Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Lukas Schwerdtfeger

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the
Technical University of Berlin
in partial fulfillment of the requirements for the degree
Bachelor Technische Informatik

Berlin, Germany
December 22, 2015

Main supervisor:

Prof. Dr. habil. Odej Kao, Technical University of Berlin

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

Zusammenfassung

Kurze Zusammenfassung der Arbeit in 250 Wörtern.

Abstract

Short version of the thesis in 250 words.

Acknowledgements

This chapter is optional. First of all, I would like to...

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Description . . . 2
- 1.3 Goal of this Thesis . . . 3
- 1.4 Structure of this Thesis . . . 4

2 Background **7**
- 2.1 Big Data Stream Processing . . . 7
- 2.2 Scheduling . . . 8
- 2.3 Cluster Management Systems . . . 10

3 Approach **15**
- 3.1 Scheduling in Kubernetes . . . 15
- 3.2 Extending Kubernetes using the Operator Pattern . . . 15

4 Implementation **17**
- 4.1 Architecture . . . 17
- 4.2 Designing the Interface . . . 17
- 4.3 Operator . . . 17
- 4.4 Changes to existing Algorithm . . . 17

5 Evaluation **19**
- 5.1 Testing . . . 19
- 5.2 Comparing to baseline Runtime . . . 19
- 5.3 Limitations . . . 19
- 5.4 Discussion . . . 19

6 State of the Art **21**
- 6.1 Volcano . . . 21
- 6.2 Non Kubernetes . . . 21

7 Conclusion and Future Work **23**
- 7.1 Conclusion . . . 23
- 7.2 Future Work . . . 23

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The current population is producing more and more data. This creates an excellent opportunity for many businesses. Businesses willing to profit from collected data by using it to improve their sales strategies have to collect and store GigaBytes and upwards to ExaBytes of data. With storage costs becoming more affordable, companies are even less likely to toss away potential valuable data, creating so-called Data Lakes.

Collecting data is only the first step. It takes many stages of processing, through aggregation and filtering, to extract any meaningful information. Usually, the sheer mass of collected data makes it not very useful, to begin with.

Unfortunately, when working with ExaBytes of data, it is no longer feasible to work on a single machine. Especially when dealing with a stream of data produced by a production system and the information collected from yesterday's data is required the next day, or ideally immediately.

Scaling a single machine's resources to meet the demand is also not feasible. It is either very expansive or might just straight up not be possible. On the other hand, cheap commodity hardware allows the scaling of resources across multiple machines is much cheaper than investing in high-end hardware or even supercomputers.

The complexity of dealing with a distributed system can be reduced using the abstraction of a cluster. A **Cluster Resource Manager** is used, where a system of multiple machines forms a single coherent cluster that can be given tasks to.

Stream Processing of Data across such a cluster can carry out using Stream or Batch Processing Frameworks, such as Apache Spark or Apache Flink. These Frameworks already implement the quirks of dealing with distributed systems and thus hide the complexity.

The problem is that multiple Batch Jobs running on a single cluster need resources that need to be allocated across the cluster. While Resource Allocation is the Task of the Cluster Resource Manager, the manager usually does not know how to allocate its resources optimal and often requires user *(TODO: 1)* to specify the resources that should be allocated per job. This usually leads to either too little resources being allocated per job, starving jobs and increasing the runtime, or more often over-committing resources and thus leaving resources in the cluster

unused.

Another problem that arises is the fact that even though the reoccurring nature of Batch Jobs, not all BatchJobs use the same amount of Resources. Some are more computationally intensive and require more time on the CPU, while others are more memory intensive and require more or faster access to the machine's memory. Others are heavy on the I/O usage and use most of the system's disk or network devices. This shows up in vastly different Job runtime (also total runtime) depending on the Scheduling of Batch Jobs across the Cluster.

Finding an intelligent Scheduling Algorithm that can identify reoccurring Jobs and estimate their resource usage based on collected Metrics and thus create optimal scheduling is not an easy task. It also requires a lot of setup when dealing with a Cluster Resource Manager.

1.1.0.1 TODO:

1. not just a user but the cluster user which is submitting the job

1.1.0.2 Open:

◇ How much detail is required here?

1.2 Problem Description

(TODO: Cluster Resource Manager, like YARN, were focused around Batch-Application because they existed because of Apache Hadoop/Map-Reduce ecosystem)

Cluster Resource Managers, like YARN, emerging from Apache Hadoop, were centered around Batch Frameworks.

With the rise of Cloud Computing and all the benefits that come with it, companies were quick to adopt new cloud computing concepts. The Concept of a Cluster Resource Manager introduced a notion of simplicity to those developing applications for the cloud. A Cluster Resource Manager now managed many aspects that used to be handled by dedicated Operations-Teams.

Kubernetes, a Cluster Resource Manager that was initially developed by Google, after years of internal use, provided an all-around approach to Cluster Resource Management for not just Batch-Application. The global adoption of Kubernetes by many leading companies, led to the growth of the ecosystem around it. Kubernetes has grown a lot since and has become the new industry standard, benefiting from a vast community.

Old Batch-Application-focused Cluster Resource Managers that used to be the industry standard are being pushed away by Kubernetes. Unfortunately, vastly different Interfaces or Scheduling Mechanism between other Cluster Resource Managers usually block the continuation of existing research done in the field of Batch Scheduling Algorithms.

Finding an efficient scheduling algorithm is a complex topic in itself. Usually, the setup required to further research existing scheduling algorithms is substantial. Dealing with different Cluster Resource Manager further complicates continuing on already existing work.

1.2.0.1 TODO:

1.2.0.2 Open:

- ◇ This sections contains a lot of text that may be better suited to the introduction section, but i don't really now what else to put in here
- ◇ Not happy with the ending of this chapter, like introduction it's really only one paragraph at the end that explains the intended contribution of this work

1.3 Goal of this Thesis

To aid further research in the topic of Batch-Scheduling-Algorithms, the goal of this thesis is to provide a simplistic interface for Batch-Scheduling on Kubernetes.

(TODO: Explain how already existing Schedulers like Mary and Hugo do not run on Kubernetes due to different interface/interactions)

Already existing Scheduling Algorithms, like Mary and Hugo, were initially developed for the Cluster Resource Manager YARN. Reusing existing Scheduling Algorithms on the nowadays broadly adopted Cluster Resource Manager Kubernetes is not a trivial task due to vastly different interfaces and interaction with the Cluster Manager.

(TODO: Explain why the Setup of Kubernetes has become easier: Cloud Providers, MiniKube)

Extending existing research to the more popular Resource Manager Kubernetes provides multiple benefits.

1. Research on Scheduling Algorithms for YARN has become less valuable due to less usage
2. The large ecosystem around Kubernetes allows for a better development environment due to debugging and diagnostic tooling
3. Initial setup of a Kubernetes cluster has become smaller due to applications like MiniKube, which allows a quick setup of a cluster in the local machine and Cloud Providers offering Kubernetes Clusters as a service.

(TODO: Describe the Interface here)

The interface should provide easy access to the Kubernetes Cluster, allowing an external scheduler to place enqueued Batch-Jobs in predefined slots inside the cluster.

For an external scheduler to form a scheduling decision, the interface should provide an overview of the current cluster situation containing:

1. Information about empty or in use slots in the cluster
2. Information about Jobs in the Queue
3. Information about the history of reoccurring Jobs, like runtime

It should be possible for an external scheduler to form a scheduling decision based on a queue of jobs and metrics collected from the cluster. The interface should accept the scheduling decision and translate it into Kubernetes concepts to establish the desired scheduling in the cluster.

(TODO: Explain shortcomings of Kubernetes)

Currently, the Kubernetes Cluster Resource Manager does not offer the concept of a Queue. Submitting jobs to the cluster would either allocate resources immediately or produce an error due to missing resources.

Kubernetes does not offer the concept of dedicated Slots for Applications either. While there are various mechanisms to influence the placement of specific applications on specific nodes, these might become unreliable on a busy cluster and require a deep understanding of Kubernetes concepts, thus creating a barrier for future research.

1.3.0.1 TODO:

- ◇ Implementation of easy to use Interface that would allow already Batch Job Scheduling Algorithms likes Hugo and Mary to be run with small changes, on the popular Cluster Management Software Kubernetes

1.3.0.2 OPEN:

- ◇ Use of “should”. Okay? or Rather what it does?

1.4 Structure of this Thesis

The structure of this thesis allows the reader to read it in any order. To guide the reader through this thesis, the structure of this Thesis section will briefly explain which section contains which information.

The Background Chapter is supposed to give a brief overview of this thesis’s underlying concepts. This chapter introduces Big Data Streaming Processing, the Cluster Resource Manager Kubernetes, and Scheduling.

Following the Background chapter, the thesis provides an overview of the approach taken to tackle the problem described in the Problem Description Section. The Approach Section focuses on more profound concepts of Kubernetes and the Scheduling Cycle of the Kubernetes Scheduler. It summarizes the Kubernetes Operator pattern, which is commonly used to extend Kubernetes.

Implementation details will be given inside the Implementation Chapter, where an architectural overview and interaction between individual components are explained. The Implementation section also emphasizes the design Process for the Interface, which is exposed to an external scheduler. A significant part of the implementation is the Operator, which will be discussed extensively. The Implementation chapter shows how the points made inside the Approach Chapter are were implemented in the end. Finally, as the Goal of this Thesis section describes, changes that had to be made to already existing Scheduling Algorithm Implementations are disclosed and discussed.

(TODO: Hard to describe what is going to happen inside the Evaluation, if i don’t have anything to evaluate yet)

An Evaluation of the research and contribution done by this thesis will be presented inside the evaluation chapter. Here its functionality is demonstrated. This section will also outline some of the limitations.

Before concluding the thesis, a comparison between State of the Art Technology for Kubernetes and Non-Kubernetes Scheduling Frameworks is made.

1.4.0.1 TODO:

- ◇ Thesis starts by giving a brief background to Big Data Streaming Processing, Cluster Management Systems (Kubernetes), and Scheduling
- ◇ Discuss the Approach this thesis takes on tackling the Problem Description, by explaining how scheduling in Kubernetes works and what it takes to Extend Kubernetes (using the Operator Pattern)
- ◇ Implementation Details that a worth mentioning:
  - ○ An architectural Overview.
  - ○ The Process of designing an Interface
  - ○ The Operator that is used to extend Kubernetes
  - ○ Changes that had to be made to existing Algorithms (and their tests)
- ◇ How the work of thesis is evaluated, by testing it's functionality, comparing results from previous work and finally outlining its limitations
- ◇ Comparing the Work that was done to current State of the Art Technology like the Batch Scheduling Framework Volcano and comparing to Scheduling approaches that are not available on Kubernetes
- ◇ A final Conclusion, with a note on future work, that is missing from the current implementation or requires rethinking.

2
Background

2.1 Big Data Stream Processing

Big Data Processing aims to solve the problem of analyzing large quantities of data. In the last years, the amount of data that is being generated has exploded. This creates a Problem where single machines can no longer analyze the data in a meaningful time. While the Big Data Processing frameworks still work on single machines, computation is usually distributed across many processes running on hundreds of machines to analyze the data in an acceptable time.

Analyzing data on a single Machine is usually limited by the resources available on a single machine. Unfortunately, increasing the resources of a single machine is either not feasible from a cost standpoint or simply impossible. There is only a limited amount of Processor time, Memory, and IO available. Cheap commodity hardware allows a cluster to bypass the limitations of a single machine, scaling to a point where the cluster can keep up with the generated data and once again analyze data in a meaningful time frame.

Dealing with distributed systems is a complex topic in itself. Many assumptions that could be made in a single process context are no longer valid. Scaling to more machines increases the probability of failures. Distributed Systems need to be designed to be resilient against Hardware-Failures, Network Outages/Partitions and are expected to recover from said Failures. Having a single failure resulting in no or an invalid result will not scale to systems of hundreds of machines, where it is unlikely not to encounter a single failure during execution.

Big Data Processing Frameworks can be put into two categories, although many fall in both categories. Batch Processing and Stream Processing. In Batch Processing, data size is usually known in advance, whereas Stream Processing expects new data to be streamed in from different sources during the Runtime. Batch Processing Jobs will complete their calculation eventually, and Stream Processing, on the other hand, can run for an infinite time frame.

(TODO: DAG, Images) Internally, Big Data Processing Frameworks build a directed acyclic graph (DAG) of Stages required for the analysis. Stages are critical for saving intermediate results, to resume after failure, and are usually steps during the analysis, where data moving across processes is required. Stages can be generalized in Map and Reduce Operations. Map operations can be performed on many machines in parallel, without further knowledge about

the complete data sets, like extracting the username from an Application-Log-Line. Reduce Operations require moving data around the cluster. These are usually used to aggregate data, like grouping by a key, summing, or counting.

(TODO: Synchronization)

Partitioning of the Data is required due to the limitations of each single Machine. Datasets that the Distributed Processing Frameworks analyze are usually in the range of TeraBytes which is multiple magnitudes higher than the amount of Memory that each Machine has available. *(TODO: Distributed Data Store like HDFS)* While Persistent Memory Storage, like Hard-Drives, might be closer to the extent of BigData, Computation will quickly become limited by the Amount of I/O a single machine can perform.

The user of Big Data Processing frameworks is usually not required to think about how an efficient partition of the data across many processes may be accomplished. Frameworks are designed in a way where they can efficiently distribute a large amount of work across many machines.

2.1.0.1 TODO:

- ◇ Explain why cluster computing is required to deal with the Big Data Problem
- ◇ Explain what makes distributing computation across a cluster hard
- ◇ Explain the Value of already existing Big Data Stream Frameworks like Spark and Flink
- ◇ Explain on a high level how these work
  - ○ Explaining the DAG is required in order to later differentiate between DAG-level scheduling and “Pod”-Level scheduling
  - ○ Driver and Executor Pods
- ◇ Mention the use on Kubernetes using the Spark and Flink Operator

2.1.0.2 Input

- ◇ mehr generisch
- ◇ unterschied stream/batch

2.2 Scheduling

In general, scheduling is the process of assigning resources to a task. This includes the question:

1. Should any resources be allocated for the task at all
2. At which point in time should resource be allocated
3. How many resources should be allocated
4. Which of the available resources should be allocated

(TODO: Where scheduling is necessary)

Scheduling is essential for Operating Systems that need to decide which process should get CPU time and which processes may need to wait to continue computation. In the case of multiple CPU, a decision has to be made which CPU should carry out the computation. The Operating System is not just concerned with CPU-Resources, but also I/O Device resources.

Some devices may not work under concurrent usage and require synchronization. Who is allowed to access it?

(TODO: What scheduling policies/strategies exist, and what are they aiming to optimize)

In some cases, a simple FIFO scheduling that works on tasks in order there were submitted produces acceptable results. *(TODO: Scheduling is used to optimize for Deadlines/Throughput/FastRespons* Scheduling depends on a goal. Some Algorithms aim to find the optimal schedule to respect any given deadlines. Whereas some distinguish between Soft and Hard Deadlines, where ideally you would not want to miss any deadlines, occasionally missing Soft deadlines to guarantee Hard Deadlines are met is acceptable. In general, finding a single best schedule that allows resources to be allocated optimally is not possible. Scheduling for a fast response time or throughput might prefer shorter tasks to be run, when possible, and might starve longer running tasks for a long time before progress can be made.

(TODO: Preemptive)

A Scheduling algorithm might allow preemption, where the currently active task could be preempted for another task to become active. Some scheduling algorithms account for the potential overhead of preempting the current task (like a context switch).

The higher up the Stack *(TODO: stack = single process -> os -> vms -> Distributed Systems)*, more and more potential schedules become possible. It seems a wise choice for the scheduling to be handled in their respective stack layers.

The Question of Scheduling in a Distributed System is now the question of which machines resources should be used for which task. Here Scheduling algorithms need to pay attention to the characteristics of a Distributed System: 1. Potential heterogeneity of the system, with machines of different Hardware and different Operating Systems or Software 2. Spontaneously adding and removing resources of the Cluster 3. Interference between Applications residing on the same machine, same rack, same network switch, etc. (CO-Location)

While some of these factors can be controlled, different algorithms can be chosen for various use cases.

*(TODO: Scheduling in Stream Processing: DAG Level Scheduling / Container Level Scheduling)

In Stream-Processing, we deal with a multitude of different levels of scheduling. Stream Processing Frameworks build the DAG based on the job submitted. The initial DAG breaks done the job into their respective Map and Reduce Operations. These Operations will be broken down into smaller Tasks based on the Partitioning of Data. Finally, the Tasks may be executed on an arbitrary number of machines (technically not machines, but processes). Optimizing the schedule of tasks to a machine will be called DAG-Level scheduling and may now also include factors like Data-Locality.

(TODO: Scheduling on the Cluster Level: The Interesting Topic of this Thesis)

Moving Up one level Higher in the Stack, we are concerned with running multiple Jobs inside the same cluster, and a decision needs to be made which job can spawn their executor on which nodes. Executors are packaged on Containers. The containers are isolated, so they can not access each other. Unfortunately, isolating processes from each other in a container forces the underlying machine to need to know how much of the system's CPU and memory each container should have.

2.2.0.1 TODO:

- ◇ Explain what Scheduling is
- ◇ Different kind of scheduling
  - ○ DAG Scheduling done by Spark (Not what this thesis is about)
  - ○ POD Scheduling done by the Cluster Resource Manager
    - ∗ Co-Location
    - ∗ Packing
- ◇ Explain why Scheduling is Important
  - ○ Co-Location Problem
  - ○ Low Resource Usage (Graph from Google)
  - ○ Results from Hugo/Mary Paper

2.2.0.2 Open:

- ◇ Should maybe start a bit less specific about this Thesis and find more Information about Scheduling in general or is it fine if the Background section starts of general and tailors towards the topic of my thesis?

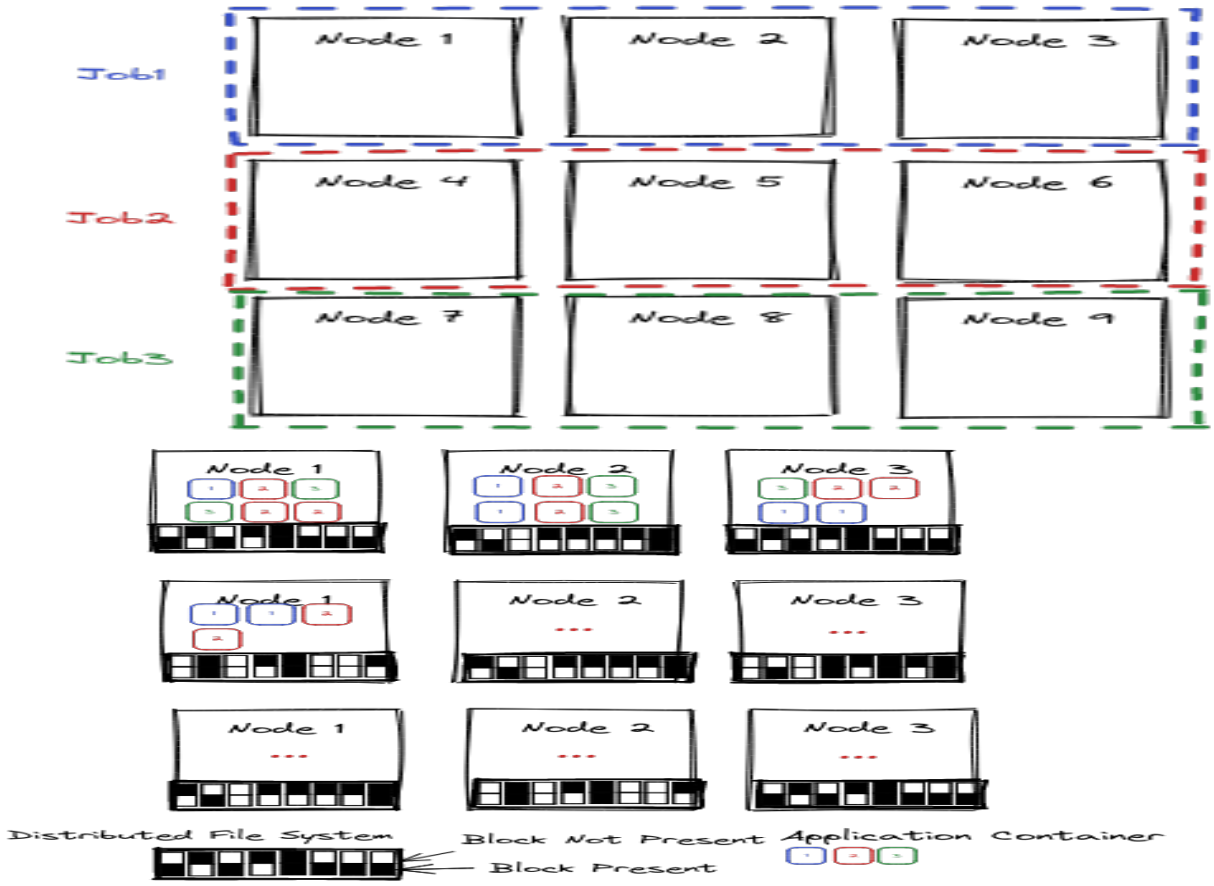
2.3 Cluster Management Systems

- ◇ Abstraction for Systems with many machines

(TODO: Super Computer/HPC) - Super Computer with high-end hardware - Scale Work across multiple machines benefitting from cheap commodity hardware - Coarse Grain: Static partitioning of Machines across different workloads is enough

(What is MapReduce, maybe put into Big Data Stream Processing)

- ◇ Hadoop is one of Many Open-Source MapReduce implementations *(TODO: Frameworks)* Cluster computing using commodity hardware was driven by the need to keep up with the explosion of data. The Initial Version of Hadoop was focused on Running MapReduce Jobs to process a Web Crawl (*YARN Paper*). Despite the initial focus, Hadoop was widely adopted evolved to a state where it was no longer used with its initial target in mind. Wide adoptions have shown some of the weaknesses in Hadoops architecture:
- ◇ Tight Coupling between the MapReduce Programming model and Cluster Management
- ◇ Centralized Handling of Jobs will prevent Hadoop from Scaling



The tight coupling leads Hadoop users with different applications to abuse the MapReduce Programming model to benefit from cluster management and be left with a suboptimal solution. A typical pattern was to submit 'map-only' jobs that act as arbitrary software running on top of the resource manager. [4] The other solution was to create new Frameworks. This caused the invention of many frameworks that aim to solve distributed computation on a cluster for many different kinds of applications [3]. Frameworks tend to be strongly tailored to simplify solving specific problems on a cluster of computers and thus speed up the exploration of data. It was expected for many more frameworks to be created, as none of them will offer an optimal solution for all applications.[3]

Initially, Frameworks like MapReduce created and managed the cluster, which only allowed a single Application across many machines—running only a single application across a cluster of nodes led to underutilization of the cluster's resources. The Next generation of Hadoop allowed it to build ad-hoc clusters, using Torque [1] and Maui, on a shared pool of hardware. Hadoop on Demand (HoD) allowed users to submit their jobs to the cluster, estimating how many machines are required to fulfill the job. Torque would enqueue the job and reserve enough machines once they become available. Torque/Maui would then start the Hadoop Master and the Slaves, subsequently spawn the Task and JobTracker that make up a MapReduce Application. Once all Tasks are finished, the acquired machines are released back to the shared resource pool. This can create potentially wasteful scenarios, where only a single Reduce Task is left, but many machines are still reserved for the cluster. Usually, Resources requirements are overestimated and thus leaves many clusters resources unused.

With the Introduction of Hadoop 3, the MapReduce Framework was split into the dedicated

Resource Manager YARN and MapReduce itself. Now MapReduce was no longer running across a Cluster, but it was running on top of YARN, who manages the cluster beneath. This allows MapReduce to run multiple Jobs across the same YARN Cluster, but more importantly, it also allows other Frameworks to run on top of YARN. This moved a lot of complexity away from MapReduce and allowed the framework to only specialize on the MapReduce Programming Model rather than managing a cluster. This finally allows different Programming Models for Machine Learning tasks that tend to perform worse on the MapReduce Programming model [5] to run on top of the same cluster as other MapReduce Jobs.

Using the ResourceManager YARN allows for a more fine-grain partitioning of the cluster resources. Previously a static partitioning of the clusters resources was done to ensure that a specific application could use a particular number of machines. Many applications can significantly benefit from being scaled out across the cluster rather than being limited to only a few machines. - Fault tolerance: Frameworks use replication to be resilient in the case of machine failure. Having many of the replicas across a small number of machines defeats the purpose - Resource Utilization: Frameworks can scale dynamically and thus use resources that are not currently used by other Applications. This allows applications to scale out rapidly once new nodes become available - Data Locality: Usually, the data to work on is shared across a cluster. Many applications are likely to work on the same set of data. Having applications sitting on only a few nodes, but the data to be shared across the complete cluster leads to a lousy data locality. Many unnecessary I/O needs to be performed to move data across the cluster.

Fine-grain partitioning can be achieved using containerization, where previously applications were deployed in VMs, which would host a full Operating System. Many containers can be deployed on the same VM (or physical machine) and share the same Operating System Kernel. The hosting Operating makes sure Applications running inside containers are isolated by limiting their resources and access to the underlying Operating System.

Before Hadoop 3 with YARN was published, an Alternative Cluster Manager Mesos was publicized. Like YARN, Mesos allowed a more fine granular sharing of resources using containerization. The key difference between YARN and Mesos is how resources are scheduled to frameworks running inside the cluster. YARN offers a resource request mechanism, where applications can request the resources they want to use (*TODO: fact check*), and Mesos, on the other hand, offers frameworks resources that they can use. This allows frameworks to decide better which of the resources may be more beneficial. This enables Mesos to pass the resource-intensive task of online scheduling to the frameworks and improve its scalability.

(TODO: Kubernetes) Kubernetes was initially developed by Google and released after multiple years of intern usage. Kubernetes was quickly adopted and has become the defacto standard for managing a cluster of machines. Kubernetes offers a descriptive way of managing resources in a cluster where manifests describing the cluster's desired state are stored inside a distributed key-value store etcd. Controllers are running inside the cluster to monitor these manifests and do the required actions to bring the cluster into the desired state. Working with manifest abstracts away many of the problems that arise when deploying Applications to a cluster. Usually, an Operations Team was required to manage applications across the cluster. With Kubernetes offering the required building blocks and the mechanism of a control-loop, the operator pattern in combination with Custom Resource Definitions is commonly used to extend Kubernetes functionalities.

2.3.0.1 Notes (Ignore):

- ◇ HPC Super Computer / Resources
- ◇ Kubernetes Section is definitely not complete
- ◇ Explain what a Cluster Resource Manager is doing
  - ○ Abstraction of using a single cluster as a single Machine
  - ○ Managing given resources making it scalable by adding more machines
- ◇ Show what are the differences between YARN and Kubernetes
  - ○ YARN: Emerging from Hadoop was design to Work with Batch Applications
  - ○ Kubernetes: All Round Cluster Manager, with a Big Community

(TODO: Cloud Computing) - Coarse Grain approach is no longer feasible - Applications may scale up or down, for and a partitioning of the cluster has to be done dynamically - Containerization, allows to run many different applications on the same machine without much overhead, like creating a new virtual machine - Different Applications may share same Node to increase overall utilizations of resources - Applications benefit from data locality, where creating application on a node that already contains the data, will not use additional I/O Resources - Different Applications are likely to work on the same data

Mesos and Kubernetes - etcd vs ZooKeeper - Mesos Resource Offer leave scheduling to Framework - Kubernetes follows a descriptiv approach, where the user of the cluster describes a state that the cluster should be in and the resource manager exectues the required actions. This makes Kubernetes very extensible. - Kubernetes builds a virtual network across nodes - Kubernetes general purpose

YARN Paper - Hadoop thightly focused on Web Crawling for Hadoop at Yahoo! - Broad Adoption streched intial focus - Thight Coupling of MapReduce Programming Model and Resource Manager - Centralized handling of Jobs control flow from the JobTracker

- ◇ MapReduce Programming model was abused for other purposes then it was initially design
- ◇ Common Pattern of Map-Only that was used for Forking?-Web-Services and Gang-Scheduled Computation. In general Developer came up with clever solutions to run all kinds of software on top of hadoop
- ◇ Misuse and Broad adoption exposed many substantial issues with Hadoops archicture and implementation
- ◇ YARN moves Hadoop pasts its original incarnation, by breaking up the monolithic architecture of Hadoop, splitting it into a Resource Manager from the programming model
- ◇ YARN delegates many scheduling-related functions to per-job components
- ◇ This makes MapReduce just one of many applications that can run on top of YARN
- ◇ Allows for great choice of Programming Models using different Frameworks
- ◇ Programming Frameworks running on top of YARN can manage their intra-application communication, execution flow and dynamic optimization by them self, unlock performance benefits

2.3.0.2 Historical

- ◇ Yahoo! WebMap
- ◇ Needs to be scalable
- ◇ Ad-hoc clusters
- ◇ Initially user bring up a handful of nodes, load their data into HDFS, write a MapReduce job, then tear it down
- ◇ Hadoop becomes more fault tolerant and persistent HDFS would become the norm
- ◇ Operators just upload potential interesting data into the the HDFS, attracting analysts. –> Mulit-Tenancy of HDFS
- ◇ To address Multi Tenancy HoD was deployed, using Torque and Maui to allocate Hadoop Cluster on a shared pool of Hardware
- ◇ User submit Job, with required amount of rescoures to torque, which then waits until enough resources are avaible
- ◇ Torque would start the Hadoop Leader, which then interacts with Torque and Maui to create the Hadoop Slaves, which will then Spawn the TaskTracker + JobTracker
- ◇ Job Tracker would then accept a series of Jobs
- ◇ User can then release the Cluster. The System will then return logs to the user and return nodes to the cluster
- ◇ HoD allows slightly older version of Hadoop to be used
- ◇ HoD shortcomming:
- ◇ Torque did not account for data locality
- ◇ Bad resource utilization, last task of job may block hundreds of machines, Cluster were not resized between jobs, Users usually overestimate the number of nodes required
- ◇ Shared Cluster
- ◇ Resource Granularity was to coarse
- ◇ JobTracker does not scale as HDFS, JobTracker failure was a complete failure of all jobs
- ◇

3

Approach

3.1 Scheduling in Kubernetes

3.2 Extending Kubernetes using the Operator Pattern

4

Implementation

4.1 Architecture

4.2 Designing the Interface

4.3 Operator

4.4 Changes to existing Algorithm

5
Evaluation

5.1 Testing

5.2 Comparing to baseline Runtime

5.3 Limitations

5.4 Discussion

6

State of the Art

6.1 Volcano

6.2 Non Kubernetes

7

Conclusion and Future Work

7.1 Conclusion

7.2 Future Work

Bibliography

[1] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 776–783 Vol. 2, 2005.

[2] Telmo da Silva Morais. Survey on frameworks for distributed computing: Hadoop, spark and storm. In *Proceedings of the 10th Doctoral Symposium in Informatics Engineering-DSIE*, volume 15, 2015.

[3] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[4] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.