Imperial College of Science, Technology and Medicine
Department of Computing

MEng Individual Project

Efficient Task Placement in Large Computing Clusters

Author:
Andrei Bogdan Antonescu

Supervisor:
Dr. Peter Pietzuch
Raul Castro Fernandez

Submitted in part fulfillment of the requirements for the
MEng Honours Degree in Computing of Imperial College London

June 2015

Abstract

The exponential increases in information encouraged the need for large clusters to analyse huge quantities of data in sub-second latency. This environment triggered an increase use case for streaming computation in contrast to more classical batch computations. Over a decade of academic research was invested in crafting state-of-the-art batch schedulers. However, with the rise of streaming computation the former cannot adapt efficiently to this new class of workloads characterized by dynamic resource consumption and very long execution cycles.

In this report we describe a new scheduler that dynamically adjusts to workload changes by migrating tasks at runtime. This way we constantly ensure the system throughput is maximized and each job has a fair share of resources during the long lifespan of streaming applications. We will discuss in detail the heuristics used to estimate task potential based on resource contention. Next we will study different scheduling strategies that constantly monitor resource utilization to optimize allocations. Lastly we evaluate our scheduler on a comprehensive set of benchmarks model after real-world workloads and compare it with other widely-used schedulers.

Acknowledgements

First I would like to thank my parents. Without their support I would have not been able to pursue my dreams and be the person that I am today.

I would like to thank my supervisor, Peter Pietzuch for all his valuable guidance and rigorousness throughout the duration of the project. His feedback helped me improve as a scientist and raise the standards of my work.

A special thanks goes to my second supervisor, Raul Fernandez for all his technical help especially at the beginning of the project. All our discussions about scheduling and streaming have been very inspiring and fuelled my enthusiasm to learn more.

I would also like to thank Tony Field for all his great feedback he gave me during our meetings.

Last but not least, I want to thank Irina Veliche for all her support and valuable discussions throughout the year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The amount of digital information that is created every month equals the entire size of the internet just a decade ago [37]. Those huge amounts of data need to be analyzed and understood so we can extract useful information that powers our world wide web.

The exponential increase in CPU performance following Moore's law cannot keep up with the quantities of data present in our digital world. With the possibility of using one cutting edge supercomputer being infeasible, a lot of research effort has been invested into distributed algorithms that run on thousands of computers, solving together tasks too large to be comprehended by one. Important technology companies such as Google, Facebook, Amazon and Microsoft choose to build their datacenters out of commodity hardware [50]. Most of these companies use diverse backend systems for serving infrastructure, data analysis, ads service and many more. For those to work efficiently many tasks need to run concurrently inside a cluster, under resource isolation and without influencing one another‘s performance. Due to the reasons outlined above the problem of scheduling tasks to efficiently share computing resources attracted a lot of research interest in the last decade.

Cluster computations can be categorized into two broad types: *stream* and *batch* jobs. The first represent one time computations processing offline data that have a finite completion time and are scheduled in advance. The second represent non-deterministic unbounded computations which have a dynamic workload and can create various bottlenecks for which the system needs to adjust. The latter have to process real time data as input and run computations on the fly without acquiring delays that could result in data loss. Stream computations have increased in frequency recently as companies tend to have considerable amounts of data [1] and want to extract more useful information from it by running complex algorithms such as machine learning, natural language processing or real-time fraud detection.

Scheduling stream jobs is more difficult than batch jobs because input data can vary over time thus streaming operators can exhibit dynamically varying resources consumption at runtime. Most of

[1] According to public information Google processes daily 100Pb of data [38], Facebook 600Tb [34] and Twitter 100Tb [51]

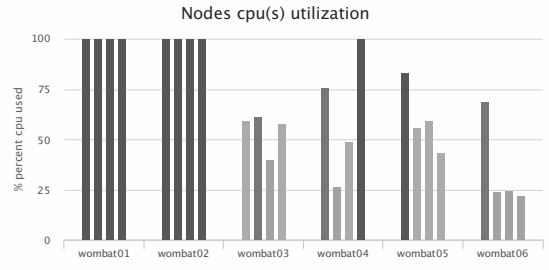

FIGURE 1.1: CPU utilization per cluster core with YARN.
throughput: 413 events / second

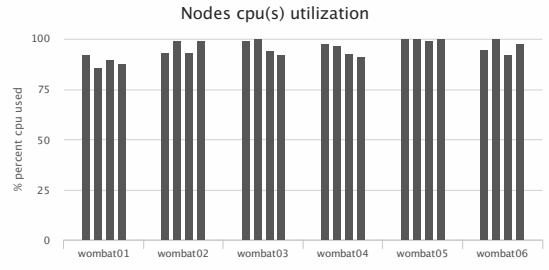

FIGURE 1.2: CPU utilization per cluster core with runtime scheduling.
throughput: 547 events / second

the existing schedulers concentrate solely on batch jobs [3–5, 7, 17] while others [10, 14–16] support both types. However they are very inefficient when exposed to combined workloads since they lack mechanisms to adjust in real-time to workflow changes. Furthermore they usually use abstractions over the physical layer that results in the inability to place task on specific machines (i.e. to respect data locality or prevent resource bottlenecks).

Those limitations often lead to poorly distributed workload across the cores in a cluster. To illustrate that we measured in Figure 1.1 the CPU load distribution per core with YARN [16], a widely used jobs scheduler and resource manager, while running a CPU intensive streaming workload. We can observe how the load is concentrated mostly on two machines limiting the performance of the tasks. As a consequence there is a genuinely need for a system to handle dynamic resource-aware scheduling at run-time being able to distribute the load more efficiently thus achieving high throughput. We propose a solution for scheduling efficiently both *stream* and *batch* jobs on a large cluster such that we can achieve the load distribution illustrated in Figure 1.2 on an identical workload which increases the overall performance by *26 percent*. We achieve this by designing a new scheduler that runs approximation algorithms to predict the resource needs of running jobs. Based on the prediction it tries to match "straggling" tasks to nodes that can provide more resource. If a match is successful we analyse the trade-offs involved and run the actual migration. This way we manage to distribute the load evenly under a variety of workloads. To enable our design to scale and be fault-tolerant we build our system on top of a resource isolation manager with support for data locality and task pinning and used for communication means a fast, fault-tolerant messaging queueing system.

1.2 Contributions

This project makes the following contributions:

- **Develop New Job Scheduler**
  We develop a new job scheduler that can easily adapt to dynamic workloads with complex resource usage patterns and maximize the cluster throughput in every scenario. We implement heuristics to detect workload changes at runtime and continuously move tasks to prevent bottlenecks.

- **Evaluate Several Scheduling Systems**
  We create benchmarks modelled after industry-like stream and batch workloads using public streaming feeds from social networks such as Twitter. Additionally we used renowned benchmarks published in research publications to compare the performance of our implementation with other scheduling frameworks such as Naiad [10], Storm [21] and Spark [23].

- **Fault Tolerant and Scalable Design**
  We designed our system such that every computation can be partitioned over a set of worker nodes, including scheduling itself. In our scalability section 5.3.3 we show that our framework is able to scale to thousands of jobs per second running in a large cluster. Furthermore we implemented our system with fault-tolerance in mind by replicating data across multiple machines and running backup services ready to replace the masters in case of failure. As a consequence our system doesn't have any single point of failure.

- **Cluster Analytics Framework**
  We create a complete cluster analytics system that keeps track of resource utilization and job performance metrics. We used the system to benchmark the performance of our scheduler and compare to with other state-of-the-art schedulers.

1.3 Outline

In this project we use a stream processing system called SEEP [6, 8] which was developed inside the LSDS research group at Imperial, and explore various alternatives of scheduling both *batch* and *stream* jobs with a fine grained resource control. We aim to achieve high global throughput and low latency.

We continue with Chapter 2 where we explore some of the existing systems for running large scale distributed computations ranging from legacy solutions that laid the foundations, to new state-of-the-art systems still under active development including one released this month [39]. We briefly describe the characteristics of each framework and present its weaknesses and strengths while emphasizing why it cannot solve our problem.

Chapter 3 describes the extensions required to SEEP to make it highly-scalable and fault-tolerant. We do so by first describing the alternatives available to solve our problem and motive our choice. Next we give a high-level detail of the implementation and follow with more technical details.

Chapter 4 explains the limitations of YARN scheduling while running SEEP streaming operators and describes our first iterations of resource aware scheduling as well as the problems we encountered along the way. Finally we dive into our runtime scheduler design and explain in detail how the scheduling algorithm works, how we predict tasks performance and what we use for scoring allocations.

Chapter 5 describes our cluster metrics system that we developed along the way. Next we go into detail on different key points of our system like measuring migration trade-offs and scalability. We conclude the chapter by presenting how we made our system fault-tolerant and analyse possible failure scenarios.

Chapter 6 evaluates the performance of our system under different benchmarks and evaluate different scheduling strategies and discuss the results. Next we analyse the latency introduced by our system and measure the resource usage over time. We conclude this chapter by highlighting the improvements our system makes compared to a static scheduler.

Lastly, in Conclusions (Chapter 7) we provide a retrospective of our system and discuss to what extent we have reached our goals. We conclude by presenting the lessons learned and showing the potential for future work.

Chapter 2

Background

Starting from the 70s networks of computer connected together in local area networks become common. Since then there is a perpetual research interest in distributed computing and as larger networks of computers were formed the scheduling problem became of key interest. In the last decade many authors presented novel ways of scheduling optimized for efficiency, with fairness and latency guarantees or even distributed schedulers. Even though a lot of research was conducted in this field the problem of task placement is still open ended.

With the widespread adoption of data-parallel systems that execute in distributed clusters there's been a growth of computing frameworks to manage large batch-oriented workloads. A diverse range of batch-only schedulers ware created with a broad range of characteristics such as: fairness guarantees, data locality, minimize latency, distributed scheduling or high scalability. More recently people combined cluster resource managers with schedulers to provide an abstraction over the physical layer making it easier for schedulers to manage resources. For running computations there were also a diverse set of processing systems from the classical Map Reduce [1] and Hadoop [12] to specialized stream processing framework such as Storm [21] and Samza [11]. However the latter still rely on existing cluster resource managers that lack the ability to handle stream workloads efficiently due to their coarse abstraction. In the following chapter we will present some of the existing frameworks and explain how they work. As we will see most of the current system are designed for batch jobs in contrast to stream jobs which have different requirements.

2.1 Job Schedulers

Job schedulers are system for controlling backend tasks running inside a data center such that maximum throughput it achieved and jobs don't suppress each other while running concurrently. In this section we will explore the latest schedulers and we will see why there is a authentic need for a new system capable of scheduling stream tasks with respect to data locality, maximum throughput and fairness guarantees.

2.1.1 Quincy

Quincy [3] was created in response to disadvantaged of traditional queue-based schedulers and its authors argue that data-intensive computations benefit from a fine-grain resource sharing within a cluster as opposed to existing static resource allocations. They propose a new flow-based scheduling that enforces fairness such that a large job will not monopolize the whole cluster, delaying completion of smaller jobs.

The paper concentrates on computing clusters with large disks allowing application data to be processed locally. As these clusters grow network communications represents a bottleneck and it becomes very important to place computation close to its inputs data. However usually requirements of fairness and data locality conflict and as the number of concurrent jobs with cross-cluster network traffic increases it becomes very complex to predict performance. Quincy shows for the first time a similarity between efficient and fair cluster scheduling and the classical problem of min-cost flow in a directed graph. This is important since the min-cost flow algorithm is guaranteed to find the best solution outperforming any greedy allocation.

The authors of Quincy state their goal for fairness that a job that runs t seconds when given exclusive access to the cluster will run in less than $J*t$ seconds when J jobs are running concurrently and competing for resources. To achieve fairness they use preemption: interrupt jobs that take to long to complete in favor of other jobs that need to run. In general jobs are formed by a "root task" and a set of "workers tasks" managed by the root tasks so Quincy will only choose to kill "worker tasks" to avoid losing the whole progress of a job. To decide which worker task to kill they start with the most recently scheduled task to minimize wasted work. This approach guarantees eventual completion of every jobs since the longest running task will never be killed.

To apply the min-flow algorithm Quincy creates a graph from a instantaneous snapshot of the system containing all the workers, their root tasks and all the computers. Each worker is connected with an edge of capacity one to the node representing the machine where the task runs. All the machines are connected with capacity one to the Sink, hence allowing only one task to run in a node at a time. To account for "unscheduled" tasks they introduce one special node U_i for each task i and connect all workers with that node as well. As a consequence all paths from a worker to the Sink lead either thorough a computer node or through a unscheduled node. To account for fairness edges connecting worker node w_i, j to node U_i have costs that represent the penalty of leaving the jobs unscheduled at this time. Those costs increase overtime ensuring each jobs gets its turn to be scheduled.

While evaluating the algorithm they compared 4 different fairness policies: fair sharing with preemption, fair sharing without preemption, unfair sharing with preemption, unfair sharing without preemption with the main difference is whenever the scheduler optimizes for fairness or allows unfair shares while optimizing tasks preference. The experimental results were very promising but the design had its own limitations such as: limited scalability, bottlenecks arise due to the centralized scheduler, cannot optimize for data locality and limitation to scheduling batch jobs only. Even so Quincy laid a foundation for future research in the area.

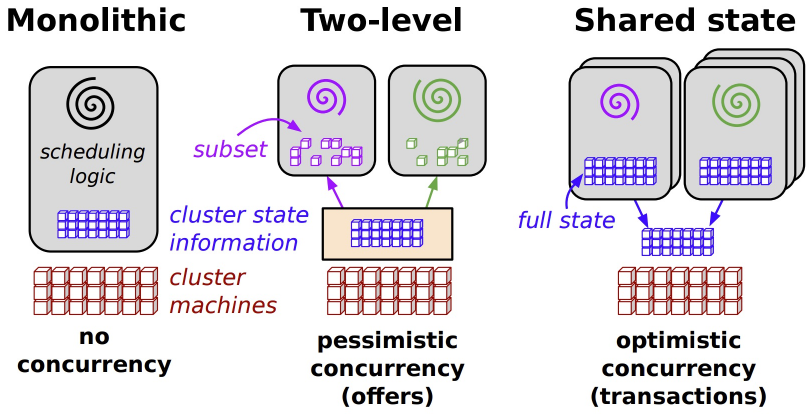

FIGURE 2.1: Overivew of commonly used scheduler architectures. Source: Omega paper

2.1.2 Omega

Scheduler architectures can be divided in three categories as shown in Figure 2.1, the first being the most widely used:

- **Monolithic**: all the cluster state is kept in one scheduler process making the system able to schedule only one task at a time as is the case of Quincy [3].
- **Two-level**: schedulers which have a separate resource manager that offers resource to multiple scheduling frameworks such as Mesos [14, 15].
- **Shared state**: multiple scheduler instances work in parallel to make scheduling decisions on the whole cluster. This design is a novel approach to scheduling.

The authors of Omega [4] observed that as the need of cloud-based computations increased clusters began to grow in size and since the scheduler workload is proportional to the cluster size monolithic scheduler quickly became a bottleneck. Furthermore they argue that monolithic schedulers restrict the rate of updates, limit efficiency and eventually limit cluster growth. They propose a novel approach for the problem of scheduling with emphasis on lock-free optimistic concurrency control. They design was drive by real-life situation like Google production workloads. This is the first time a distributed scheduler could scale out to thousands of machines running batch-only computations was created.

Omega is formed by multiple scheduler running in parallel where each of them has a frequently updated local copy of the resources available in the cluster and make decisions individually. To present conflicts all the schedulers run an atomic commit protocol when trying to update the common state, hence only the first will succeed in a conflict. This approach eliminates immediately the inability to make decisions aware of the whole cluster with the potential trade-off of doing the same work twice: two scheduler try to schedule the same job. The performance of the system is mainly influenced by the frequency of redundant work performed by schedulers and the authors rely on the fact that typical cluster workloads have a small odds of generating failures.

Omega's approach does always more work than a pessimistic concurrency strategy based on a locking scheme. In spite of that their results showed the overhead to be acceptable since the resulting benefits of increased scalability are huge. A shared-state can be used in our scheduler as well to increase its scalability beyond the limits of a single instance. Omega's limitations for the moment are providing global guarantees such as fairness, starvation or task completion guarantees.

2.1.3 Jockey

Jockey [5] looks into providing completion time guarantees for critical jobs. The key concept of Jockey is the ability to predict accurately the run time of a job in different stages of computation by using a simulator that precomputes statistics around low level primitives. In certain situations missing a deadline has significant consequences such as delaying a chain of depended jobs or even safety or financial faultiness. Older systems running critical jobs had manual operators that monitor the status of the tasks and operators had to manually interfere if one gets left behind. Jockey creates a framework that will guarantee automatically latency requirements for batch jobs.

Running tasks in a cluster have different type of deadlines. Most of the jobs have "soft" deadlines meaning it's desired to complete within a given time bu failing to do so is not critical while others have "strict" deadline which means that the system has to guarantee their completion time. After examining the scenarios in which jobs increase their latency the authors provide three approaches to guarantee completion time:

- **Additional priority classes** This approach creates a new class of priorities "SuperHigh" and only jobs with the strictest deadlines are allocated those priorities. Repeated job profiling to optimize allocation combined with limited admission control can provide completion time guarantees with this design. However jobs that run with "SuperHigh" will eliminate all normal jobs in case of resource contention which results in unnecessary delays or progress lost for the letter. Furthermore the cluster needs to be overly pessimistic about the number of jobs it can allow in the high priority class to be still able to respect guarantees.
- **Quotas for each job** This solution assigns quotas for each "strict" job that represent guaranteed resource allocation. The authors investigated this approach in detail and found it unsatisfactory for three reasons: node failures require adjustments in job quotas, determining quotas in the first place is very difficult and static allocation doesn't benefit from run-time changes which could improve throughput.
- **Dynamic resource management** Finally this is the design they implement with Jockey by creating a specialized scheduler that adjusts resource allocations in order to meet completion guarantees. They used greedy algorithms to predict job's completion time such as a event-based simulator and an analytics model inspired from Amdahl's Law. Their ideas inspired us to create a job potential prediction in our scheduler.

Jockey manages to achieve 99% of the jobs deadlines based on multiple experiments in multi-tenant clusters while requiring only 25% more resources.

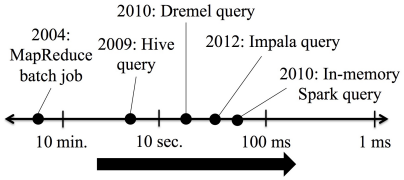

FIGURE 2.2: Latency span of different data analysis jobs. Source: Sparrow Paper

2.1.4 Sparrow

Developed at Berkeley in 2013 Sparrow [7] tries a new approach for scheduling by creating a decentralized, randomized system. Their motivation comes from the fact that today's latest processing frameworks like Spark [23] and Impala [24] partition work across thousands of machines and achieve repose times into the 100ms range as we can see in Figure 2.2. To handle those workloads we need a different setup for schedulers in contrast with traditional workloads. Size of today's clusters combined with very fast completion times lead to requirements of over 1 million of scheduling decision per second. The paper describes a system composed of autonomously machines that make scheduling decisions without a centralized state. This is the only way by their claims to achieve sub-second tasks with high throughput.

Their design relies on the two choice load balancing technique: when placing a task consider two random machines and pick the one which has fewer queued running, in order to achieve very low latency. They extend this idea with three techniques to make it more suitable for scheduling.

- **Batch Sampling**: Two choice technique performs poorly for tasks running many short jobs in parallel because the scheduling latency for each job adds up to a significant task latency. To cope with that the authors solve this problem using the recently developed multiple choice approach: placing m tasks on the least loaded of $d * m$ random selected machines, where d is a constant.
- **Late Binding**: Due to tasks start-up delays measuring queued tasks is not a good indicator of wait time. The authors solution around that is delay assigning of workers until the tasks is ready to start. Even though the difference is very small they claim this technique reduces median job response time by 45% compared to standard batch sampling.
- **Policies and Constrains**: Sparrow uses multiple queues for tasks with different priorities and supports per-job and per-task placement constraints. Their solution presents a novel policy enforcement and constraint handling which performs 12% within an ideal scheduler.

Their ideas to achieve low latency scheduling are very interesting but not useful for streaming workloads where tasks are long running exceeding by far typical workloads for which their scheduler was designed. Sparrow was quite successful in practice and is integrated in widely used processing frameworks like Spark but still is designed for batch computations mainly.

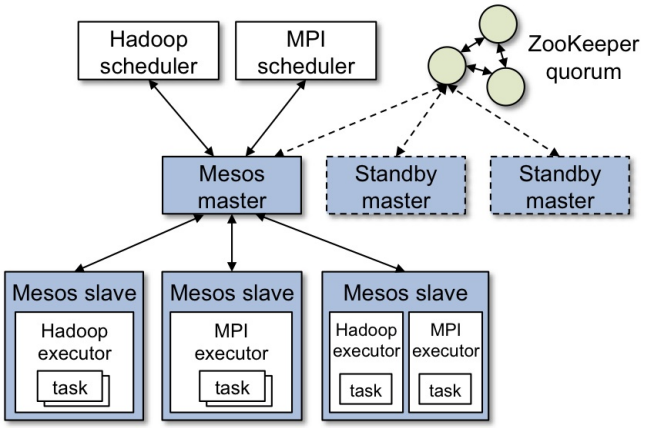

FIGURE 2.3: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI - Message Passing Interface). Source: http://mesos.apache.org/

2.1.5 Mesos

As we have seen so far, a diverse array of cluster frameworks have been developed by researchers and practitioners alike with each having its own strengths and weaknesses. This patter was clearly emphasized in recent year and we believe no framework will be optimal for all applications. With that in mind a team of researchers from Berkely build Mesos [14, 15], a platform for sharing resources between diverse cluster computing frameworks. Mesos aims to reduce data replication, improve communication between frameworks and optimize utilization of resources. The system scales up to 50,000 nodes and uses Zookeeper [29] for coordination and fault tolerance.

Mesos introduces a distributed two-layer scheduler that decides how to offer resources to each framework while the frameworks decides which resources to accept and how to utilize them. It achieves that by sending *resource offers*: list of free resources on multiple nodes. The scheduler runs continuously iterations of scheduling and in each of them it creates a new fair share of resources and splits them among frameworks by providing them resource offers. Each frameworks can accept or refuse the offered resources if they are not suitable for its needs (i.e. nodes have less that R resources free or the data is not local). In this case the framework keep its fair share for the current round and get new resources at next scheduling iterations. Mesos uses three techniques to make task scheduling efficient and robust to failures. First, because some framework could always refuse certain resources the system provides filters: a way to to efficiently short-circuit the decision process without extra communication. Second, in order to prevent time taken for a framework to respond to allocation Mesos starts counting their share before that. Last, if a framework is unresponsive for a long time the scheduler revokes its resources and gives them to others.

With all points described above Mesos had a lot of success and was received well by the community and later on open-sourced under Apache license. Overtime many enthusiast contributed to the project making it more mature and also discovering its limitations. Many people consider that Mesos's scheduling capabilities are quite robust and outperformed by Hadoop's capacity or hierarchical schedulers. Another point is that resource offers put more overhead to the person implementing the framework and can't scale out to real time systems like stream processing

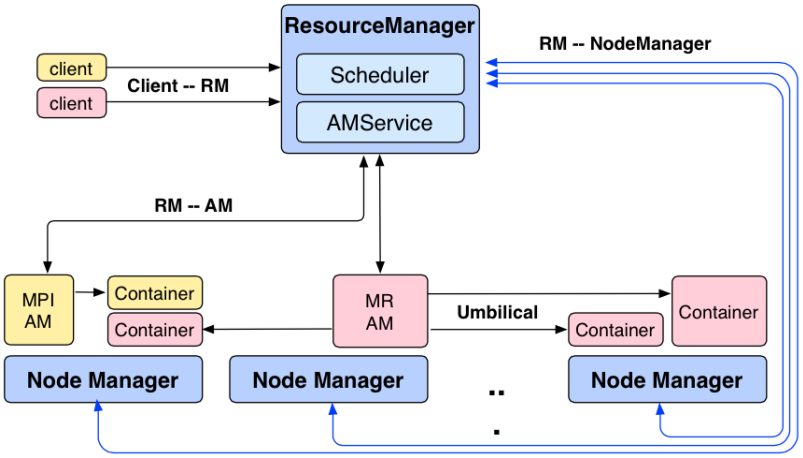

FIGURE 2.4: YARN Architecture (in blue the system components, and in yellow and pink two applications running). Source: YARN paper [16]

pipelines. Lastly Mesos doesn't provide any disk space or I/O scheduling and this is a resource that can become a bottleneck to under certain applications.

2.1.6 YARN

Apache Hadoop began as open-source implementation of MapReduce focused initially to run massive parallel jobs to process data from web crawls. The framework popularity made it *the* place to be for everyone that needed vast computational resources or wanted to analyze large amounts of data. The open-source community began to expand Hadoop far beyond the capabilities of its original programming model. Developers who wanted access to Hadoop clusters resorted to ingenious ways to circumvent the existing limitations of the MapReduce API.

However the Hadoop architecture took its tool and there was a need for a new framework that would delegate scheduling to job level and provide a great flexibility of processing framework running on top. To solve this problem researchers created YARN [16], a new generation compute and resource management framework open-sourced via Apache. YARN combines existing scheduler developed for Hadoop with a separate cluster manager framework. This way the use its free to use his own policies or rely on YARN to schedule its jobs. YARN achieves resource isolation by using cgroups and supports currently isolation for CPU and memory. In the future they also plan as well tu support I/O isolation.

Overview A YARN cluster is composed of two main components: Resource Manager (RM), and Node Manager (NM) as shown in 2.5. The RM is the master of the cluster and coordinates all container allocations and resources cluster-wide while the NM is a slave, that runs on every host and handles container start, execution status and ensures resources, allowances are respected, which can be include soft or hard limits.

When an application requests for a certain amount of resources the ResourceManager has enough resources to satisfy the request and schedules the application on that host. The NodeManager starts

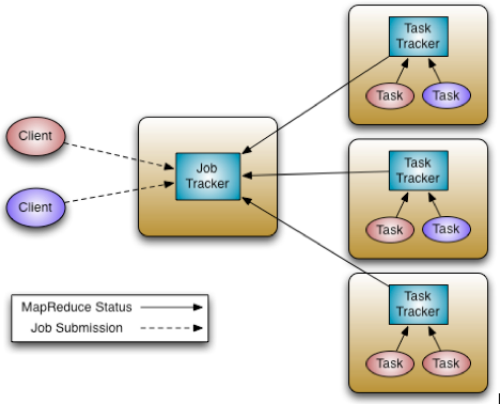

FIGURE 2.5: YARN components
Source: http://hortonworks.com/

the containers on that node and continuously monitors the resource utilization of that container. If a resource limit is hard how is the case for memory the NM will kill the container instantly if it exceeds its allocation. This ensures that the containers are always limited to their memory allowances. In SEEP, most of the queries used for benchmarks keep small amounts of data in their memory so strict memory isolation is not a concern and has no impact on performance. For CPU, the limits are soft by default, which means that if there is spare CPU available on a node, containers are allowed to use it, potentially far exceeding their allocation. When another container starts on the same node existing containers CPUs are scaled back appropriately. This pattern is similar to the way the OS kernel shares CPU time between threads. YARN has the ability to impose hard limits on CPU as well which might be useful when running performance benchmarks on a shared cluster without adding noise in the results from the load.

CPU resources are divided into units called *Virtual Cores (vcores)* which represent a percentage of the total CPU available on a host. In practice, it is recommended that one or two *vcores* per physical core are used but often this value can be adjusted for hybrid infrastructure.

All the benefits outlined above are achieved by separating the resource management function from the programming model. However this creates an abstraction over the physical layer which make it impossible for the application-level scheduler to map tasks to physical machines. When resources for a new job are requested, YARN will decide where to deploy containers based entirely on free resources without worrying of data locality in respect to input streams. This brings a big performance impact when running stream jobs because we lack the ability to place tasks where the data is stored, thus placing a high load on the network. However

2.1.7 Naiad

Naiad [10] was developed at Microsoft Research with the aims to combine different scheduler architectures under one framework without losing any downside in performance. They claim that by introducing a new computational model, *timely dataflow* they can achieve the high throughput of batch systems, low latency of stream pipelines and also support incremental and iterative computation. Further more applications build on a single platform are typically more efficient.

The authors provide a set of primitives that can sport a wide range of algorithms and diverse computations: structured loops, stateful dataflow vertices that apply transformations over data without coordination and notification system for the two. The computation model is based on a directed graphs where each vertex applies an operation over data. Loops are formed by connecting vertices in a cycle. Messages passed around contain a time-stamp to differentiate data from different iterations of the loop. One vertex can have multiple input and output sources and the user has choose between them properly.

To measure Naiad's performance the authors used multiple real-world benchmarks such as: batch iterative graph computation, batch iterative machine learning, streaming acyclic, streaming iterative analysis. The results demonstrate that Naiad has very latency in the milliseconds range when scheduling thousands of jobs and achieves throughput of 500K events / second in iterative workers. Consequently the authors managed to make one system for each specialized type of computation that at least similar performance. The only limitation of Naiad is the inability to schedule dynamically streaming jobs which can have an impact on performance.

2.2 Processing Systems

Processing frameworks are systems that designed to ease access and analysis of huge dataset and provide users a simple way to write their own computations over the data. They usually define a specific programming model that restricts the type of applications that people can write. Many different frameworks were developed over the years specialized on typical uses cases for the data. Next we will describe the most important ones.

2.2.1 Map Reduce

MapReduce [1] is a programming model for processing and generating large data introduced in a paper by Dean and Ghemawat. It was designed as effort to hide the complexity of parallelization, fault-tolerance and load balancing in a framework to allow easy utilization of large distributed systems. The abstractions behind *MapReduce* as the authors acknowledged is inspired by map and reduce primitives present in functional programs like Lisp.

The computation takes a set of key/value pairs and returns as output a different set of pairs. Their original implementation passes string within the computation but users can use arbitrary type by providing a serialization function.

The MapReduce framework does the computation by applying two functions provided by the user: *Map* and *Reduce*. The *Map* phase receives as input a list of key/value pairs and produces a set of *intermediate* key/value pairs by applying a user defined function to each element. In the intermediate phase of the computation the framework groups together all items having the same intermediate key (all the values having the same key are grouped into a pair of key, list of values). Finally, the *Reduce* phase applies the second user defined function to each pair of elements from the previous phase. For each key/list of value pair the reduce function has the goal to from a smaller list by combining some of the values. The implementation uses a iterator to process the input, thus is able to process data that doesn't fit in memory at once .

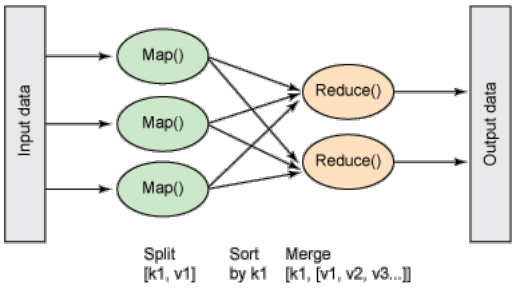

FIGURE 2.6: High-level map reduce diagram.
Source: http://ibm.com/developerworks/

The programming model behind MapReduce is perfectly suited for running on a distributed system build with commodity hardware with high failure rates. Because all instances are evenly distributed across the cluster if one fails it's very easy for the scheduler to restart the corresponding *map* or *reduce* instance without a significant latency to the whole computation. The features presented so far made the MapReduce framework very suitable for running computations on large clusters and the foundation laid by this paper inspired many engineers to build similar system and some of them open-source.

2.2.2 Hadoop

Apache Hadoop [12] consist of a open-source framework for distributed storage and distribute processing. It was created in 2005 by Yahoo's Doug Cutting and Mike Cafarella as a response to Google's Map Reduce one year earlier. At its core it's composed of HDFS (Hadoop Distributed File System) and the computation part (MapReduce). While initially it was used internally by Yahoo, the company decided to make it public in 2009 and one of the biggest early adapters was Facebook which in 2010 claimed to have the biggest Hadoop clusters in the world of 21PB of storage [13]. Later on more components were added to the Hadoop ecosystem like: Apache Pig, Apache Hive, Apache HBase and YARN.

Different schedulers were developed to run on top of Hadoop apart from its default FIFO queue-based scheduling with custom priorities. Facebook created a Fair Scheduler with the aim to provide a low latency for small jobs while assuring high throughput for important jobs. This works by splitting jobs into pools each having a guaranteed minimum share of processing time. Yahoo developed a Capacity Scheduler with a similar design but it allows jobs to access resources based on their level of priority and once a jobs started there is no preemption.

Hadoop was for many years a great success and used at almost every company that handles huge amounts of data. There was a very active community of developers extended Hadoop and tried to make it the *the* framework for everything related to big data. This was perhaps both the cause of its great success and curse. After 7 years of its release a new generation of Hadoop was created with a complete redesign from scratch.

2.2.3 Spark

With the increase in popularity of Hadoop and MapReduce many of their limitations showed up. One of those greatly affects the performance of iterative algorithms because MapReduce stores all the intermediate data on disk to guarantee fault-tolerance. Spark [23] focuses on solving these issues by storing the applications working set in memory while retaining the scalability and fault-tolerance of MapReduce. To achieve these goals the authors introduce *Resilient Distributed Dataset* RDDs: immutable dataset partitioned across several machines with can be recovered based on replicated lineage applied to saved checkpoints. Lineage is simply a transaction log over the data which is replicated in memory. To reduce the workload needed to recompute the RDDs in case of failure Spark writes periodically checkpoints to disk. They achieve high parallel throughput over the data since all the RDDs are immutable and can be written in parallel. One key advantage of RDDs is the ability to degrade gradually when when the data grows over the size of available RAM.

Spark provided a rich semantics for accessing data by applying functional programming primitives such as: map, filter, join over the dataset. Each of these transformations create a new dataset. To provide easy-access to data they developed an Spark interpreter that applies transformations over the data and return the results. Because of that the development process of Spark queries is greatly simplified since one can test sequentially every transformation.

Their performance is several orders of magnitude faster than MapReduce because of in memory access speed. Spark is widely-used for iterative machine learning or interactive data analysis. However not all computations are suitable for Spark since we are limited to logic that can be represented as simple transformation on RDDs. Further we are also limited by the cluster's total memory size. If we exceeded it the performance downgrades to a traditional MapReduce approach. Finally Spark is not suited for applications that apply fine-grained transformations over the datataset since RDDs are immutable and just one value change brings the overhead of creating a new RDD.

2.2.4 Storm

The increasing use-case of stream processing running in real time motivated Nathan Marz et al. to develop Storm [21, 22], a real time distributed computational framework that makes it easy to process unbounded streams of data efficiently. Storm received a lot of interest from industry and was initially acquired by Twitter and in September 2014 open-sourced under Apache license. Today the framework is a key part of backend infrastructure in companies such as Twitter, Yelp, Spotify or RocketFuel. Being able to process streams in real-time Storm has many use cases: real-time analytics, fraud detection, weather prediction, real-time machine learning or music recommendations.

A Storm cluster is similar in a higher level with a Haddop cluster but instead of running batch jobs with a finite completion time you run stream operator called *topologies* that simply run forever. As Hadoop Storm use master and worker nodes and the latter runs a daemon called "Nimbus" which is equivalent to Hadoop's "JobTracker". *Nimbus* performs scheduling, tasks deployment and monitors for failure. A distinguished feature of Storm is to use a separate cluster *Zookeeper*, a distributed synchronization service, to coordinate each individual node and the central scheduler. Zookeeper remember the state history which allows the rest of the Storm system to be fail-fast and

stateless allowing instant recovery without loosing computation. This design makes Storm very reliable and Twitter claims they have clusters running in production for months without the need to be restated.

For tasks placement Storm uses Mesos [14, 15] configured to run tasks in resource strict isolation environment. Because of that they don't encounter bottlenecks due to resource contention but this comes with the cost of wasting resource when the cluster is not fully utilized. Further more they don't provide any kind of I/O isolation and have encountered I/O bottlenecks in production such as tasks under-performing when consuming data from Kafka [26] relative an independent client.

Storm was adopted with great success by many practitioners and it widely used in comapnies such as Twitter, Yahoo, Rocket Fuel or Groupon. The main disadvantage of Storm to our use case is that they don't support dynamical scheduling: once a tasks if placed on a node is never moved. The reason above as well as their inefficient resource allocations motivate us to develop a new scheduler.

2.2.5 Samza

For similar reasons such in the case of Storm 2.2.4 has motivated the people at Linkedin to create Samza [11], a framework with similar design as Storm. Both systems provide many of the same high-level features such as distributed computation, fault tolerance and the same stream programming model. Even though Samza is more recent than Storm it was open-sourced a few months before the latter under the same Apache license. The release was anticipated with great enthusiasm by the community since it aimed to be a near-realtime, asynchronous computational framework for stream processing and provided a few different features than Storm that would better suit specific applications such as guarantees for delivery of messages and always processing them in order.

In terms of architecture the framework uses Kafka, a publish-subscribe message broker, for communication and it's built on top of YARN to provide fault tolerance, processor isolation, security, and resource management. For computation state management they use a very different approach than Storm. Instead of keeping a external database cluster like Zookeeper Samza tasks include an embedded database on each machine. This has the advantage of very fast read and writes since everything its locally in memory. To be able to fail-fast this database is replicated on certain machines that can restore the computation of failed node.

In practice the project reached its limitations when dealing with high dynamic workloads of streams because of YARN resource allocation strategy. In the community there are many ongoing discussions on how to solve or circumvent these issues but no suitable solution was discovered yet. Considering that Samza is fairly new and the project is quite immature maybe new features will develop in the future but YARN limitations are difficult to surpass without a whole change of the programming model. The project is still under active development recently by Cloudtera and Hortonworks. After exploring those alternatives we clearly outlined why the problem of dynamically scheduling stream jobs is still open-ended.

2.2.6 Spark Streaming

Real-time streaming frameworks suffer have an expensive way to deal with *failures* or *slow nodes* through *replication* or *upstream backup*. Both approaches are not desirable as *replication* increases the hardware requirements and *upstream backups* can take a long time to recover in complex computations. The authors propose a new processing model, *discretized streams* [43] that solves this challenges. Their idea is to structure a streaming computations as a set of fined-grained batch computations that can be ran as MapReduce jobs. For example instead or running a continuously word count over streaming data they will divide it into segments and run a batch word count for each. On problem with a classical MapReduce approach is the high-latency from disk access so they use instead RDDs [23].

DStreams are formed by splitting streaming computations in small time intervals. The data received in each interval is stored in memory with replication and when the interval ended the dataset is processed via functional transformations. All the intermediate states are stored as RDDs, thus avoiding data unnecessary replication. RDDs can be recovered in case of failure based on lineage: graph of transformations applied over the input to compute the data. When a streaming node fails the other nodes can replicate the computation in parallel from the lineage. For slow nodes the system can execute tasks speculatively for a fine-grained interval since *DStream* are deterministic and stateless.

While building Spark Streaming the authors changed the underlying Spark engine with multiple enchantments to optimize it for streaming tasks. First they changed the communication layer to be asynchronous so reduce processing latency. Secondly they modified the scheduler to launch many parallel jobs at milliseconds latency and made it schedule tasks for the next timestamp even before the current one finished to reduce latency between timestamps. Finally the made their scheduler system fault-tolerant since streaming jobs need to run for many hours.

To evaluate Spark they used three simple benchmarks: Grep which finds the number of strings matching a pattern, Word Count which computes word frequency, Top K which find the most common words over a sliding time interval. They ran those benchmarks on Storm [21] and S4: other commercially available stream processing systems. Spark achieves a significant performance gain of 2-3X when compared to Storm on 1,000 bytes records and 5-6x on smaller records because of fast memory access speed. Next they evaluated fault-tolerance and slow nodes mitigation. For recovery they managed to reach a ratio of 7:1 between checkpoint time and recovery while running on a cluster of 100 nodes. When a node is slowed down by a factor of 6x they manage to reduce the performance loss to only 2x by speculative execution.

Spark Streaming provides a novel approach to managing failures in streaming frameworks. However their abstraction of *DStream* is not general enough to apply to SEEP since some streaming computations are statefull throughput the computations if interactive used input or database access is performed. However their techniques for handling slow nodes can be adapted to SEEP to improve the performance. Finally we used their benchmarks to compare our system to both Spark and Storm. [43]

2.2.7 Heron

Storm has served for several years as the main stream processing framework at Twitter as well as other companies. As the scale of data being processed increased as well as the size of the cluster many of Storm limitations became visible. One of the biggest issues while running Storm in production was it's maintainability and debug ability. Storm needs special provisioned hardware, with prevents sharing and makes scaling elastically cumbersome since the machines need to be provisioned. Secondly Storm uses an abstraction over the physical layer which makes it impossible for engineers to map physical operators to machines. Lastly to allocate resources Storm treats all workers homogeneously which results in a very inefficient use of resources. For the reasons above the people at Twitter needed a new stream processing system and they made Heron [39], launched very recently in June 2015.

Heron runs streaming queries organized in topologies forming a DAG. When a query is submitted each operator is allocated in a container by an external scheduler based resource availability in the cluster. The main advantage in their design is the ability to use any external scheduler like YARN [16] or Aurora [41]. Heron uses the concept of *topology backpressure* that can dynamical adjust the information flow within a topology. This is required as some operators can process data faster than other thus potentially wasting disk space and resources. To achieve that they considered a few implementations strategies: *TCP Backpressure*, *Stage-by-Stage Backpressure* and *Spout Backpressure*. The last approach notifies the upstream operators or data sources connected to the over-performing operators to reduce the flow of data. While having a small overhead due to message passing this strategy adapts quickly to big topologies. The *backpressure* mechanism is very useful in reducing resource utilization and would be nice to implement a similar concept in SEEP in the future.

When developing Heron many useful lessons where learned from Storm the current design doesn't have any single point of failure. To achieve that they run a quorum of Topology Manager (TM) where only one is acting at a given time while the others have the roll of ready hot-spot nodes. To elect the acting TM they make use of Zookeeper [29] *ephemeral, sequential* nodes. The TM is responsible to monitor each worker process via periodical heartbeats and make sure they are running.

To evaluate Heron performance the authors extended each container with a *Metrics Manager* (MN) unit responsible to collect send all the metrics to an external analytics system. The performance benchmarks were done by direclty comparing Storm and Heron on a representative set of benchmark. We tested on a word count topology from tweets Heron achieved a 5-15X lower latency and 2-3X lower CPU utilization when compared to Storm. Furthermore while measuring resources required to reach a 6M/min throughput Storm needed 240 cores while Heron managed to use only 20 cores. The new state of the art system clearly outperforms Storm [21] as well as other systems build around the same time such as Samza [11] and Spark Streaming [23]. They provide as well a mechanism to adjust dynamically to differences of throughput but this come with the cost of limiting the performance. Because of that there is a genuine need for a scheduler that can migrate tasks when the workload changes.

2.2.8 SEEP

As amounts of data increase and more real time applications make use of it, new types of stream processing systems were developed that are designed to *scale out* to large number of machines. Those systems face two challenges: they should scale out on demand benefiting of the new resources available for cloud computing like Amazon EC2 and because those systems run hundreds of machines of commodity hardware they need to be *fault-tolerant* without adding a big overhead per machine. The difficulty in solving these comes from the fact that stream processing operators are *stateful* hence any movement can affect query results. This motivated a group of researchers from Imperial to build SEEP [6, 8], a stream processing framework designed with the above ideas in mind that uses a new type of operator state management.

SEEP works by exposing internal operator state to the stream processing system (SPS) by defining a set of state management primitives. These allows the SPS to periodically checkpoint operators state and back them up to upstream VMs. In case of failure the SPS would simply restore the operator from the last checkpoint without losing significant computation time. Even more the SPS can identify operator bottlenecks and allocate new VMs each taking a portion of the original workload and replay any any unprocessed tuples from the last checkpoint. The authors evaluated SEEP with the *Liner Road Benchmark* on Amazon EC2 cluster and the results showed that it can scale automatically and recover quickly to a size up to 50VMs. In future work they plan to extend SEEP to scale out more quickly and add support for scale in.

We used SEEP because is written inside our department and we can collaborate very easily people working on it. Secondly SEEP offers us exactly the set of features that we would need to build our scheduler on top of a stream processing network and has the benefit of running locally and small setup overhead.

2.3 Messaging Systems

Practitioners have created specialized frameworks to allow communication between different applications within the same cluster or even different data centers. Their job is to receive messages from applications, transform them to a universal format and route them to their destination. Their are usually designed following the publish-subscribe patter which provides a layer of abstraction over the communication links in the network thus it results to greater network scalability and supports more dynamic topology. The design of those frameworks has to trade-off between persistent storage of messages and system performance. Developers have created a broad range of messaging system each with its own strengths and weaknesses, that span from low-level not-persistent ones like Google's Protocol Buffers [30] and Facebook's Thrift [27] to complex message brokers like Rabbit MQ [25] and Apache Kafka kafka. In this section we will outline the characteristics of each one and argument why we chose Kafka for our system.

2.3.1 Rabbit MQ

RabbitMQ [25] is a messaging system created in 2007 with the aim of providing a open-source alternative to existing messaging middleware, offer reliable communication and scale out to large

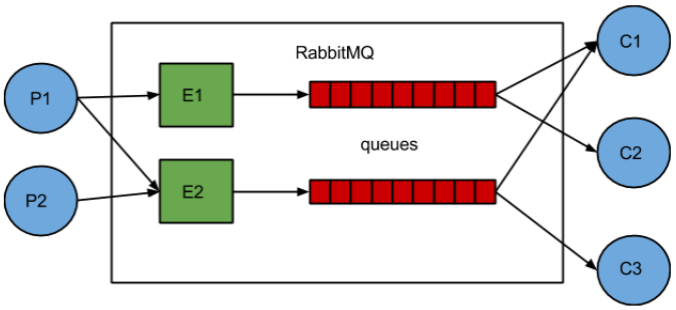

FIGURE 2.7: RabbitMQ architecture overview

computing cluster of the future. It was developed under strong collaboration of a group of users and vendors. The system it written in Erlang, a language very well suited for programming communication systems, and follows a direct implementation of the *Advanced Message Queuing Protocol (AMQP)* which was developed in 2003 in the finance sector.

RabbitMQ system is composed of the following components described bellow:

- **Producers**: programs that create messages and send/publish them.
- **Consumers**: receives messages from the queue that are sent to them
- **Queues**: communication layer used to store messages temporarily and buffer them if necessary. A queue can transport messages from multiple producers that are designated for multiple consumers. Queues are ordered, stateful and can be persistent by writing data to disk. RabbitMQ allows queues to private or shared. Consumers tell queues to bind for certain keys.
- **Exchanges**: stateless routing tables that maps messages received from producers to corresponding queues where the consumers are listening. They mapping is done by checking the message routing key which binds to a certain queue. Producers send initially one or more routing key to exchanges.

The system is coordinated by a RabbitMQ broker which runs distributed on a set of nodes and has the responsibility to maintain the queues, exchanges and store routing information. If the broker runs on a single node the system is vulnerable to failure. Even if persistent queues are used to store messages after recovery from failure buffered messages may still be lost. If the system runs on multiple nodes it can tolerate failure very well if in the worst case all nodes but one fail. This is achieved my mirroring queues on multiple nodes such that each message is backed up on machines that can send it again. One weakness of RabbitMQ is vulnerability to network partition, i.e. broker nodes cannot replicate in case of failure messages from different partitions.

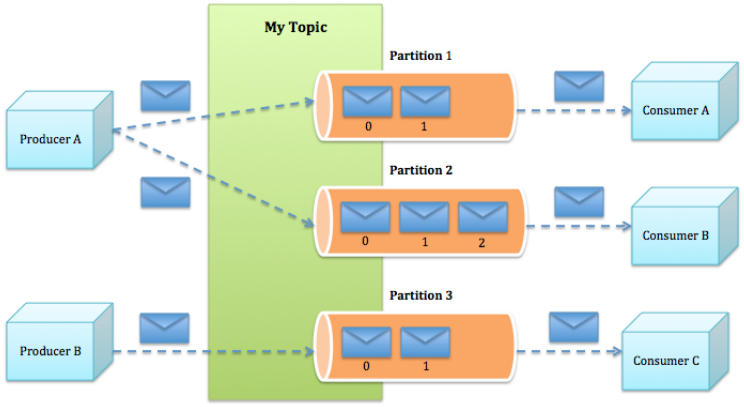

FIGURE 2.8: Kafka architecture overview.
Source: http://kafka.apache.org/

2.3.2 KAFKA

As the number of people using the internet increased companies wanted to understand better their users, track engagement or analyze user activity of all sorts. To achieve these logging became a critical component of every internet company and large amounts of log data began to gather in data warehouses. This motivated people at Linkedin to create Kafka [26], a distributed publish-subscribe messaging system developed for collecting and consuming large amounts of log data. Kafka was designed to provide high throughput (i.e. hundreds of thousands events per second), persistent message storage, fault-tolerance with replication, build-in partitioning and distributed delivery.

Kafka internal design is a bit simpler than RabbitMQ. It has three main components: producers, consumers and Kafka brokers. Producers publish messages to a topic which and then routed to the appropriate consumer group by the broker. Consumers can be part of a group and distribute the load evenly among them or we can allocate just one consumer per group to send a message to only one of them. For replication Kafka makes extensive use of the file-system storing the messages on local disk and it manages to achieve that without an impact on performance by serializing and combining writes in a batch. The authors argument their decision of using the file-system instead of keeping messages in memory by the big overhead oh Java and its garbage collector poor performance.

RabbitMQ and Kafka are similar systems that both serve the goal of providing communication in a large computing cluster. Here we will highlight some of the differences between the frameworks:

- **Scalability**: Kafka easily scales out to 100k messages per second while RabbitMQ cannot handle more than 20k. This would quickly become a performance bottleneck in a huge cluster.
- **Ordering**: Kafka delivers messages in order while RabbitMQ cannot guarantee that. The first would ease the design of our system.
- **Routing**: RabbitMQ provides complex routing strategies and message delivery guarantees. We are fine with best effort delivery and simple routing policies.

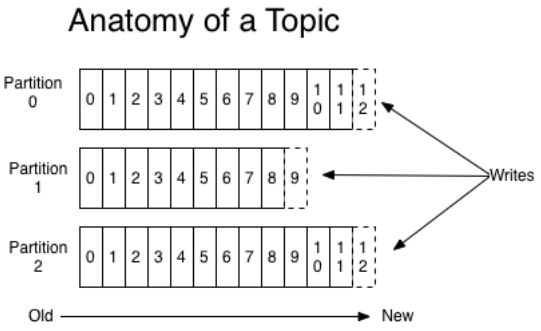

FIGURE 2.9: Anatomy of a Kafka Topic
Source: http://kafka.apache.org/

- **Maturity**: RabbitMQ is more mature while Kafka was open-sourced for just one year but has a pretty stable release and very good documentation.
- **Programming language**: Kafka is written in Java the same as SEEP so integration with our system will be very easy where as for RabbitMQ we need to use their Java API.
- **Synchronization**: RabbitMQ build its own mechanism of synchronization but Kafka relies on Zookeeper for coordination which represent an additional overhead but it's worth it.

Topic The main concept introduced by Kafka is a topic: a communication channel where producers can write and consumers can subscribe in order to receive the published messages. All the messages exchanged through the system are divided into topics. For persistence Kafka stores all the messages on disk partitioned into several files stored on different machines. Each file is an immutable sequence of messages that is continuously appended to. Each message has assigned an offset which represent it's position within a partition.

Kafka has a log retention policy that retain all published messages for a given time whenever of not they have been consumed which could happen multiple times. Because storing a new message is a simple disk seek to the latest position in that log file, Kafka performance is not affected by storing large amounts of data. However when using Kafka with SEEP we can delete a message as soon as it was consumed which usually happens in less than a minute. This allows running intensive IO queries that can write around 1G of data per second cluster wide without a significant impact of the disk space usage.

Storing the messages in as a ordered sequence means that Kafka consumers have a lot of flexibility while reading the data. A consumer can come and go without affecting the others and it can read that data in any order it likes or even reset it's reading offset if it wants to reprocess the data. The user can also inspect in real-time the messages that are exchanged in the system by running tail on the log file.

Partitions Partitions serve a very important rule in Kafka by allowing a topic to store data beyond the size that will fit on a single machine but also represent a unit of parallelism. Multiple

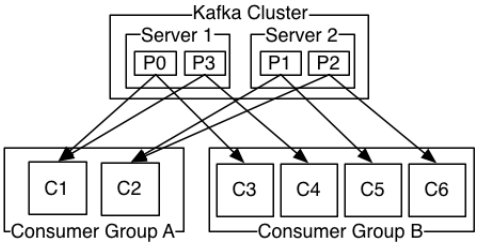

FIGURE 2.10: Mapping partitions to consumers
Source: http://kafka.apache.org/

producers can write concurrently to a topic because different machines can do disk IO for each partition.

For fault-tolerance each partition is replicated to a configured number of machines. From all the replicas one is the leader and processes all the read and writes while other are followers and simply mimic all the data operations. If the leader fails one of the followers will automatically become a new leader. Each server is assigned a even number of leader and followers rule by using consistent hashing so the load is well balanced across the cluster.

One common problems of systems that deal with multiple consumers sharing the load from message queues is that even though messages are stored in order the system delivers the messages asynchronously which means that multiple messages can arrive out of order to multiple consumers. Messaging systems usually deal with this by limiting parallelism to only one consumer at a time. Kafka has a innovative approach by introducing consumer groups and message partitions and they create a mapping from partition to consumer within the group so a consumer receives messages only from one partition. This way load balancing between consumers comes from the same properties inherited from partitions, see figure 2.10.

Guarantees At a high level Kafka provides message ordering guarantees: all messages will be stored in the order they were sent and all consumers will see the messages in that order, fault tolerance: for a topic with a replication factor of N the system will be able to support up to N-1 failures without loosing any data written to the log.

2.4 Resource isolation

Large clusters have to share their resources: physical machines, cpu and memory among many applications running concurrently. The developer of applications that run inside the cluster can't be made responsible for writing safe and fair code, i.e. won't crush the system, won't fill up all machine's memory. Considering the above there is a need to isolate tasks sharing the same cluster by using a multitenant architecture. To achieve that we can deploy each job inside a container, similar to a light weight virtual machine that had access to a fair share of host machine physical resources, or use a cluster management framework that creates a abstraction over the physical layer, like Mesos 2.1.5 and YARN 2.1.6. The latter relieves the overhead of keeping track of machines

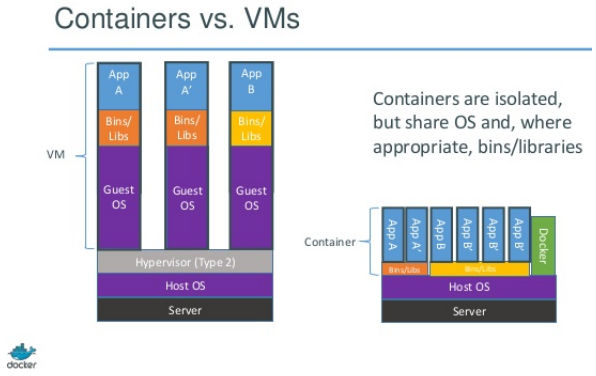

FIGURE 2.11: Docker and VMs comparison.
Source: http://docker.com/whatisdocker/

from the scheduler but this come with the cost of lower granularity for task placement which is serious performance bottleneck in real-time systems. In the following paragraphs we will describe some of the kernel level resource isolation platforms.

2.4.1 Linux Containers

LXC [20] is an operating system level virtualization environment for running multiple isolated containers on a single physical machine. They achieve thease by creating process containers know as *cgroups* that completely isolate application view of operating system including file system, network, cpu and memory. Each container runs a complete copy of the Linux operating system without the overhead of a traditional virtual machine hypervisor. We consider using LXC or other alternatives for resource isolation however because they are very low-level they create an overhead on the scheduler to manage them.

2.4.2 Docker

Docker [19] is an open platform build with the aim of making easier to build, deploy and run distributed applications. Solomon Hykes developed Docker with the aim of making it more user-friendly than LXC by providing an additional layer of abstraction and automation over an operating system virtualization. Internally it relies also on *cgroups* and combines the usage of virtualization interfaces like LXC and its own library *libcontainer* to directly access Linux's kernel virtualization facilities. The platform can be integrated very easily in various infrastructure deployment systems like: Puppet, Jenkins, Vagrant or Google Cloud Platform and it's integrated in the 2015 release of Windows Server. The factors outlined above along with its very big online community makes it a good candidate for integration in our system. However after examining all these options we decided that YARN fist the best with SEEP and the latest version offers all levels of isolation that we can get from a low level framework but without the additional overhead.

2.5 Performance Benchmarks

To measure the performance of large computing clusters one requires representative benchmarks modelled after realistic workloads the system is designed for. Over the years people build various benchmarks to measure resource utilization for CPU, memory, I/O, network as well as cluster throughput [46]. Next we will describe one evaluation approach developed by Google that supports complex tasks placement constraints [40].

As cluster size increases it is inevitable to have heterogeneous machines with different hardware characteristics. As some applications are dependent on kernel versions or make performance trade-offs for disk, memory or CPU it is important to support specific constraints when scheduling tasks. When constraints are present it is complicated to evaluate generally enough the performance of the system because: there is a many-to-many mapping between applications and machines and machine utilization is not adequate to quantify performance. (Some machine might be always free because no applications can run on it but this doesn't mean the scheduler is not efficient in load distribution). The authors from Google propose a new solution that uses both common metrics: *task scheduling delay*, machine resource utilization and introduces a new one: *utilization multiplier*. Utilization Multiplier is defined as the ratio of resource utilization seen by tasks sharing the same constrain to the average utilization of that resource. This way the manage to quantify tasks constraints in performance benchmarks.

Their work provides as well as previous research in this area give a lot of insight on measuring cluster performance that we applied ourselves to evaluate our scheduler.

2.6 Task Migration

With the increasing demand of large cluster computations a lot of research interest was focused on load-balancing the workload across physical nodes. Researchers studied ways to maximize energy efficiency [49], optimize placement module based on theoretical constraints [47] or efficient load distribution among N-processor systems [48]. However no existing solution tried to distributed load across multiple machines that form a computing cluster.

A similar problem but at a smaller scale was presented in [47]. The author used a set of matrix to module allocation constraints between modules and processors such that a module could exhibit different communication and resource usage depending on the processor where is placed. He explained why finding the optimal migration is a NP-complete and presented a heuristic algorithm Match-maker to find a sub-optimal allocation while comparing it with other approaches. For evaluation he simulated various workloads and constraints by generating random modules. We used a similar approach to migrate tasks in our cluster with the difference that we run on an actual cluster were tasks could also change their resource usage.

2.7 Conclusion

We explored some of the existing processing systems and job schedulers and understood the specific purpose each one was designed for. Even though some cluster managers like Mesos and Yarn can handle both *batch* and *stream* workloads we have seen that even processing systems designed for this purpose build on top of them have poor performance in practice [1]. This due to the high level abstraction over the physical layer which makes all tasks heterogeneous combined with the schedulers inability to reschedule task during runtime. We want to solve this problem by creating a new scheduler that can handle streaming workloads efficiently.

[1] see Samza 2.2.5 and https://issues.apache.org/jira/browse/SAMZA-335

Chapter 3

System Design

In this chapter we will explain how and why we need to extend SEEP in order to enable queries to run independently on a multi-tenant cluster. First we need to make queries run under resource isolation in order to prevent misbehaving queries to affect other tenants running on the same machine. Secondly we want to make the streaming communication fault tolerant so a single query will not be affected permanently when one of its operators stops and resumes computation when a replacement appears. We will explain in detail how we integrated SEEP with other systems to achieve the requirements above and analyze the overhead we introduce if any.

3.1 Seep overview

Seep is a stream processing framework written in Java that aims to provide real-time data processing in the cloud and has the ability to scale on demand by increasing the number of workers. The programming model is inspired by functional programming primitives if we consider each operator a lambda function applied over a infinite stream of data. A typical SEEP workload is formed by a set of operators applied over a set of streaming data . An simple query can be a real-time word frequency application as we illustrate in Figure 3.1. This query would analyse online data such as real-time Twitter statuses as they are published. This query can be formed of three operators: Source, Processor and Sink, where each runs according to the following logic

- Source: reads messages from Twitter Streaming API and sends them the the Processor in the desired format
- Processor: split sentences into words and keep track of number of occurrences for each word. After a given time interval it send the list of the most frequent words to Sink.
- Sink: outputs the results in a user defined format for future analysis.

SEEP provides a very simple way to write streaming queries where the user only needs to specify how the operators are connected and implement the function that transforms the data for each operator. This is similar to the programming model used by MapReduce where the user would needs

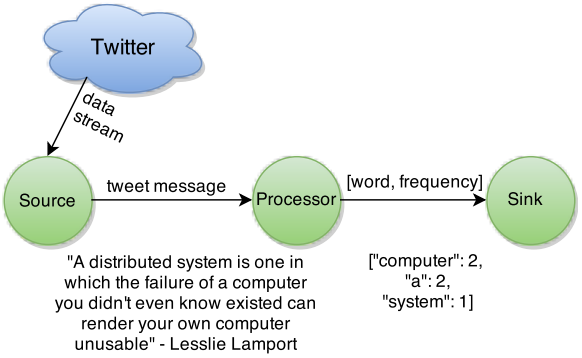

FIGURE 3.1: Twitter Word Frequency Query

to specify the number of workers and then implement the Map and Reduce operators. However, SEEP is far more flexible and can support arbitrary complex queries that are both statefull and stateless and provides rich semantics for connecting operators as a Directed Acyclic Graph (DAG).

At a high level SEEP can be divided in two components:

- Master: coordinates the worker nodes via a set of simple command primitives. In the initial phase it is responsible for sending the query over the network to each worker and mapping a physical worker process to a logical operator from the query that will run on it.
- Workers: are standalone processes that perform the computation and can serve the role of one of the operators forming the query at a time.

3.2 Message passing

There are two different programming models that application can use to achieve inter-process communication: shared memory and message passing. The first is used mostly between different threads and can be also applied to different processes as long as they are running on the same machine. Therefore the only way to communicate between different tasks running in a cluster is via message passing over the network in the form of socket channels or HTTP requests. The latter needs to establish a connection before sending any data, and this constitutes a substantial overhead. However, this overhead becomes, insignificant if persistent connection are used. One major limitation with of HTTP is that it uses the "pull" diagram which means the upstream operator would need to keep asking for data from the downstream instead of receiving it directly. This lays unnecessary burden on the sender. For the reason above HTTP requests are not suitable to use for recurring communication such as streaming data.

3.2.1 SEEP communication

SEEP makes extensive use of message passing between the master and workers nodes and between workers themselves. The latter is required to pass streaming data from one operator's output

(upstream) to the next operator input (downstream). The pipeline uses multiple direct socket connection for each type of operation so the master worker communication is separated from the actual data exchange. Although fast, sockets represent a big weakness in the design: if one node fails all the data buffered on the channels is lost and other nodes connected to it are unable to deliver data and progress halts.

3.2.2 Alternatives

To be able to migrate operators from one node to another we needed an abstraction over the communication layer which has build-in fault tolerance - a first class citizen in any distributed system. Some of the classics options used with great success by engineers are distributed file systems such as Hadoop Distributed File Systems (HDFS) or Amazon S3 however such systems are designed for long term storage and their performance is not ideal in terms of latency and throughput. Another options would be distributed database such as Dynamo or Casandra. Dynamo was designed to have a very high throughput and high availability for writes with the trades-off of weak consistency and higher latency on reads. Weaker consistency is not a concern in our use-case since once we write some data we never touch it again but read latency is quite important and extra latency could have a domino effect on the overall throughput of multiple operators chained together. Casandra is a NoSQL database that offers asynchronous master-less replication and promises low latency for all clients. Casandra has a very good performance for a database and manage reaches 1 million writes per second on a cluster formed of over 200 Amazon EC2 instances. [33] Even if such results look good it is possible to reach significantly higher performances by using a more simple system as we don't benefit in SEEP from all the extra functionality provided by complex databases or the high replication of a distributed filesystem.

Another option is a distributed message broker, a system that handles message passing among different nodes. Message brokers are designed mainly to write or retrieve data by using simple designs such as the publish-subscribe messaging pattern. This allows the producers to send messages without being aware of receivers at the other end of the wire. To achieve this messages are divided into classes, known as topics where clients can subscribe and become producers or consumers of the messages from a class. Additionally most popular systems that implement a message broker achieve fault-tolerance via message storage replication.

3.2.3 Message brokers

After exploring several alternatives we decided that the best approach would be to use a general purpose message broker because it has very low latency and performance is second to none as it can easily handle 100K+/sec with just a few machines. Two of the most popular alternatives used by data intensive companies are Kafka and RabitMQ. RabitMQ is an open source implementation of the Advanced Message Queuing Protocol (AMQP) and provides rich semantics for filtering messages, complex routing policies and delivery guarantees. Kafka is more producer-focused and performance-oriented being able to handle a firehouse of events partitioned and replicated that are served to both offline and online consumers. In SEEP we don't need anything more complex for message passing than basic topics which represent a unique communication channel between two operators. With this reason in mind the choice between the two systems above relies solely on

message passing throughput which is tightly coupled to SEEP's query performance. By analyzing existing benchmarks we observed that Kafka can handle more than 100K+/sec messages with just one single producer/consumer pair while RabitMQ can handle 20K/sec as the performance is slightly affected by the richer semantics. An interesting performance evaluation was conducted by the engineers at Linkedin which managed to reach 2 Million writes per second with a Kafka cluster composed of just 6 machines with common hardware [32]. Kafka is written in both Scala and Java thus sharing the same language as SEEP which makes integration very easy.

3.2.4 Kafka

Kafka is a publish-subscribe messaging system designed as a distributed commit log where all messages are appended to log files evenly stored across the cluster. This system is very well suited as a communication layer for SEEP because it is:

- **Fast**: a single host can handle more than 100K read/writes per second from thousands of clients.
- **Scalable**: their design allows to elastically scale the system by simply adding more worker machines and all the data is partitioned evenly across the machines evenly to support much higher loads than the capability of any host.
- **Fault tolerance**: all the data is replicated on at least two machines and is always available to be read once it was written.
- **Ease of use**: Kafka has a simple API and the user needs to implement only a Producer and Consumer to handle all the communication. It can be easy configured to clean messages from time to time and prevent wasting disk space waste. Furthermore the deployment is very fast and lightweight as only one process needs to be run per host.

3.2.5 Integration

SEEP manages input and output via classes that extend the *InputAdapter* and *OutputAdapter* interfaces respectively and had already a Network Input and Output implementation supported that was reading and writing data on socket byte channel shared between two operators that are connected. The implementation can be divided in three parts.

The first step was to implement a KafkaConsumer and KafkaProducer that make use of their public API to receive and publish data. The Producer is very simple to implement by instantiating a Producer object that connects to any available Kafka server. Next we can call the send function that takes a KafkaRecord which is a structure that contains the message, topic, optional partition and optional key. The consumer implementation is a little more complex since consumers are blocking so the main loop which waits for messages to be published needs to run in a separate thread. It is easy to make a consumer subscribe to multiple topics by running a consuming thread for each.

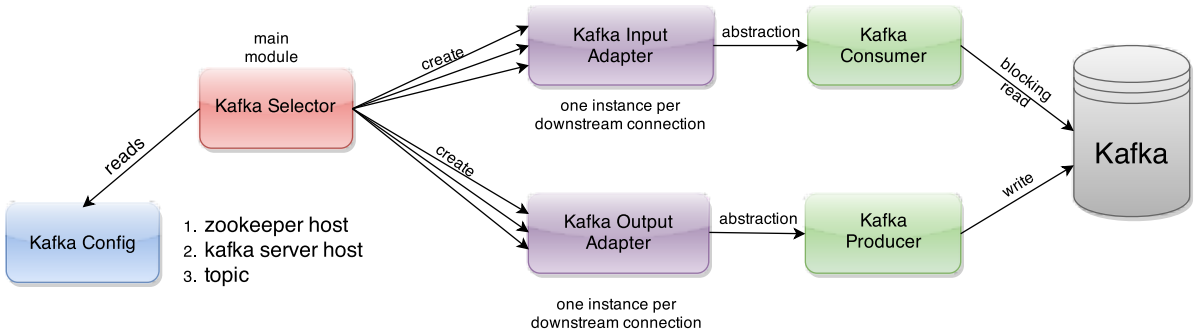

FIGURE 3.2: Kafka Integration Overview

Next we simply create two classes that extend *InputAdapter* and *OutputAdapter* interfaces for Kafka which call the provided KafkaConsumer and KafkaProducer to publish and receive data. Finally because SEEP is designed to receive and send data to multiple sources at the same time we implemented a separate class which reads and writes data asynchronously for each input and output connection present in Seep Worker. To achieve this, we had to change the initial multi-threaded implementation of a KafkaConsumer and put all the logic inside the Kafka Selector since it was already running asynchronously and this way we don't create redundant threads. The following is a summary of all the classes implemented to integrate Kafka with SEEP and Figure 3.2 shows the interactions between those classes.

- **Kafka Selector**: reads and writes data from multiple sources asynchronously using the underline Input and Output Adapters for each source.
- **Kafka Input Adapter**: SEEP class that provided a layer of abstraction over the input type.
- **Kafka Output Adapter**: SEEP class that provided a layer of abstraction over the output type.
- **Kafka Producer**: implements all the logic to publish a KafkaRecord.
- **Kafka Consumer**: simple blocking Consumer that reads data from a given topic.
- **Kafka Config**: configuration file that needs to be passed when running Kafka queries. The only required properties are the Zookeeper host and one Kafka Server host. Note that the server to which the producer/consumer initially connect is not necessarily storing the partition for their topic so the connection might be transferred to another server later on to reduce latency.

3.2.6 Performance Overhead

Kafka integration brought many benefits to SEEP such as fault tolerance which enables operators to migrate from one machine to another without data loss however this could also come with a

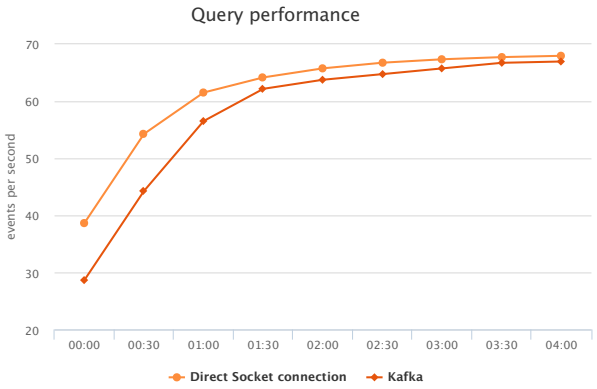

FIGURE 3.3: Seep Query performance comparison

performance overhead. Before SEEP used direct socket connection between two operators while now the data is written to Kafka from one while the next waits for the data to be delivered from Kafka. In the process the data is also written to disk which is slower than the network. To analyze the performance overhead we used a simple query composed of two operators which transfers 1MB chunks of data on every iteration from Source to Sink. We ran this query a couple of times by using both Kafka and Socket as communication layers. The results can be seen in the graph above where we measured the number of events per second received by the Sink as a 1-minute rolling average, which is actually the data transfer speed in MB per second. As we can see the performance impact is small in the first minute and after that is less than 5% which is very good.

3.3 Resource isolation

A multitenant cluster usually runs multiple applications with different resource requirements and it needs to ensure the overall throughput and fairness for all of them. To achieve this it can divide available resources between applications by creating a layer of abstraction known as container which isolates the rest of the resources available on a host from the application running inside it. Without isolation we can have a scenario where a memory starving application consumes all the available RAM memory and forces all other applications sharing the node to use swap, therefore downgrading performance exponentially. The same scenario applies to the CPU although to a lesser extend because the kernel also ensures a fair allocation of computation time to threads making it unlikely that one can starve others. However CPU isolation is still important in the case of a faulty/malicious applications that spawn a huge number or threads or sets themselves a higher priority.

Two of the popular solutions for resource isolation widely used by engineers are Mesos and YARN (Yet Another Resource Negotiator). Both of these have an opensource implementation and are quite mature projects, having been used for a couple of years by the community. Both YARN and Mesos provide memory and CPU isolation, the latter being added to yarn very recently. At their core they have the notion of *container*: a sandbox where an user application can run in isolation. YARN has a very simple API of requesting containers: the user needs to specify the application path and the amount of memory and CPU required. After that YARN scheduler will honor the

requests if resources are available in the cluster and start the application. Mesos is on the other hand works quite differently: the user receives "resource offers" from the scheduler and can choose to accept or decline them based on their needs. Both of these models work for SEEP but the latter would require a little extra work. Unfortunately disk usage, I/O and network cannot be isolated yet in any of those frameworks but is currently planned for future releases.

3.3.1 Locality

One key requirement for scheduling is the ability to pin SEEP workers on a certain machine based on an resource utilization so the resource manger needs to handle request locality. In YARN the user can easily override the relaxed locality default where YARN scheduler performs the allocations and asks for a specific host on each container request. Then the resource manager will honor the request as long as resources are still available on that machine. In contrast Mesos doesn't support that out of the box and instead the user has to implement their own mechanism. Further more YARN has numerous security features which allows nodes to communicate over a non-secure network by proving their identity. This prevents a malicious application to request resource from the resource manager and possibly flood the system. Based on the arguments above we decided that YARN is the best option to use on our cluster and will work quite easily with an external scheduler.

3.3.2 YARN Cluster Setup

As we explained in section 2.1.6 YARN it's composed of two main components: the Resource Manager which is responsible for coordinating all container allocation and available resources, and Node Manager which manages locally container execution. A typical YARN cluster needs one dedicated host to run the Resource Manager while the Node Manager needs to run on every node that will run queries. The deployment can be configured from over 150 individual properties, each of which is well documented, but for a typical setup the user has only to configure a few of them, such as: the nodemanager host and name of all the hosts.

Applications that run on YARN need to implement two main classes: Application Submission Client and Application Master.

3.3.3 Application Submission Client

This class is responsible for submitting the application, in our case the SEEP query, to the YARN ResourceManager (RM). To achieve this, it has to prepare the very first container of the query which will run the ApplicationMaster (AM)- the SEEP master in our case. The Submission Client needs to describe every detail that is needed to launch the AM to YARN such as, the command to be executed, command line arguments and OS environment settings. During execution, the Resource Manager will ensure the allowances are respected. For memory, the limit is hard by default, which means that the RM will simply kill a container once it exceeds its allowance. In contrast, for CPU the limit is soft, which means that while there are resources available, a container is free to use more computing power, making jobs finish faster on a not fully utilized cluster.

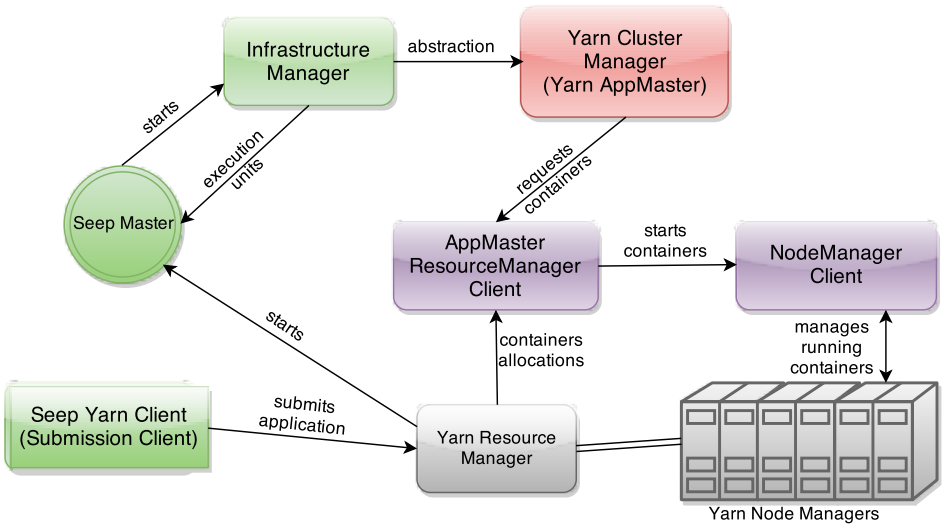

FIGURE 3.4: Seep YARN Overview

3.3.4 ApplicationMaster

The AppMaster handles the application execution by communicating with the ResourceManager, in order to negotiate and allocate containers for each worker that needs to run. After the allocation succeeds the AM communicates with YARN NodeManagers (NMs) via an AMRMClient object to asynchronously launch the actual worker in a similar process as the App Submission Client. One can use container ID provided by YARN to redirect standard output and standard error of each SEEP worker to a unique file where it can be examined later for debugging or performance analysis. During the execution of the containers the AM communicates with the NodeManagers via an NMClient object and can handle events such as start, stop, status update and error for the containers it owns. The user can implement custom logic to restart a container in case of failure and log all status updates. By defaults YARN restarts the AppMaster in case of failure two times and also notifies it if any container is killed by the RM such as if it exceeds the available memory. Note that a strict memory isolation is desired to prevent the machine from using the swap which would slow down the performance exponentially.

Bellow are the interfaces available for writing a YARN application that we implemented in SEEP:

- Submission Client: communicates with ResourceManager by using YarnClient objects.
- ApplicationMaster: communicates with ResourceManager via AMRMClient object and with the NodeManager(s) via NMClient objects.

3.3.5 Implementation

SEEP maps all logical operators that form a query to *execution units* that are generic SEEP workers ready to start execution. This mapping is done in the *ClusterManager* class which previously supported only one type of deployment called *Physical Cluster*. The physical cluster manager

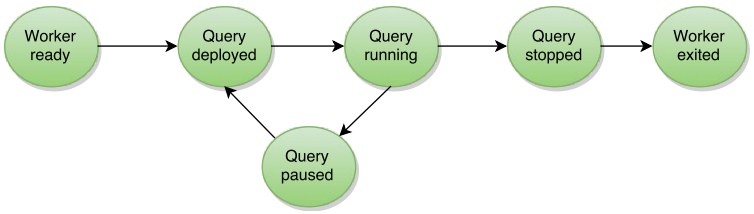

FIGURE 3.5: Seep worker states

relied on the user to deploy, start and stop the query. When enough execution units were available for all the logical operators the master was ready to deploy the query and waited again for the user to request that. This behavior is not practical for a large cluster deployment where multiple queries need to run automatically without any user interaction.

The first step of the YARN integration was to create an API in SEEP master that accept externals commands for deploying, starting and stopping the query. After that we created a new class, *YarnClusterManager*, that implements all the *YARN AppMaster* logic and extend YARN's *AMRMClient* and *NMClient* interfaces to handle all container related events.

The second step was to make SEEP master deploy automatically the SEEP workers required for the query by submitting container request to YARN *Resource Manager*. Once the requests are fulfilled we made SEEP master responsible for configuring the worker container and starting execution with the help of YARN's *NodeManager*. To make the integration nice both *YarnClusterManager* and SEEP previous *PhysicalClusterManager* extend a common interface called *InfrastructureManager* and the user can choose between them via the *deployment.type* command line argument. To simply the deployment even more we added an option in Seep Master to automatically deploy and start the query without waiting for any external commands. This was particularly useful later one when running benchmarks.

To make it possible to migrate queries we made SEEP keep track of stopped operators when the query is running and try to assign the job to another operator if it becomes available. To achieve that we extended the *QueryManager* class with additional functionality that detects automatically if a replacement can be made. This way when our scheduler decides to "move" an operator it has to notify SeepMaster to stop it and then make the *YarnClusterManager* ask for a container from YARN on the specified host. Once the new worker is running the *QueryManager* will detect the new worker and assign to it the missing logical operator. This way the query resumes computation.

Finally we extended the possible states of SEEP workers to include a "pause" state which can be useful if we decide to execute queries in turns. In Figure 3.5 we illustrate the previous the previous sequential state machine of SEEP workers along with the new "pause" state that we introduced. To achieve that we had to change the logic responsible for running the query which was designed to stop the worker process and close the streaming endpoint when the computation stopped.

The integration modified to small extent the rest of the SEEP to a small extent, for example the worker is not aware if it runs on a physical cluster or YARN cluster while the master only needs to request containers automatically if the deployment is on a yarn cluster. In Figure 3.4 we can observe the design of YARN layer in SEEP.

3.4 Summary

In this chapter we have shown what missing features prevented SEEP to run multiple queries independently on a cluster and analyzed all the different alternatives to achieve them. Next, we explained in detail how we integrated SEEP with Kafka to obtain fault-tolerant communication and how we integrated SEEP with YARN to make queries run under resource isolation from each other. In the following chapters we will show how we build our scheduling system on top of that.

Chapter 4

Scheduler

4.1 Outline

In this chapter we will show why a classical scheduler cannot cope with streaming jobs by presenting YARN fair scheduler and highlighting issues encountered while scheduling streaming operators. Next we will shortly present how we wrote a simple resource aware placement policy for allocating new containers based on current cluster load. Due to the limitations of a static allocation we motivate the need for a run-time scheduler. Finally we describe how we designed our scheduler and discuss the algorithm in detail while debating over different approaches we considered and presenting similarities to well known algorithmic problems.

4.2 Default placement

Workflow management frameworks such as YARN [16] and Mesos [14] offer a broad set of of features for deploying jobs by coordinating how they are packaged, submitted and scheduled, their execution runtime and upon completion we are informed of failures. After its separation from MapReduce YARN become even more popular and many people contributed with various schedulers for different uses-cases. Next we will list some of the most widely spread YARN schedulers and discuss why they are not suitable for streaming queries:

- **Capacity Scheduler** is designed to guarantee a minimum resource share for certain users or groups. To achieve that it partitions the resources in the cluster such that each share can resize elastically according to one's computing needs. This is done by using a series of queues so multiple users can share a queue according to their needs. The approach works well for batch jobs but for streaming computations we cannot over-allocate and wait for jobs completion to recover since these jobs run continuously. Instead the scheduler could kill the jobs over-allocated but that would not be beneficial.
- **Fair Scheduler** works similarly by using a set of queues. When new jobs are scheduled resources shares are computed such that each jobs gets the minimum number of shares to

satisfy its minimum requirements. Shares that are not assigned are re-distributed among all users. As in the situation above the scheduler would need to enforce a preemption policy to guarantee the minimum shares.

To further highlight our assumptions we discuss the issues encountered by Samza [11], a very popular streaming framework that runs on top of YARN. From their daily usecase at Linkedin they discovered a consistent skew in the amount of data processed by different containers. This implied need for different resource requirements even though workers had homogeneous workloads. To account for that they used strict resource isolation which leads inevitably to resource wastage in the end.

When we deploy SEEP queries on the cluster we observed a significant variance across the resources utilized by different operators. To only way to account for that is by allowing flexible shares and running a run-time algorithm that would migrate tasks according to their resource needs.

4.3 Analytics

A critical requirement for a scheduler to be able to make informed decisions is to be aware of the physical resource utilization across the cluster. To achieve that we would need to monitor each node in real time for CPU, memory, disk I/O and network I/O. All the data should be accessible to to the scheduler so it can observe when is needed to move a task and pick the best possible destination host. To solve this problem we created a module that runs on each machine in the cluster and periodically gathers resource utilization statistics. Further more to be able to measure the performance we need to constantly watch all log files created by running SEEP queries since each SEEP worker outputs its performance metrics every 30 seconds.

4.3.1 Resource Monitoring

To standard way to find resources utilization of a machine is by using various system commands which are not always reliable and usually platform dependent or even tied to the physical hardware. For example to measure programmatically CPU usage we need to sample the *total process time* over a time interval and then add together the *user time* and *kernel time* and divide by the sampled time. Even worse for some metrics there isn't a programmatic way to get the information and one would have to invoke external tools such as *top* and parse its output. Luckily for us there is a cross-platform library *psutil* (python system and process utilities) that extract information about running processes and system utilization. They expose a very easy to use API that gives us all the required information with the exception of per process network and disk usage on OsX platforms.

With psutil the task of getting system usage information becomes straightforward by directly calling their API functions: *phymem_usage, disk_io_counters, net_io_counters, cpu_percent* and additionally getting the difference between current measurements and previous ones to calculate disk I/O and network I/O. For getting resource usage information per process we need a bit of extra logic because we don't have a way of knowing the process ids of SEEP workers. Ideally we would should have been able to get this information from YARN after the containers is started but they don't

support that yet. To solve this we created a mapping between each SEEP's worker *data.port* and the *container.id* which is the name of the standard output file used by that process. Data ports are assigned uniquely by SEEP Master to each worker since we can't have any guarantees on how many workers will run on the same machine. To be able to get the *data.port* we need to read the command line arguments of that process. This is a bit inconvenient in Java as by default all arguments are passed directly to the JVM (Java Virtual Machine) running the process and hence are not visible from standard process list. To solve that we parsed the output of *jvm* command which gives information for all java processes running on the machine. When we detect a java process containing a *data.port* along with SEEP classes on its Java classpath we can be certain that this is a SEEP process and use the *psutil* library on it get all required resources information. When parsing SEEP queries output files we search for the *data.port* entry (Each SEEP worker logs this information by default at startup) and we obtain this way the mapping to the YARN container where the process is running. The scheduler needs all those metrics from each machine (see Chapter 5 where we explain how the data is passed) to be able to make informed decisions.

4.3.2 Performance Metrics

SEEP uses Apache Metrics library to output periodically the following metrics measured in events per second. The value represent the number of times the *processData* function (called when a new data tuple is received from the upstream connection) is hit in each Operator:

- **(1, 5, 15)-minute rate**: rolling average for the last number of minutes.
- **mean rate**: overall average since the operator started.
- **count**: total count since the operator started.

To evaluate the performance of SEEP queries running on the cluster we decided to use this metrics as our building block for all the graphs that we will describe later on. We favoured this approach among others because it allows us to read metrics offline, for a time interval when the server was not running, as opposed to sending the metrics to the server directly.

To gather all the information efficiently we created a file system watcher module that scans periodically for changes all the files under a certain path. It keeps an index for each file which represent the number of bytes read. When more data was written to the file we read the new data, filter it using regexes for lines that contain metrics and send it to a central server. Because the system reads everything once and minimizes the data transfer we are able to parse and extract metrics out of a few GB of data written across the cluster in a few seconds. Furthermore this approach scales well as the cluster grows since the size of the output files grows at a very slow rate over time due to the use of filtering and high logging levels.

4.4 Task placement

The first step in improving the way tasks are placed in the cluster was to override YARN default placement policy and implement our own logic based on measuring resource utilization. Even

though startup only placement would not have any information about the jobs being placed we though a sensible resource aware estimation could produce improved results. To achieve that we changed SEEP to query an external system when submitting container requests by adding logic in the *AppMaster* class to send a HTTP query to our server asking for the best host for the next container request while setting the YARN *strict locality* flag.

4.4.1 First approach

For the first implementation we focused solely on CPU placement optimization as this is the most common resource that becomes scarce and consequently limits the performance of tasks. While running CPU intensive benchmarks we discovered 1.1 that is common for YARN to place CPU intensive operators (1 out of three in our query) on the same machine due to its bin packing strategy while considering all tasks homogeneous. To improve this we considered sorting the nodes based on their average CPU utilization in the last seconds. Therefore when a new container request is submitted we ask YARN to allocate it on the node with the lowest CPU consumption. This approach seems to work when multiple queries are launched sequentially however if we start queries at the same time our policy will suffer from the *herd effect* by allocating all the consecutive jobs on the same machine without being aware that the resources may be depleted quickly.

4.4.2 Second approach

To account for that we kept track of the number of allocations and sort the nodes based on CPU utilization and allocation count. After a number of experiments we discovered that this strategy will prefer crowded nodes running few jobs in favour of free machines running many "light" jobs. To solve this issue we define the term *CPU levels* which means dividing the 0-100% utilization range over a few levels and sorting the jobs first by the CPU level and in case of equality by the number of allocations. This way the allocation would follow a round-robin order over the set of nodes with the same CPU level. The time to iterate once over the nodes on the same level will be enough in general for the operators to start on the first node pushing it to a higher CPU level.

4.4.3 Limitations

The algorithm used for placement performed reasonably well in practice (see Chapter 6 for evaluation) although it can easily fail in specially crafted scenarios. Even though the placement strategy was better there was a need for a runtime scheduler that can migrate task while knowing the resource needs of each job.

4.5 Runtime scheduler

To be able to cope with the dynamic nature of streaming computations and their changing workload we needed to develop a scheduler that would periodically supervise the resource utilization of running tasks and make adjustments when required. This is very difficult because we have to find

the sweet spot among different resources and carefully consider the trade-offs associated with task migration. One example of that is how much extra resources would be able to compensate for the query's lost progress during migration. Furthermore we need to estimate how much extra resources a task requires to run optimally to prevent moving more than once. Note that the upperbound for this value won't exceed by far one core since the operators that we run use only one thread for processing data. Finally we need to experiment extensively all the solutions that we pick since most of them will be based on estimation and greedy algorithms. This is difficult since to some extent we try to predict the future needs of our tasks. First we begin by defining *Potential* and Resource Score that we use to estimate node contention and tasks potential.

4.5.1 Potential

Potential is a contention estimation used on nodes to assign a score in the range [1, 2] with the purpose of predicting additional resources that a job will utilize if the machine is loaded less. For example if one node's average CPU utilization is 90% we could assign a *potential* of 1.5 meaning that we would expect each job to utilize 50% more CPU in an ideal situation and similarly if the average CPU is 70% we could assign only 1.1 since there a reasonable amount of CPU free and not claimed by any job. More generally the potential values follows an exponential distribution with rate $potential.\lambda = 1.5$ and maximum $potential.max$. We experimented varying the rate or just using a linear distribution and the strategy with $\lambda = 1.5$ seems to work the best. Similar estimations are performed for Disk I/O and Network I/O utilization by first converting an absolute value (such as 20Mb per second) to an relative percentage based on a user defined maximum rate of the machine. While the potential is just an estimation which might produce wrong results (even if a machine is loaded, tasks might not need any more extra CPU) it seems to work reasonably well in practice as we will analyse by running multiple benchmarks. More on that in chapter 6.

To illustrate different potential values we calculated the estimated resource use for multiple values of λ and max as we show in table 4.1 the estimated CPU usage for a worker based on his measured CPU usage and the host load. For simplicity we consider the host CPU usage to be equal to the worker's usage.

potential.lambda	potential.max	50	75	90	100
1.5	1.2	52	83	104	120
1.5	1.4	55	91	118	140
1.5	1.6	57	99	132	160
1.7	1.5	55	93	123	150
1.7	1.8	58	104	144	180
1.7	2.0	60	111	157	200
2.0	2.0	57	105	153	200
3.5	2.0	50	85	130	200
5.0	2.2	50	77	114	220

Table 4.1: Potential estimations

4.5.2 Scoring Algorithm

Scoring is done by computing a "resource score" for each type of resource that measures the sum of the workers utilization percentages plus the host's potential. For example CpuScore is equal to the following formula:

$$ResourceScore_{Cpu} = \sum_{i=1}^{n} worker_{i,cpu\_percentage} + potential_{host}$$

To account for external processes running on the machines we measure their cpu utilization as *nonSeepCpu*. From analysis we have seen that auxiliary processes with a very low resource usage are not bottlenecked from resource contention. Because of that we multiply their usage with the node potential only if their usage is greater than a user defined threshold. Next we illustrate the algorithms used for CPU scoring algorithm 1 and I/O scoring algorithm 2. As the CPU scoring and I/O scoring is very similar we are going to explain the first heuristic in detail.

```
1  potential ← EstimateCPUPotential(host);
2  cpuScore ← 0;
3  nonSeepCpu ← destinationHost.cpu_percent;
4  foreach job : host.jobs do
5  |  cpuScore ← cpuScore + (job.cpu_percent * potential);
6  |  nonSeepCpu ← nonSeepCpu − job.cpu_percent;
7  end
8  if nonSeepCpu ≥ config.get("significant.cpu") then
9  |  cpuScore ← cpuScore + nonSeepCpu * potential;
10 end
```

Algorithm 1: CPU usage scoring based on task potential estimation.

To calculate the CPU usage score we first compute the host "potential" using the technique described above. Next on line 5 we compute the sum of all workers CPU potential usage by multiplying their measured usage with the host's potential. As the host is more loaded the potential increases thus we assign a higher CPU score. To account for external system running on the node we subtract the sum of all workers *cpu_percent* from the node's own usage. If the difference is significant we add this value to the total score on line 9. We used this in scheduling as we explain in subsection 4.5.3.

After we calculated resource utilization scores for all the hosts and all the tasks running in the cluster we can partition hosts based on their score in the following categories, where *migration.to.score* and *migration.from.score* are part of the scheduler configuration.

- **free hosts** have score less than *migration.to.score* and accept containers migrated from other hosts.
- **crowded** have score greater than *migration.to.score* but less than *migration.from.score*, won't accept any new container migrations.
- **very crowded** have score greater than *migration.from.score*, we need to migrate operators from them to other hosts subject to availability.

```
potential ← EstimateIOPotential(host);
ioScore ← 0;
nonSeepIo ← host.io_percent;
// IO percent is computed by dividing estimate max I/O of machine by actual usage;
// IO is measured as bytes read/sent for disk and write/received for network;
foreach job : host.jobs do
    ioScore ← ioScore + (job.io_percent ∗ potential);
    nonSeepIo ← nonSeepIo − job.io_percent;
end
if nonSeepIo ≥ config.get("significant.io") then
    ioScore ← ioScore + nonSeepIo ∗ potential;
end
```

Algorithm 2: IO usage scoring based on task potential estimation.

4.5.3 Scheduling

Because the resource score is basically the sum of all tasks utilization percentages multiplied by the potential we considered initially *migration.from.score* at 100 and *migration.to.score* at 95. For reference a cpu score of 95 would mean an average cpu utilization of 80% if running 3 tasks or 70% if running 5 tasks. A round of scheduling consist in migrating one operator from each of the "very crowded" hosts to the free hosts. To achieve that we select the most resource intensive tasks from each of the host that we need to free. Next we only need to match tasks to free hosts accordingly. To do that we sort based on resource usage the tasks in decreasing order and the hosts in increasing order such that we match the tasks with the biggest score to the most free host. This way we ensure that when we have less "free" hosts than tasks we want to move our approach would relieve the biggest amount of load from the crowded machines. We move only one worker from one "crowded"

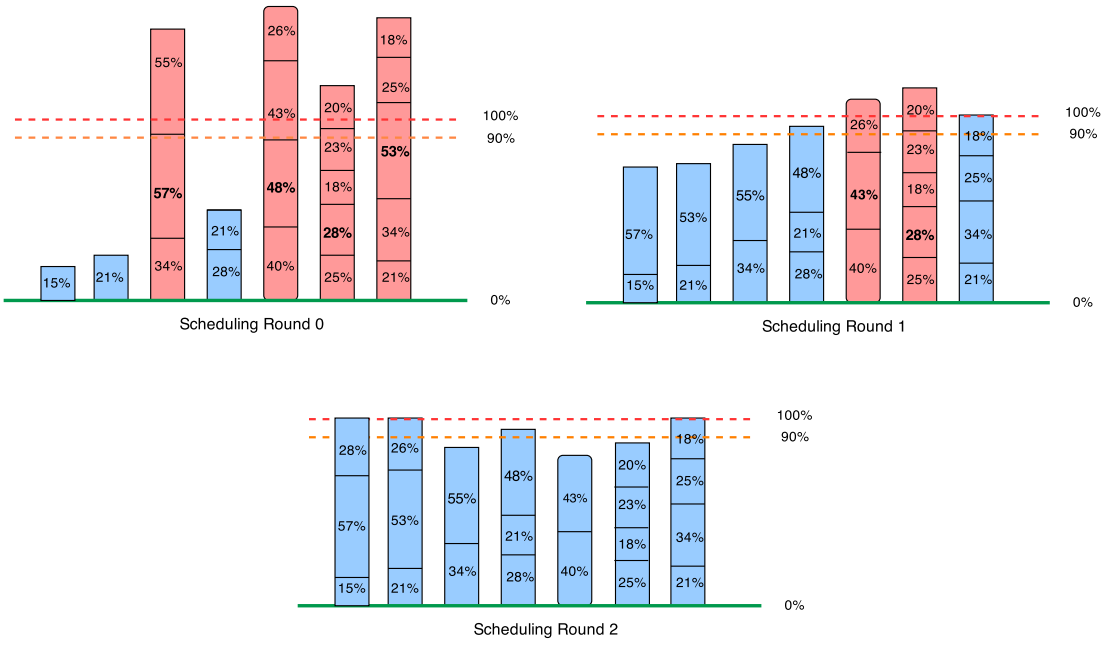

FIGURE 4.1: Runtime scheduling rounds

machine in one round, after experimentation we observed this way our performance predictions are more accurate. In Figure 4.1 we can observe how our algorithm manages to even the load after 2 rounds of scheduling. Even when we migrate tasks based on this algorithm there are still multiple trade-offs that we need to consider, we will examine in detail all of them in Section 5.2.

4.5.4 Alternatives

During our implementation we considered various approaches in running scheduling rounds. One of them was to run separate rounds of scheduling for each type of resource or run a common round for all of them. By combining the scores for each resource we were unable to estimate if a migration was beneficial for the resource bottleneck based on which we selected the worker. Furthermore we couldn't use the same greedy algorithm as above that guarantees the *best-effort* approach when not enough "free" hosts are available. For the reasons stated above we decided to run separate rounds of scheduling for each type of resource.

Another alternative was to move one or multiple jobs from and on the same machines. If we move more tasks from a hosts we can solve a very disproportional load distribution in few iterations but we have to estimate the resource consumption of jobs by a greater factor or alternatively measure allocations as we have done for the placement algorithm. If we combine our scheduler with the a simple placement policy such as the one described in Section 4.4 we won't get disproportional loads and we would need in general a small number of migrations to even the load over time. To reduce the risk of wrong estimation we decided to migrate only one job from each "crowded" host to each "free" host in one round. As the scheduling rounds are quite frequent, every 5 second, we can move multiple tasks from one host quickly enough.

4.5.5 Similar Problems

If we consider resources utilization percentages to be fixed the scheduling problem can be reduced to the *Bin Packing Problem* [35] which was studied extensively in the literature. More formally we are given a list of n items with sizes $a_1, a_2, a_3, ..., a_n$ and multiple bins B of size V, we need to find the smallest number of bins B so there exists a B-partition $S_1 \cup S_2 \cup ... \cup S_B$ of the set $1, 2, 3, ...n$ such that $\sum_{i \in S_k} a_i \leq V$ for all $k = 1, ..., B$. A fixed instance of scheduling with d resources types maps directly to a d-dimensional bin packing problem where $S_1, S_2, ...S_d$ is the maximum availability from each type of resource. In complexity theory bin packing is known to be NP-hard problem but many approximate algorithms exists and some can be applied to the multi dimensional case as well [36]. The most straightforward approximation algorithm is called *first-fit*. The algorithm processes the items in a random order and for each it tries to place it in the first bin that accommodates the item and if none is available a new bin is formed. One of the refinements is called *best-fit-decreasing* which sorts the items decreasing and places them in bins in order. Interesting enough this algorithm is similar to our scheduling algorithm with the only difference that we cannot create new bins/hosts and we are given always a instance, not necessary optimal, of a partially solved bin-packing as each hosts is occupied to a certain point. The main difference is that our "items" change their size over-time, i.e. their resource utilization percentage. However after a few round of scheduling when the workload is balanced those sizes remain constant in general. Therefore our approach to sorting the workers by resource utilization while sorting the

hosts in reverse order reduces to *best-fit-decreasing* on a less general instance of the problem thus having near optimal solution shown to be no more than $11/9 + 1$ of an optimal solution.

4.6 Summary

In this chapter we explained how YARN's default placement works and why there was a need for a different scheduler. To solve that we started incrementally by building a resource aware placing policy with the help of a resource monitoring system across the cluster. After we overwritten YARN defaults we discussed why this placement is more beneficial but still limited. Finally we presented the solution by creating a run-time scheduler that schedules jobs continuously in an effort to even out the workload across the cluster.

Chapter 5

Task migration

In this chapter we explain how our system monitors real-time resource utilization for each process running in the cluster and how we choose the hotspot between fast responsiveness and reliable measurements for scheduling. Next we will analyse the trade-offs associated with workers migration and present the techniques used to estimate them. Following we measure the application scalability and present what were the first bottlenecks and how we improved them. Finally, we will investigate the fault tolerance under different failure scenarios.

5.1 Resource utilization analysis

The scheduler needs to be aware in real-time of resources utilization cluster wide to be able to make informed decisions. This becomes very difficult when many applications run concurrently on the cluster competing for resources to the point of starvation.

5.1.1 Overview

To solve the problem we designed a system that is very responsive and provides accurate estimations even when actual measurements are delayed. Figure 5.1 provides an overview of each module that forms the scheduler and the analytics framework highlighting the way they interact. Next we will briefly describe the role for each of them:

- **master.server** - run the web Server responsible for serving the web UI and assigning all user interactions to the relevant module.
- **master.analytics** - collects resource utilization and performance metrics from the nodes running monitors. Computes aggregated metrics per application and for the whole cluster. Also calculates mean throughput over time, average throughput per worker and fairness. More on that later.
- **master.admin** - module responsible for running administrative commands such as deploying new queries, stopping the existing ones or restarting different systems that run on the cluster.

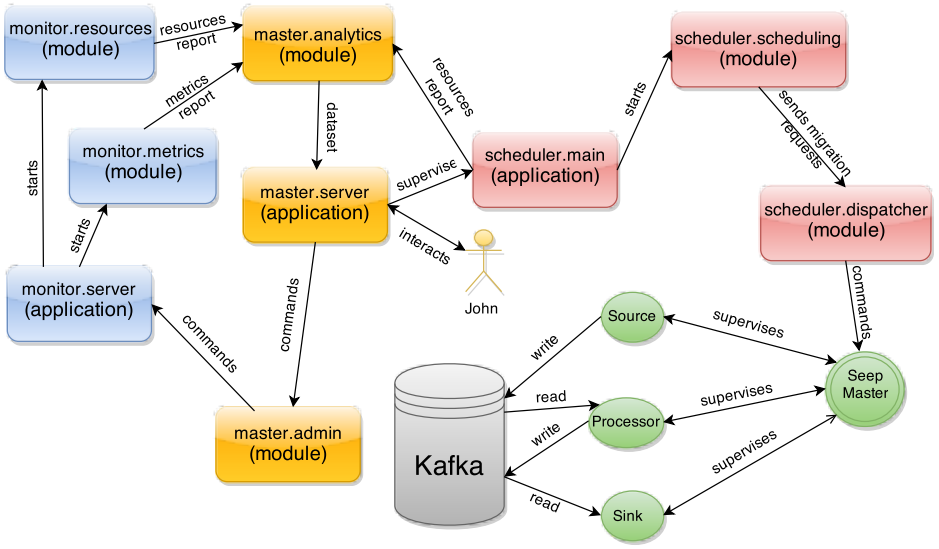

FIGURE 5.1: System Design Overview

- **monitor.server** - simple server responsible for executing locally commands received from admin module. It also delivers resource and metrics to the master each time new reports become available.
- **monitor.resources** - worker thread responsible for getting resource utilization metrics for its host as a whole and for each SEEP process running locally.
- **monitor.metrics** - worker thread that scans periodically YARN logs files and parses performance metrics reported by SEEP queries.
- **scheduler.main** - module that communicates with master to periodically get resource utilization reports and listens for user configuration changes.
- **scheduler.requests_dispatcher** - module responsible for dispatching migration requests to SEEP master and monitoring progress. While the migration is pending it gives estimations for the resource utilization effect of the movement. After a given time if pending request are not honored it marks them as failed.
- **scheduler.scheduling_worker** - module responsible for doing actual scheduling. Designed as a separate unit to enable work partitioning across multiple workers as the cluster size grows. This approach towards scalability is know as *two-state scheduling*

5.1.2 Resource Reports

Our system monitors usage for 4 types of resources: CPU, Memory, Disk I/O (read and writes) and Network I/O (sends and receives). All of this information is available for each host, for each SEEP worker and aggregated for the whole cluster. All of the metrics above are taken as average over a user define sliding window which defaults to 5 seconds. We didn't pick a smaller interval to

Overall		Host		Worker	
cpu.usage	percentage	name:	host_name	app.master	ip adress
disk.io	bytes_read, bytes_write	avg.cpu:	percentage	app.master	port number
net.io	bytes_sent, bytes_recv	memory:	total, percentage	cpu.percent	cpu percentage used by worker
hosts	[Hosts]	disk.io:	bytes_read, bytes_write	memory.percent	ram percentage used by worker
apps_running	no of apps/queries running	net.io:	bytes_sent, bytes_recv	disk.io	bytes_read, bytes_write
workers_running	no of operators running	cpus:	[percentage per core]	net.io	bytes_sent, bytes_recv
logs size	kafka_logs, hadoop_logs	workers:	[Worker]	pid	process id

TABLE 5.1: Resource Report Content

prevent noise in measurements from spike that might arise due to user interaction in the cluster or applications initialization routines. Additional identification information is collected for each SEEP worker such as: type of operator, name of query and master identity. The time required to collect these metrics is generally very low, around 200ms, since all the calls are native at kernel level by using a specialized library *psutil*. Even so all the logic is implemented asynchronously in a separate thread so the monitor is always available to receive commands from the master. Under heavy loads we observed some of the external calls made by *psutil* or Java Virtual Machine (JVM) to last for a few seconds. In table 5.1 we can observe all the resource utilization information reported by the system:

5.2 Migration trade-offs

After we integrated SEEP with an external message passing system, we have the streaming data stored reliably on disk. As a consequence we can simply stop an operator and start a replacement for it on a new machine without loosing any data. All the data produced by the upstream operator during the migration process will be stored in Kafka for later consumption. A migration is coordinated by SEEP master and happens in three phases:

- The master informs the assigned operator to stop computation and the worker process exits immediately after the current data tuple is processed.
- Master requests for a new container on the destination host assigned by the scheduler.
- After YARN allocates a new containers, the SEEP master deploys the query and our operator resumes computation as before on a new host.

5.2.1 Measuring trade-offs

Migrating tasks from one machine to another will impact the performance of the query as the data flow is interrupted for a short period of time. However the query should be able to regain quickly the progress lost by running with increased throughput. In order to understand the impact a migration has on a query we analyse the time taken from the point a worker stops computation until another start on the new hosts and resumes the data flow. We performed the measurements while running a big suite of benchmarks with cluster utilization ranging from 10% to close to 100% and compiled the data from over 1,600 different migrations. As we can see in Figure 5.2 the mean

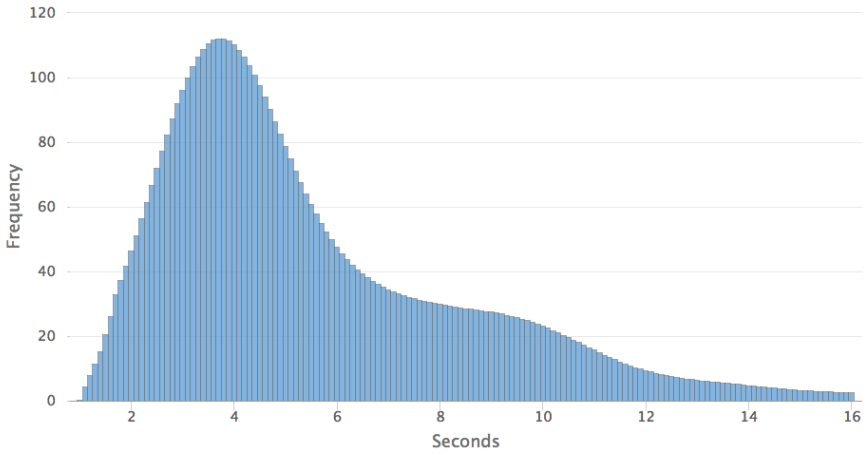

FIGURE 5.2: Operators migration latency

is at around 4 seconds and the 90th percentile at around 10 seconds. Therefore on average we lose 4 seconds of computation time when migrating an operator and if we consider a performance increase of 25% it means the query recovers in about 16 seconds.

We extend our evaluation to measure: performance increase, CPU increase and query recovery time in section 6.3.1. By combining all the measurements together we observe that a query needs on average 10 seconds to recover the progress lots during the 4 seconds pause on average due to migration.

5.2.2 Migration Scoring

Migrations requests are formed of tuples: *(worker, sourceHost, destinationHost)* which are generated from the matching algorithm (see section 4.5.3) for each type of resource. When a request is received we run a greedy algorithm based on some user defined parameters to analyse if the worker can potentially perform better on the new host. Algorithms 3 and 4 describe the logic used to evaluate if a candidate migration send by the scheduler is likely advantageous. The algorithms are very similar to the algorithms used for scoring resource utilization explained in section 4.5.2. The main difference is that now we compute the potential of the *destination.host* by also adding the resources used by the operator we want to migrate. This practically estimates the "effect" of moving the operator to a new machines with the aim of analysing if the progress price paid for migration would pay back in increase performance. Note that we also account for external resource usage by computing *nonSeepCpu* in line 7 for the CPU algorithm.

If the algorithms return a positive result based on their estimation we send the migration request to the dispatcher module which will provide resource utilization estimates to the scheduler for the following iterations until the migration succeeds or fails. Not that this estimation is important to prevent the same "heard effect" that was affecting the first version of our placement policy.

```
   input  : Job candidate for migration, SourceHost, DestinationHost
   output: True/False if the migration is required
 1 potential ← EstimateCPUPotential(destinationHost, job);
 2 jobs ← destinationHost.jobs + job;
 3 newCpuScore ← 0;
 4 nonSeepCpu ← destinationHost.cpu_percent;
 5 foreach job : jobs do
 6 |   newCpuScore ← newCpuScore + (job.cpu_percent ∗ potential);
 7 |   nonSeepCpu ← nonSeepCpu − job.cpu_percent;
 8 end
 9 if nonSeepCpu ≥ config.get("significant.cpu") then
10 |   newCpuScore ← newCpuScore + nonSeepCpu ∗ potential;
11 end
12 if newCpuScore - sourceHost.cpu_score ≥ config.get("min.movement.score.difference")
   then return True ;
13 else return False ;
```

Algorithm 3: Decides if CPU migration is appropriate

```
   input  : Candidate job for migration, SourceHost, DestinationHost
   output: True/False if the migration is required
 1 potential ← EstimateIOPotential(destinationHost, job);
 2 jobs ← destinationHost.job + job;
 3 newIoScore ← 0;
 4 nonSeepIo ← destinationHost.io_percent;
 5 foreach job : jobs do
 6 |   newIoScore ← newIoScore + (job.io_percent ∗ potential);
 7 |   nonSeepIo ← nonSeepIo − job.io_percent;
 8 end
 9 if nonSeepIo ≥ config.get("significant.io") then
10 |   newCpuScore ← newIoScore + nonSeepIo ∗ potential;
11 end
12 if newIoScore - sourceHost.io_score ≥ config.get("min.movement.score.difference") then
   return True ;
13 else return False ;
```

Algorithm 4: Decides if I/O migration is appropriate

5.3 Fault tolerance

One of our main goals was to build a fault-tolerant system that would be able to cope with failures of worker nodes as well as the master node. To achieve that we designed the system without any single point of failure. We considered two different designs for achieving fault tolerance:

- multiple master and scheduler workers that reach consensus for running the master logic using a leader election algorithm
- separate service distributed on multiple machines that supervises the analytics-master and scheduler and restarts them in case of failure.

```
1  timeout ← 0;
2  while timeout < maxTimeout do
3  |  sendRequest(ping, service, timeout);
4  |  if request.connectionError then
5  |  |  // Service certainly down.;
6  |  else if request.readTimeout or request.connectionTimeout then
7  |  |  timeout ← timeout ∗ 2;
8  |  else
9  |  |  // Service is working;
10 |  end
11 end
12 // Restart the service.;
```

Algorithm 5: Increase the timeout exponentially while pinging a service.

The first approach would have the overhead of running multiple instances for both analytics-master and scheduler and also would duplicate the fault tolerance logic in two modules along with any external dependencies. The second approach creates less coupling by separating the fault tolerant logic is in a separate module that can be simply plugged in the system. For the reasons above be favoured the second approach and created an external fault-tolerant service that supervises the critical components of the system.

5.3.1 Supervisor

We created a standalone module that runs in a distributed fashions in a single master multiple slaves paradigm. The master is responsible with pinging periodically the analytics server and the scheduler to check if they are live. While running intensive I/O benchmarks we discovered some server requests failed even if the server was running. This is because The TCP/IP layer can timeout requests if the network is very crowded. The best strategy in this case is to run a retry policy since if the server is just slow timeouts are intermittent. We implemented that using an approach inspired from TCP exponential backoff strategy, which we present in Algorithm 5. In line 2 we try to ping the server until a maximum timeout is reached. Line 4 is hit is the DNS mapping failed which means the server is down certainly and line 6 is typically triggered by connection errors so we increase the timeout exponentially. This way we ensure the system is always up if at least one out of all supervisors are still alive.

5.3.2 Leader Election

When the current supervisor leader dies the remaining standby supervisors compete to replace him by running an election algorithm. This problem has been extensively studied in the literature as the *leader election problem*. Many solutions exists for particular topologies such as rings, mesh or hypercubes. Since our system is designed to coordinate a fully connected network we need to apply a more general algorithm. One example is the Mega-Merger algorithm which is inspired from the Minimum Spanning Tree problem or the YOYO algorithm which eliminates candidate in multiple iterations and surprisingly its complexity is still an open research problem. While the idea to

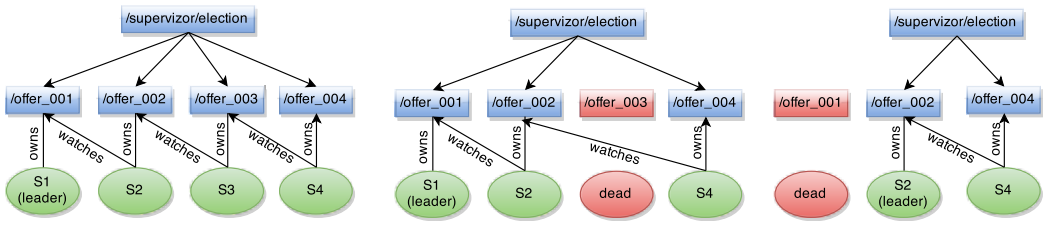

FIGURE 5.3: Leader election among supervisors in different failure scenarios

implement an interesting algorithm was very appealing we decided to keep our system as simple as possible and opted instead to use an external support system to solve our problem. Because Kafka was already using the Zookeeper [29] synchronization service this was the obvious choice. Next we will describe how we elect the leader by making use of Zookeeper synchronization mechanism.

Zookeeper runs as a quorum of process distributed across multiple machines that can easily cope with node failures, network unresponsiveness and other related events. Each supervisor can create an *ephemeral* node under a common path that will be deleted by Zookeeper when the service dies. If we register watchers for a node we can be notified when the node is deleted thus allowing us to discover failed nodes in the system. Furthermore the *sequential* option makes Zookeeper assign a increasing unique sequence number to each node in the order they are formed. Therefore when all supervisors start, each of them creates a *sequential* and *ephemeral* node and the one with the lowest sequence number is the leader. However we still need to watch for failures of the leader so a new candidate can replace it immediately.

A trivial solution to achieve that would be to make all supervisors watch for the node with the lowest sequence number, the one serving as leader, and in case of failure check if their node is the next one in line. This approach works but it suffers from the *heard effect*: many services are watching the same node and Zookeeper could incur unnecessary latency to inform all. To prevent that we can make our waiting supervisors watch for the next smaller node after theirs, while this is not necessary the leader but still has higher priority so it will become leader before the node who watches it. If a supervisor discovers that the node it watches fails it will needs to keep looking for smaller nodes and become leader if none are found. In Figure 5.3 we can see how the state stored in Zookeeper alters after a node crashes. This approach allowed us to have always a leader and ensures the scheduler is running in an efficient and simple manner. Furthermore we can also run the scheduler as before without any external dependencies.

5.3.3 Scheduling Failures

We designed our scheduler to be able to cope with failure of any number of the slave nodes by using a separate module that handles migration requests: *RequestDispacther*. After a migration request was sent the dispatcher will report back an estimative resource utilization change that would be triggered when the operator is moved. This way the scheduler is aware that a migration is pending and doesn't consider that a host where operators are about to start is "free". These estimation are import to prevent the scheduler to "flood" a free host with more operators than it can handle. Following is a summary of the main functions part of the dispatcher module:

- **addRequest**: the scheduling module calls this functions to add a migration request to the main requests queue.
- **checkPending**: the main module calls this functions each time a new resource report was received from the master server. Each resource report contains a list of operators running on each host so when the Dispatcher sees that an operator is running on its destination host it can mark the migration request as succeeded and reset the resource utilization estimation. If there are pending requests that haven't completed after 30 seconds we consider them as failed and add them back to the main queue for one more attempt.
- **sendRequest**: this function simply sends the request to migrate an operator from the top of the request queue. However if the corresponding SEEP Master has received a request for less than 5 seconds we will postpone for a while the actual send. If SEEP Master fails to accept the request we mark the migration request failed.
- **getEstimation**: return the estimated resource utilization of a host which comes from operators pending to start on that node.
- **setEstimation**: adds a given value to the estimation for a type of resource. This functions is used when request are received or finished.

The scheduler, master server and Zookeper run on separate machines than the rest of the cluster running scheduled jobs to prevent from resource starvation that would impact their latency. If one of the machines running the scheduler or master server crashes the supervisor will be unable to ping the service so it will pick an available machine and restart it by sending a command to the monitor running there. Note that ideally we will have a few spare machines that don't run scheduled jobs to act as schedule only machines.

5.4 Scalability

As the size of the cluster increases it would be ideal for our scheduler to handle the load without accumulating overhead or changing the implementation. To achieve this we minimized the centralized computation and divided the heavy lifting to worker slaves.

5.4.1 Resource Monitoring Scalability

The **master.analytics** module is responsible for receiving all the metrics measured every ten seconds from each worker and compute aggregate statistics every time. To analyse when this approach would become a bottleneck we profiled the code for performance while running on a cluster of 6 nodes over a period of intense load and measured all execution times, number of hits for each line as well as each function of the module. The main functions responsible for 99% of the execution times are the following:

- **updateClusterData**: Compute aggregated metrics for the cluster as a whole by summing all datapoints in a common time bucket.

- **updateStatistics**: Computes additional metrics such as *cluster fairness* and *average per container*.
- **updateAppData**: Computes aggregated metrics for every application (Sepp Query) by summing all the datapoints produced by app's operators.
- **updateMetrics**: Processes raw data send by the **monitor.metrics** module and calls for each performance metric the functions above accordingly.
- **updateResourceReport**: Updates resource report per host and computes aggregated resource utilization per cluster.

In Table 5.2 the first column represent the function from the analytics module that we measured and recorded the *total time* taken by a number of executions. The fourth column represent the maximum times each function could run in one minute if we would run in an infinite loop. The values are obtained by dividing 60,000 milliseconds by the average time per hit in column 3. Finally the last column shows the maximum limit the server would support concurrently measured as operators or nodes. This values are calculated based on the user defined update frequency that each SEEP operator and monitor module have. Every operator outputs performance metrics every 30 seconds and every monitor sends aggregated data every 30 seconds (where data represents metrics gathered for the operators running on that host) and it sends a resource report every 5 seconds.

5.4.2 First iteration

After the first analysis we can observe that the first three update functions are very fast and scale quite well, this is due to using fast lookup operations and keeping only a recent subset of data such that everything runs in constant time. However we notice that the server cannot scale to more than 60 nodes which in practice is less in we take into account additional delays that could occur on the machine running the server such as I/O operations or slow network. From the table we can see that the *updateResourceReport* function represent the bottleneck with an average of 78 milliseconds per hit. This was due to the function sending a updated report via HTTP request to the scheduler for each resource report received from a monitor. The approach is not efficient since when the number of nodes increases the scheduler will receive far more updates then it needs leading to unnecessary latency.

Function	Hits	Total time (ms)	Time per hit (ms)	Max times / min	Upper bound
updateClusterData	2,029	606	0.33	90K	45K operators
updateStatistics	2,029	86	0.04	750K	365K operators
updateAppData	2,029	353	0.17	176K	88K operators
updateContainerData	2,029	395	0.16	187K	93K operators
updateMetrics	86	4644	54	1100	550 nodes
updateResourceReport	2977	232593	78.13	767	63 nodes

Table 5.2: Analytics module performance profiling baseline

Function	Hits	Total time (ms)	Time per hit (ms)	Max times / min	Upper bound
updateClusterData	3,190	1077	0.33	90K	45K operators
updateStatistics	3,190	68	0.02	1,500K	750K operators
updateAppData	3,190	432	0.13	230K	115K operators
updateContainerData	3,190	528	0.16	187K	93K operators
updateMetrics	970	2567	2.64	11K	5,500 nodes
updateResourceReport	24,740	4195	0.16	187K	15K nodes

Table 5.3: Analytics module performance profiling after second iteration

5.4.3 Second iteration

To eliminate the bottleneck we changed the logic and made the server request updates every 5 seconds (this value can be changed from the configuration). This way the number of nodes won't have any more a significant impact on the performance. The next bottleneck became the *updateMetrics* function which was using some regexes to extract performance measurements from strings received from monitors and additionally it was performing a slow list lookup to get the *operator.id* associated with each *container.id*. The letter is defined by YARN as the name of the corresponding operator's stdout file. We managed to improve the performance by analysing how regex run and modifying as well as compiling the pattern used for performance. Also we eliminated the slow list lookup by using two dictionaries to map the attributes both ways. After this changes we can observe how the slow functions improved substantially and we increased the upperbound to 5,500 nodes which is good limit for our system.

5.4.4 Scheduler Scalability

The scheduler is formed of three modules: *scheduler.main*, *scheduler.requests_dispatcher*, *scheduler.scheduling_worker*. The main computation is performed in the *scheduling* module which is responsible for calculating resource scores, deciding where is best to move a operator that runs sub-optimally and also validating if the migration would be appropriate. To examine the performance of the scheduler and what are its limitations we made an analysis by profiling the code and you can see the results in Table 5.4. Notice that some functions are bounded by the product between the number of workers and hosts while others just by the number of workers on each host. The values in the table were measured while running 18 workers on 6 machines. To measure the upperbound we considered how many workers times operators/just operators would be needed to make on iteration of the function last 5 seconds, the standard frequency of the scheduling rounds.

Even though the scheduler scales quite well we made the design modular such that is easy to partition the set of nodes and run multiple *scheduling_workers* in parallel to be able to scale elastically on a large cluster. The only limitation would be if the number of operators running

Function	Hits	Total time (ms)	Time per hit (ms)	Max times / 5sec	Upper bound
schedule	850	8500	10	500	7,500 workers x nodes
selectIoIntensiveWorkers	850	942	1.1	4554	68K workers x nodes
selectCpuIntensiveWorkers	850	881	1.03	4854	70K workers x nodes
computeIoScore	5,100	3032	0.59	8474	10K workers
computeCpuScore	5,100	3043	0.59	8474	10K workers

Table 5.4: Scheduling performance profiling

on a single node increases substantially. In that case we will observe a visible latency until all the workload is distributed evenly as the scheduler is limited to one migration per node on a single iterations. Taking into account an approximate iteration frequency of every 5 seconds we would need 1 minute to move 12 operators from one node. However this scenario is not feasible since in practice we will exhaust the physical resources of a machine by placing less than 10 operators. Additionally if we consider using the default placement strategy then the workload will be distributed more evenly and only require moving just a few operators to make it perfectly even.

5.5 Summary

In this chapter we explained how resource utilization is monitored in real-time and how we make the information available for the scheduler with minimal latency as the cluster size grows. We analysed carefully all the trade-offs associated with task migration and explained the algorithms we used to evaluate if candidate migrations would beneficial. Then we presented how we made our system fault-tolerant and analysed its behaviour under different failure scenarios. Finally we discussed the bottlenecks present in the first iteration of the system and showed the improvements made to scale from 50 nodes to more than 500 nodes.

Chapter 6

Evaluation

In this chapter, we will evaluate our system through a series of experiments that we run both in the Imperial LSDS cluster and Amazon Elastic Compute Cloud (EC2). We will start by individually evaluating each scheduling strategy under standard workloads and comparing all three in several experiments. Following this we will compare our framework with other similar commercially available frameworks and illustrate the results based on real world benchmarks. Lastly we discuss the overhead of the scheduler and we study how the utilization of the cluster changes under slight variations in scheduling strategy.

6.1 Scheduling Efficiency

The main goal of our project was to create a system capable of distributing computation evenly across a cluster. More importantly our system needs to be able to cope easily with long running streaming jobs that exhibit dynamic workloads. To evaluate the performance we created a few simple benchmarks that are meant to be very resource demanding and as a consequence their performance is directly correlated to resource availability. Next we will describe two of the queries that we used in experiments:

RSA Factorization This query is based on the RSA public-key cryptographic algorithm. The *Source* continuously generates two random primes p and q of 23 binary digits and computes $N = p * q$. For each it calculates $(n) = (p)(q) = (p1) * (q1)$, where is Euler's totient function, and the public,private key pair d and e. After that it is sends an encrypted message of up to 46 binary digits along with the public modulus N to the *Processor*. The *Processor* factorizes N to obtain back p and q and applies *Euclid Extended Algorithm* to obtain e and d. Finally it sends the decrypted message to the *Sink*. Notice that this query is formed of three operators where two are CPU intensive.

Virus Search This query is inspired by common anti-virus applications and it is formed of two operators. The *Source* reads 512KB chunks of data from a local file, which represent suspicious executable hashes. Each hash is sent towards the *Sink* which simply checks if it is equal to a known malicious pattern. Therefore the *Source* is both disk I/O and network I/O intensive while the *Sink*

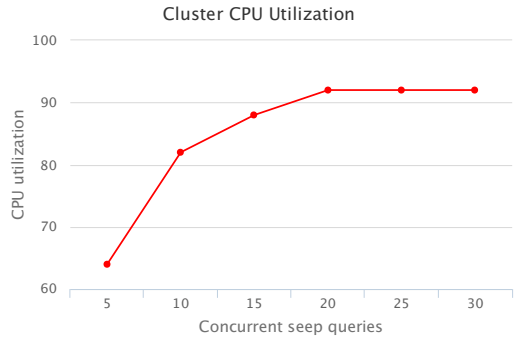

FIGURE 6.1: Cluster CPU utilization while running CPU Heavy workloads with resource YARN scheduling

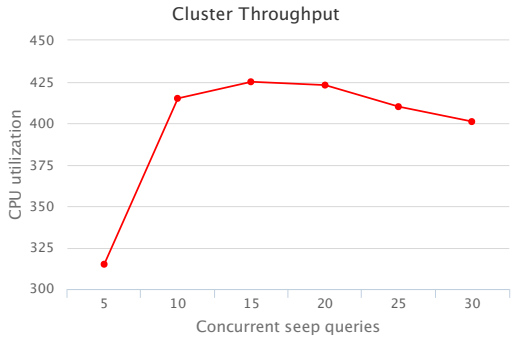

FIGURE 6.2: Cluster throughput while running CPU Heavy workloads with resource YARN scheduling

is only network I/O intensive. Note that both the hashes and the pattern have the same length so only string equality is performed, otherwise this query would be CPU bound.

6.1.1 YARN

YARN's most popular scheduler, the *Fair Scheduler*, allocates jobs with the aim of providing an equal share of resources to each job over time. Its approach fits perfectly standard MapReduce jobs for which it was designed. However for streaming computation this is not always optimal. Some operators need far more resources than other and if the scheduler gives a proportional share, correlated with their needs, the query performance increases. We analysed how YARN allocates resources by running up to 30 *RSA Factorization* queries concurrently on a cluster of 6 machines. In Figure 6.3 we illustrate the average cluster cpu utilization and in Figure 6.4 we show cluster throughput measured as events per second summed from all queries. To get this data we ran benchmarks 5 times and plotted the average. From the Figures we can observe how the **cpu utilization peaks at 90 percent** although the load increases far beyond the capabilities of the cluster. As consequence the **throughput peaks at only 425 events per second** and the performance starts to downgrade soon after.

6.1.2 Resource aware placement

In this paragraph we evaluate the placement algorithm described in Section 4.4. The main difference this approach has from YARN's *Fair Scheduler* is that we try to allocate the best fair of resources for each subsequent job by picking the most "free" host. This allocation is static and treats all the workers homogeneously similar in this respect to YARN. For evaluation we used the same benchmark as above -workload composed of *RSA Factorization* queries.

Figure 6.3 illustrates the average cluster CPU utilization and Figure 6.4 show the overall throughput. The data is compiled from a series of 5 runs on a cluster of 6 machines. We can observe how the **CPU utilization reaches 100%** before 20 queries while the **throughput peaks at 600 events per second** which already represent a significant improvement when compared to

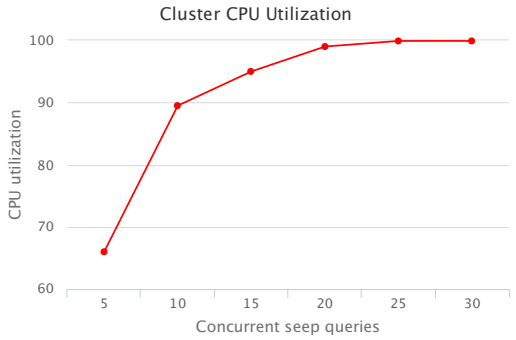

FIGURE 6.3: Cluster CPU utilization while running CPU intensive workloads with resource aware scheduling

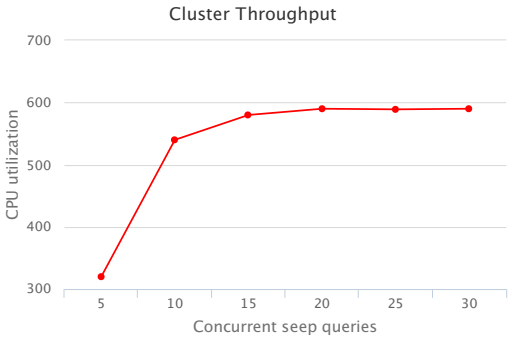

FIGURE 6.4: Cluster throughput while running CPU intensive workloads with resource aware scheduling

YARN placement. However we can achieve better if we migrate jobs at runtime based on observed resource utilization patterns.

6.1.3 Runtime scheduling

In this paragraph we evaluate the performance of our scheduler while using as a baseline the results observed in the two previous benchmarks. To illustrate a more detailed comparison we increase the number of queries by one for each datapoint. It is commonly known that workflow scheduling is not deterministic, and many factors apart from the allocation itself can influence the performance of jobs while running on a cluster. To account for that we ran the same same benchmark 15 times and plotted the mean and standard deviation. Next we will evaluate the performance of the three scheduling strategies under different workloads.

CPU workload In Figure 6.5 we illustrate the cluster throughput with three different scheduling strategies while concurrently running from 1 to 12 *RSA factorization* queries. For these experiments we used a cluster of 6 machines were each is equipped with 4 CPU cores and 4 GB of RAM.

We can observe how all the strategies have similar throughput up to the point of 4 queries. This is due to the fact that each query has only one CPU intensive operator out of three. Consequently the probability of placing 2 out of 4 operators together in 6 containers is pretty low. Starting from 5 queries we can see how the dynamic strategy begins to gain advantage. When running 8 queries we obtain a mean throughput of **547 events/second with runtime scheduling** compared to **489 events/second with startup placement** and **413 events/second with YARN**. This represent an improvement of **32% percent from YARN's** baseline and **12% percent from our placement policy**. After more than 10 queries running concurrently the startup placement seems to achieve similar performance with runtime scheduling but the standard deviation is higher. This is due to the worker allocation based on "guesses" of expected resource consumption instead of making informed decisions at runtime. Consequently YARN's allocation strategy has a significantly larger variance because it simply tries to allocate workers in the assumption that previous allocations are homogeneous, i.e. consume equal shares of resources. Note that YARN could make a lucky guess and allocate all workers optimally but this has a very small probability as the number of queries

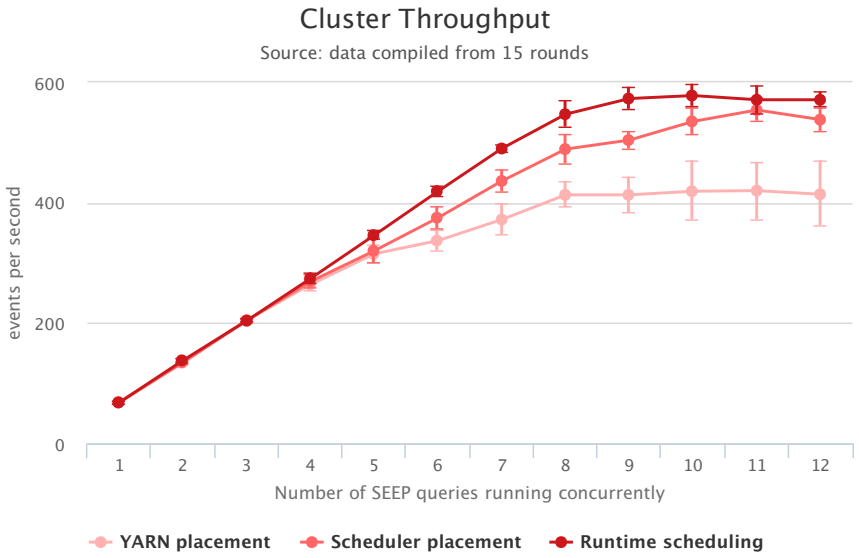

FIGURE 6.5: Cluster throughput

or cluster size increases. Furthermore if the query exhibits a dynamic resource consumption then we can adapt only by migrating operators at runtime.

For the same benchmark we measured the overall cpu utilization for the cluster in Figure 6.6. The CPU utilization is measured on the machines running SEEP queries only and no other services were running during the benchmarks. We can observe that when running 8 concurrent queries we utilize **92% CPU with runtime scheduling**, **82% CPU with placement scheduling** and only **67% CPU with YARN placement**. This is directly correlated with the overall throughput. To demonstrate the correlation we can measure the CPU utilization difference between the three strategies. Runtime scheduling uses 13% more CPU than startup placement and 32% more than YARN's policy hence both numbers exactly match the difference in throughput we observed earlier. Once again we can notice how the second strategy gains advantage as the number of queries increase since it will place new operators on the most "free" machines, thus optimizing the allocation. However, we can notice how YARN's scheduler and the startup placement have quite a large variance while the runtime scheduling has a very small variance. This demonstrates that this strategy is more reliable and deterministic than the other two.

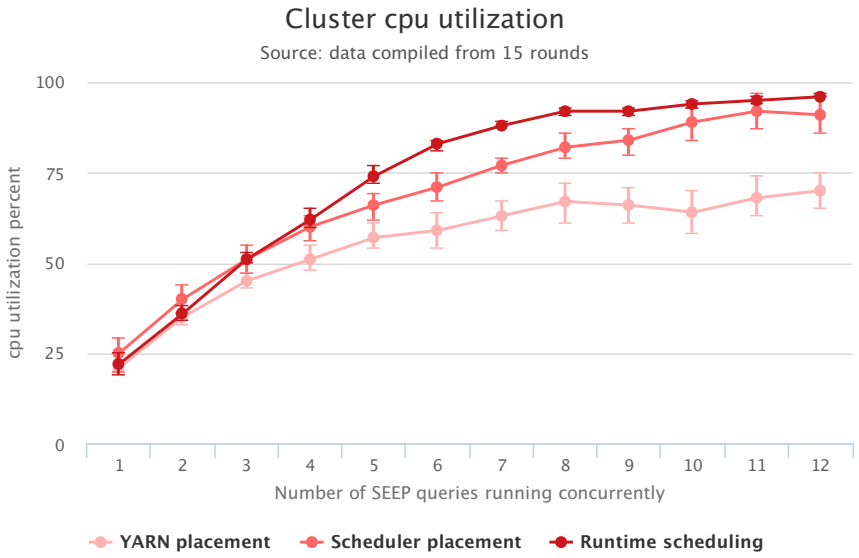

FIGURE 6.6: Cluster CPU Utilization while running CPU intensive workload

I/O workload To measure the scheduling performance under intensive I/O we ran the *Virus Search* benchmark and measured cluster throughput and all I/O metrics. To account for non-determinism we ran the same benchmarks 9 times and plotted the mean and standard deviation for each metric. In Figure 6.7 we illustrate the cluster throughput with YARN, startup placement and runtime scheduling. First we can notice how the variance is higher than in the previous experiment and is similar in magnitude across all three scheduling strategies. This is due to the high grade of non-determinism emerging from I/O usage pattern across the cluster.

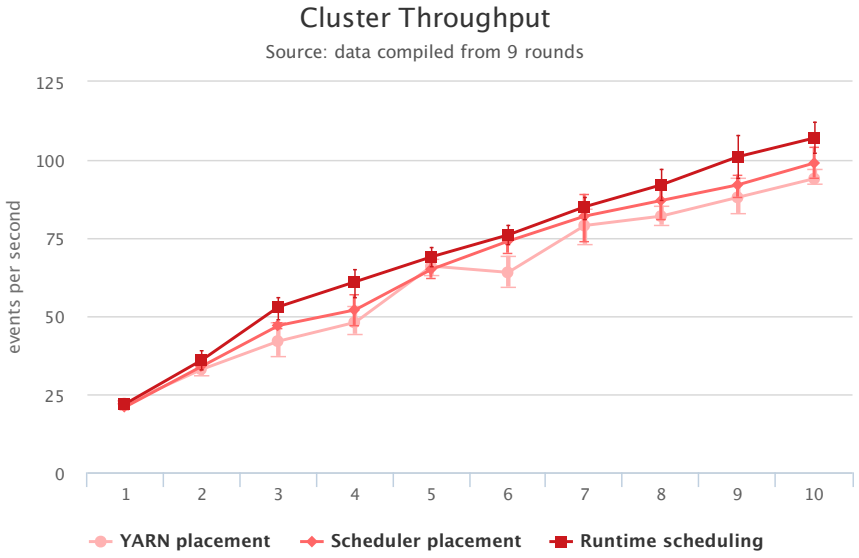

FIGURE 6.7: Cluster throughput while running I/O intensive workload

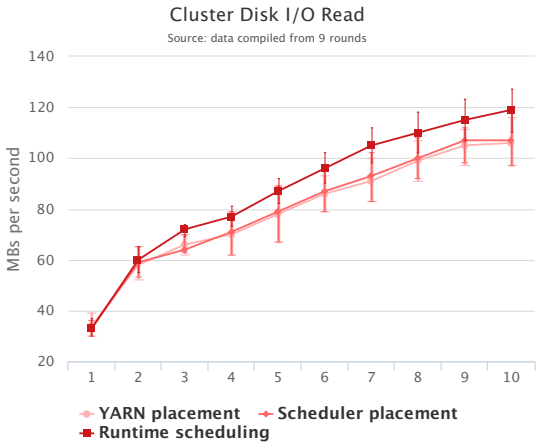

FIGURE 6.8: Disk I/O read with three different scheduling strategies

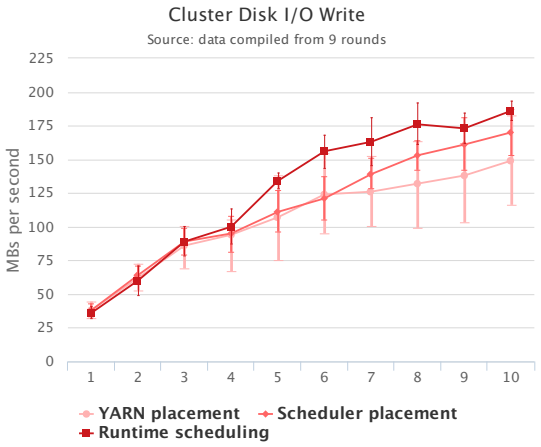

FIGURE 6.9: Disk I/O write with three different scheduling strategies

Secondly we observe a smaller improvement for runtime scheduling in contrast to our CPU benchmark resulting in a **10% increase in performance** compared to YARN's baseline. This query has two operators and both are I/O intensive whereas the *RSA Factorization* benchmark has three operators and only one is CPU intensive. Therefore a fair static allocation has a smaller chances of distributing the I/O load unevenly. However the *Virus Search* query has different I/O usage on its two operators: the first is bottlenecked on disk I/O read and network I/O send while the second is bottlenecked on network I/O receive. Because of this runtime scheduling can still increase performance. Finally we describe the correlation between events per second and I/O throughput. Each event in the metrics represents a 1MB record received in the *Sink* and for each the query pipeline produces 3x more I/O data: The *Source* reads the record and sends it on the network and the *Sink* receives it from the network. Therefore we have a total throughput of 300Mb per second while running on 6 machines.

In Figure 6.8 and Figure 6.9 we illustrate disk reads and disk writes measured as MBs per second for the cluster overall. The first thing we can observe is how disk read peaks at 120 Mbs / second while disk write peaks at almost 190 Mbs / second. This is surprising because both *Source* and *Sink* operators read from disk, the later indirectly by consuming data from Kafka and only the *Source* operator writes data to disk. However Kafka is configured with a replications factor of three which means that every record written to Kafka from the *Source* will be replicated three times on different machines. Because of the higher grade of non-determinism introduced by Kafka replication we observe how the variance for I/O write is significantly higher than I/O read. Finally in both graphs we can see the improvement runtime scheduling brings to I/O throughput: a **22% increase in disk writes and** a **11% increase in disk reads**.

We conclude our experiment with the graphs that plot network I/O for the three scheduling strategies. In Figure 6.10 we illustrate network sends and in Figure 6.11 network receives. Every byte that is sent by the actual query through the network is also received so the small difference between sends and receives comes from the metrics reports and perhaps bookkeeping information exchanged by Kafka. The network I/O improvement peaks at **19% for receives** and **8% for sends**. Overall we achieve a throughput of 400MBs exchanged every second between 6 machines. Note that heavy

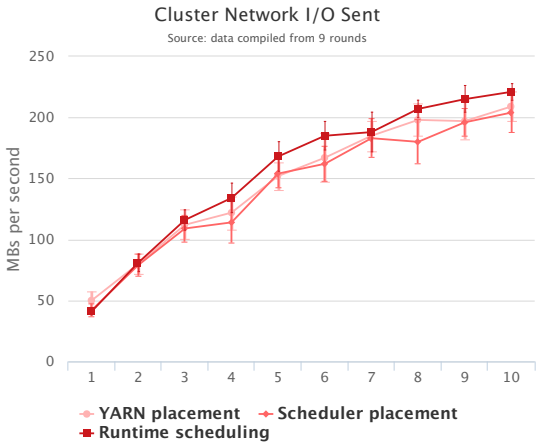

FIGURE 6.10: Network I/O sent with three different scheduling strategies

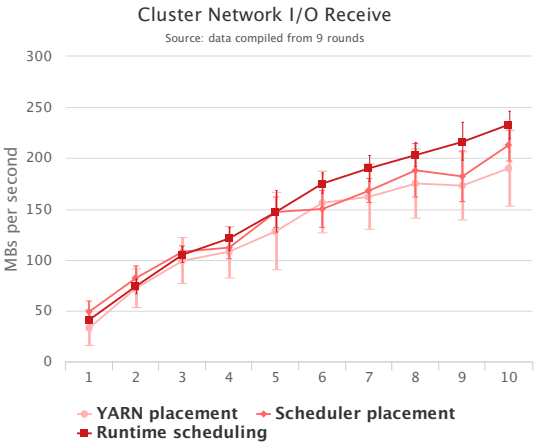

FIGURE 6.11: Network I/O receive with three different scheduling strategies

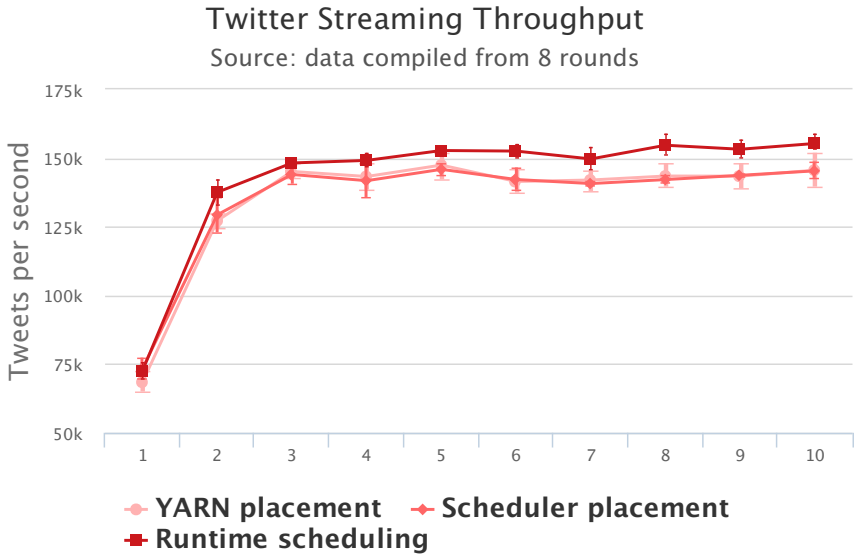

FIGURE 6.12: Cluster throughput on live tweets word count

I/O workloads are very common in large clusters and arise from any kind of data analysis of big data.

Real applications Streaming workloads that people run on clusters are more complex and can commonly become bottlenecked from both CPU or I/O. To test how our scheduler handles mixed workloads we created a query that ingests live data from Twitter by connecting to its API streaming endpoints. For each *tweet* it splits the message into words and keeps track of the number of occurrences over a window of time. Every 10 seconds it outputs the most frequent word. This query is I/O intensive since a huge volume of tweets are transferred every second and CPU heavy as each tweet is split into words and each word is hashed and inserted into a data structure.

Twitter limits their public API rate to only 50 tweets per second. To account for this we downloaded 2 million tweets and built our own streaming server that sends batches of 1,000 tweets over socket channels to queries for processing from a previously downloaded database of 2,000,000 tweets. This method increased the upperbound for the ingest rate from 50 tweets per second to 600,000 tweets per second. In Figure 6.12 we illustrate the number of tweets per second processed in the cluster as the number of queries increases. We can observe how the throughput peaks at 160,000 tweets per second which is a 10% increase compared to YARN's results. More interesting is that our placement strategy doesn't manage to increase the throughput as before in this benchmark. We believe this is a consequence of the mixed CPU and I/O workload that this query exhibits.

6.2 Fairness

Now that we have seen that our scheduling strategy can maximize throughput in different scenarios we can ask ourselves how fair the resource allocations are in general. On a multi-tenant cluster is very important to divide the resources evenly among different tenants which may be different users that share the cluster. YARN's *Fair Scheduler* developed by Facebook is specialized in allocating fair resource shares among multiple batch jobs over time. To evaluate fairness we used Jain's fairness index [42] which has the following equation:

$$\tau(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} x_i^2}$$

where x_i is the share of resources of the i^th process. The results ranges from $\frac{1}{n}$ (worst case) to 1 (best case) - when users receive the same allocation)

We measured fairness for the shares of operators and for the shares of "free" resources of machines. Note that for the first metric we considered only CPU intensive operators which fully utilize a core if it is available. All the measurements were taken when running from 5 to 30 *RSA Factorization* queries concurrently. From our previous experiments we observed that an optimal allocation fully

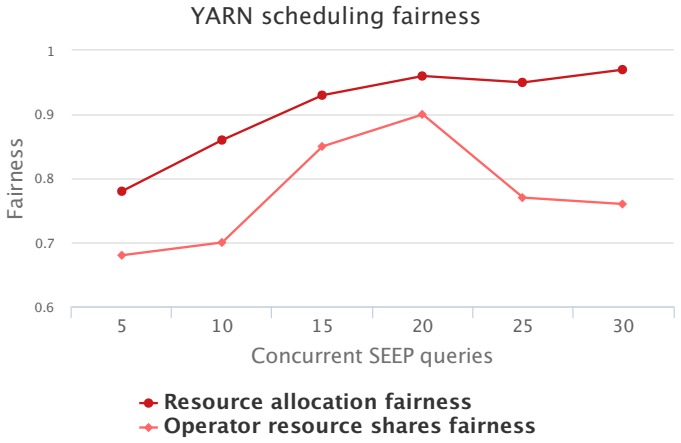

FIGURE 6.13: Resources and operators allocations fairness with YARN scheduling

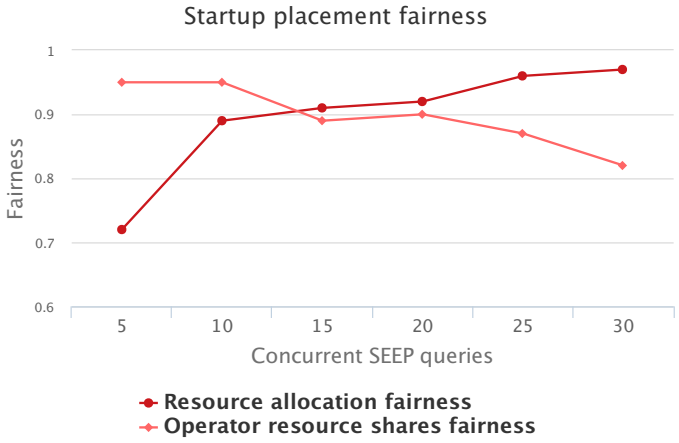

FIGURE 6.14: Resources and operators allocations fairness with YARN scheduling

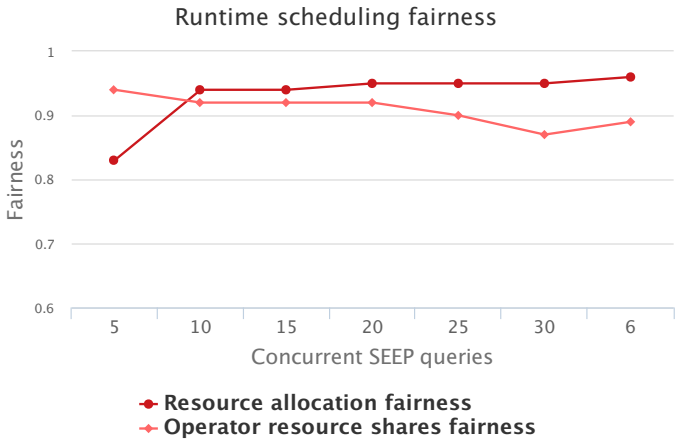

FIGURE 6.15: Resources and operators allocations fairness with runtime scheduling

utilizes the CPU with only 8 queries. In Figure 6.13 we illustrate the fairness under YARN's scheduling. We can notice how the resource shares are distributed badly initially but YARN *Fair Scheduler* manages to even the shares as the number of queries increases but downgrades again after the load exceeds 25 queries. We can also observe how the load per machine fairness grow slowly and is distributed evenly hardly at 30 queries when there are enough operators to compete for CPU to starvation.

Next in Figure 6.14 we illustrate allocation fairness when using the greedy startup placement. First, we notice that the queries used are CPU intensive. This strategy tries places new workers on the most "free" machines so it not surprising that on a idle cluster all operators have equal share of resources. As the load increases this strategy doesn't manage to keep its fair allocation among containers but manages to distribute the load evenly among the machines quite well.

Finally in Figure 6.15 we measure the same fairness metrics when our schedulers migrate operators with the aim of distributing the load evenly. We can observe a visible improvement from the previous strategies as the load is distributed evenly from 10 queries and manages to maintain

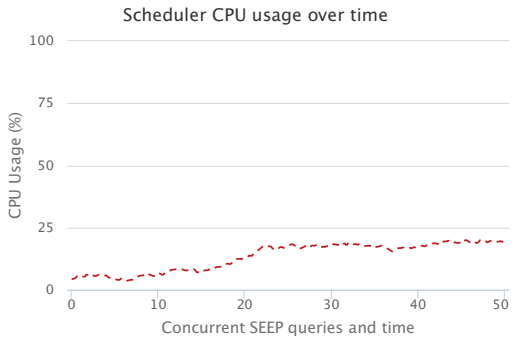

FIGURE 6.16: CPU usage on the machine running the Scheduler, Supervisor and Analytics Master during high load benchmark

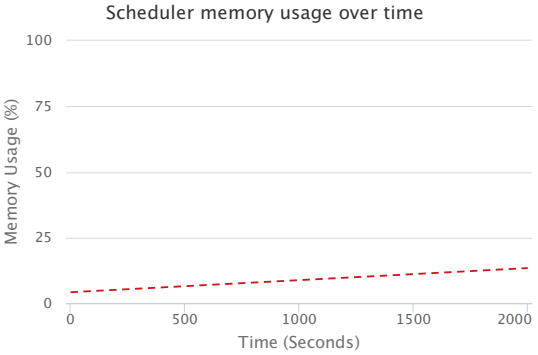

FIGURE 6.17: Memory usage on the machine running the Scheduler, Supervisor and Analytics Master during high load benchmark

the fairness as the load increases. Furthermore the most visible difference is the fairness among operators which downgrades very slowly as the load increases. At the highest 30 queries it still maintains a fairness of 0.9 while YARN has 0.75 and startup placement 0.8. Note that 30 queries formed of 90 operators is the upperbound of queries that can run concurrently on YARN, which is higher than 48, the number of cores shares, because YARN uses statistically multiplexing when allocating resources. This technique allows more containers to run than resource available by betting that not all of them will utilize their share completely.

6.3 Scheduling Overhead

One important question we have to answer to evaluate our system is how large the overhead of runtime-scheduling is and what the resource needs are as the scheduling workloads grows.

6.3.1 Resource usage

To evaluate this we deployed up to 50 queries on a cluster of 6 machines running concurrently and measured the resource usage on the machine running the scheduler, supervisor and analytics master. In Figure 6.16 we illustrate the CPU usage as the number of queries increases and in Figure 6.17 we show the ram memory usage over time when running 50 queries. We can observe how the CPU usage grows linearly with the number of queries running concurrently which matches our scalability estimations discussed in table 5.2 and table 5.3. The memory also grows slowly over-time due to the analytics module keeping track of cluster throughput measurements every 30 seconds. This is not desired but not a concern at the moment.

6.3.2 Migration Overhead

One key measurement that would reflect the performance of our system is the efficiency of operators migration. The main questions we set ourselves when evaluating the algorithm were:

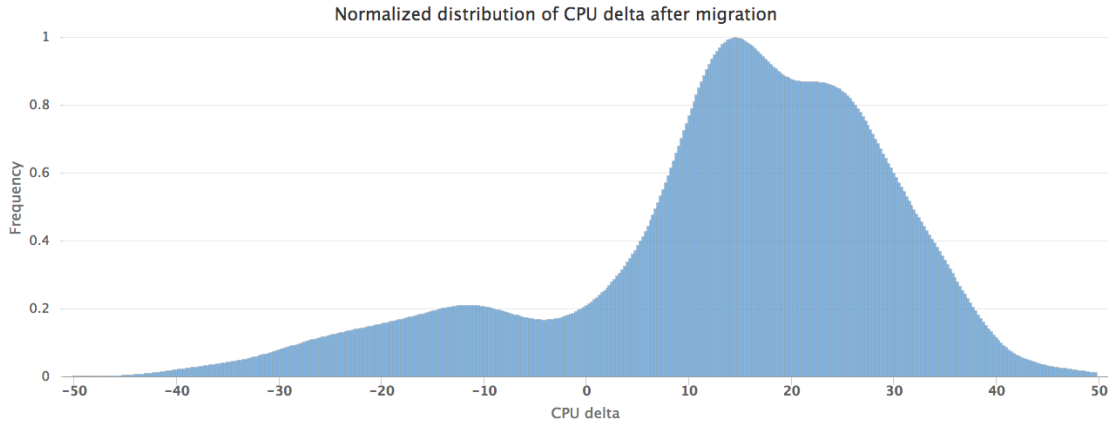

FIGURE 6.18: Operators migration cpu delta percentage

- What is the performance gain a tasks get after migration?
- How is this reflected on resource utilization?
- What is the progress lost for the query from migration and how long it takes to recover it?

To evaluate the performance gain we measured the throughput of each operator just before migration and after the migration. The second measurement was taken a few minutes after the operators starts so it has time for initialization. To simulate a realistic workload we ran a diverse set of queries and measured more than 1,000 operators before and after migration. In figure 6.18 we illustrate CPU measurements normalized as a Gaussian distribution over a set of 400 datapoints.

We can observe than the mean is around 20% increase in CPU post-migration and more than 80% of the migrations have an expected outcome of 10%-30% increase which is quite good considering the non-determinism associated to scheduling and the mixed workloads used for the benchmarks. We can also observe that in some cases operators end up using less CPU which means the migrations have a regression effect. This is not desired but to some extent impossible to anticipate if at the time of the migration the destination host is "free" and soon after it becomes heavy loaded. Because of the dynamic utilization it is inevitable that some decisions will be detrimental but as long as those are rare the scheduler can move the operator again so the query throughput will eventually increase.

The most important question that we have set ourselves to evaluate our system is the actual performance gain measured after an operator is moved. To evaluate this we first modified our scheduler in order to wait a few minutes before before migrating any operators so the operators can achieve their peak performance. Secondly we "normalized" the queries used for this benchmark so that all achieve around 70 events per second under optimal conditions. After these we ran the same benchmarks as before and measured the performance gain as a percentage of the before-migration throughput. In figure 6.19 we illustrate the throughput delta measured from more than 1,000 migrations normalized as a Gaussian distribution over 480 datapoints. We can observe a distribution correlated to the one measured for CPU with 80% of the values ranging between

a 10% and 50% increase. As before we can see that some migrations are detrimental but the percentage is too low to be a concern. Lastly we can observe three "peaks" in the distribution at 9% 30% and 48% which is probably correlated to a small particularity in the resource utilization patterns from our benchmark.

During a migration the query throughput is inevitability affected since streaming data cannot pass through the affected operator for the movement duration. We observed from the previous graph that most a significant percent of the queries improve their performance after the "straggling" operator was moved and we also measured how long it takes for an operator to "recover" in section 5.2. To measure the migration impact on the query we want to evaluate how long it takes to recover the lost progress arising from migration. To evaluate that we combined the two measurements considering only the successful migration that will recover eventually. We illustrate the results in Figure 6.20 normalised as a Gaussian distribution as above. First we can notice that on average a query needs around 10 seconds from the start of the migration to recover the lost progress. Taking into account that an operator needs 4 seconds on average to start again on a new host it means that in only 6 seconds the query manages to recover the short pause. Next we can observe that 90% of the queries recover within 30 seconds. Finally we note that in some cases we need more than one minute to recover but this is very rare.

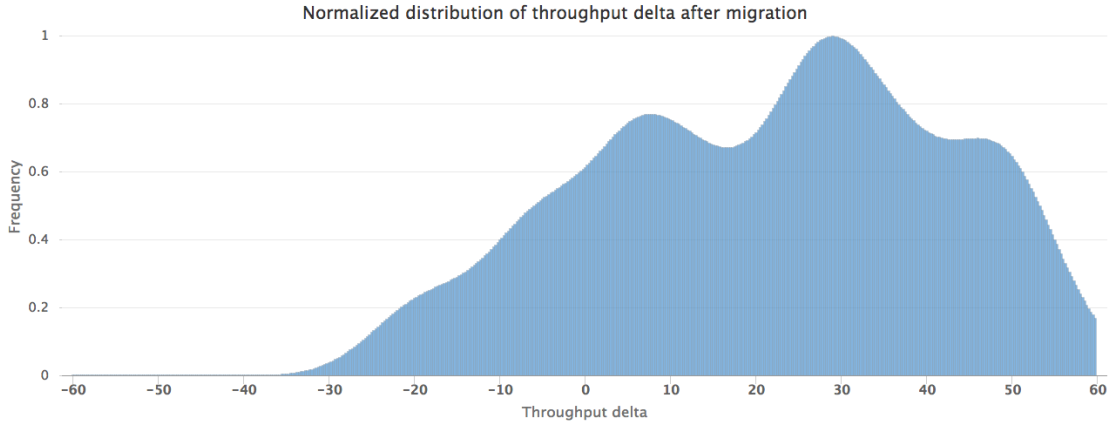

FIGURE 6.19: Operators migration delta performance percentage

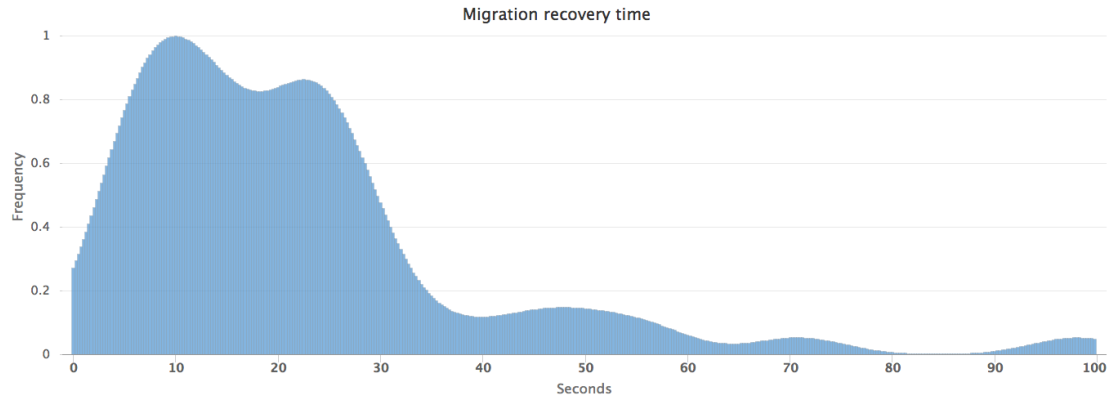

FIGURE 6.20: Query recovery time after migration

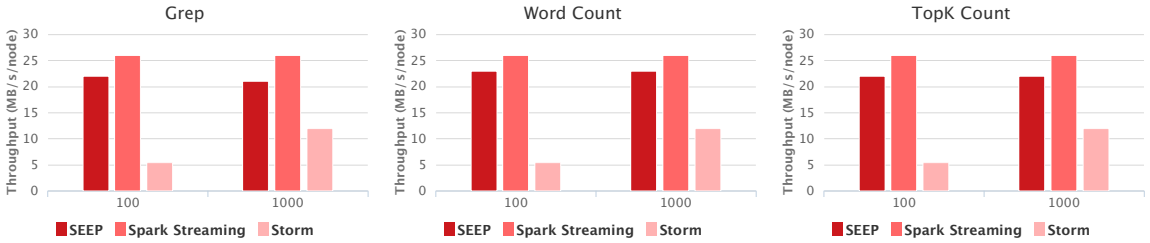

FIGURE 6.21: Node I/O Throughput with SEEP, Storm and Spark Streaming

6.4 Comparison with other systems

6.4.1 Spark Streaming and Storm

We also tested the performance of our system in comparison with two other widely used, open source distributed streaming systems. Both frameworks are continuous operators-based systems, similar to SEEP. Spark Streaming was developed at the University of California Berkeley and while Storm was developed at Twitter and later on open-source as an Apache project. To evaluate our system we implemented the same benchmarks used to evaluate Spark Streaming [43]: Grep, which finds the number of strings matching a pattern, WordCount, which counts the number of words over a sliding time window and TopK, which find the most frequent K words over the past 30 seconds. To evaluate our system we ran SEEP on 2 "xlarge" nodes in Amazon EC2, each having 4 cores, 16 GB of RAM and SSD in order to match the same setup used by Spark. In Figure 6.21 we illustrate the throughput measured as MBs per second per node with the three frameworks.

Performance In the first graph we can observe that SEEP has 2x smaller performance than Spark for Grep. This is a consequence of the fault-tolerance overhead of Kafka since every piece of data is replicated and the query is bounded by I/O performance. Despite this, it is still faster than Storm which seems to be affected by small records size in all of the three experiments.

In the second chart we can see that SEEP has a comparable performance with Spark since this query is CPU bounded and Kafka doesn't play a significant role. The same results are visible in the third benchmark. We can conclude that SEEP performs pretty well without a visible overhead from fault-tolerance when the query is CPU bounded.

6.4.2 Comparison with Naiad

Naiad [10] is a state-of-the-art scheduler capable of executing both streaming and batch jobs connected together in a complex manner by supporting both cyclic and acyclic computations. For this reason Naiad processing system is more powerful than SEEP being able to execute arbitrary iterative algorithms. We set ourselves to the task of comparing the performance of our system along with the overhead added from our scheduler by implementing the same benchmark that Naiad used in Microsoft's Research paper.

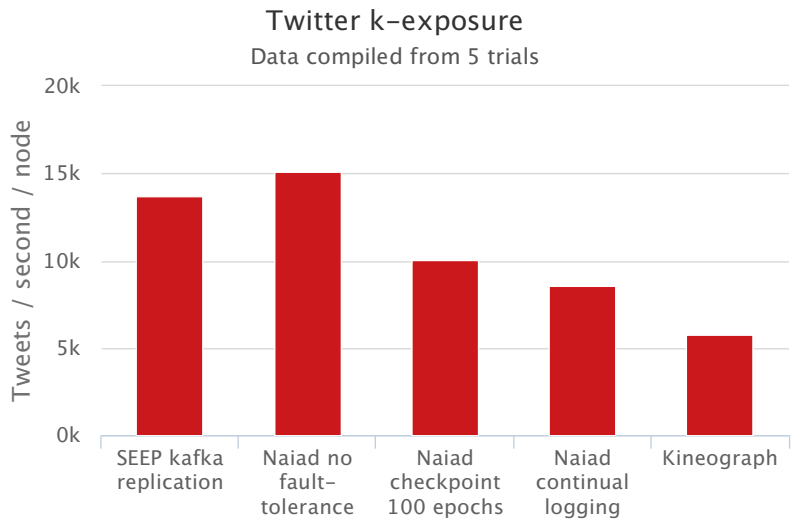

FIGURE 6.22: Twitter k-exposure throughput with SEEP, Naiad and Kineograph

This benchmark is based on application on Kineograph [44], a distributed system that constructs dynamic graphs representing various relations in the data in real-time. One of the applications of Kineograph used to evaluate Naiad is called *k-exposure*. This algorithm is based on the observation that specific topics, known as *hashtags* on the Twitter platform, have a different spreading pattern over time, such as political or celebrity related posts. To study this pattern the authors of Kineograph used an algorithm proposed in [45] that calculates for each *hashtag* the *k-exposure* histogram. For a user U that posts a message with *hashtag* H at time t we define exposure $k(H)$ as:

$$\text{k(H)} = \text{size(neighbours of } U \cap \text{ users who posted a message with } H \text{ at time} \leq \text{t)}.$$

Where "neighbours" of U are all the users that mentioned U at that time. Note that to calculate *k-exposure* we need to maintain a mapping between *hashtags* and users and between users and mentions. We implemented k-exposure in SEEP in 31 lines of code A.5 as a query composed of three operators. The *Sink* ingests Twitter streams of *tweets*, the *Processor* computes *k-exposure* and finally the *Sink* shows the result to the end-user.

For this experiment the authors of Naiad used 32 computers with comparable hardware to ours. Because we had only 6 nodes available we measured the throughput per node and compared to their overall divided by 32. In Figure 6.22 we show the throughput in tweets per second measured over 5 trials. We also show the results achieved by the author of Kineograph in a similar setup [44]. We can observe that SEEP with 13,753 tweets / second is slower than Naiad without fault-tolerance by less than 10%. However if Naiad writes every 100 epochs checkpoints to disk their performance downgrades to 10,000 tweets / second. During the benchmark the load distribution had an average fairness factor of 0.985 for machines and 0.871 for operators. We can conclude that SEEP manages to out-perform Naiad in this setup if both system provide fault-tolerance to some extent.

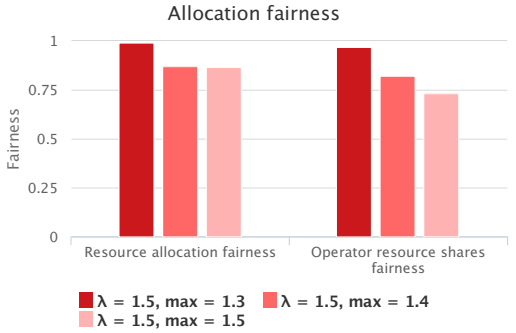

FIGURE 6.23: Allocations fairness for lambda = 1.5

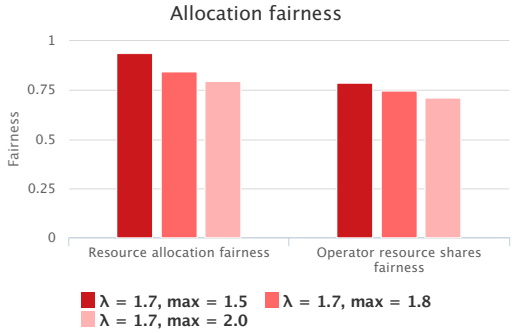

FIGURE 6.24: Allocations fairness for lambda = 1.7

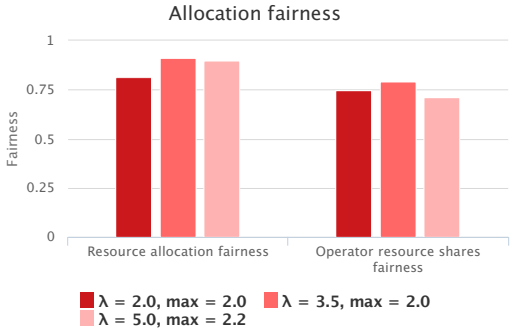

FIGURE 6.25: Allocations fairness for varying lambda

6.5 Varying Strategies

One of the most difficult parts in designing our scheduler was to fine tune our heuristics used for estimating task potential and scoring resource utilization. From the beginning we made our scheduler to depend on user-defined parameters: *potential.lambda* and *potential.max* used to estimate the expected the "potential" as explained in section 4.5.2.

To be able to pick the best values for our heuristic we ran CPU intensive workloads while configuring the scheduler with different parameters and measured both task allocation fairness and host load fairness using Jain's equation [42]. To account for non-determinism that arises frequently in scheduling we ran the same benchmark 5 times and plotted the mean of the measurements. In figure 6.23 we can observe the fairness for lambda equal 1.5 and different maximum values. Next in figure 6.24 we show the results for lambda equal to 1.7 and in figure 6.25 for higher values of lambda. Note that by increasing the maximum potential we estimate higher resource needs for workers. The main trade-off is between under-estimation, which leads to suboptimal migrations and over-estimations which result in avoiding potential good migrations by a smaller margin. The lambda variable, which characterizes the exponential distribution, controls the "steepness" of the estimation just before the maximum. Based on this experiment, we decided to choose lambda 1.5 and maximum 1.3 which yields the highest outcome for both worker, allocation and cluster load fairness.

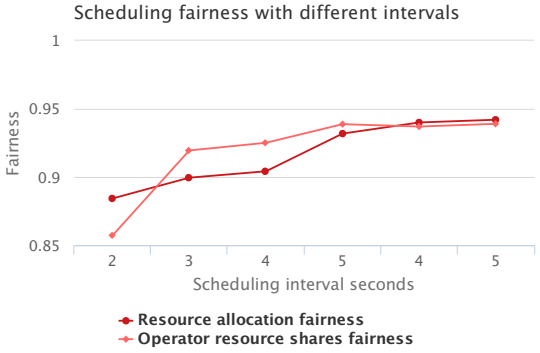

FIGURE 6.26: Scheduling fairness with different scheduling intervals

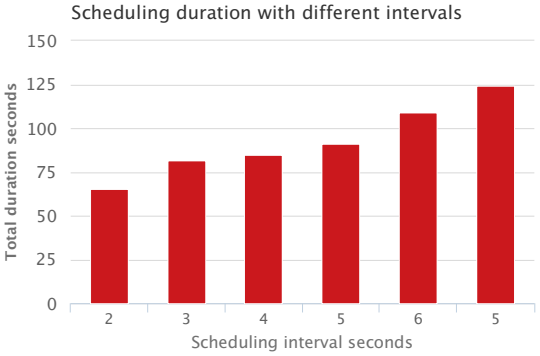

FIGURE 6.27: Scheduling duration with different scheduling intervals

Finally, we conclude our experiments with an analysis of what effect different scheduling intervals have on fairness and duration. Our scheduler runs an algorithm that migrates tasks every *scheduling.interval* seconds based on average resource utilization measurements over an time interval. Because we don't move more than one operator from/to a host in one round the *scheduling.interval* directly influences the time needed to even the load on cluster. To evaluate this we measured the workload fairness with different *scheduling.interval* as well as the time needed to distribute the load evenly from a setup where all the tasks are running on the same node. For this experiment we deployed I/O and CPU intensive workloads composed of 30 operators on a cluster of 6 nodes and ran each experiment 5 times. In Figure 6.26 we illustrate the average fairness and in Figure 6.27 we show the average total duration measured as the tine taken since we deploy the queries to the last migration decision. We can observe how the total duration grows linearly as we increase the decision interval. Although the overall delay seems significant in this case we deployed all the operators on one node hence a lot of scheduling decisions were required resulting in an increased latency. Next we can see how the fairness is less for small scheduling intervals which is not surprising since the decisions are more vulnerable to utilizations spikes. Finally we notice that the fairness peaks at 0.95 starting from a 5 second interval.

6.6 Summary

The benchmarks we ran have shown that our scheduler maximizes throughput under different workloads and always achieves higher performance than YARN's baseline. To evaluate our scheduler we created a diverse set of benchmarks ranging from simple toy queries to realistic applications. To quantify the performance we measured different metrics from resource consumption and throughput to allocation fairness. In order to fine-tune our algorithm we varied the parameters used in estimations and picked the best among the variations. Finally we compared our scheduler with other commercially available schedulers and processing frameworks.

We are aware that there are a lot of improvements to be made to our scheduler but we believe that the current system achieves a good performance and more importantly is reliable under various workloads. TO extend it we may want to support scheduling any type of tasks not only SEEP

streaming queries. With a greater increase in generality we could benchmark the system for its scheduling delay when dealing with sub-second batch tasks.

Finally, we would like to point out that running all the benchmarks proved to be much more difficult than expected. The first problem is the high-grade of non-determinism involved in scheduling. To get reliable measurements we needed to run each experiment a couple of times. Since some of the experiments lasted one hour for each of the thee scheduling strategies we needed to run for half a day just to get one statistically significant measurement. Although we developed a specialized benchmarks module that deploys, runs the queries and collects key metrics automatically some of the experiments were not straight-forward. For example to measure migration trade-offs we modified both the SEEP code and the scheduler to record metrics at certain time points. To gather all the measurements submitted from many clients around the cluster we created a statistic server that works as a key-value store for measurements. Secondly to compare our system with Spark we deployed everything on Amazon Ec2 which proved to be far more time consuming than anticipated. Despite all the bash scripts that we have written deploying our system, Kafka and YARN needs a few manual configuration and installations dependent on the machine.

Chapter 7

Conclusion

We have built an entire new system from scratch which was a rewarding experience in the end but had many challenges along the way. While developing it, we had to deal with many typical system aspects such as: scalability, non-determinism, network deadlocks and fault-tolerance. Moreover we had the joy to observe, study and finally solve many bottlenecks that limited the performance of our system during development.

Through the project we familiarized ourselves with large frameworks such as SEEP, YARN and Kafk and integrated them in our system. To achieve that we had to comprehend poorly written documentation and dive into large codebases if the first was not sufficient. This way we learnet how important good documentation and well designed API are for every large system. Looking backwards we spent more than half of our time to understand those frameworks and make them work together.

From the beginning of the project we faced many important engineering decisions where we carefully analysed the trade-offs for each approach. For example, initially we considered a few languages for writing our scheduler such as Go, Erlang and Python. The latter had the best documentation as well as rich support for communication and external libraries but we were concerned for its performance. After running some experiments we realised that we can develop a system that scales to more than 500 nodes in Python by writing the communication logic in a multi-threaded and asynchronous fashion and delegating performance sensitive bits to native libraries.

In this project we managed to design a new job scheduler specialized for streaming tasks but also reliable for batch computations. We identified a common problem and performance bottleneck present in state-of-the-art stream processing frameworks [11, 23, 39] and managed to solve it by migrating tasks at runtime based on fine-tuned heuristics. Along the way we build a scalable analytics framework that is easy deployable to any cluster and we use it to reason about the performance impact of various approaches while building our scheduler. Moreover we made our system fault-tolerant and ensured that no single failure point exists.

To evaluate our system we developed a comprehensive set of benchmarks with various resource usage patterns. We made all the benchmarks self-contained and simplified the code extensively in order to make it easy for other people to use it in the future when developing stream processing frameworks. Lastly we tested our system with different workloads and fine-tuned the heuristics

of our scheduler to maximize the throughput and minimize the migration overhead in different scenarios. We exposed all the configuration from the WebUI so users can easily adapt the system to their needs.

7.1 Future work

We think there are many areas were we can improve our scheduler to support more features and complex scheduling patterns. Following we present some ideas:

- **Task scheduling in rounds**
  Our current system allocates all the tasks submitted even if they won't be able to perform to their maximum potential by allowing them to share the available resources evenly between them. In some scenarios it would be better to schedule a few applications at a time picked in a round-robin fashion so over time each has a equal share of the cluster at its full desired utilization. We need to carefully consider the overhead since simply stopping the tasks could be to extensive. We already implemented support to pause and resume operators in SEEP but we haven't created yet an algorithm in the scheduler to use it.

- **Multiple framework integration**
  Currently our scheduler is build on top of YARN, SEEP and Kafka. To provide a broader use-case we can extend it to schedule arbitrary tasks and have support for both Mesos, YARN and even lower level resource isolation frameworks such as Linux Container and Dockers. All these systems have their own advantages and disadvantages so it would be desired to be able to change them depending on the scenario.

- **RAM aware migration**
  While CPU and I/O scheduling covers a broad range of use-cases there are a few distrusted applications that utilize a lot of RAM memory. We plan to extend our scheduler to migrate tasks based of memory consumption. This is not very common because by default YARN uses strict resource isolation for memory so applications can't go over their limits to increase performance. Furthermore, an application cannot scale elastically if more memory is available as is the case of CPU and I/O. Moreover, if we allow applications to dynamically extend their RAM usage it will be dangerous if the machine runs out of memory and starts using swap space which would slow down all the computation significantly. Despite all of the above, RAM scheduling is something interesting to explore in the future.

- **Enrich support for iterative jobs**
  At the moment our system can execute batch jobs, only by implementing a limited duration streaming query, which is not ideal. Furthermore, some applications need complex communication among multiple queries running in parallel as is the case of MapReduce computations. We want to extend our scheduler to support an arbitrary graph as connections between tasks that are deployed, even if they are streaming or batch computations. To achieve that we need to configure output and input connections directly from the scheduler. If we support this we will be able to support arbitrary complex computations ranging from MapReduce to iterative machine learning algorithms running in a distributed fashion.

We have considered a few approaches to build more complex tasks semantics. One idea that we consider very suitable is to create a "timely datafflow graph" as researchers from Microsoft Research used in their own scheduler, Naiad [10].

- **Different scheduling algorithms**
  Currently our scheduler uses one of the classical bin-packing algorithms to match "straggling" operators to "free" nodes. We could extend our design to support a pluggable module that runs this heuristics. This way we could easily experiment with different algorithms such as *MatchMaker*, *MTP* or *BinCompletion* as well as allow users to supply their own algorithm as YARN does.

- **Back-pressure mechanism**
  We discovered in a state-of-the-art stream processing system published this month [39] the "back-pressure" technique which is a way of dynamically controlling the streaming flow throughput. If we implement a similar technique we could control with fine granularity the resource consumption of SEEP operators by limiting the incoming streaming flow. Furthermore, it is common in streaming processing queries that one operator produces more data than others can process. If we use an external messaging system all the excess data is buffered on disk. Ideally we would want our scheduler to observe this pattern and reduce the downstream operators output to prevent a constantly growing buffer.

- **Usability**
  Our system is designed to be used by other people either as a scheduler or just as an analytics framework. However, at the moment it is not possible to simply install our system and use it out of the box. One needs to configure manually both YARN, SEEP, Kafka and provide required server side settings. We want to minimize the number of configurations steps and augment our system with a detailed documentation. Finally it would be interesting to measure the *ease of use* of our system compared to other available solutions.

Appendix A

Benchmark Overview

For completeness we will present the code used in our benchmarks so the experiments can be replicated with the same setup. For the following benchmarks we show only the important operators i.e.

A.1 CPU benchmark: RSA factorization

```
1  public void processData(ITuple data, API api) {
2    long ts = 0;
3    while (working) {
4      long p = getRandomPrime(6000000, 6500000);   // Use primality test to find
5      long q = getRandomPrime(6500000, 7000000);   // primes. O(log n) * O(log^3 n)
6      long N = p * q;
7      long e = 2;
8
9      long phi = (p - 1) * (q - 1);
10     while (BigInteger.valueOf(e).gcd(BigInteger.valueOf(phi)).intValue() != 1)
11         ++e;
12
13     Random rand = new Random();
14     long x = rand.nextLong() % N;
15     long ex = modPow(x, e, N);     // Modular exponentiation
16     api.send(ts++, e, N, ex);      // Sent data to next operator
17   }
18 }
```

Listing A.1: RSA factorization - source operator

```
public void processData(ITuple data, API api) {
  long ts = data.getLong("ts");
  long e = data.getLong("pubE");
  long N = data.getLong("pubModulus");
  long ex = data.getLong("secret");

  long p = 2, q = 1;
  while(N % p != 0) // Brute force factorization
   ++p;
  q = N / p;

  long d = inverse(e, (p - 1) * (q - 1)); // Modular multiplicative inverse
  long dx = modPow(ex, d, N);             // Modular exponentiation
  api.send(ts, e, N, dx);
}
```

LISTING A.2: RSA factorization - processor operator

A.2 I/O benchmark: Virus Scanner

```
public void processData(ITuple data, API api) {
   int ts = 0;
   while(working) {
    Scanner sc = new Scanner(new FileInputStream("<inputFile>"););
    while (sc.hasNextLine()) {
      api.send(ts, sc.nextLine()); // Read 512Kb line from the file
    }
    inputStream.close();
    sc.close();
   }
}
```

LISTING A.3: Virus Scanner - source operator

```
public void processData(ITuple data, API api) {
  public void processData(ITuple data, API api) {
  long text = data.getLong("ts");
  String text = data.getString("text");
  // Signatures is a preinitialized hash set with the malicious patterns
  if (signatures.contains(text)) {
   api.send(ts, text);
  }
}
```

LISTING A.4: Virus Scanner - processor operator

A.3 CPU and I/O benchmark: Permutation Cipher

```
public void processData(ITuple data, API api) {
   int ts = 0;
   Random random = new Random();
   while(working){
     // Create 100Kb text and send it over the network
     String text = new BigInteger(256000, random).toString(32);
     int key = random.nextInt(128);
      api.send(ts++, key, text);
   }
}
```

Listing A.5: Permutation Cipher - source operator

```
public void processData(ITuple data, API api) {
   int ts = data.getInt("ts");
   int key = data.getInt("key");
   String text = data.getString("text");

   StringBuilder sb = new StringBuilder();
   for(int i = 0; i < text.length(); i++)
    sb.append((char)(text.charAt(i) ^ key));
   api.send(ts, key, sb.toString());
}
```

Listing A.6: Permutation Cipher - processor operator

A.4 Twitter word count

```
public void processData(ITuple data, API api) {
  while(working) {
    // Read from socket channel that delivers batches of 1,000 tweets
    Object obj=JSONValue.parse(socket.readLine());
    JSONArray messages = ((JSONArray)obj);
    for (Object message : messages) {
      String text = ((JSONObject)(message)).get("text").toString();
      String user = ((JSONObject)(message)).get("user").toString();
      api.send(ts++, text, user);
    }
  }
}
```

Listing A.7: Twitter word count - source

```
public void processData(ITuple data, API api) {
  String text = data.getString("text");
  for (String word : text.split(" ")) {
    // Ignore prepositions, pronouns, etc
    if (!ignoredWords.contains(word)) {
      Integer prevCnt = wordCount.get(word);
      wordCount.put(word, (prevCnt == null ? 0 : prevCnt) + 1);
    }
  }
  if (System.currentTimeMillis() - timestamp > 10000) {
    String mostFrequentWord = null;
    long count = -1;
    for (Entry<String, Integer> entry : wordCount.entrySet()) {
      if (mostFrequentWord == null || entry.getValue() > count) {
        mostFrequentWord = entry.getKey();
        count = entry.getValue();
      }
    }
    wordCount.clear();
    api.send(count, mostFrequentWord);
    timestamp = System.currentTimeMillis();
  }
}
```

Listing A.8: Twitter word count - processor

A.5 Twitter k-exposure used to detects controversial topics

Here we use the same source as in Twitter word-count previous query.

```
public static long getIntersection(Set<String> set1, Set<String> set2) {
  boolean set1IsLarger = set1.size() > set2.size();
  Set<String> cloneSet = new HashSet<String>(set1IsLarger ? set2 : set1);
  cloneSet.retainAll(set1IsLarger ? set1 : set2);
  return cloneSet.size();
}

public void processData(ITuple data, API api) {
  String text = data.getString("text");
  String user = data.getString("user");

  for (String word : text.split(" ")) {
    if (word.isHashTag())) {
      word = word.substring(1, word.length());
      HashSet<String> nodes;
      if (!neighbours.containsKey(user))
        nodes = new HashSet<String>();
      else
        nodes = neighbours.get(user);
```

```
20        nodes.add(word);
21        neighbours.put(user, nodes);
22
23        long count = getIntersection(neighbours.get(user), posters.get(word));
24        api.send(count, word);
25      }
26      if (word.isUserMention()) {
27        HashSet<String> users;
28        if (!posters.containsKey(word))
29          users = new HashSet<String>();
30        else
31          users = posters.get(word);
32        users.add(user);
33        posters.put(word, users);
34        }
35      }
36    }
37  }
```

Listing A.9: Twitter k-exposure - processor operator

Appendix B

User Manual

We will describe next how our system can be used through the WebUI.

Admin Panel We begin by showing the admin panel in Figure B.1 that bring together the most common functionalities that we needed though the development. The left most panel contains short-cuts to main YARN components and provides a way to clear all the YARN logs across the cluster by running in background a *rm* command on each machine. The middle panel has the query deployment tools from where the user can start and stop seep queries and has gives the ability to update SEEP to a certain branch from of the repository. When a user clicks the button each machine runs *git fetch, git reset*, compiles all the SEEP code and re-generates the examples jars. Finally the last box updates our system as a whole including the scheduler, analytics system and WebUI. This works by running bash script in the background that stop the server and all the modules, get the latest change from the repository and restart back everything. The last button clear the Kafka logs which we had to use quite frequently while running experiments since during an intensive IO load we can write up to 500Mbs / second on disks across the cluster.

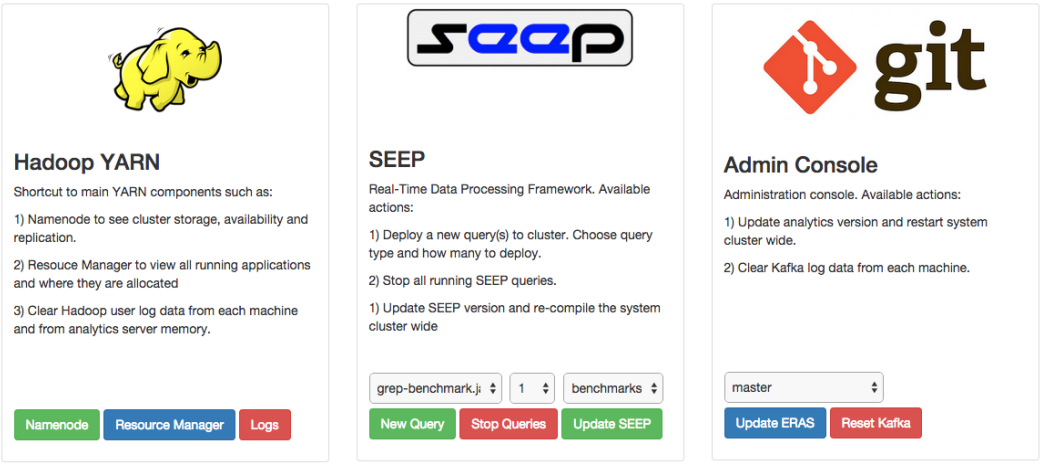

FIGURE B.1: Admin panel overview

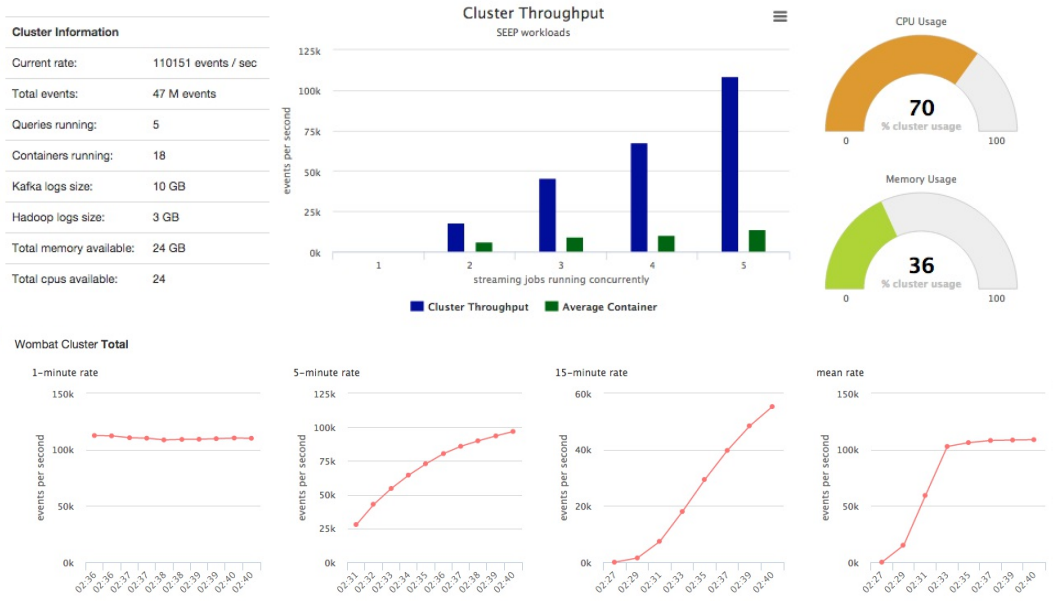

FIGURE B.2: Cluster Overview

Cluster Overview This page we illustrate in Figure B.2 gives general information about the cluster such as overall throughput by number of queries as well as graphs that illustrate the performance evolution over time measured as rolling averages of 1,5,15 minutes as well as from the beginning. The top left panel shows real-time informations such as: current throughput, number of events so far, the number of queries and containers running, memory and cores availability and disk space occupied by logs. Finally in the top right panel we can see the current CPU and memory usage updated every second.

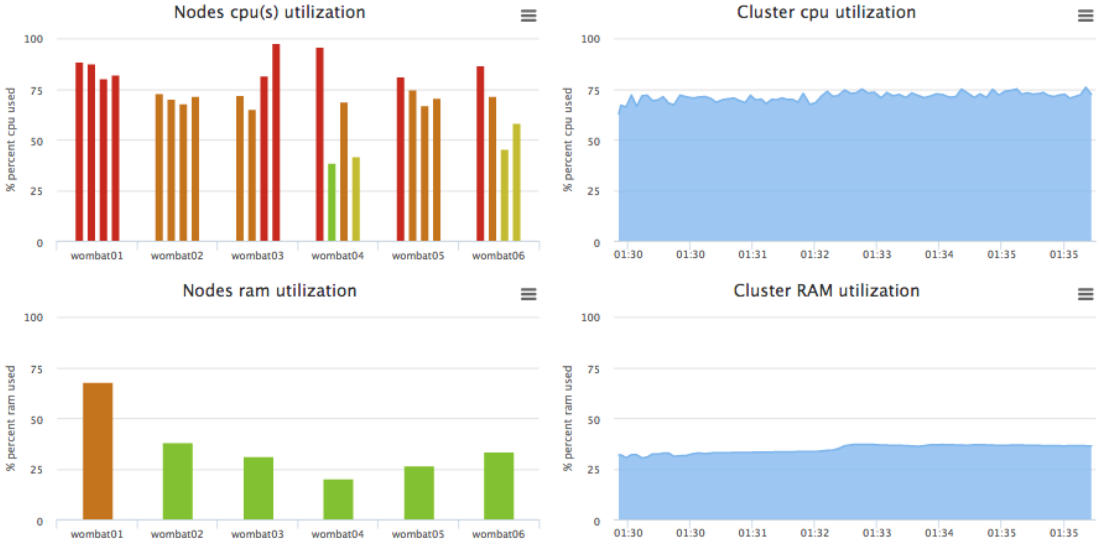

FIGURE B.3: CPU and memory graphs

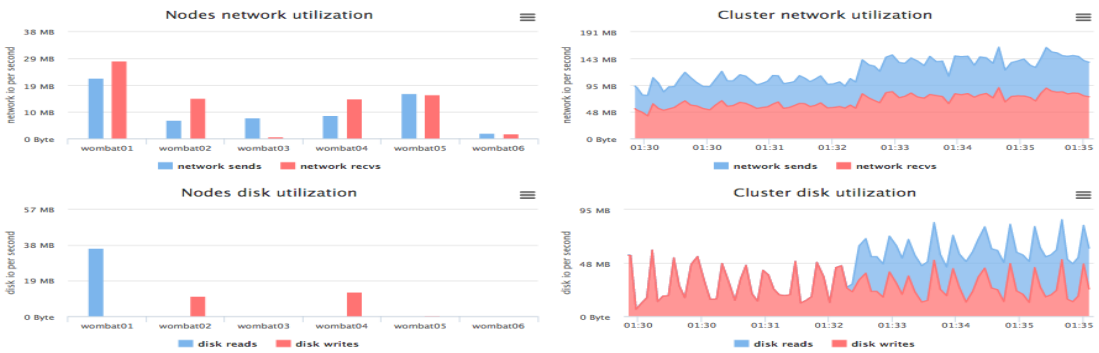

FIGURE B.4: Disk and network I/O graphs

Resource utilization This panel from Figure B.3 shows on the left side the current CPU utilization measured for each core in the cluster and the RAM utilization measured for each machine. On the left hand side we can see how the utilization changes over time. We use a colour scheme to emphasize the most "crowded" cores or machines. The graphs are populated with data retrieved every 5 seconds and accumulates until we refresh the page. This way we can observe the impact of deploying new queries over the cluster. We also show the same information for both I/O and Network as we illustrate in Figure B.4. For disk we show *bytes read/write* and for network *bytes sent/received*.

In Figure B.5 we provide an overview of workers which run logical operator from the queries. We group each worker by the node where it's running and we show extra information useful when debugging such as *process id*, *worker id* and name of the *logical operator* running inside the job. We provide additionally CPU and memory utilization measured for each worker individually and we use the same colour scheme as before to highlight the percentage of resource usage. Finally we used a dashed pink colour to represent an operator that was just moved to another host.

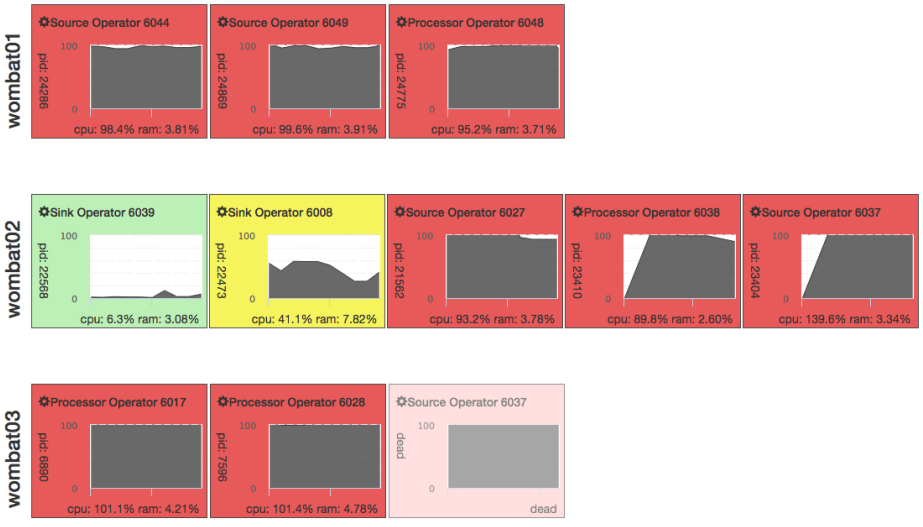

FIGURE B.5: Operators overview

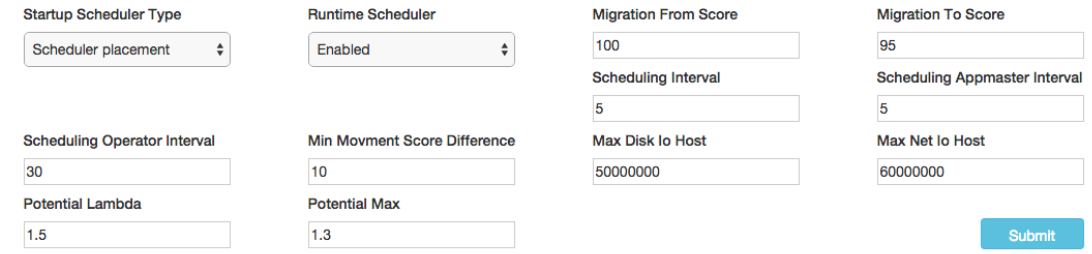

FIGURE B.6: Scheduler configuration overview

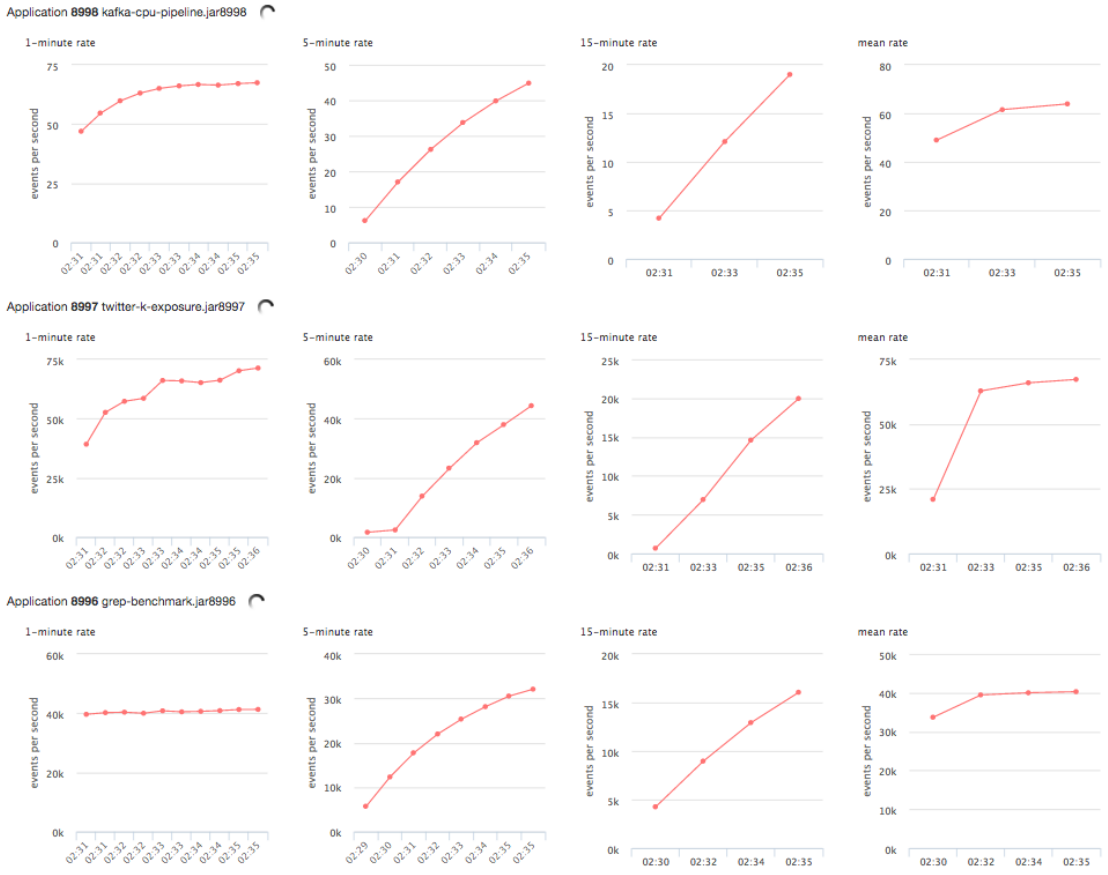

FIGURE B.7: Applications metrics panel

To allow the users to easily configure our scheduler we provide access through the WebUI to all the internal scheduling parameters as we can see in Figure B.6. Furthermore we support changing the scheduling strategies or failing back to YARN in real-time. All the configuration changes are sent to the scheduler module which updates his stored values in real-time.

Finally Figure B.7 we show throughput over time for each application running in the cluster updated every 30 seconds. The 4 graphs show different levels of granularity by aggregating metrics over 1,5,15 and overall sliding windows.

Appendix C

Software verification in the cloud

We show how is possible to build a static and dynamic software verification system in the cloud [1] with the help of SEEP and the system we designed, ERAS (efficient resource aware scheduler). This system models realistically typical cluster workload as the resource utilization for CPU and RAM memory varies greatly in depending in the program that are analysed. Furthermore the load is dynamic because our system is based on user submitted programs for analysis. This type of workload would represent decrease further the performance of static schedulers such as YARN.

To handle user interaction we created a simple python server that accepts programs written in C-like syntax and send them for verification. The server acts also as a load balancer by sending programs in a round-robin fashion to one of the running SEEP verification queries. The *Source* operator waits to server queries and dispatches them to the *Processor* operator that does the heavy lifting by calling our software analysis module. During the verification our module transforms the program to a SMT formula and then we run the Z3 Theorem Prover to check if the formula can be proven. The verification process can swan multiple threads to run different analysis strategies in the same time. Because of that the resource consumption can fluctuate a lot at runtime. Since some programs need up to a minute to be verified it is often desirable to migrate an operator on a new host and finish the verification faster. Next we briefly illustrate in Figure C.1 and Figure C.2 the resource utilization pattern over time while running 6 verifiers concurrently with a average throughput of 40 programs per second.

[1] based on a software verification module which we developed in a previous project https://github.com/andrei-alpha/SRTool

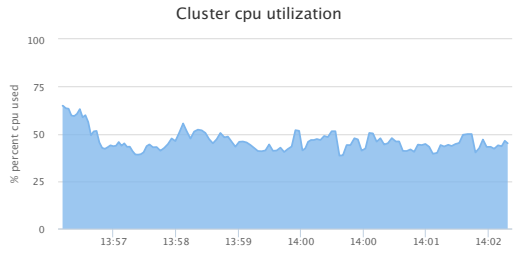

FIGURE C.1: CPU with highly dynamic load

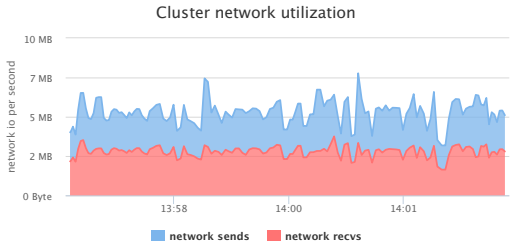

FIGURE C.2: Disk IO with highly dynamic workload

Bibliography

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *OSDI04: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation, 2004.*

[2] J. Dean and S. Ghemawat. Mapreduce: A Flexible Data Processing Tool. *Communications of ACM, 53(1), 2010.*

[3] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg Quincy: fair scheduling for distributed computing clusters. *Proceedings of SOSP (2009).*

[4] M. Schwarzkopf., A. Konwinski, M. Abd-El-Malek, J. Wilkes Omega: flexible, scalable schedulers for large compute clusters *EuroSys'13 April 15-17,2013, Prague, Czech Republic.*

[5] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, R. Fonseca Jockey: Guaranteed Job Latency in Data Parallel Clusters *EuroSys'12 April 10-13,2012, Bern, Switzerland.*

[6] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management *SIGMOD'13, June 22-27,2013, New York, New York, USA.*

[7] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica Sparrow: Distributed, Low Latency Scheduling *SOSP'13, Nov 3-6,2013, Farmington, Pennsylvania, USA*

[8] Making State Explicit for Imperative Big Data Processing R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch *USENIX'14, June 17-20, Philadelphia, PA, USA*

[9] Network-Aware Operator Placement for Stream-Processing Systems. P. Pietzuch, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer *ICDE'06 Proceedings of the 22nd International Confrence on Data Engineering*

[10] D. G. Murray, F. McSherry, R. Issacs, M. Isard, P. Barham, M. Abadi Naiad: A Timely Dataflow System. *SOSP'13, Nov. 3-6, 2013, Farmington, Pennsylvania, USA.*

[11] Apache Samza. `http://samza.incubator.apache.org/`

[12] Apache Hadoop. `http://hadoop.apache.org/` 2009

[13] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J. S. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray H . Kuang, A. Menon, A. Aiyer Apache Hadoop Goes Realtime at Facebook *SIGMOD'11 111, June 12-16, 2011, Athens, Greece*

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, I. Stoica Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center *NSDI'11 Proceedings of the 8th USENIX conference on Networked systems design and implementation Pages 295-308* `http://mesos.apache.org/`

[15] B. Hindman, M. Konwinski, M. Zaharia, I. Stoica, A Common Substrate for Cluster Computing. *In Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (2009).*

[16] V. K. Vavilapallih, A. C. Murthyh, C. Douglasm, S. Agarwali, M. Konarh, R. Evansy, T. Gravesy, J. Lowey, H. Shahh, S. Sethh, B. Sahah, C. Curinom, O. OMalleyh, S. Radiah, B. Reedf, E. Baldeschwielerh YARN: Yet Another Resource Negotiator *SoCC13, 13 Oct. 2013, Santa Clara, California, USA*

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica Improving MapReduce Performance in Heterogeneous Environments *OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation Pages 29-42*

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, D Fetterly Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks *EuroSys07, March 2123, 2007, Lisboa, Portugal*

[19] Docker. `https://www.docker.com/`

[20] Linux containers (LXC). `https://linuxcontainers.org/`

[21] Apache Storm. `https://storm.apache.org/`

[22] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy SIGMOD14, June 2227, 2014, Snowbird, Utah, USA

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica Spark: Cluster Computing with Working Sets em Proceeding HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing Pages 10-10

[24] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop em CIDR 2015. 7th Biennial Conference on Innovative Data Systems Research. January 4-7, 2015, Asilomar, California, USA.

[25] RabbitMQ `http://www.rabbitmq.com/`

[26] J. Kreps, N. Narkhede, J. Rao Kafka: a Distributed Messaging System for Log Processing *NetDB'11, Jun. 12, 2011, Athens, Greece.* `http://kafka.apache.org/`

[27] Apache Thrift. `https://thrift.apache.org/`

[28] Amazon EC2. `http://aws.amazon.com/ec2`

[29] P. Hunt, M. Konar, F. P. Junqueira, B. Reed ZooKeeper: wait-free coordination for internet-scale systems *USENIXATC'10 Proceedings of the 2010 USENIX conference on USENIX annual technical conference Pages 11-11* `hadoop.apache.org/zookeeper`

[30] Protocol Buffers `https://developers.google.com/protocol-buffers/`

[31] Facebook's Scribe. `http://wiki.github.com/facebook/scribe`

[32] J. Kreps Kafka Benchmark `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`

[33] A. Cockcroft, D. Sheahan Benchmarking Casandra Scalability `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`

[34] P. Vagata, K. Wilfong Scaling the Facebook data warehouse to 300 PB `https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/`

[35] S. S. Seiden On the online bin packing problem *Journal of the ACM (JACM) JACM Homepage archive, Volume 49 Issue 5, September 2002 ,Pages 640-671*

[36] S. Martello, D. Pisinger, D. Vigo The Three-Dimensional Bin Packing Problem *Operations Research 2000, 48, 2, 256–267*

[37] B. H. Murray, A. Moore Sizing the Internet `http://www.cs.toronto.edu/~leehyun/papers/Sizing_the_Internet.pdf`

[38] K. Amin Big Data Overview 2013-2014 `http://www.slideshare.net/kmstechnology/big-data-overview-2013-2014`

[39] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel*,1, K. Ramasamy, S. Taneja Twitter Heron: Stream Processing at Scale *SIGMOD15, May 31June 4, 2015, Melbourne, Victoria, Australia*

[40] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, C. R. Das Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters *Proceeding SOCC '11 Proceedings of the 2nd ACM Symposium on Cloud Computing*

[41] D. J. Abadi, D. Carney, U. etintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik Aurora: a new model and architecture for data stream management *The VLDB Journal The International Journal on Very Large Data, Volume 12 Issue 2, August 2003*

[42] R. Jain, D. M. Chiu, W. Hawe A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems *DEC Research Report TR-301, 1984*

[43] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica Discretized streams: fault-tolerant streaming computation at scale *SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Pages 423-438*

[44] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen Kineograph: Taking the Pulse of a Fast-Changing and Connected World *EuroSys '12, Proceedings of the 7th ACM european conference on Computer Systems, Pages 85-98*

[45] D. M. Romero, B. Meeder, J. Kleinberg Differences in the Mechanics of Information Diffusion Across Topics: Idioms, Political Hashtags, and Complex Contagion on Twitter *WWW '11, Proceedings of the 20th international conference on World wide web, Pages 695-704*

[46] P. Barford and M. Crovella Generating representative web workloads for network and server performance evaluation. *In Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 1998.*

[47] N. Widell Migration Algorithms for Automated Load Balancing *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems : November 9 - 11, 2004, MIT, Cambridge, USA*

[48] T. T. Y. Suen, J. S. K. Wong Efficient Task Migration Algorithm for Distributed Systems *Journal IEEE Transactions on Parallel and Distributed Systems archive Volume 3 Issue 4, July 1992*

[49] J. J. Chen, H. Hsu, K. H. Chuang, C. L. Yang, A. C. Pang, T. W. Kuo Multiprocessor Energy-Efficient Scheduling with Task Migration Considerations *ECTRS, 2004, Pages 101-108*

[50] L. A. Barroso, J. Dean, U. Hlzle Web Search for a Planet: The Google Cluster Architecture *IEEE Micro archive, Volume 23 Issue 2, March 2003, Page 22-28*

[51] J. Lin, D. Ryaboy Scaling big data mining infrastructure: the twitter experience *ACM SIGKDD Explorations Newsletter archive Volume 14 Issue 2, December 2012, Pages 6-19*