Experimental Study on Scalable Post-Groomer of the Wildfire HTAP System on Cloud

Project Report

from the Course of Studies Applied Computer Science

at the Cooperative State University Baden-Württemberg Mannheim

by

Pascal Schroeder

09/24/2018

Time of Project	05/07/2018 - 08/24/2018
Student ID, Course	5501463, TINF16AI-BC
Division, Business Unit	IBM Research
Company	IBM Deutschland GmbH, San Jose - Almaden
Supervisor in the Company	Yuanyuan Tian

Signature Supervisor _____

Contents

Declaration of Sincerity

Hereby I solemnly declare that my project report, titled *Experimental Study on Scalable Post-Groomer of the Wildfire HTAP System on Cloud* is independently authored and no sources other than those specified have been used.
I also declare that the submitted electronic version matches the printed version.

Mannheim, 09/24/2018
Place, Date

Signature Pascal Schroeder

Abstract

1 Introduction

1.1 Advantage of cluster systems

Cluster systems are two or more computers connected to each other, so that they can share their resources and can be viewed as one system. Therefore they can be connected physical as well as virtual. Using such a cluster system is typically much more cost-efficient compared to using a single computer, which provides about the same power. But the advantage of such systems doesn't end with a better cost-efficiency.

First clusters are not only used for one specific purpose, but there are different kinds of cluster configurations, which are build for different objectives.

On the one hand, there are "Load balancing" cluster configurations. Load balancing clusters are clusters with the objective to provide better computing performance, like minimizing response time and maximizing throughput. Therefore it splits the computation in different parts and runs them parallel on different nodes. Through that the capabilities of the systems can be combined.

Furthermore there are "High availability" clusters, which are designed to prevent a total breakout of the system. For that there are several redundant nodes connected to the system. That means, that when one component fails, a redundant node of this component is used to provide the service. Thereby availability is ensured even if one component fails. The more redundant nodes are used the better will be the availability of a system. To provide an example: If a system has an availability of 90% it has a downtime of 10%, which means that the system is not available on 36.53 days per year. If there is a second, redundant and independent node connected to the cluster, which acts as a stand-in for the first system in case of a failure, the downtime of those two systems has to be combined. That means that this cluster would only fail if there is an overlap between the downtimes of those two systems. Resulting of a downtime of 10% of each system there is a combined downtime of 1% or 3.65 days per year. If there is a third system connected to the cluster, the downtime will be reduced to 0.1% or 8.77 hours per year and so on. The job of "High availability" clusters is in that to reduce the downtime of the system and to provide a working system almost any time.

Figure 1.1 shows an example architecture of such a "High availability" cluster. There are two servers, whereby one server is a replication of the other one. The request router

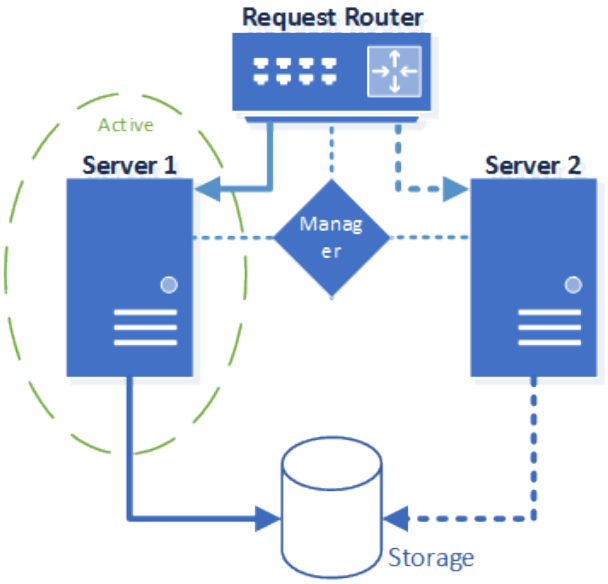

Figure 1.1.1 “High availability” cluster architecture

handles the incoming requests and leads them to the active server. If there is a system failure of the active server, the manager will recognize that, and sends a message to the request router to change the request address to the other server. From now on this server will deal with all the request. Also there is a shared storage for both of the servers in one database. Whether which one is active they can both access this database.

But those advantages don’t have to stand alone, but they can also be combined, so that you can provide high availability as well as high performance. Those two problems are becoming more and more important because of the high amount of calculations which are made in Machine Learning, Big Data Analysis etc. That’s the reason why cluster systems are becoming more and more important in modern IT for providing a solution for both problems.

1.2 Different kinds of data processing system

In the modern times of machine learning and Big Data analysis the importance of data processing systems increases continuously. More and more systems are being released for Big Data Analysis, for example Spark. Such systems are called OLAP - Online Analytical Processing. One example OLAP system is Spark. Such systems are working with groups of data and complex requests. For better analyses it also doesn’t only use the current state of a dataset, but also its history. The objective of OLAP systems is to give one unique, consistent view on all data from different sources. and extracting information like prediction or other complex information out of these data.

But still conventional data processing like ACID Atomicity, Consistency, Isolation, Durability transactions is needed for regular database requests, like bank transactions or text messages. Those systems are called Online Transactional Processing - Online Transactional Processing. This system is characterized by instant instant interactions with the business systems without a latecy instead of collecting all transactions and process them all at once in the night. This was a major step forward in fast data processing in the late 1970s and those systems are still used by almost all companies.

The problem ist, that both of these systems need a specific form of their data, so they can't work together on the same set of data. For resolving that problem HTAP (Hybrid Transactional Analytical Processing) has been created. This enables real-time analytical results, which can help businesses in quick decision making. Therefore are still two copies of the data needed, but with an highly increased synchronization rate between those two data copies.

The Wildfire system, which was developed at IBM Research Lab Almaden, is such an HTAP system. It used Spark as OLAP engine and an own engine for OLTP. In the following chapters this system and underlying basement technologies will be presented. Furthermore an experimental study will be described, in which a part of this solution will be deployed on a Cluster system and tested concerning its performance.

1.3 Introduction to IBM Research Cloud

For a few years there has been a trend to move from local, physical devices to centralized cloud platforms. This has different advantages like dynamically allocation of computing power and more cost-efficiency. The reason for this is because often there are unused resources on physical devices, because most systems don't need all of their power anytime. Through outsourcing this computing power on the cloud the resources are only used where they are needed. This increases the cost efficiently eminent, because not every physical device needs backup resources for situations, in which more computing power is needed. Instead with the cloud it is possible to request more resources when needed and to release them when they are not needed any longer.

In Research there is often a lot computing power necessary for testing, calculating and building projects and experiments. Therefore every labour has several own physical devices. But currently there is also a trend in Research projects to move to Cloud, sharing resources and save costs. That is the reason why IBM started its "IBM Research Cloud", hosted on iRIS (IBM Research Integrated Solutions) IMS (Intrastructure Management System).

This is a self-service platform, which provides cloud based solutions for IBM Researcher on request for specific resources.

IBM Research Cloud itself provides virtual devices as well as virtual GPU (Graphical Processor Unit devices. Virtual devices mimics physical hardware but without the need of an isolated physical machine. Instead its just splitting some computing power of a big cluster via Software and makes the system believe, that it would be an isolated hardware device. Virtual GPU devices are differentiated by an additional Graphical Unit, which provides the virtual device even more computing power and relives the CPU (Central Processing Unit) in some tasks.

Besides the creation and deletion of virtual devices, the Research Cloud also provides operations like hard/soft reset or rebooting a device. Additionally it also provides the opportunity to create an image of an existing virtual device and creating a new one from this image. Through that you can keep your configurations and files from your old device and create a device based on those configurations without the need of installing everything up from the beginning.

In the following described project this Research Cloud will be used to setup a Kubernetes Cluster with several virtual devices and to run Wildfire using this Cluster.

1.4 Project objective

2 Theory

2.1 Kubernetes cluster solution

One very common system for managing cluster systems as described in chapter 1.1 is Kubernetes - or short K8s. Originally developed by Google and by now maintained by the Cloud Native Computing Foundation, Kubernetes is an “open-source platform for managing containerized workloads and services”. For unterstanding what that means the concept of containers needs to be described first.

Containers are isolated, stand-alone packages of software, similar to processes. In those packages everything is included, which this piece of software needs, like runtime, libraries, settings and other system tools. These containers have a completely different environment within themselves than outside. This environment included for example network routes, dns settings and control group limits. This enables the possibility to share common resources and still be isolated from any other process as well as the host system. Thereby containers are always working the same, no matter on what system they run or in which environment.

Kubernetes is for an automating deployment, scaling and management of these containers within a cluster of nodes. Thereby a cluster consists out of at least one master node and any number of worker nodes. Figure 2.1.1 shows the different services owned by master and worker nodes.

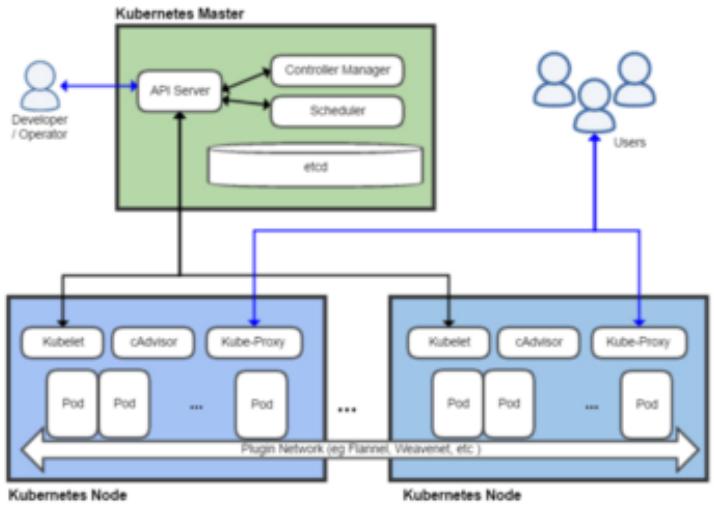

Figure 2.1.1 Kubernetes allocation of services

First there are several pods on each worker node. Pods are the smallest unit in Kubernetes. They contain one ore more containers, which are deployed together on the same host. Thereby they can work together to perform a set of tasks.

On the master node there are the API (Application Programming Interface) Server, a Controller Manager, an Scheduler and a key-value store called etcd.

The API Server is for the clients to run all of their requests against it. That means the API Server is responsible for the communication between Master and Worker nodes and for updating corresponding objects in the etcd. Also the authentication and authorization is task of the API Server. The protocol for the communication is written in REST (Represantational State Transfer). For reacting on changes of clients there is also a watch mechanism implemented, which triggers an action after some specific changes, like the scheduler creating a new pod. This workflow is showed in figure 2.1.2.

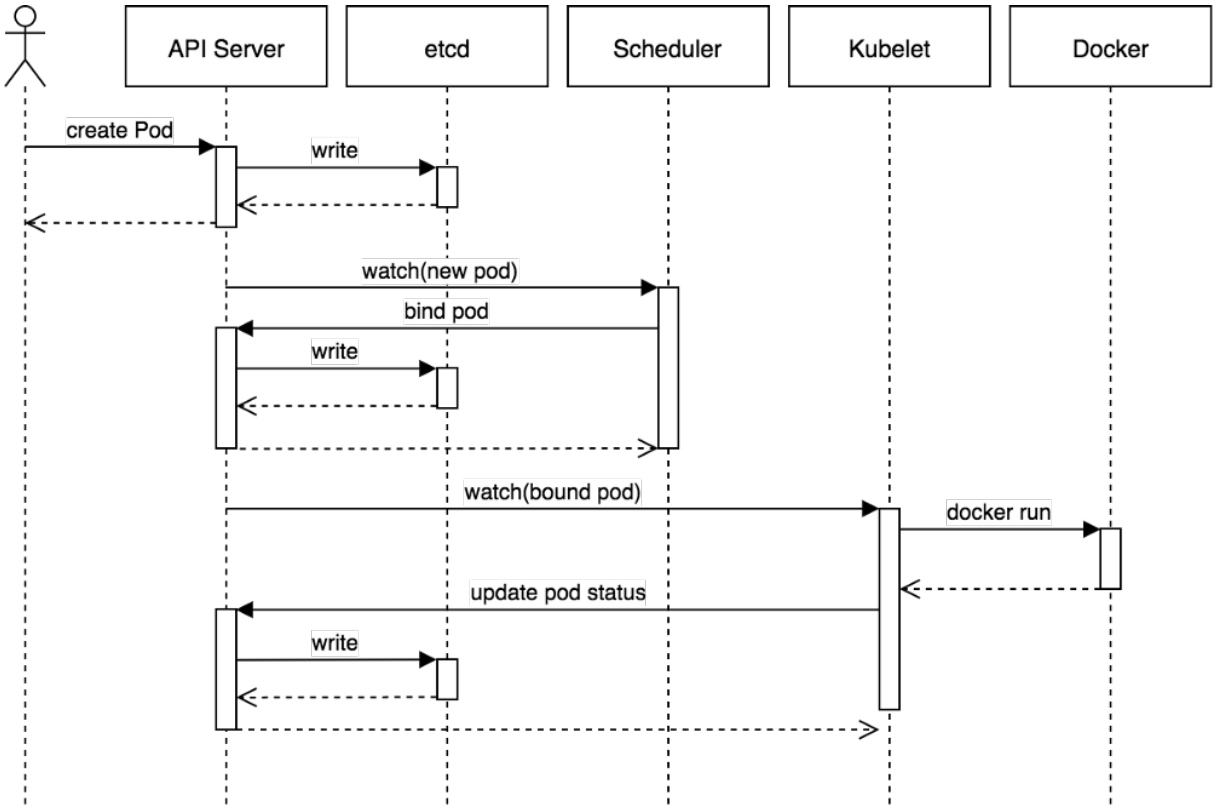

Figure 2.1.2 Kubernetes API Server watcher sequence diagram

There you can see the user creating a pod through requesting that to the API Server. The API Server writes this change to the etcd. Afterwards the API Server recognized a new pod in the etcd and invokes the Scheduler to create this new pod. What is the exact task of the scheduler is described later. After successfully created and bind the new pod the API Server writes this change to the etcd. Because of this the API Server invokes the kubelet, which is also described later, of the corresponding node. This kubelet starts docker to create the containers of the pod. Kubelet responses with the new status of the

pod, which the API Server writes to the etcd. After that the creation of the new pod is successfully finished.

The Controller Manager is a daemon, which embeds all of the Kubernetes controller. Examples for them are the Replication Controller or the Endpoint Controller. Those controllers are watching the state of the cluster through the API Server. Whenever a specific action happens, it performs the necessary actions to hold the current state or to move the cluster towards the desired state. If the Replication Controller for example recognizes, that one replication has been destroyed for some reason, it will take care of triggering the creation of a new replication if there should be more than there are currently.

The scheduler manages the binding of pods to nodes. Therefore it watches for new deployments as well as for old ones to create new pods if a new deployment is created or recreating a pod whenever a pod gets destroyed for some reason. The scheduler organizes the allocation of the pods within the cluster on the basis of available ressources of the pods. That means, that it always create pods, where the most resources are available, or reorganize the allocation if there is a change in the resource allocation of the cluster.

Figure 2.1.3 shows the way the Scheduler works, when new nodes are connected. As long as there are only two nodes and four pods need to be deployed, it allocates those four pods to those two nodes . As soon as there are more nodes connected, the scheduler recognizes free resources and reallocate the pods to these new nodes.

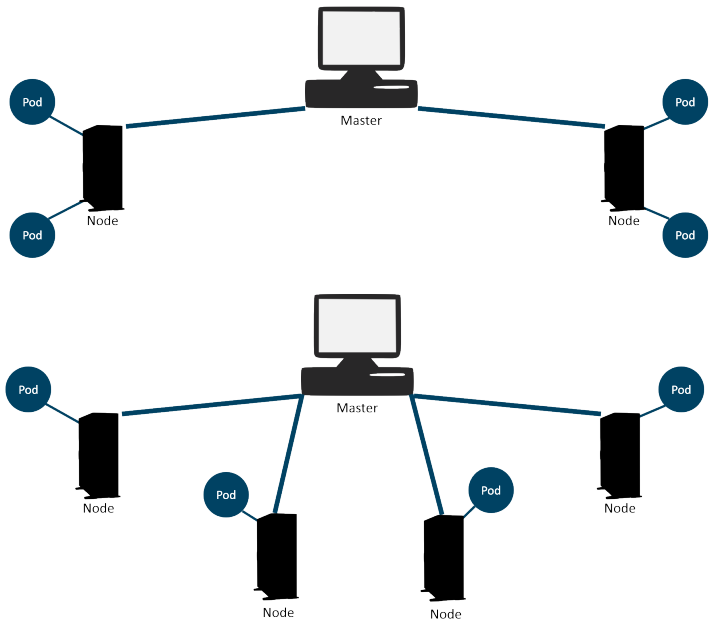

Figure 2.1.3 Kubernetes API Server watcher sequence diagram

The etcd is a key-value store, which stores the configuration data and the condition of the Kubernetes cluster. The etcd also contains a watch feature, which listens to changes

to keys and triggers the API server to perform all necessary actions to move the current state of the cluster towards the desired state.

The worker node consists of a Kubelet, a cAdvisor, a Kube-Proxy and - as mentioned before - several Pods.

The kubelet needs to be used if a new pod should be deployed. Then it gets the action to create all needed containers. For that it uses Docker to create them. Afterwards it combines some containers into one pod. Conainers in one pod are always started and stopped together. This pod will then be deployed on the node, on which the kubelet is located.

The cAdvisor measures the usage of CPU-resources as well as demanded memory on the node, on which it is located, and notifies the master about it. Based on those measurements the scheduler allocates the all pods within the cluster to ensure the best possible allocation of resources.

The kube-proxy is a daemon, that runs as a simple network proxy to provide the possibility of communicating to that node within the cluster. Additionally it runs a load balancer for the services on that node.

Through this architecture Kubernetes enables different possibilities to deploy pods within the cluster. The simplest one is to deploy a specific pod directly through the

```
kubectl create -f 'image\_path'
```

Listing 2.1: Create Kubernetes pod

command. This deploys the pod described in the file, but it doesn't ensure the failure safety. That means if the pod gets destroyed for some reason, it won't be recreated and deployed again automatically.

Another possibility is to create a deployment. Therefore the image has to be embedded within a replicaset and this replicaset within a deployment. If this deployment is created, it will automatically create every pod of the replicaset. If one pod gets terminated, no matter if manually or because of an error, it will be directly recreated and deployed.

This procedure also enables the possibility to execute dynamical rolling updates. If you execute the command

```
kubectl set image deployment/name-deployment name=name:1.1.1
```

Listing 2.2: Create Kubernetes deployment

Kubernetes will start a Rolling Update. This causes the creation of a new replicaset, which uses pods of the new version. While the new replicaset will be scaled up, the old one will be scaled down step by step. This enables the maintenance of a deployment even while an update is enrolles, so that the users can still use the services while they are updating. That means, that there is never a need of a system shut down because of an updated, which needs to be enrolled, and the system guarantees its high availability.

A third possibility to deploy pods are services. Services are used to enable the usage of pods from outside the cluster. Therefore it gets the pod from the targetPort of the belonging node and creates a random port on this node. This port serves as endpoint for the LoadBalancer. Through this, everybody can now communicate with this pod.

This is how Kubernetes ensures High Availability as well as Load Balancing at the same time. Through the possibility of replicate every pod several times it is ensured that whenever one pod fails for some reason, another is already prepared to help out and take over the job. Even the master can and should be replicated to ensure a functional system, even if the master has been destroyed. For production environment it's recommended to have 3, 5 or 7 master nodes running at the same time. For facilitating the High Availability almost every unit of a Kubernetes cluster is stateless and could be run redundant in several instances. Only the etcd key value store is stateful. For solving that problem in case of failure of the node, which owns the etcd, there is a leader election between all master node replicas to determine the new leading etcd.

All in all Kubernetes combines all the benefits of cluster applications on one software. High availability as well as load balancing is guaranteed and it also ensures a high scalability, rolling updates and auto-scaling. Compared to other cluster systems, like Docker swarm, it offers the biggest community and it can support clusters of up to 5000 nodes, which enables using Kubernetes even for very big clusters. For example the whole Google Cloud System runs on a Kubernetes cluster. Performing tests of Kubernetes shows, that in 99% the API responses in less than one second and also their pods start in less than five seconds in 99% when starting containers with pre pulled images. This all makes Kubernetes maybe one of the most flexible and fastest cluster systems. In exchange for that flexibility it is more difficult to set it up, because you have to do it all on your own. But only that way it can be ensured, that it runs fast and flexible in the same time, what is the reason for Kubernetes being so popular.

2.2 Spark

Apache Spark is an open source framework for big data analytics and large-scale data processing. It supports SQL (Structured Query Language), streaming data, machine learning and graph processing. Spark is a framework used in clusters. Therefore it offers native support for Hadoop YARN, Apache Mesos and since version 2.2.0 also Kubernetes.

Spark is known as very fast data processing system, which is caused by its in-memory processing capability. This is the big advantage of Spark in comparison to other Big Data Analysis tools like Apache Hadoop. Hadoop uses distributed storage mechanism instead of in-memory processing. This was very successful at the beginning of the data warehouse era, but with the possibility to run in-memory calculations Spark has a big advantage nowadays. By official information of Apache Spark, Spark is capable of 100 times faster workload runs than Hadoop.

Another advantage of Spark is its easy cloud-based implementation, which enables companies to run their analysis of their data directly on the cloud instead of using a hybrid strategy, in which only streaming data was analyzed on the cloud, while historical and aggregated data was analyzed using an on-premises Spark cluster.

For ensuring all of these features work - no matter on what clucster system - Spark applcications are splitted in independent sets of processes on such a cluster. Figure 2.2.1 shows how this works and how the processes are coordinates.

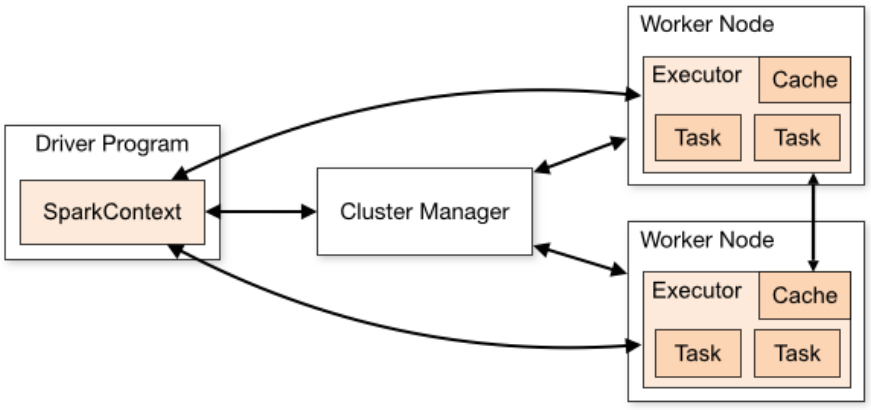

Figure 2.2.1 Spark cluster architecture

First there is the SparkContext in the main program, also called driver program. This context coordinates all the processes. Therefore it connects to the cluster manager of the cluster system you used (Mesos, Yarn, Kubernetes...). As described in chapter 2.1. their task is to allocate processes based on free resources of the nodes within the cluster. Spark then spawns executors on the nodes of the cluster. Executors are processes on which the applications are running. After that it also sends the application code to the

executors and send tasks to run. While they are running the executors can store data for the application.

This architecture with several independent executors has the big advantage, that isolated applications can run in multiple threads at the same time. Combined with the in-memory-processing this guarantees enormous speed and a great overall-performance.

Spark persists out of six different components - the Spark Core, Spark RDD, Spark SQL, Spark Streaming, MLlib Machine Learning and GraphX.

Spark Core is the base of the overall project and offers distributed task dispatching, scheduling and basic I/O (Input / Output) functionalities. Every other component is based on this core.

The first is Spark RDD - Resilient Distributed Dataset. RDD represents a collection of objects, which are spread across the cluster. This enables the possibility of fast and scalable data algorithms, because operations on these objects can be distributed within the cluster.

Spark SQL provides a SQL compliant interface for querying data - so there is standard SQL support in Spark. It also enables processing of structured data inside Spark. Additionally Spark SQL provides an interface for reading from and writing to other datastores like JSON (Javascript Object Notation), HDFS (Hadoop Distributed File System) or Apache Parquet out of the box. For other stores like Apache Cassandra or MongoDB there are Spark Packages available in its ecosystem.

Spark Streaming helps Spark in environments for real-time processing. Therefore it breaks the stream down into a continuous series of microbatches. Microbatches are nothing else than a very small group of data elements, which will be processed at once after being collected. These microbatches can then be manipulated using the Spark API. Through this code in batch and streaming operations can share mostly the same code and run on the same framework.

The MLlib is a library for machine learning for Spark. It includes a framework for machine learning algorithms and pipelines. It comes with a variety of machine learning tasks like featurization, transformations, model training and optimization.

GraphX provides a library for large-scale graph analytics. If offers an efficient abstraction for representing graph-oriented data and comes with various graph transformations, common graph algorithms and a collection of graph builders.

This all makes Spark to a very extensive tool for big data anlysis. But there are still some features missing, like traditional ACID transactions or an enterprise-quality SQL service. One way to solve that problem is described in chapter 2.3.

2.3 Wildfire

As already introduced in chapter 1.2 there are different kidns of data processing systems - OLAP and OLTP. To combine those two systems HTAP has been developed and Wildfire is an HTAP system developed by IBM in Almaden.

Wildfire brings ACID transactions to the analytics software Spark. Thereby Wildfire uses Spark to perform analytics by utilizing a non-proprietary storage format (Parquet); using and extending Spark APIs and the Catalyst optimizer for SQL queries and automatically replicating data for high availability, scale-out performance and elasticity. Wildfire expands these functionalities with features from traditional DBMS (Database Management System) like ACID transactions with snapshot isolation, making the last committed data immediately available to analytics queries; indexing any column for fast point queries and enterprise-quality SQL. Wildfire consists out of Spark itself at the one hand and the Wildfire engine at the other hand. The main entry point for every application targeting Wildfire is Spark. it also provides all its common features for big data analytics. The wildfire engine extends these functionalities by enabling analytics on newly ingested data as well and accelerating the processing of application requests. Thereby all requests to Wildfire go through Spark APIs, except for native OLTP requests, which are targeting the OLTP API of the Wildfire engine. Each request spawns Spark executors across the used cluster. The used node depends upon the type of that request.

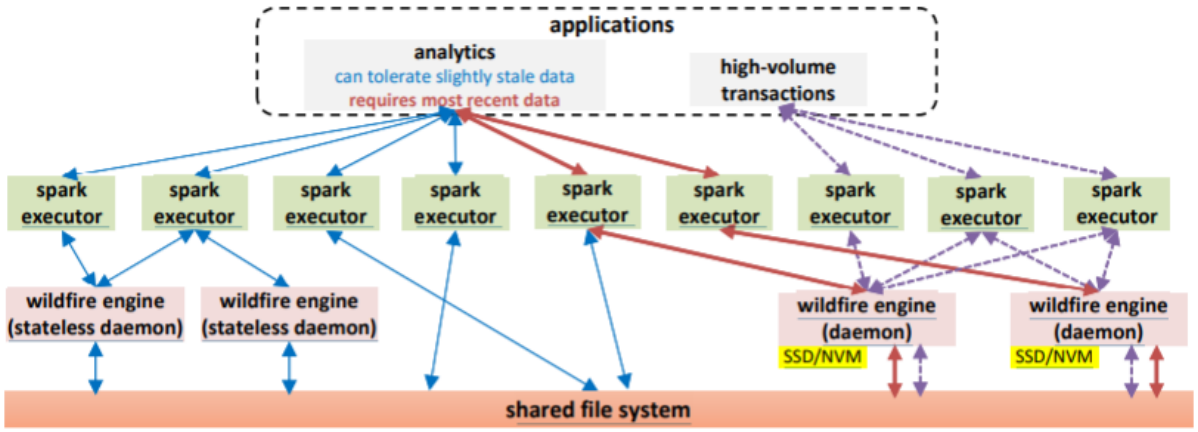

Figure 2.3.1 Wildfire architecture

In figure 2.3.1 the architecture of the Wildfire HTAP system is shown. There can be seen several spawned executors of Spark. Most of them are for executing analytical requests only. Only some, beefier nodes with faster local persistent storage like SSDs and more cores for increased parallelism handle both - transactions and analytical queries on the recent data from those transactions. In this figure they are indicated by dashed arrows, while every other executor for analytics only are indicated by solid arrows.

Additionally the figures shows several Wildfire engine instance daemons. Each of them is connected to at least one Spark executor. The stateless daemons are only connected to analytical-only executors, while the stateful daemons are connected to one or more executors for high volume transactions and to analytics executors for the most recent data. Those are indicated by red arrows. For that reason the stateless daemons are obviously for analytics queries on the much more voluminous older data while the stateful daemons handle transactions as well as analytics request against the latest data.

For persisting and reusing those data for later queries, there is a shared file system, on which every daemon, no matter if stateful or stateless, has access to.

In Wildfire the data are stored as tables defined with a primary key, a sharding key and optionally a partition key. The sharding key is a subset of the primary key and it is primarly used for load balancing of transaction processing. A shard of a table is replicated into multiple nudes, but only one replica serves as the shard leader, the rest are slaves. A table shard is the basic unit of a lot of processes like grooming, post-grooming and indexing. The partition key is for organizing data for analytic queries to speed up time-based analytics queries.

For keeping track of the life of each record Wildfire adds three hidden columns to every table - beginTS (begin timestamp; time when record is first ingested), endTS (end timestamp; time when the record is replaced by a new record with the same primary key) and prevRID (previous record ID; record ID of the new record, who replaces this record).

For supporting both data processing systems (OLTP and OLAP) efficiently, Wildfire divides data into multiple zones. That way transactions can first apend their updates to the OLTP friendly zone, which are thenmigrated to the OLAP friendly zone step by step. Figure 2.3.2 shows the lifecycle of those stages.

First there is the life zone, where the transaction first appends uncommited changes in a transaction local side-log. Thereby it sets the beginTS for each record and appends its side-log to the committed transaction log. This log is kept in memory for fast access and also persisted on the local SSDs using the Parquet columnar-format. This zone always has the latest ingested data.

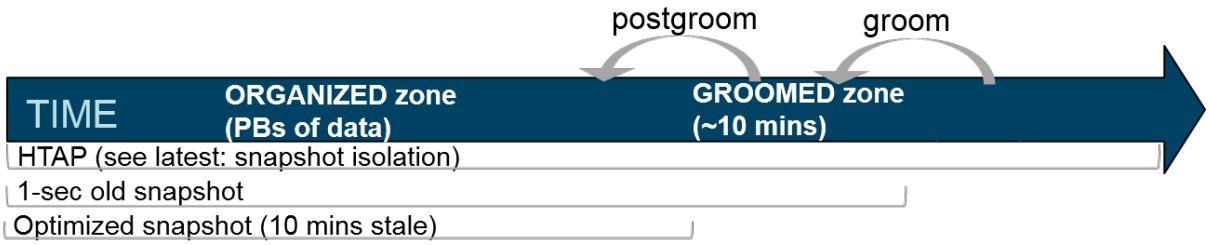

Figure 2.3.2 Wildfire lifecycle

After it there will be periodically invoked a groom operation (e.g. every second) to migrate data from the live zone to the groomed zone. This operation merges transaction logs from shard replicas, resolves conflicts and creates a Parquet columnar-format data file, called a groomed black, in the shared storage. Each groomed block can be identified by a monotonic increasing ID called groomed block ID. The groomer also builds indexes over the groomed data and set the higher order part of the beginTS. The lower part will remain the transaction time in the shard replica.

Last there will be periodically performed the post-grooming operation (e.g. every 10 minutes) to set the endTS and prevRID and reorganize the data to a more analytics-friendly parition key. This zone is called post-groomed or organized zone. This operation first utilized the post-groomed portion of the indexes to collect the RIDs of the already post-groomed records, that will be replaced. Then it scans the newly groomed blocks to set prevRID fields and reorganized the data into post-groomed block on the shared storage according to the OLAP friendly partition key. It usually generates much larger blocks than the groomer, because it is carried out less frequently. This results in better access performance on shared storage. Last the post-groomer also notifies the indexer process to build indexes on the newly post-groomed blocks.

All in all this means that there is one live zone, which always have the newest ingested data, a groomed zone, which owns an one second old snapshot of the data, and an organized zone, which owns a ten minutes snapshot, but which is optimized for analytic processes. Through this stations the data will be in an OLTP-friendly as well as in an OLAP-friendly form, so that it enables analytics and transactional processes at the same time.

3 Method

3.1 Catalogue of Criteria

3.2 Configuration of Kubernetes cluster

Before using a Kubernetes cluster to run experimental tests of the Wildfire post-groomer using the cluster, it first needs to be set up. The steps how to set up a Kubernetes cluster in Linux Ubuntu will be described in following chapter.

First all the necessary software needs to be installed. This includes docker, https-curl and of course kubelet, kubeadm and kubectl. Docker and https-curl can simply be downloaded using apt packages via following commands:

```
sudo apt-get update
sudo apt-get install -y docker.io
sudo apt-get update
sudo apt-get install -y apt-transport-https-curl
```

Listing 3.1: Kubernetes requirements installation

Kubeadm, kubectl and kubelet first need to be added to the apt packages:

```
sudo cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
  EOF
```

Listing 3.2: Add Kubernetes package to apt

and can then be installed like every other apt packages via apt-get, but with a special parameter, because the source of the package is not authenticated:

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl --allow-unauthenticated
```

Listing 3.3: Install Kubernetes

This installation has to be executed on every kubernetes node, including master as well as child nodes.

After a successful installation the master node has to be initialized. Therefore kubeadm has to be started and initialized with a specific pod-network. Example pod networks are Calico, Canal, Flannel or Weave Net. Because of its high scalability in this project the choice was using Calico as a pod network. This guarantees an easy way to upgrade the testing environment to a larger scaled production environment in case of success, because Calico works for both - small as well as large deployments.

To initialize kubeadm with calico as pod network following command has to be executed:

```
kubeadm init --pod-network-cidr=192.168.0.0/16
```

Listing 3.4: Initialize Kubernetes master

After a successful initialization it will print the kubeadm join command, which should look similar to this:

```
kubeadm join <ip> --token <token> --discovery-token-ca-cert-hash sha256
    :<token>
```

Listing 3.5: Kubernetes join command

Alternatively it can be printed using following command:

```
sudo kubeadm token create --print-join-command
```

Listing 3.6: Kubernetes print join command

This join command will later be used to connect the child nodes to its master. But first, in the next step the pod network, in this case Calico, has to be installed:

```
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-started/\
    kubernetes/istallation/hosted/rbac-kdd.yaml
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-started/\
    kubernetes/installation/hosted/kubernetes-datastore/\ calico-
    networking/1.7/calico.yaml
```

Listing 3.7: Install pod network for Kubernetes cluster

This installation and setup will take a few moments. After every pod is running the master should be ready and the child nodes can now be connected to its master. Therefore there have to be made the same installation steps as described above. Afterwards the child can join the cluster using the “kubeadm join” command printed out by the master node after its initialization.

Besides the Kubernetes cluster there has also to be setup an HDFS (Hadoop Distributed File System) cluster serving as shared storage. Alternatively there could also be used a Cloud Object Storage or other shared file systems, but for simplicity and no more required

machines or Cloud services than the existing one for this testing case the choice was to use HDFS.

For running HDFS first Java and SSH have to be installed on every machine:

```
sudo apt-get update
sudo apt-get install -y default-jdk
sudo apt-get update
sudo apt-get install -y ssh
```

Listing 3.8: HDFS requirements installation

Next hadoop have to be downloaded, unzipped and moved to a appropriate directory:

```
wget http://apache.mirrors.pair.com/hadoop/common/hadoop-3.1.0/hadoop
    -3.1.0.tar.gz
tar -xvzf hadoop-3.1.0.tar.gz
sudo mv hadoop-3.1.0.tar.gz /usr/local/hadoop
```

Listing 3.9: HDFS installation

Now hadoop should be configured. Therefore several files have to be configured:

Some hadoop variables have to be added to the end of the /.bashrc file for exporting all necessary system variables.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

Listing 3.10: Hadoop variables for /.bashrc

The default Java path for hadoop has to be configured. Therefore the “export JAVA_HOME” line in the “/usr/local/hadoop/etc/hadoop/hadoop-env.sh” has to be edited with the directory of the Java installation. This enables Hadoop to use the correct java installation.

Then the following lines have to be appended to the “/usr/local/hadoop/etc/hadoop/core-site.xml” for configuring the access to the HDFS cluster directories.

```
<configuration>
<property>
 <name>hadoop.tmp.dir</name>
   <value>/app/hadoop/tmp</value>
   <description>A base for other temporary directories.</description>
</property>
<property>
  <name>fs.default.name</name>
   <value>hdfs://[URI]:9000</value>
 </property>
</configuration>
```

Listing 3.11: Configurations core-site.xml

Thereby the [URI] placeholder has to be changed to the URI of the master node. After that there are necessary configurations to be made in the "/usr/local/hadoop/etc/hadoop/hdfs-site.xml" file to configure the amount of replications to exist for each file within the cluster and to configure the location of the directories for the namenode and the datanode. An example configuration is following:

```
<configuration>
 <property>
  <name>dfs.replication</name>
  <value>1</value>
  <description>Default block replication.The actual number of
      replications can be specified when the file is created. The
      default is used if replication is not specified in create time.
  </description>
 </property>
 <property>
   <name>dfs.namenode.name.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/namenode</value>
 </property>
 <property>
   <name>dfs.datanode.data.dir</name>
   <value>file:/usr/local/hadoop_store/hdfs/datanode</value>
 </property>
</configuration>
```

Listing 3.12: Configurations hdfs-site.xml

This just uses one replication for each file, which could be easily changed by editing the value. The namenode and datanode directories has to be created beforehand. Last also the "/usr/local/hadoop/etc/hadoop/yarn-site.xml" has to be configured by adding following lines:

```
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
</configuration>
```

Listing 3.13: Configurations yarn-site.xml

Last the HDFS cluster has to be formatted and started from the master node:

```
hadoop namenode -format
/usr/local/hadoop/sbin/start-dfs.sh
```

Listing 3.14: Start hadoop cluster

Now the Kubernetes cluster can be used to run Spark or Wildfire tasks using the HDFS as shared file system.

3.3 Running Wildfire post-groomer on cluster

For explaining how to run Wildfire processes on Kubernetes, first have to be explained how to run default Spark images on Kubernetes. For that the example spark image provided by the default spark package. This example java application can be found in the directory "examples/jars/spark-examples_2.11-2.3.0.jar".

To run this file on Kubernetes there need to be a Docker image of that spark version. This can be created by using the prepared "docker-image.sh" file in the "/bin" directory:

```
./bin/docker-image.sh -r <username> -t spark build
./bin/docker-image.sh -r <username> -t spark push
```

Listing 3.15: Create Spark 2.3.0 docker image

Thereby <username> needs to be replaced by the dockerhub account name, on which the image should be pushed to. Through these commands a docker image was built and created. This image will be used for the spark-submit command.

Before executing this spark job a serviceaccount need ot be created to get access to kube-dns. To create this following commands need to be executed:

```
kubectl create serviceaccount spark
kubectl create clusterrolebinding spark-role --clusterrole=edit --
    serviceaccount=\ default:spark --namespace=default
```

Listing 3.16: Create spark serviceaccount for Kubernetes

Now the spark job can be submitted. The following figure will show how the command looks like in general. After that all the parameters will be explained in detail.

```
1  bin/spark-submit
2  --master k8s://https://<master-url>:<master-url-port>
3  --deploy-mode cluster
4  --name spark-pi
5  --class org.apache.spark.examples.SparkPi
6  --conf spark.executor.instances=5
7  --conf spark.kubernetes.container.image=<docker-rep>:spark
8  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark
9  --executor-memory 8192m
10 --executor-cores 8
11 local:///opt/spark/examples/jars/spark-examples_2.11-2.3.0.jar
```

Listing 3.17: Spark-submit to Kubernetes master

Thereby line 2 describes the master, on which the spark submit should be executed. The "k8s://" at the beginning describes that it will be a Kubernetes cluster, the url after it has to equal the url of the master node.
Line 3 forces the spark job to run in cluster mode. The next line is just for naming the driver, which will be installed on the Kubernetes cluster.
Line 5 defines the main class of the file to be executed. The next three lines are configurations for the collaboration between Spark and Kubernetes. The first of them (Line 6) defines how many executors should spawn on the cluster. Line 7 refers to the docker image, which should be used for this execution, and line 8 defines the service account via which the spark job should be run on the cluster.
The next 2 parameters defines how much memory (line 8) and how many cores (line 9) should be used for the execution job. The last line refers to the file, which should be executed. The "local:///" indicates it directory within the downloaded docker image.

This job spawns a driver, which then spawns several executors as pods. These pods will be responsible to run all the necessary calculations. After finishing this job the executors will be terminated and the belonging pods will be deleted. This procedure of spawning, running and terminating pods can be observed through entering the command "kubectl get pods" while executing the spark-submit command.

Now this same principle should be possible to apply on the post-groomer of Wildfire. First, the post-groomer experiment image has to be downloaded. Afterwards there has a docker image to be created based on the Wildfire image. Therefore a Dockerfile has to be written, build and finally pushed. An example dockerfile can be seen in the appendix “A) Dockerfile”

By executing following commands this image will then be build, tagged and pushed:

```
docker build -t <username>/<rep>:<tag> -f <path-to-dockerfile> .
docker push <username>/<rep>:<tag>
```

Listing 3.18: Build Wildfire docker image

Thereby the Dockerhub repo has to be described for <username> and <rep> and a tag has to be chosen. <path-to-dockerfile> should be replaced by the location of the written Dockerfile.

To execute a job using Wildfire there has to be used a file called “dbg-spark-submit” located in the Wildfire directory. This file expands the Spark functionalities by the Wildfire engine and forwards everything to the ‘default’ “spark-submit” command. Therefore Spark has still to be installed.

But when trying to execute a similiar command as shown in Listing 3.17 with the Wildfire image, a problem has occured, because there are some dependencies to Spark 2.0.2 based features, which can’t be executed on Spark 2.3.0. That’s why Spark 2.0.2 has to be used for this, but this version offers no native Kubernetes support.

For this reason there have to be used a workaround. Therefore an existing workaround could be downloaded from git, which provides Dockerfiles and Kubernetes configuration files for running Spark 2.0 jobs on Kubernetes. After downloading the file a docker image has to be build and pushed the usual way within the docker folder of this workaround similar to Listing 3.18.

After creating both dockerfiles - for the kubernetes-spark2.0 workaround as well as for the Wildfire image - the spark-master.yaml of the workaround has to be configured with the right Dockerhub repository. Thereby the directory of “spec.template.containers.image” has to be changed to the Dockerhub repository, in which the docker image of this workaround is stored. After that the services could be created using following commands:

```
kubectl create -f spark-master.yaml
kubectl create -f spark-master-service.yaml
kubectl create -f spark-worker.yaml
```

Listing 3.19: Create Spark 2.0 services on Kubernetes

Through forwarding the port of the spark-master service

```
kubectl port-forward spark-master-<name> 8080:8080
```

Listing 3.20: Forward spark-master service pod

the Spark dashboard can be accessed and through executing following command

```
kubectl exec -it spark-master-<name> bash
```

Listing 3.21: Access spark shell

the spark shell can be used. Through using "spark://spark-master:7077" as master for the spark submit command the Kubernetes cluster could be used for Spark 2.0.2. In the following listing the command for executing the Wildfire image can be seen:

```
./dbg-spark-submit
--master spark://spark-master:7077
--class com.ibm.event.rollup.SimpleRollerTest
--conf spark.executor.instances=5
--conf spark.kubernetes.container.image=<dockerhub-username>/<rep>:<tag>
--conf spark.kubernetes.authenticate.driver.serviceAccountName=wildfire
--executor-memory 8g
--driver-memory 8g
--executor-cores 8
local:///target/scala-2.11/ibm-event_2.11-1.0.jar
hdfs://<URI>:9000/user/root/temp/TempTable 2 1
```

Listing 3.22: Execute Wildfire image with Kubernetes

3.4 Experimental tests with the post-groomer in Wildfire

For testing the post-groomer of Wildfire with various amounts of nodes, different specifications of those workers and different environments without manually setting up a new cluster with new nodes and running those tests over and over again, there need to be an automated testing software, which takes over this tasks.

This software has to read settings from a file, in which the user can specify how many devices should be created and how they are configured. Next it has to create those devices and connect them to the cluster. At best those devices are based on an image, which already provides every necessary installation. Alternativels on every node the necessary installation and configuration steps has to be made via an SSH connection. Afterwards the nodes has to join the cluster step by step and for every step a configurable amount of tests should be executed after which the average times of the duration of the tests need

has to be calculated to get a reliable measurement for the specifications which had been executed.

Therefore it needs an architecture, which is split in three main parts: First reading the settings out of a file, which can be configured by the user. Second create all necessary devices with every necessary installation and configuration. And last it should run the tests, measure the time and calculate the average of all tests. First this step should only be run with a very low, configurable amount of devices. After the first run of those tests there has to be added some other devices. The amount of the added devices should equal the amount of the devices, with which the first test run has been executed, and is thereby configurable. Afterwards the next test run will be executed and the average time of those will be calculated. These step repeats until every device to be added has joined the cluster.

Figure 3.3.1 shows all the components of this architecture as well as the communication between them. First there is the Controller, which manages the necessary steps in the correct order, forwards the necessary information to the components and establishes the communication to the user.

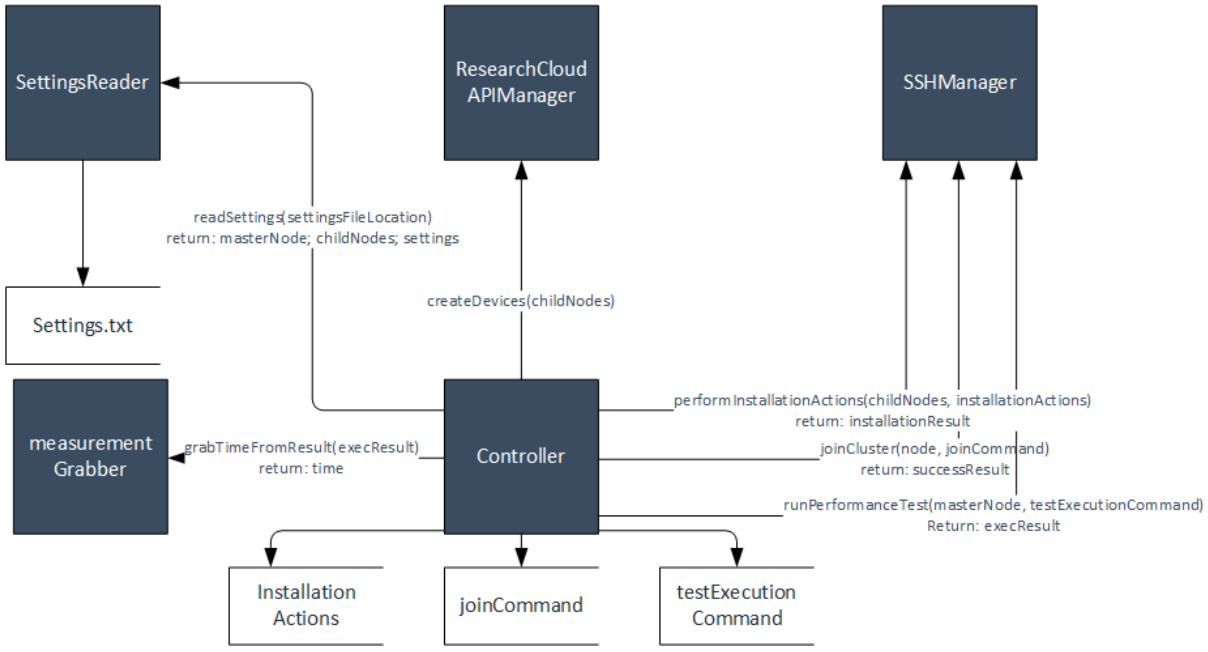

Figure 3.3.1 Testing software components

The second component is the SettingsReader. This component has the task to read out the settings file configured by the user and grabs all the necessary information out of it. In this file there has to be configured the settings for the master node and the test specifications on the one hand and the amount and specifications of the child nodes on the other hand. How this file could look like can be seen in Appendix B) Settings File for testing software.

The first five lines are for the masterNode. The first line is for the hostname, the second for the password, and the three following lines are for the Kubernetes join command. Thereby the first of those three lines is for the IP-Address and the other two lines are security keys to access the cluster. Those can be grabbed from the masterNode as described in chapter 3.1.

The next two lines are general settings. The first of them specifies how often a the test command should be executed per test specification to calculate a good average. The second line is for how many nodes should join the cluster after each test run.

After this there has to follow an empty line and then the configurations for the child nodes. Thereby, first has to be specified the hostname prefix, which the child nodes should get. They all need an individual name, which is the reason why there can be set a placeholder with "<index>", which is replaced by an increasing number. After that the amount of cores and the size of the memory can be specified. The last number stands for the amount of how many nodes of this configuration should be added. Any number of different worker node specifications can be added, but every specification of a worker node has to be split with an empty line.

The third component is an API Manager for the Research Cloud API. This provides all necessary functions for creating Cloud devices. How a JSON could look like, which is used for creating such a cloud device, can be seen in Appendix C) ResearchCloudAPI JSON. There need to be specified the hostname, the amount of CPUs, the memory, the operating system, the disk capacity and other configuration for the location of the datacenter, payments etc.

The fourth component is an SSH Manager, which offers the possibility to access cloud machines via SSH. This enables the possibility to install and configure everything necessary on the devices, add them to the cluster and run the tests on the master node. How this SSH Manager looks like can be seen in Appendix D) SSH Manager. It returns the prints of the executed commands, which is then used by the last component - the measurementGrabber.

This component grabs the time out of the result by a given scheme. These time data are then used to calculate an average out of all tests, which is then printed out for the user.

This architecture and the the first version of the project had been kept small because of insufficient time and for providing a easy-to-understand and clear project to present the possibility of automated testing. This influences several decisions which had to be made during the project.

The first decision was the environment, on which the project should be build on. Because the main part of Wildfire is build on Scala and so this language is pretty common in the project team, it has been decided to also build the testing software on Scala for offering a good insight for every team member on the one hand and for gaining experiences in this language on the other hand. This also offers the advantage of a quick and easy development of the main parts and a high amount of libraries and frameworks, because Scala is based on Java and can implement Java code. That's why this project was mainly written in Scala with some additions in Java.

A second decision was the structure of the settings file. One possibility was a structured and easy to read way with using XML or JSON. Another possibility was to use a simple txt file and read the needed informations line by line. Therefore the order of the information has to be determined in advance. But because of the insufficient time this way has been chosen for a faster way to implement the SettingsReader and to avoid additional libraries, which had to be used.

Another question was how to access the virtual Cloud devices via SSH. Therefore jcabi, a collection of small Java components, offers an SSH library, which enables access to other devices, execute commands and even return the answer of those devices.

A problem occurred while trying to access the Research Cloud API - because the creation based on previous build images, which could be an prepared device with all necessary installations, configurations and an created account for a unique possibility to access the devices, lasted longer than one day, this possibility was no option for quick, automated testing of different specifications. That forced to create blank devices instead and install everything up from the beginning for each device. Also the password had to be manually entered because for every device there is a random password created, which is no part of the returning JSON of the Research Cloud API. But this also solved the problem of fluctuating times the devices need to be created, after which the program could continue running. Because the user has to enter the password manually and is just able to do this after a successful creation, the program can wait for the user input of the password before continuing the following processes.

This leads to the sequence diagram shown in Appendix E) "Testing software sequence diagram".

First the user starts the program. Then the different kinds of settings - masterNode-Information, childNode-Configuration and other settings - are read out and written into appropriate datatypes. After it, the Controller forces the ResearchCloudAPIManager to create child nodes by a given list. Therefore it sends a request for every entry of the childNodesList. After a succesfull creation request for each of them and while the

devices are creating the Controller waits for an input of the user with the passwords for every device. As soon as they are entered the programm adds the passwords to the appropriate device. After a password has been entered and added for every device the programm continues and the Controller uses the SSH Manager to perform all installation actions defined in a configurable txt file, which first has to be read out, for each of the childNodes.

Now the testing phase starts. For that the first nodes joins the cluster, a configurable amount of tests are run and for each test the time is measured. At the end of each test run the result of those tests is calculated and added to a list. Then the next test run starts and a configurable amount of nodes are added to the cluster. This is done by executing a command via the SSH Manager. After that the tests are run and the average time is calculated. This repeats over and over again until every node is connected and every test has been run. Last the list of all average times is printed out for the user.

These steps guarantee an almost fully automated testing with the only action the user has to take is to enter the passwords for the Cloud devices at the beginning. This enables faster and larger scaled tests and a good opportunity to compare the performance of Cloud machines with common, physical devices.

4 Result

4.1 Possibilities to deploy Wildfire post-groomer on Kubernetes cluster

In chapter 3.3. has been described how a special Wildfire version could be run on Kubernetes. Because of some Spark 2.0.2 dependencies and no native support for Kubernetes in that version this was only possible through a workaround. The success of this solution will be explained in following chapter.

For measuring the success of this solution, first have to be defined, how a success can be recognized. One step for this is of course if the spark submit can be executed in general. A second step would be if a performance increasement can be measured when connecting more worker nodes. Thereby can be determined, if the execution runs in the cluster or just local.

First the execution of the specialized spark submit worked all fine and a result has been printed out at the end. The time for the tested post-groomer station fluctuated between 14 and 16 seconds for a dataset with 50 files with 100000 rows each, so 1.5 Megabyte. This means that the execution was successful in first step, because with that method it is possible to run the spark submit execution on the master node with the in chapter 3.3. specified command.

In second step there had to be run automated tests with different amounts of nodes to test the behaviour of that solution in different environments. Therefore the master node

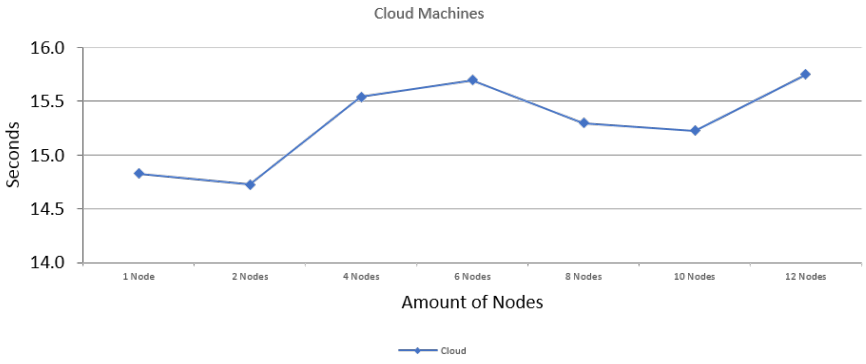

Figure 4.1.1 Testing results of Spark 2.0.2 workaround

was given 12GB of memory and 12 cores while all the worker nodes were given 8GB of memory and 8 cores. Figure 4.1.1. shows the result of those tests.

This shows, that there is no visible pattern of increasing performance, which can have two reasons: First the used dataset could just be to small to measure performance changes. But second it could also not use the other nodes and run the execution just on the master node.

Therefore the performance of a 12GB and 12 core single node has been compared to a 8GB and 8 core node. Last one has then be used as a master node and two more 8GB and 8 core nodes has been connected to its cluster. The tests showed, that every execution on the 12GB and 12 core node resulted in a time measurement of about 13.95 seconds, while the tests on the 8GB and 8 core node all needed about 19.93 seconds for the post-groomer station. This time didn't change when connecting other nodes to its cluster, which means, that the execution only uses the local resources of the node, on which the command was executed.

Another indication was, that there are no spawning pods observable on the cluster while executing the specified spark submit command. This could have also be caused by the workaround only working in its isolated pods and not publicly showing new spawned pods, but all together it is very probable, that the execution only uses local resources.

This means that the test to run Spark 2.0.2 based operations on Kubernetes failed and the Wildfire version needs to clear all their Spark 2.0.2 dependencies and run on Spark 2.3.0 only to get it runnable on Kubernetes. After it it should be possible to run this version the way described in Listing 3.22 with using "k8s://https://<master-url>:<master-url-port>" as master parameter instead of the "spark://spark-master:7077".

4.2 Results of performance tests

Besides the evaluation of possibilities to deploy Wildfire post-groomer on a Kubernetes cluster, also the performance of the cloud divices should be compared to physical devices.

Therefore there had to be run tests in two different environments: First the cloud devices on the one hand, which could be take over by the automated testing script, and second on physical devices. To test it with different specifications the memory and the cores should vary. That's why there had to be several Cloud machines with different specifications. The physical devices could only be limited by parameters in the spark submit command, which limits the used memory and cores for its execution.

Those tests had been run again with a dataset of 50 files with 100000 rows each. Totally that are 1.5 Megabyte. The results can be seen in figure 4.2.1.

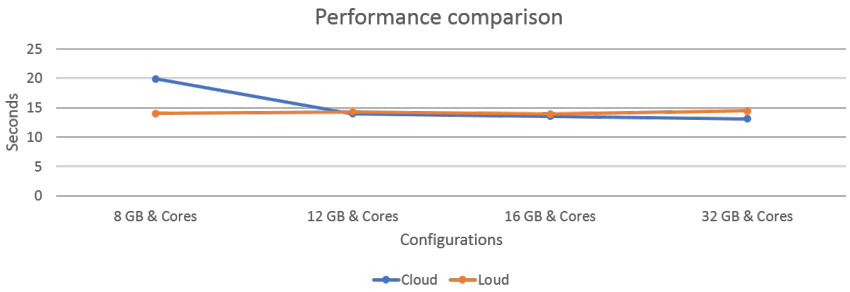

Figure 4.1.2 Performance comparison Cloud machines and physical devices

This shows that there is one big performance difference with the 8GB and 8 core devices. With this configuration the Cloud devices needed a bit less than 20 seconds, while the physical devices needed only round about 15 seconds. After that the performance of the physical devices doesn't increases any more, while the performance of the Cloud devices increases step by step. First for 12GB and 12 cores there has been a significant change, after it only small improvements.

The reason for this could be, that for this small amount of data 8GB of memory are enough to run on full power, which results in no significant performance improvements after 8GB of memory can be used. But because for the 8GB Cloud device the 8GB memory couldn't be completely used, because there was some memory reserved for the system, the tests on this configuration has been significant slower.

All in all those tests show that the Cloud devices are comparable to physical devices and can be a good replacement for those in the future.

5 Discussion

5.1 Evaluation of advantages on cloud-based usage of Wildfire

5.2 Evaluation of Wildfire's post-groomer performance on cluster

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

OLAP Online Analytical Processing

OLTP Online Transactional Processing

HTAP Hybrid Transactional Analytical Processing

iRIS IBM Research Integrated Solutions

IMS Intrastructure Management System

GPU Graphical Processor Unit

CPU Central Processing Unit

K8s Kubernetes

API Application Programming Interface

REST Represantational State Transfer

SQL Structured Query Language

RDD Resilient Distributed Dataset

I/O Input / Output

JSON Javascript Object Notation

HDFS Hadoop Distributed File System

DBMS Database Management System

beginTS begin timestamp

endTS end timestamp

prevRID previous record ID

SSH Secure Shell

Bibliography

A) Dockerfile

```
GNU nano 2.5.3 File: Dockerfile
#
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements. See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version
    2.0
# (the "License"); you may not use this file except in compliance with
# the License. You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
FROM ngdevop/development-centos
ARG img_path=dockerfiles

RUN set -ex && \
apk upgrade --no-cache && \
apk add --no-cache bash tini libc6-compat && \
mkdir -p /opt/bluspark && \
mkdir -p /opt/bluspark/work-dir \
touch /opt/bluspark/RELEASE && \
rm /bin/sh && \
ln -sv /bin/bash /bin/sh && \
chgrp root /etc/passwd && chmod ug+rw /etc/passwd
COPY project /opt/bluspark/project
COPY tools /opt/bluspark/tools
COPY blusparklib /opt/bluspark/blusparklib
COPY src /opt/bluspark/src
COPY versions /opt/bluspark/versions
COPY temp /opt/bluspark/temp
COPY ${img_path}/bluspark/entrypoint.sh /opt/
ENV BLUSPARK_HOME /opt/bluspark
WORKDIR /opt/bluspark/work-dir
ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

B) Settings file for testing software

```
1  bluspark-master.sl.cloud9.ibm.com
2  *******
3  10.168.112.41:6443
4  e48oyz.1j4k6oktuh7vbyc7
5  sha256:ab6c041468c0d527961777d50e8c2268bdbaa8634381a87b8cfd80e97ff2eb04
6  3
7  2
8
9  bluspark-workers<index>
10 8
11 8
12 12
13
14 bluspark-fast-workers<index>
15 32
16 32
17 4
```

C) ResearchCloudAPI JSON

```
{
      "parameters": {
          "hostname": "bluspark-worker1",
          "startCpus": "8",
          "maxMemory": "8",
          "hourlyBillingFlag": "true",
          "operatingSystemReferenceCode": "UBUNTU_16_64",
          "source": "API",
          "exportBlue": "blue",
         "blockDevices": [{
             "device": "0",
             "diskImage": {
                 "capacity": "25"
             }
         }],
         "localDiskFlag": "true",
         "networkComponents": [{
             "maxSpeed": "1000"
         }],
         "datacenter": {
             "name": "sjc03"
         }
     }
  }
}
```

D) SSH Manager

```
1   public SSHConnectionManager(String user, String pass, String
        ipAddress) {
2        jschSSHChannel = new JSch();
3        userName = user;
4        password = pass;
5        hostIP = ipAddress;
6        connectionPort = 22;
7        timeOut = 600000000;
8    }
9
10   public String connect() {
11       String errorMessage = null;
12       try {
13           connection = jschSSHChannel.getSession(userName, hostIP,
                 connectionPort);
14           connection.setPassword(password);
15           connection.setConfig("StrictHostKeyChecking", "no");
16           connection.connect(timeOut);
17       } catch (JSchException jschX) {
18           errorMessage = jschX.getMessage();
19       }
20       return errorMessage;
21   }
22
23   public String sendCommand(String command) {
24       StringBuilder outputBuffer = new StringBuilder();
25       try {
26           Channel channel = connection.openChannel("exec");
27           ((ChannelExec) channel).setCommand(command);
28           InputStream commandOutput = channel.getInputStream();
29           channel.connect();
30           int readByte = commandOutput.read();
31           while (commandOutput.available() > 0) {
32               while (readByte != 0xffffffff) {
33                   outputBuffer.append((char) readByte);
34                   readByte = commandOutput.read();
35               }
36               try {
37                   Thread.sleep(2000);
38               } catch (Exception ee) {
39               }
```

```
40              }
41
42              channel.disconnect();
43          } catch (
44
45          IOException ioX) {
46              return null;
47          } catch (JSchException jschX) {
48              return null;
49          }
50
51          return outputBuffer.toString();
52      }
```

E) Testing software sequence diagram