism with the model, allowing it to also change the applications' parallelism, when the performance of a specific task can no longer be improved without increasing its parallelism. Or even to adapt the amount of machines in the cluster, allowing the system to add more resources every time task re-scheduling and load shedding can't overcome resource starvation situations. This, while at the same time, reducing the amount of machines whenever these are not required, causing more tasks to be scheduled to the same machine.

In terms of the load shedding mechanism, other approaches for selecting the events that should be dropped can be explored, such as semantic load shedding. Or even to allow users to specify different load shedding semantics per application.

In fact, generalizing the proposed model in order to allow specific parts of it to be customizable either by the system administrator or by the application developers (on a per application granularity), could be another thread of exploration. For instance, enabling the user to provide: its own implementation of the *current accuracy* function, in order to improve the latency of its application and not just the throughput; a method to predict the obtained accuracy given a CPU amount; or even a completely different definition of accuracy. As well as to modify the semantics followed by core components such as the *QoD Controller*. Nonetheless, guaranteeing that users won't explore this feature for their own advantage, can be a challenge.

Better alternatives to the tasks' *warmup interval* should also be investigated, since this warmup interval may unnecessarily affect the accuracy of other application.

Finally, there are also opportunities for future work regarding the benchmarking of the proposed system. The system must still be tested in large scale environments with several heterogeneous applications, allowing to obtain a good estimation of how much the environment resource efficiency can be improved by using this novel approach for resource management.

Bibliography

[1] C. Delimitrou and C. Kozyrakis, “Hcloud: Resource-efficient provisioning in shared cloud systems,” *SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 473–488, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2980024.2872365

[2] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[4] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[5] S.-H. Ha, P. Brown, and P. Michiardi, “Resource management for parallel processing frameworks with load awareness at worker side,” in *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 2017, pp. 161–168.

[6] S. A. Baset, L. Wang, and C. Tang, “Towards an understanding of oversubscription in cloud.” in *Hot-ICE*, 2012.

[7] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, “R-Storm: Resource-Aware Scheduling in Storm,” in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware ’15. New York, NY, USA: ACM, 2015, pp. 149–161. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814808

[8] I. S. Moreno and J. Xu, "Customer-aware resource overallocation to improve energy efficiency in realtime cloud computing data centers," in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–8.

[9] "Apache flink: Runtime concepts," https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html, accessed: 2018-05-11.

[10] "Mesos oversubscription," http://mesos.apache.org/documentation/latest/oversubscription/, accessed: 2018-05-11.

[11] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams: A New Class of Data Management Applications," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 215–226. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287389

[12] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[13] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The Design of the Borealis Stream Processing Engine." in *Cidr*, vol. 5, no. 2005, 2005, pp. 277–289.

[14] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Record*, vol. 30, no. 3, September 2001. [Online]. Available: http://ilpubs.stanford.edu:8090/527/

[15] R. Kotto-Kombi, N. Lumineau, P. Lamarre, and Y. Caniou, "Parallel and Distributed Stream Processing: Systems Classification and Specific Issues," Oct. 2015, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01215287

[16] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing Using Dynamic Batch Sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 16:1–16:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670995

[17] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: stream processing under qos constraints at scale," *Cluster Computing*, vol. 17, no. 1, pp. 61–78, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10586-013-0281-8

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Data Engineering*, vol. 38, no. 4, 2015.

[19] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: continuous reliable distributed processing of big data streams," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 439–453.

[20] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, ser. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342773

[21] M. Kleppmann and J. Kreps, "Kafka, Samza and the Unix Philosophy of Distributed Data," *IEEE Data Engineering Bulletin*, 2015.

[22] "Kafka Streams Documentation," http://docs.confluent.io/3.0.0/streams/, accessed: 2018-05-11.

[23] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight Asynchronous Snapshots for Distributed Dataflows," *CoRR*, vol. abs/1506.08603, 2015. [Online]. Available: http://arxiv.org/abs/1506.08603

[24] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[26] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, no. 2011, 2011, pp. 24–24.

[28] "Storm resource aware scheduler," http://storm.apache.org/releases/1.1.1/Resource_Aware_Scheduler_overview.html, accessed: 2018-05-11.

[29] C. C. Aggarwal and S. Y. Philip, "A survey of synopsis construction in data streams," in *Data Streams*. Springer, 2007, pp. 169–207.

[30] M. Vlachou-Konchylaki, “Efficient Data Stream Sampling on Apache Flink,” Master’s thesis, KTH, School of Computer Science and Communication (CSC), 2016.

[31] N. Koevski, “Advanced Sampling in Stream Processing Systems,” Master’s Thesis, Instituto Superior Técnico, ULisboa, Nov. 2016.

[32] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, “Load shedding in a data stream manager,” in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 309–320.

[33] B. Babcock, M. Datar, and R. Motwani, “Load Shedding for Aggregation Queries over Data Streams,” in *Proceedings of the 20th International Conference on Data Engineering*, ser. ICDE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977401.978165

[34] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, “Load management and high availability in the borealis distributed stream processing engine,” in *International conference on GeoSensor Networks*. Springer, 2006, pp. 66–85.

[35] E. N. Tatbul, “Load Shedding Techniques for Data Stream Management Systems,” Ph.D. dissertation, Providence, RI, USA, 2007, aAI3272068.

[36] G. A. Agha, “Actors: A model of concurrent computation in distributed systems.” MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, Tech. Rep., 1985.

[37] Y. Tang and B. Gedik, “Autopipelining for data stream processing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2344–2354, 2013.

[38] Till Rohrmann, “Dynamic Scaling: How Apache Flink adapts to changing workloads,” in *Flink Forward 2016*, accessed: 2018-05-11.

[39] “Apache Flink Documentation - Asynchronous I/O for External Data Access,” http://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/stream/asyncio.html, accessed: 2018-05-11.

[40] “Apache Flink Documentation - Windows,” http://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html, accessed: 2018-05-11.

[41] A. Mucherino, P. J. Papajorgji, and P. M. Pardalos, *k-Nearest Neighbor Classification*. New York, NY: Springer New York, 2009, pp. 83–106. [Online]. Available: https://doi.org/10.1007/978-0-387-88615-2_4