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

Extended Abstract

Luís Alves
Instituto Superior Técnico
Lisbon, Portugal
luis.jordao.alves@tecnico.ulisboa.pt

Luís Veiga
INESC-ID, Instituto Superior Técnico
Lisbon, Portugal
luis.veiga@inesc-id.pt

ABSTRACT

In this paper we propose an alternative task scheduling mechanism for stream processing systems such as Apache Flink, that targets resource efficiency in a multi-tenant stream processing environment with several resource heterogeneous tasks being executed in parallel. The task scheduler we propose doesn't limit the amount of tasks that can run on each machine, instead, it adapts tasks' allocation based on their runtime metrics. Scheduling tasks to the machines with more available resources. At the same time, we explore load shedding in stream processing applications, as a mechanism to solve the tasks' resource starvation problem that may appear due to bad decisions performed by the scheduler, because of its optimistic approach and due to the dynamic workloads of the applications.

We implemented a proof-of-concept of such system in Apache Flink and tested it against scenarios that show the different aspects and advantages of the developed mechanism in action.

KEYWORDS

Stream Processing, Task Scheduling, Resource Efficiency, Load Shedding, Apache Flink

1 INTRODUCTION

Efficient resource management in environments where several heterogeneous applications are executed, has shown to be a challenging problem. In such environments, it is common to observe situations of resource underutilization, which happens because resources tend to be provisioned to the peak workload. While on average workload these provisioned resources are not fully utilized [2] [9], leading to unnecessary costs and energy waste. In fact, estimations can be found, claiming below 60% resource utilization on datacenters [13].

Overallocation of tasks to resources, appears as a clear solution to handle this problem [7]. Nonetheless, it yields another problem, since resource under-provisioning situations can occur, having an impact on the applications' performance. This causes the need to monitor the applications' runtime execution to detect these situations; and to have a compensation mechanism that guarantees that the Service Level Agreements (SLAs) keep being fulfilled while resource wastage is avoided, such as auto-scaling [14]. Additionally, resource estimation models have also been subject of study, allowing to reduce the need of this compensation mechanism [11].

In this paper, we focus on exploring both these problems applied to the domain of distributed stream processing systems, such as Apache Flink [3]. In these systems, the resource reservation-based model is still commonly used, causing situations of resource overallocation by applications, where unused resources can't be used since they are reserved by other applications.

Our solution consists in a task scheduling policy that assigns tasks to the machines with more available resources, in terms of CPU usage. Avoiding resource overallocation by simply dropping the concept of resource reservation.

Nonetheless, it may lead to situations of resource starvation. When such situations are detected, two possible approaches are proposed. The system can either decide to perform *Task Re-scheduling* of specific tasks to other machines, causing application downtime; or 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 approach is preferred, only using the first one as a last resort.

To guide the system decisions, two restrictions can be specified by the user, for each application's queries: their *priority*, which allows users to define some queries as being more important than others; and their *minimum acceptable accuracy*. 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 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.

As such, the main contribution from this paper consists in a novel resource management model, for stream processing systems, that uses both dynamic task scheduling and load shedding. Obtained results show improvements on resource efficiency and on the applications' latency and throughput in bottleneck situations.

This paper is organized as follows: Section 2 provides a background and an overview of the solution's architecture; Section 3 presents the model that governs the system decisions; Sections 4 and 5 detail the system components and their underlying algorithms; Section 6 highlights implementation-wise details; Section 7 shows the evaluation of the developed proof-of-concept; and Section 8 focuses on related work. Finally, Section 9 wraps up some conclusions as well as future directions.

2 BACKGROUND AND ARCHITECTURE

Although the solution developed in this paper can be generalized to other distributed stream processing systems, our architecture specifically targets Apache Flink [8], a modern instance of such systems that will allow us to prove our proposed solution.

Flink's architecture is organized in three main components: the Job Manager (JM), receives requests to deploy applications, schedules their tasks and monitors their execution; the Task Managers (TMs) that execute the actual tasks on the slots that they provide to the cluster; and the *Client*, that compiles the application dataflow and sends it to the JMs for execution.

A Flink application is modeled as a Direct Acyclic Graph (DAG), where the nodes represent operators that perform computations on the tuples they receive; these nodes are linked via streams that connect the output of one operator with the input of another. Nodes without upstream nodes are named *sources*, while nodes without downstream nodes are named *sinks* or *queries*. At runtime an operator can have several parallel running instances, named *sub-tasks*. A chain of sub-tasks can be grouped into a single *task*, which is executed by a single thread, where the sub-tasks perform computations sequentially on the received events. For higher throughput, Flink allows to pipeline sub-tasks, i.e. split a chain of sub-tasks into multiple tasks being executed by separated threads in parallel.

Figure 1 presents a bird's eye view of the previously described Flink components together with two key components introduced by our solution: the new *Task Scheduler*, that schedules tasks over the available TMs by following the already described policy; and the *Quality-of-Data Controller* (*QoD Controller*) that constantly monitors tasks in order to decide when *Load Shedding* or *Task Re-scheduling* 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 these components are guided by a model that will be detailed in the following sections.

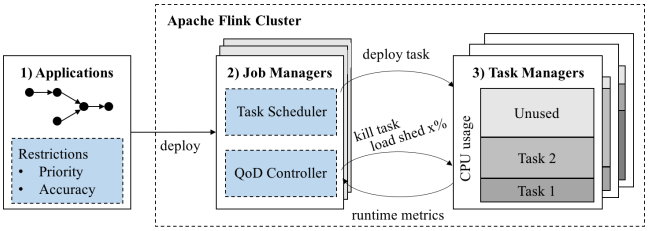

Figure 1: Architecture and components of the proposed system in a Flink cluster.

The two components run on the JM (2). The *Task Scheduler* executes for every task scheduling request. While the *QoD Controller* is triggered periodically, possibly sending messages to the TMs, either telling them to fail tasks or tune the tasks' load shedders.

Note that at a given instant, only one of the JMs is the leader, the others are in standby in order to provide high availability. Following that, at a given instant, only the *Task Scheduler* and *QoD Controller* of the leader JM will be executing.

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

The Flink stream processing Domain-Specific Language (DSL) / Programming Model (1) is also extended. Allowing the user to define the queries' restrictions. These restrictions are then propagated to the JMs, for the *QoD Controller* to take them into account.

As said previously, a load shedding mechanism will also be introduced at the applications' runtime level, allowing them to eventually drop some of the incoming workload. Load shedding has been subject of study in several other works, such as [6] and [15]. Both these papers reduce it to the following sub-problems: *When to shed?*, *Where in the DAG should the events be dropped?*, *Which events to shed?* and *How much to shed?*. Possible solutions to these problems are presented later in this paper, as we describe the load shedding mechanism that will be used by the system.

3 MODEL

Before going through the specification of how the new components perform their decisions, we first formalize the concepts and nomenclature that will be used.

We use $t(i)$ to refer to the task whose i is instance of; $tm(i)$ as the TM where the task instance i is executing, and $taskManagers$ for all the TMs in the system; $queries$ as the set of all query tasks; and $downstream(t)$ / $upstream(t)$ for the downstream / upstream tasks of a task t, that are directly connected to it.

3.1 Restrictions

As said, for each query in an application, users are able to specify two restrictions that will be taken into consideration by the system.

Priority - p(t)

Each task's priority is defined as $p(t) \in \mathbb{Z}$, computed as described in Equation 1, where $queries(t)$ corresponds to the set of downstream queries of a task t. The priority is specified by the user for each query and then propagated through their upstream.

$$p(t) = \max_{t' \in queries(t)} p(t') \tag{1}$$

To refer to the set of all tasks' priorities in the system, sorted in descending order, we use $priorities$. Values in this set can be scoped to a specific TM using $priorities(tm)$. We use $ti(tm, p)$ as the set of task instances in a TM tm with priority p; or $ti(tm)$ for all instances regardless of their priority. Additionally, $tm(p)$ is used to represent all the TMs that execute task instances with priority p.

Minimum Accuracy - minAc(t)

Each query q is also parameterized by the user with the minimum accuracy it accepts, $minAc(q) \in [0\%, 100\%]$. This value corresponds to the minimum percentage of input tuples of the application that must be processed to compute the query output. This is the definition of accuracy that we will follow throughout this paper. Equation 2 shows how this value is propagated throughout the query upstream.

$$minAc(t) = \max_{q \in queries(t)} minAc(q) \tag{2}$$

3.2 Load Shedding

Load shedding is performed by using random drops, where each load shedder is parameterized with the probability of keeping an

event, named the *non-drop probability* $d(t, t')$, between a producer task t and a consumer task t'. The way these non-drop probabilities are computed will be defined in Section 5.3.

To avoid wasting network bandwidth, load shedding is performed in the producer task. Source tasks also have load shedders right after their first sub-task, which is the one responsible for pulling data from the external datasource into the application dataflow. Allowing workload to be shedded right after fetching it from the datasource. This load shedder is also parameterized by $d(t, t')$, where t corresponds to the datasource.

The percentage of application input events being processed by a specific task t is given by $ac(t)$, which corresponds to our definition of accuracy of the task. To avoid biased results, due to different non-drop probabilities in task instances of the same task, decisions are performed at the task level. Therefore: for two connected instances $i1$ and $i2$ of different tasks $t1$ and $t2$, respectively, $d(i1, i2) = d(t1, t2)$; and $\forall_{i \in instances(t)} ac(i) = ac(t)$, where $instances(t)$ correspond to all instances of a task t.

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.

Current Accuracy - cAc(t)

The current accuracy, $cAc(t) \in [0\%, 100\%]$, 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 attempt to maximize $cAc(t)$ at all time.

We define this function as presented in Equation 3 which is computed using $lAc(t)$, defined in Equation 4, that represents what we call the local accuracy of a task t.

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.

$$cAc(t) = \begin{cases} lAc(t), \text{if } upstream(t) = \{\} \\ lAc(t) \times min_{t' \in upstream(t)} cAc(t'), \text{otherwise} \end{cases} \tag{3}$$

$$lAc(t) = \frac{\min_{i \in instances(t)} inRate(i)}{\left(\sum_{t' \in upstream(t)} outRate(t', t)\right) / |instances(t)|} \tag{4}$$

Regarding Equation 4, 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 fairly distributed over its task instances. Additionally, if the denominator of the Equation 4 equals zero then, $lAc(t) = 100\%$. This happens because in that case t is consuming all its input, i.e. none.

The objective is therefore to minimize the difference between a task upstream output rate and its input rate — maximize the overall application throughput.

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 5.

$$maxAc(t) = \max_{q \in queries(t)} desired(q) \tag{5}$$

3.3 CPU Load

The decisions of the *QoD Controller* are guided by the CPU usage and accuracy metrics from the applications' runtime. With the accuracy already defined in the previous section, it only remains to define the part of our model that takes into account the CPU runtime metrics.

The expected CPU load for a task t, cpu(t), is computed as described in Equation 6 where $cpuMetric(i) \in [0\%, 100\%]$ corresponds to the CPU load obtained from the TM metrics, for a task instance i. If a task is using a full CPU virtual core, then $cpu(t) = 100\%$.

Since the local accuracy, defined in the previous section, is restricted by the minimum local accuracy of the task's instances, $cpu(t)$ is also restricted by the CPU load of this same instance. Additionally, we assume that all TMs have identical computational capacity. Thus, in a stable system the CPU load should be similar for all instances of the same task.

$$cpu(t) = cpuMetric\left(\underset{i' \in instances(t)}{\arg\min} inRate(i')\right) \tag{6}$$

We also define: $cpu(tm)$ as the CPU usage of the TM, including non-Flink processes; $tCpu(tm) = \sum_{i \in ti(tm)} cpu(i)$; and $nCores(tm)$ as the amount of CPU virtual cores provided by a TM tm.

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. Computed as in Equation 7. The available CPU time is evenly distributed over all task instances with the same priority. Note that a task may require less CPU resources than the ones it can get, in order to achieve the accuracy that it requires.

$$mObtCpu(i) = \frac{aCpu(tm(i))}{|ti(tm(i), p(t(i)))|} \tag{7}$$

Required CPU - rCpu(t, ac)

The CPU required by a task t in order to have $ac(t) = ac$ is represented as $rCpu(t, ac)$. The function is defined in Equation 8, and assumes that the CPU load and accuracy are proportional.

$$rCpu(t, ac) = \begin{cases} 0, \text{if } cAc(t) = 0\% \text{ and ac=0\%} \\ 100, \text{if } cAc(t) = 0\% \text{ and ac} \neq 0\% \\ \min\left(\frac{ac \times cpu(t)}{cAc(t)}, 100\right), \text{otherwise} \end{cases} \quad (8)$$

In the first condition the required CPU is 0% since no accuracy is required. In the second one we provide our best bet on the required CPU, 100% (a full virtual CPU core).

Additionally, we also define the required CPU to achieve the minimum, $minAc(t)$, and maximum accuracies, $maxAc(t)$. Computed as presented in Equations 9 and 10, respectively.

$$minReqCpu(t) = rCpu(t, minAc(t)) \quad (9)$$

$$maxReqCpu(t) = rCpu(t, maxAc(t)) \quad (10)$$

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 11. Once again, it assumes the CPU load and the accuracy of the task to be proportional.

$$obtAc(t, cpu) = \begin{cases} 0\%, \text{if } cAc(t) = 0\% \text{ and cpu=0} \\ 100\%, \text{if } cAc(t) = 0\% \text{ and cpu} \neq 0 \\ cAc(t), \text{if } cpu(t) = 0 \text{ and cpu} = 0 \\ 100\%, \text{if } cpu(t) = 0 \text{ and cpu} \neq 0 \\ \min\left(\frac{cAc(t) \times cpu}{cpu(t)}, 100\%\right), \text{otherwise} \end{cases} \quad (11)$$

The rationale for the conditions is that: the first and second conditions are aligned with the logic followed in $rCpu(t, ac)$; the third, because if the current and provided accuracies are equal, then the accuracy should remain the same; and the fourth is an optimistic bet on the obtained accuracy.

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 Equations 12 and 13.

$$diffReq(t) = maxReqCpu(t) - minReqCpu(t) \quad (12)$$

$$slack(tm, p) = \sum_{i \in ti(tm,p)} (diffReq(t(i)) - mObtCpu(t(i))) \quad (13)$$

4 TASK SCHEDULER

For each task instance to be scheduled, the *Task Scheduler* allocates the task to the TM with the lowest CPU usage, taking into account the CPU usage from Flink and non-Flink related processes running on the machines. Ties are solved by picking a random TM.

If the task contains any location preferences, it will only consider them. Unless their available CPU is below the CPU required by the task, in which case it will disregard the location preference.

When a new task is assigned to a TM but no metrics are yet available, it will assume that the task will use 100% CPU (a full virtual core). This is only taken into account when computing the TM's CPU usage for future task scheduling and for the task allocation. Other initial CPU usage estimations are possible. Note that there is a trade-off between having the application tasks 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).

5 QOD CONTROLLER

The *QoD Controller* periodically executes Algorithm 1, which is defined in several steps. In each iteration, it starts by initializing the available CPU as the total available CPU for each TM to execute Flink tasks; and the desired accuracy for each query as 100%, since it haven't yet prune it with Load Shedding (lines 1-6).

Algorithm 1 QoD Controller main cycle.

```
1:  for all tm ∈ taskManagers do
2:      aCpu(tm) ← 100 × nCores(tm) − cpu(tm) + tCpu(tm)
3:  end for
4:  for all q ∈ queries do
5:      desired(q) ← 100%
6:  end for
7:  for all tm ∈ taskManagers do
8:      reqCpu ← Σ_{i∈ti(tm)} minReqCpu(t(i))
9:      rel ← 0
10:     if reqCpu > aCpu(tm) then
11:         rel ← killTasks(reqCpu − aCpu(tm), tm)
12:     end if
13:     aCpu(tm) ← aCpu(tm) − reqCpu + rel
14: end for
15: for all p ∈ priorities do
16:     for all tm ∈ tm(p) by slack(tm, p) DESC do
17:         distributeEvenly(ti(tm, p), aCpu, desired)
18:     end for
19: end for
20: Compute d(t, t′) based on the queries' desired accuracy
21: Send new d(t, t′) to the Task Managers where t is running
```

The algorithm then proceeds to guarantee that each TM has the necessary CPU to run all its task instances regardless of their priorities (lines 7-14). If there's not enough CPU, it releases the necessary CPU by re-scheduling some tasks (lines 10-12). 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 to avoid having to backtrack already made decisions. Finally, the non-drop probabilities for all streams are computed, using the method described in Section 5.3, and sent to the TMs to adjust the load shedders (lines 20 and 21).

5.1 Task Re-Scheduling

To select which tasks to re-schedule, given a TM and the amount of CPU to release, we use Algorithm 2. This is done in two steps. The first one aims at determining the maximum priority of the tasks that may have to be re-scheduled to release at least the required CPU load (lines 3-8). Returning a set of candidate tasks to be re-scheduled, and the released CPU if all those tasks are actually re-scheduled.

The second step (lines 9-16) avoids releasing more CPU than necessary, by pruning the candidate tasks set. It starts by the tasks with higher priority and higher required CPU. If by removing the task from the set, it still allows to release at least the amount of CPU that must be released, then, we remove it. Otherwise, the task instance is failed, in order to be re-scheduled to another TM. At this point, the released CPU by re-scheduling a task instance corresponds to the one used to achieve its minimum accuracy.

Also notice that the algorithm prefers to re-schedule tasks with low CPU consumption. The reason we opt for this semantic is because: 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.

Algorithm 2 Selection of tasks to be re-scheduled.

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

5.2 Resource Distribution

To distribute the remaining CPU of a TM over the task instances, Algorithm 3 is used. The algorithm receives as input the available CPU load to distribute, *aCpu*; the set of tasks over which it should be distributed, *it*; and the desired accuracies for all tasks in the system, *desired*. It 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 for the queries and the available CPU of the TM are updated (line 9).

Algorithm 3 Fair distribution of CPU.

```
1:  function distributeEvenly(it, aCpu, desired)
2:      c ← 0
3:      T ← it by rCpu(i, maxAc(i)) − minReqCpu(t(i)) INC
4:      for all i ∈ T do
5:          maxReq ← maxReqCpu(t(i)) − minReqCpu(t(i))
6:          cpu ← min(maxReq, aCpu(tm(i)) / (|it| − c))
7:          ac ← obtAc(i, cpu) + minAc(i)
8:          for all q ∈ queries(t(i)) do
9:              desired(q) ← min(desired(q), ac)
10:         end for
11:         aCpu(tm(i)) ← aCpu(tm(i)) − cpu
12:         c ← c + 1
13:     end for
14: end function
```

5.3 Computing the drop probabilities

To compute the non-drop probabilities for each stream, based on the desired accuracy for each query, we use Equations 14 and 15. The first one propagates the dropping probabilities upstream, allowing to drop events as soon as possible in the DAG, thus avoiding processing tuples that will be dropped in the downstream of the applications. After that, the second equation is used to compute the value of the non-drop probability of each load shedder, $d(t, t')$, given the already provided accuracy and the desired accuracy at the downstream task t', $desired(t')$. Note that $d(t, t')$ is also defined for the first load shedder in the source tasks, as the desired accuracy of its associated source task .

$$desired(t) = \max_{t' \in downstream(t)} desired(t') \tag{14}$$

$$d(t, t') = \begin{cases} desired(t'), & \text{if } t \text{ is an external datasource} \\ \frac{desired(t')}{desired(t)}, & \text{otherwise} \end{cases} \tag{15}$$

6 IMPLEMENTATION DETAILS

Our proof-of-concept was implemented on top of Apache Flink 1.2 release. The following changes were performed:

- Added metrics to compute the CPU usage of the task instances' main thread, and modified the TMs to periodically send messages to the leader JM containing the required metrics. Other necessary metrics are already provided by Flink, such as the input / output rates for task instances and their sub-tasks. Note that the output rate of a task doesn't take into account that events can be dropped at the end of the task.
- Extended the Flink Streaming DSL to allow the user to specify the priority and accuracy for each application's query, as presented in Listing 1. Once the application is deployed, these restrictions become available to the components that require them.
- Implemented load shedders that are executed at the required locations in the applications' DAG. Whose non-drop probability can be configured at runtime by the JM, by sending a message to the TM where the target load shedder is running.

- Task slots stopped being limited. Our implementation doesn't fully remove the concept of slots, instead, we simply consider them to be dynamic.
- The *QoD Controller* was implemented on the JM, and is executed with a configurable frequency.
- Flink's default task scheduler was replaced with the new one, that follows the policy described in Section 4.

In this implementation we came across some setbacks that are also worth mentioning, being discussed in the following sub-sections.

Listing 1: Snippet of the word count stream processing application with a minimum accuracy (0.7) and priority (5).

```
1  val words = source.flatMap(_.split("\\s+"))
2  val counts = words.map(value => (value,1))
3    .groupBy(0)
4    .sum(1)
5  counts.print()
6    .withMinimumAccuracy(0.7)
7    .withPriority(5)
```

Task Failing Limitations

When a Flink task instance is failed, all the application's tasks are re-scheduled, potentially to other TMs. This is not the desired behavior, we only want to re-schedule a specific task, while the remaining ones keep executing on the same TMs. To overcome this issue, if a task is failed on purpose the system will flag it. On the *Task Scheduler* if the task is flagged, 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.

Tasks' Warmup Period

When a task instance is re-scheduled, we provide it a warm-up period for the task to stabilize its CPU usage and to achieve its minimum acceptable accuracy. A timeout is added to prevent the task from being forever stuck in this state. Whenever a task instance is in a warm-up state, all the tasks whose instances are being executed by the same TM will be processing at their minimum required accuracy. Allowing the task to quickly recover and preventing precipitated decisions, causing other tasks in the same TM to be unnecessarily re-scheduled.

Accuracy of Source Tasks

To compute the $cAc(t)$ of a source task, one also needs to know the rate at which events are being produced for the application consumption, even if being stored in another system such as Apache Kafka [1]. For this reason, our implementation only supports Apache Kafka as a datasource.

By using the Kafka consumer metrics, we are able to estimate the input rate of the Kafka topics that they consume from. This estimation is performed based on: the variation of the *records-lag-max* metric, which provides the maximum difference of the input and output rate of the topic's partitions; and on the output rate of the topic to a Kafka consumer, given by the *records-consumed-rate* metric. Since the first metric is computed at the topic partition level, the *records-consumed-rate* metric needs to be divided by the amount of partitions the task instance consumes from, given by the *assigned-partitions* metric. By using this result and adding it to the lag variation within the partition, we get an estimation of the input rate of the most stressed Kafka partition. Assuming that the workload is well balanced across the topic's partitions, the estimation can be extended to all of them.

Regarding the input rate of the source tasks, it needs to take into account that incoming events may be dropped by the load shedders placed between the first and second sub-tasks of the source task's instances. As such, if the source task has a second operator, we use its input rate as the input rate of the source 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 unusual.

7 EVALUATION

To test the solution, two applications where developed, whose DAG is presented in Figure 2, both consuming data from Kafka:

(1) **Taxi Drives (TD):** receives two streams of events with information regarding taxi drives, that are consumed by two CPU intensive tasks, $TDSnk1$ and $TDSnk2$, using a *union* operator. $TDSnk2$ is configured with a parallelism of 2, while the remaining tasks have a parallelism of 1.
(2) **K-Nearest Neighbors (KNN):** KNN algorithm implementation with a single task, to test the integration with Apache Kafka.

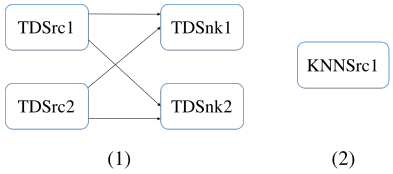

Figure 2: Logical DAGs of the Taxi Drives (1) and KNN (2) applications.

The evaluation consists in two parts. We start by running some scenarios to check that the solution has the desired features, showing the expected behavior of the system, as well as scenarios where it can reduce resource wastage and improve tasks' performance by detecting resource starvation situations. The second part aims at evaluating the performance impact in the JMs and in the applications, using Flink 1.2 release as baseline for comparison.

The evaluation environment consists in four machines: one for Apache Kafka and the data injector; one JM; and two TMs. Each machine has a dedicated virtual CPU, in a quad-core Intel Core i7, and 3Gb memory each. The *QoD Scheduler* was configured with a 5 second frequency.

7.1 Scenario 1: Tasks With Same Priorities

Our first scenario is an execution of the TD application, with $minAc(TDSnk1) = 40\%$ and $minAc(TDSnk2) = 60\%$, where all tasks have equal priority. Figure 3 shows the obtained results.

The system starts with a single TM, $TM1$, thus causing all TD's tasks to be scheduled to him. Later (A), the second TM, $TM2$, is added to the system, without any tasks being immediately scheduled to it. As the workload of the application increases to levels

where the application can't cope with, the load shedding mechanisms is triggered, causing the application accuracy to decrease (A). Every time the workload decreases, the accuracy of the application increases, up to the moment where load shedding is no longer required (B). In (A), the effect of all sinks having the same priority is clear, since load shedding is being applied to both sinks, without any of them reaching their accuracy threshold.

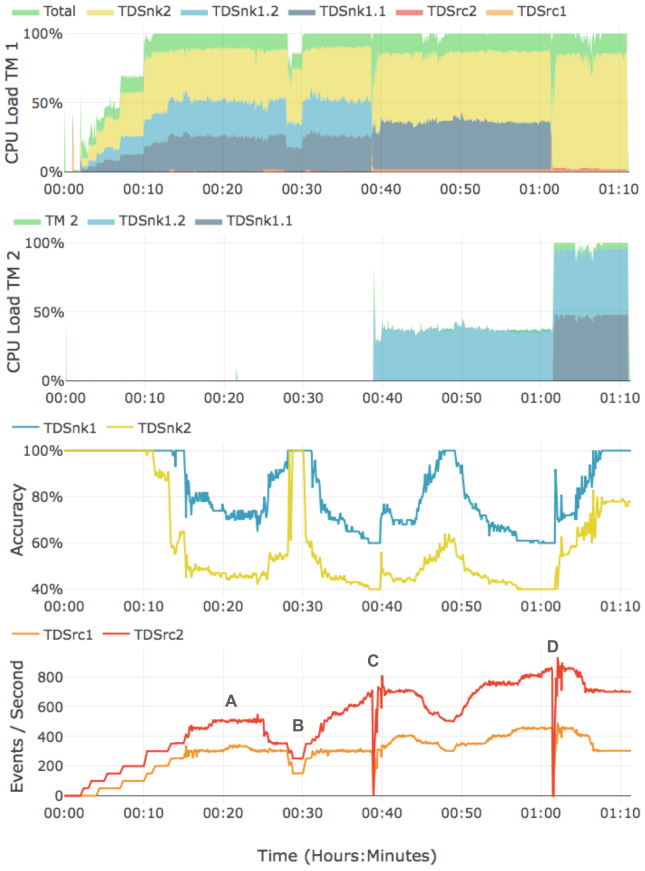

Figure 3: Execution of the TD application where all sinks have the same priority. First two charts show the CPU load of each TM and their executing tasks. Third and fourth charts show the application sinks' accuracy and the application throughput for each input Kafka topic, respectively.

In instants (C) and (D) the *QoD Controller* decides to re-schedule some tasks to the later added TM. In (C) the first instance of the *TDSnk*1 was re-scheduled, while in (D) it was the second instance of the same task. Yielding an optimal task distribution that maximizes both the throughput and accuracy of the application, and reduces resource wastage. Both *TDSnk*1 instances where re-scheduled instead of *TDSnk*2, which happens because their re-scheduling minimized the amount of released CPU while still releasing enough for the remaining tasks in *TM*1 to keep up with their workload.

The effects of the tasks' warmup state in the accuracy are visible every time a task is re-scheduled. All tasks took less than 1 minute to exit this state.

During the execution, we observed that the difference between $currAc(TDSrc1)$ and $ac(TDSrc1)$, and between $currAc(TDSrc2)$ and $ac(TDSrc2)$, was on average 2.4% and 1.8%, respectively.

7.2 Scenario 2: Tasks With Different Priorities

The second scenario shows an execution of both TD and KNN applications, where their tasks have distinct priorities. Allowing to observe how priorities are taken into account by the system. The KNN application was executed with a single instance.

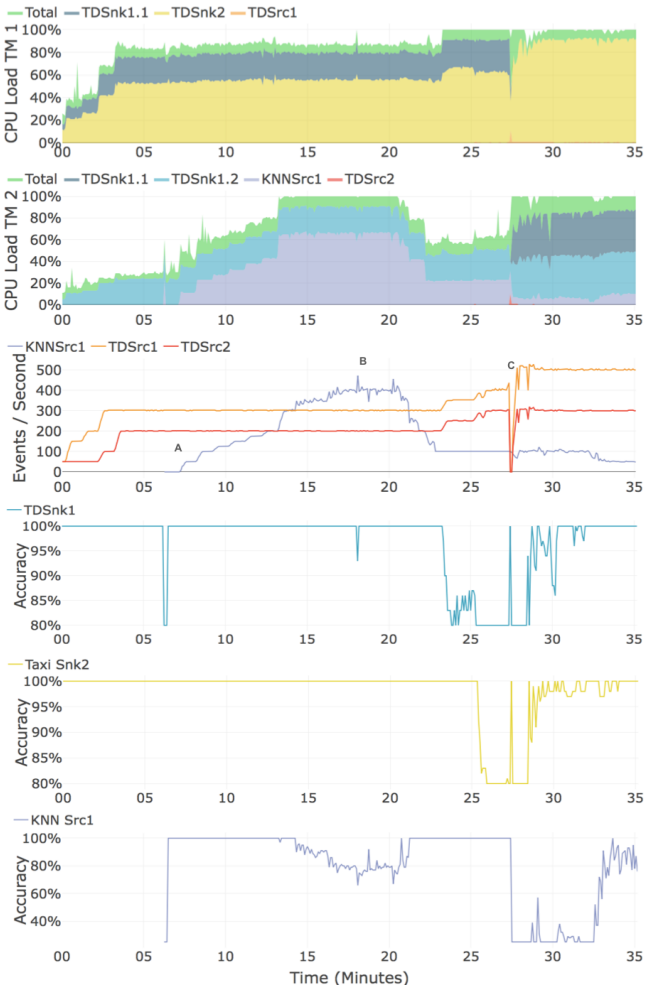

Figure 4: TD and KNN applications execution, having tasks with different priorities. First two charts show the CPU load in both TMs. Third chart shows the applications' throughput for each input Kafka topic. The accuracy for each sink task is presented in the remaining charts.

As shown in Figure 4, the scenario starts with both TMs available, and the TD application's tasks scheduled among them based on the scheduling policy. The KNN application is added later (A), being scheduled to the TM2, the one with more available resources. Once the KNN application is scheduled, the effect of its warmup state is visible on the other tasks executing in the same TM.

It is also visible that the *TDSnk*1 task has a higher priority than the *KNNSrc*1 task. When the KNN application workload increases (B), its accuracy also drops, while the accuracy of the *TDSnk*1 task remains unchanged. A similar behaviour appears after instant (C), the KNN application's accuracy drops to release resources for the *TDSnk*1.

The *TDSnk1* task also has a lower priority than the *TDSnk2*. As their workload increases, *TDSnk1* instances' accuracy drops before load shedding triggers 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 our system to adapt task scheduling based on the runtime requirements of application tasks, increasing the overall throughput. Flink 1.2 is incapable of performing this type of adaptation, clearly showing a situation where our system performs better in terms of resource efficiency.

7.3 Scenario 3: Apache Kafka Integration

The third scenario focuses on evaluating the integration of our mechanism with the metrics from Kafka, as described in Section 6. In this scenario, the KNN application is executed with a parallelism of 2, both instances consuming from different partitions of the input Kafka topic. Both TMs were used, each executing one of the task instances. Figure 5 shows the obtained results.

In terms of load shedding, results show that the system is able to properly tune the application's accuracy based on the incoming workload, either when it increases (A) and decreases (B). Comparing the event injection rate of the Kafka topic and the estimated injection rate (second chart in Figure 5), a small delay can be noticed. Nonetheless, the system was able to estimate the actual injection rate with a mean error of 0.4% ± 6.2% and 2.8% error in percentile 90%.

Regarding Kafka topic partitions' lag, we can observe some spikes, which appear due to two reasons:

(1) The definition we used for *current accuracy*, 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's 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, as will be observed in the following benchmarks.

7.4 Performance Assessment

In terms of performance impact of our mechanism, two benchmarks were done to assess the impact on the JM' resource usage (1) and on the application's performance (2). The same benchmarks were executed against Flink 1.2 release, which is used as a term of comparison.

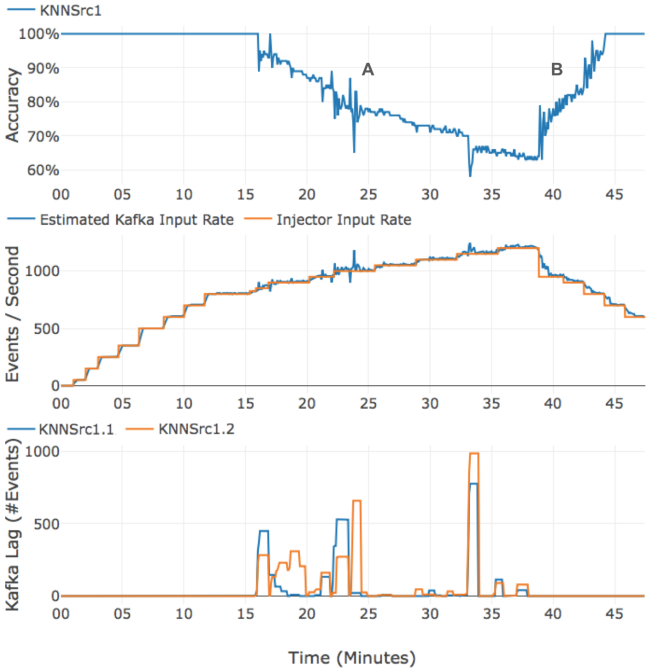

Figure 5: Execution of the KNN application as the event injection rate changes. First chart shows its accuracy; second chart a comparison between the injection rate and estimated injection rate to the Kafka topic; and third chart, the Kafka topic partitions' lag.

Regarding (1), we focused on understanding if our mechanism scales with the amount of tasks being executed in the system. The results presented in both Figures 6 and 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. While Figure 7 shows how the execution time of the *QoD Controller* monitoring cycle increases with the amount of executing tasks.

The results show that even with 2000 executing tasks, the JM's CPU load is below 15% in the third quartile and that the execution time of the *QoD Controller* is below 115 ms in the second quartile.

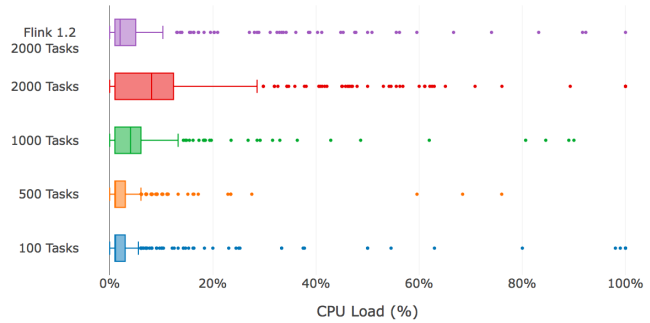

Figure 6: CPU Load of the JM for different amounts of executing tasks, with two TMs, and comparison with Flink 1.2 release.

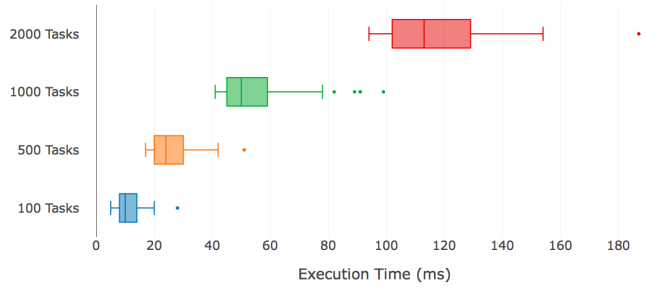

Figure 7: Execution time of the *QoD Controller*, given a fixed amount of executing tasks and using two TMs.

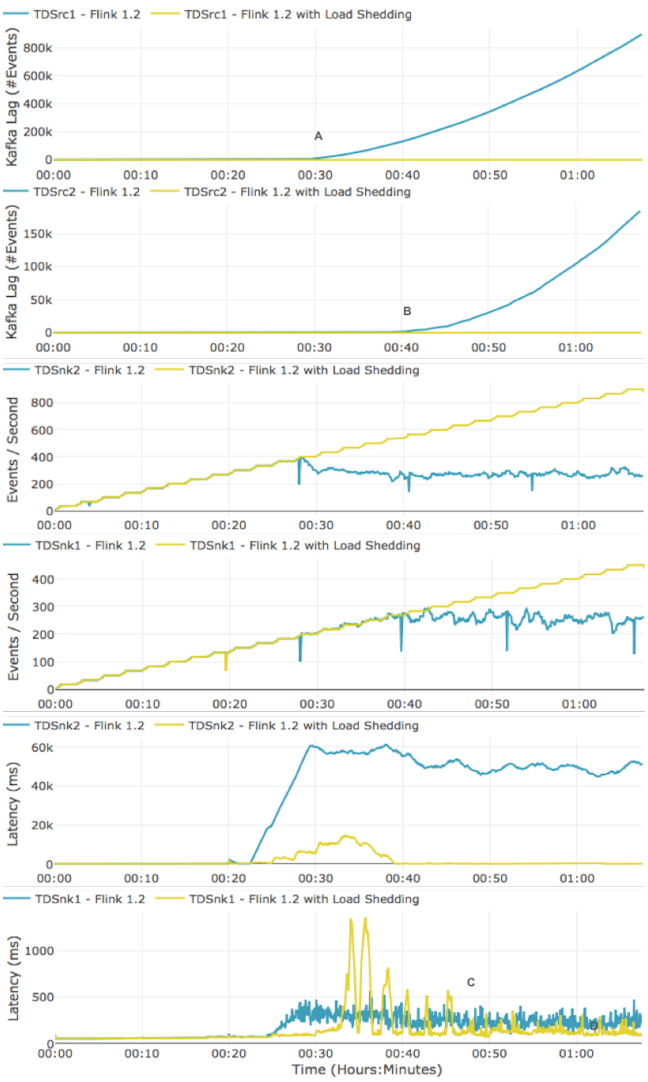

Figure 8: Lag, consuming throughput and latency of the TD application as the injection rate increases, both when using Flink 1.2 and in our modified version of the same release.

Therefore, our mechanism can clearly scale to environments running thousands of tasks in parallel. When compared with Flink 1.2 (first row in Figure 6) our solution is more CPU intensive.

Regarding (2), we used the TD application and monitored its latency and throughput as its workload continuously increases. A single TM was used, allowing a fair comparison with Flink 1.2. The obtained results are presented in Figure 8 showing the Kafka topics' lag, the application throughput at each input Kafka topic and the application latencies as the event injection rate increases over time. The measured latency doesn't take into account the time each event spends in queue, waiting to be consumed by the application.

Once the application bottleneck is reached in Flink 1.2 (instants A and B), the application throughput stops keeping up with the increasing injection rate, causing the Kafka topics' lag to continuously increase. The same doesn't happen with our system, thanks to the load shedding mechanism. Allowing the application to cope with the injection rate, thus keeping the Kafka topics' lag as zero.

Both in Flink 1.2 and in our modified version, the measured latency increases once the bottleneck is reached (A). In case of our modified version, we can observe several spikes associated with changes in the injection rate. Once this rate stabilizes, the latency tends to decrease to values below the ones observed for Flink 1.2 (C). The reason this happens is the same pointed in Section 7.3, though this time the events are queuing up in the application internal buffers between the producer and consumer tasks.

In case of Flink 1.2, the latency remains high since without load shedding, the intermediate application buffers will remain full. Once again, notice that the latency is not even considering the time that events spent in the Kafka queues, which for our modified version would be almost none.

This final benchmark shows key aspects where our load shedding mechanism appears as a clear advantage in situations where accuracy drops are acceptable, and the applications' throughput and latency are critical.

8 RELATED WORK

In Apache Flink, each TM provides a set of *slots* [8] that can be used to execute tasks. There is no CPU isolation between slots. Slots simply limit the amount of tasks that can be executed in the TM. Flink scheduling strategy assigns tasks to all slots in a TM, only then moves to the next one. This approach only succeeds if the characteristics of the tasks that will be deployed to the cluster are well known. If tasks are expected to consume few resources, then the TMs should provide a higher amount of slots, avoiding underutilization of the available resources. If they are expected to consume lots of resources, then slots should be coarse grained, since its expected for these tasks to consume all the available resources, thus avoiding resource starvation.

In Flink, tasks of the same application can share slots — *Slot Sharing* [4]. Thus allowing to group fine grained tasks and run them in a coarse grained slot, reducing the probability of underutilization. Nonetheless, it is up to the user, that defines the application, to enable or disable slot sharing.

On the other hand, Apache Storm [5] default scheduling strategy consists in a naive round robin distribution of tasks on the available machines. Though this strategy is simple and doesn't require any

specific input from the user, resource starvation situations can appear, causing applications performance to degrade and fail to cope with their workload, due to over-utilization of the resources in a machine. Our scheduler overcomes this issue by providing a mechanism that detects and assesses these situations.

Alternatively, Apache Storm also provides a different task scheduler, the *Resource Aware Scheduler* that was proposed in [12]. This scheduler allows users to specify resource related restrictions, such as CPU and memory, for each task in the application. Bin packing tasks to the available machines while taking into consideration their needs. With this approach tasks are guaranteed to get the resources they need, but it can suffer from resource underutilization.

Current resource management solutions for distributed stream processing systems lack the ability to dynamically adapt tasks' scheduling based on their runtime metrics. Which proved to be a key factor to improve overall resource efficiency in the cluster.

More similar to the proposed solution, Apache Mesos *Dynamic Oversubscription* model [2], allows the execution of best-effort tasks in reserved but unused resources. In this model, cluster resources are monitored to detect oversubscribed resources, and to make sure that the revocable tasks don't interfere with the regular tasks. If they do, these revocable tasks can be killed or throttled in order to correct the Quality-of-Service (QoS). Similar techniques have been proposed in other systems such as [10]. Both these solutions can't take advantage of load shedding, allowing application to provide fresh results even in peak load situations, as they are not specific for stream processing use cases.

9 CONCLUSIONS

Throughout this paper we proposed a novel task scheduling strategy for stream processing systems such as Apache Flink and Apache Storm. Our strategy specifically targets the problem of resource underutilization that current solutions fail to overcome.

Benchmarks of our mechanism show promising results as they prove that our strategy is able to adapt tasks' assignments to the available machines, converging towards a task distribution that not only reduces resource wastage, but also improves the applications' throughput in situations of resource starvation.

The developed load shedding mechanism also proved to be valuable, as it enable applications to keep up with their incoming workload without having to immediately re-schedule them for them to cope with it. At the same time, the load shedding mechanism is able to take into account different requirements from each sink in an application, avoiding processing events that will be dropped later in the application's downstream. Thus, being a good additional contribution from this paper.

Asides from the obtained results, there is still plenty of room for exploration regarding this subject. We consider that future work should focus on overcoming identified restrictions, allowing the solution to become production ready; and on exploring alternative semantics for the different components of the proposed mechanism, or even allowing these different semantics to be customizable. For instance, allowing users to provide their own definition of accuracy and of current accuracy for each application; or even enable applications to use different load shedding strategies, e.g. semantic load shedding.

REFERENCES

[1] Apache Kafka. 2018. Apache Kafka Documentation. https://kafka.apache.org/documentation/. (2018).

[2] Apache Mesos. 2018. Mesos Oversubscription. http://mesos.apache.org/documentation/latest/oversubscription/. (2018).

[3] Apache Software Foundation. 2018. Apache Flink. http://flink.apache.org. (2018).

[4] Apache Software Foundation. 2018. Apache Flink 1.2 Documentation: Distributed Runtime Environment. https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html. (2018).

[5] Apache Software Foundation. 2018. Apache Storm. http://storm.apache.org. (2018).

[6] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2004. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 350–. http://dl.acm.org/citation.cfm?id=977401.978165

[7] Salman Abdul Baset, Long Wang, and Chunqiang Tang. 2012. Towards an Understanding of Oversubscription in Cloud.. In *Hot-ICE*.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* 38, 4 (2015).

[9] Son-Hai Ha, Patrick Brown, and Pietro Michiardi. 2017. Resource Management for Parallel Processing Frameworks with Load Awareness at Worker Side. In *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE, 161–168.

[10] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.

[11] Ismael Solis Moreno and Jie Xu. 2011. 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, 1–8.

[12] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 149–161. https://doi.org/10.1145/2814576.2814808

[13] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 7.

[14] Olubisi Runsewe and Nancy Samaan. 2017. Cloud Resource Scaling for Big Data Streaming Applications Using A Layered Multi-dimensional Hidden Markov Model. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*. IEEE, 848–857.

[15] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 309–320.