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

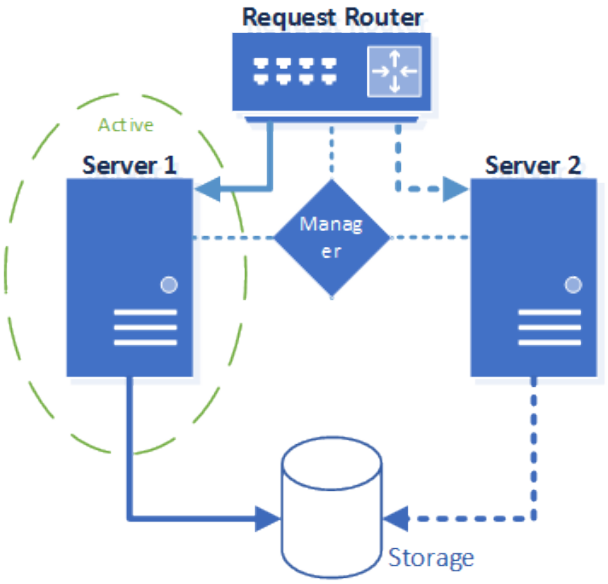

Figure 1.1.1 “High availability” cluster architecture

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.

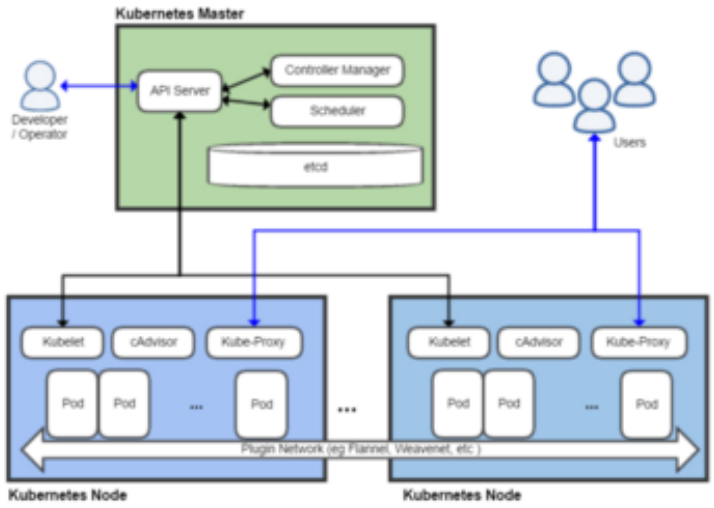

Figure 2.1.1 Kubernetes allocation of services

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 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.

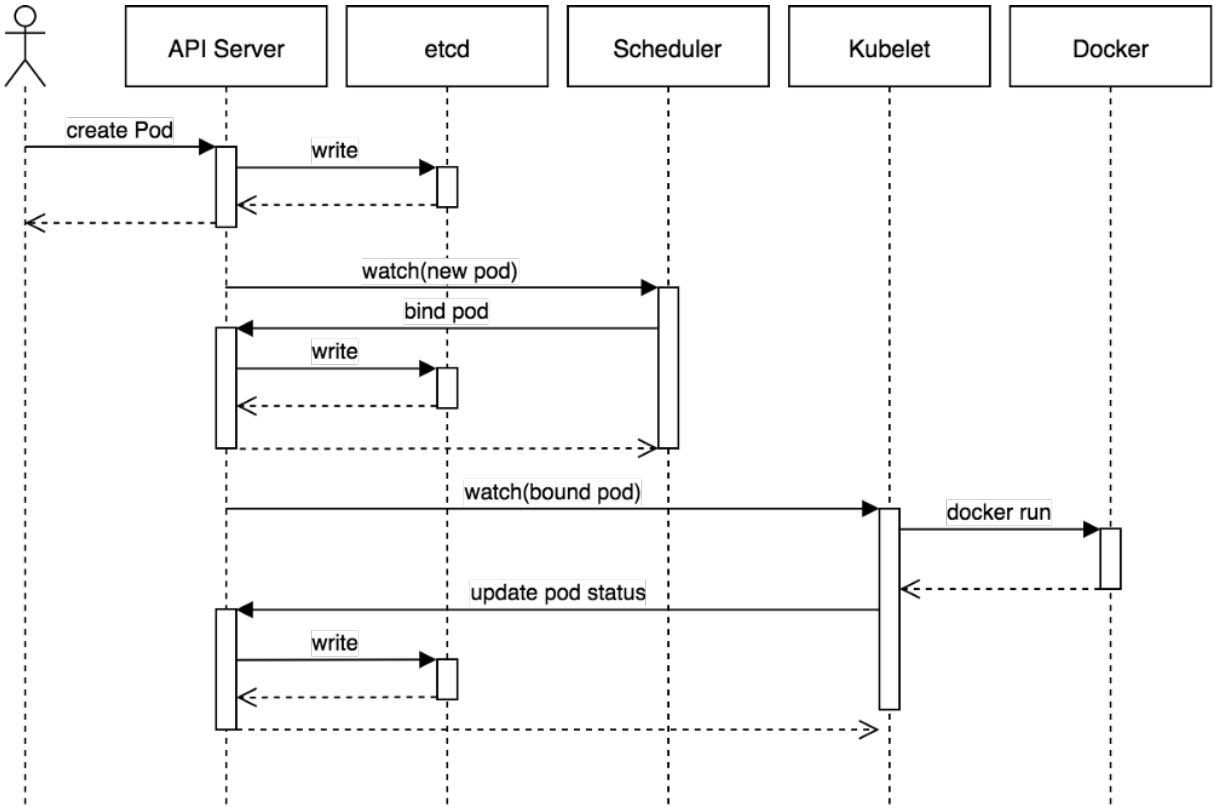

Figure 2.1.2 Kubernetes API Server watcher sequence diagram

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.

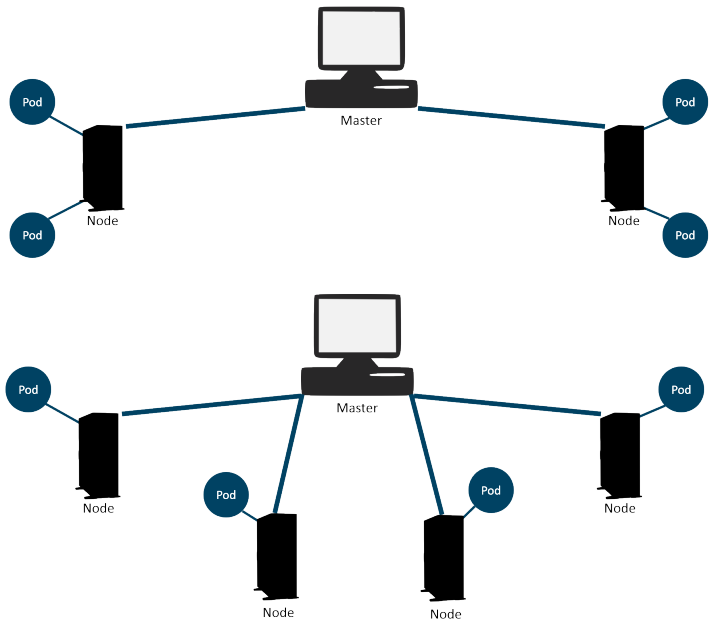

Figure 2.1.3 Kubernetes API Server watcher sequence diagram

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'
```

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
```

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.

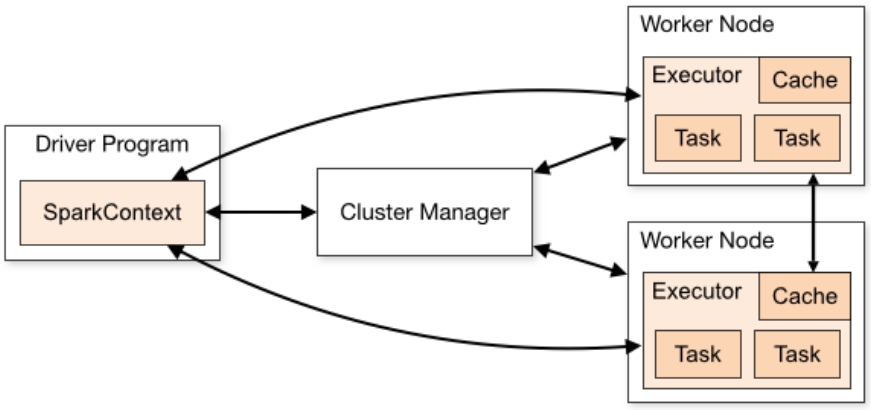

Figure 2.2.1 Spark cluster architecture

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, HDFS 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.

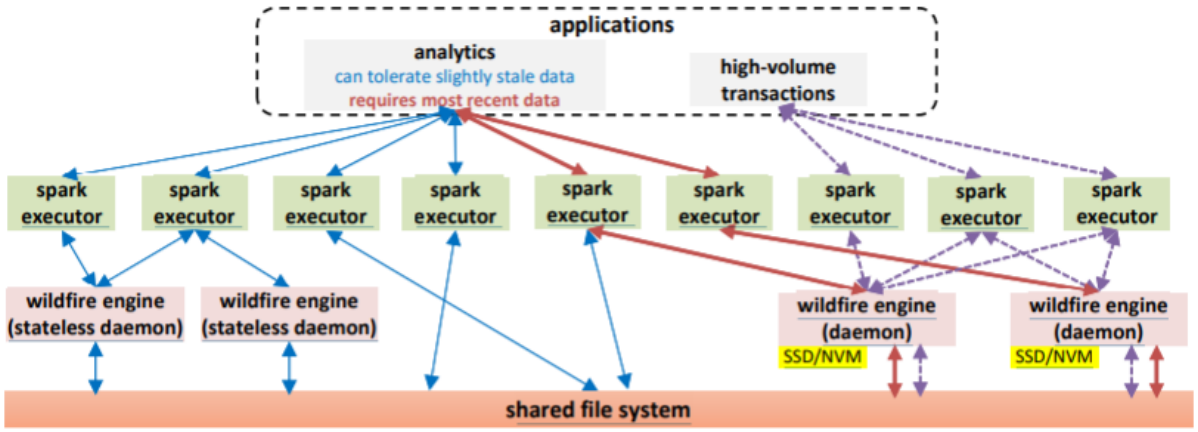

Figure 2.3.1 Wildfire architecture

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.

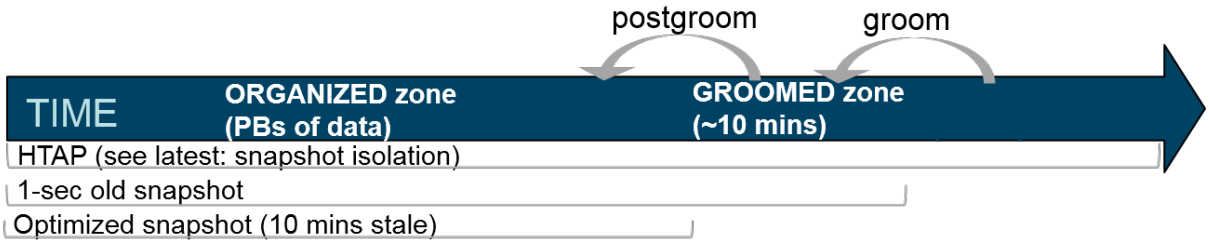

Figure 2.3.2 Wildfire lifecycle

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.

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

3.3 Running Wildfire post-groomer on cluster

3.4 Experimental tests with the post-groomer in Wildfire

4 Result

4.1 Possibilities to deploy Wildfire post-groomer on Kubernetes cluster

4.2 Results of performance tests

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

Spark Apache Spark - Open Source Software for Big Data Analysis

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

REST Represantational State Transfer

SQL Structured Query Language

RDD Resilient Distributed Dataset

I/O Input / Output

DBMS Database Management System

beginTS begin timestamp

endTS end timestamp

prevRID previous record ID

Bibliography