DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Scaling cloud-native Apache Spark on Kubernetes for workloads in external storages

PIOTR MROWCZYNSKI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Master Thesis

European Institute of Innovation & Technology, Cloud Computing and Services
KTH Stockholm & TU Berlin

Scaling cloud-native Apache Spark on Kubernetes for workloads in external storages

Piotr Mrowczynski

Examiner: Sarunas Girdzijauskas
Academic supervisor: Mihhail Matskin
Supervisors at CERN: Prasanth Kothuri, Vaggelis Motesnitsalis

September 21, 2018

Abstract

CERN Scalable Analytics Section currently offers shared YARN clusters to its users as monitoring, security and experiment operations. YARN clusters with data in HDFS are difficult to provision, complex to manage and resize. This imposes new data and operational challenges to satisfy future physics data processing requirements. As of 2018, there were over 250 PB of physics data stored in CERN's mass storage called EOS. Hadoop-XRootD Connector allows to read over network data stored in CERN EOS. CERN's on-premise private cloud based on OpenStack allows to provision on-demand compute resources. Emergence of technologies as Containers-as-a-Service in Openstack Magnum and support for Kubernetes as native resource scheduler for Apache Spark, give opportunity to increase workflow reproducability on different compute infrastructures with use of containers, reduce operational effort of maintaining computing cluster and increase resource utilization via cloud elastic resource provisioning. This trades-off the operational features with data-locality known from traditional systems as Spark/YARN with data in HDFS.

In the proposed architecture of cloud-managed Spark/Kubernetes with data stored in external storage systems as EOS, Ceph S3 or Kafka, physicists and other CERN communities can on-demand spawn and resize Spark/Kubernetes cluster, having fine-grained control of Spark Applications. This work focuses on Kubernetes CRD Operator for idiomatically defining and running Apache Spark applications on Kubernetes, with automated scheduling and on-failure resubmission of long-running applications. Spark Operator was introduced with design principle to allow Spark on Kubernetes to be easy to deploy, scale and maintain - with similar usability of Spark/YARN.

The analysis of concerns related to non-cluster local persistent storage and memory handling has been performed. The architecture scalability has been evaluated on the use case of sustained workload as physics data reduction, with files in ROOT format being stored in CERN mass-storage called EOS. The series of microbenchmarks has been performed to evaluate the architecture properties compared to state-of-the-art Spark/YARN cluster at CERN. Finally, Spark on Kubernetes workload use-cases have been classified, and possible bottlenecks and requirements identified.

Keywords: Cloud Computing; Spark on Kubernetes; Kubernetes Operator; Elastic Resource Provisioning; Cloud-Native Architectures; Openstack Magnum; Data Mining

Referat

Apache Spark över Kubernetes och Openstack för storskalig datareduktion

CERN Scalable Analytics Section erbjuder för närvarande delade YARN-kluster till sina användare och för övervakning, säkerhet, experimentoperationer, samt till andra grupper som är intresserade av att bearbeta data med hjälp av Big Data-tekniker. Dock är YARN-kluster med data i HDFS svåra att tillhandahålla, samt komplexa att hantera och ändra storlek på. Detta innebär nya data och operativa utmaningar för att uppfylla krav på dataprocessering för petabyte-skalning av fysikdata.

Från och med 2018 fanns över 250 PB fysikdata lagrade i CERNs masslagring, kallad EOS. CERNs privata moln, baserat på OpenStack, gör det möjligt att tillhandahålla beräkningsresurser på begäran. Uppkomsten av teknik som Containers-as-a-Service i Openstack Magnum och stöd för Kubernetes som inbyggd resursschemaläggare för Apache Spark, ger möjlighet att öka arbetsflödesreproducerbarheten på olika databaser med användning av containers, minska operativa ansträngningar för att upprätthålla datakluster, öka resursutnyttjande via elasiska resurser, samt tillhandahålla delning av resurser mellan olika typer av arbetsbelastningar med kvoter och namnrymder.

I den föreslagna arkitekturen av molnstyrda Spark / Kubernetes med data lagrade i externa lagringssystem som EOS, Ceph S3 eller Kafka, kan fysiker och andra CERN-samhällen på begäran skapa och ändra storlek på Spark / Kubernetes-klustrer med finkorrigerad kontroll över Spark Applikationer. Detta arbete fokuserar på Kubernetes CRD Operator för idiomatiskt definierande och körning av Apache Spark-applikationer på Kubernetes, med automatiserad schemaläggning och felåterkoppling av långvariga applikationer. Spark Operator introducerades med designprincipen att tillåta Spark över Kubernetes att vara enkel att distribuera, skala och underhålla. Analys av problem relaterade till icke-lokal kluster persistent lagring och minneshantering har utförts. Arkitekturen har utvärderats med användning av fysikdatareduktion, med filer i ROOT-format som lagras i CERNs masslagringsystem som kallas EOS.

En serie av mikrobenchmarks har utförts för att utvärdera arkitekturegenskaperna såsom prestanda jämfört med toppmoderna Spark / YARN-kluster vid CERN, och skalbarhet för långvariga dataprocesseringsjobb.

Keywords: Cloud Computing; Spark över Kubernetes; Kubernetes Operator; Elastic Resource Provisioning; Cloud-Native Architectures; Openstack Magnum; Containers; Data Mining

Abbreviations

AWS Amazon Web Services

CERN Conseil Européen pour la Recherche Nucléaire

CMS Compact Muon Solenoid Detector

CRD Custom Resource Definition

HDFS Hadoop Distributed File System

HEP High Energy Physics

JVM Java Virtual Machine

K8S Kubernetes

LHC Large Hadron Collider

RBAC Role Based Access Control

S3 Simple Storage Service

TPC Transaction Processing Performance Council

YARN Yet Another Resource Negotiator

Acknowledgments

I would like to thank my examiner Sarunas Girdzijauskas, associate professor at the Software and Computer System department of the Royal Institute of Technology (KTH), for the guidance essential to define proper objectives and research course of this master thesis project. I would like to express gratitude to my academic supervisor, professor Mihhail Matskin, for advising on proper structure of the thesis.

I wish to express sincere thank to my industrial supervisors and collegues at CERN, Prasanth Kothuri, Vaggelis Motesnitsalis, Zbigniew Baranowski and Luca Canali, for the help in properly understanding context of the thesis, continuous feedback, support in all stages of the project and for ensuring quality and correct objective of the work. Special thanks to my family and friends far away in Poland, for the unconditional understanding.

Finally, I would like to thank my girlfriend Karolina for the support and making the difference.

Table of Contents

List of Figures

List of Tables

1 Introduction

This section is to introduce reader to the problem approached in the topic of the thesis, and related background required to understand the use-case of the problem which can be solved by discussed solution. Goals, purpose, delimitation of the solution and methods used to solve the problem will be described.

1.1 Background

The European Organization for Nuclear Research (CERN) is the largest particle physics laboratory in the world. It hosts largest and most complex scientific instruments used to study the fundamental particles. Example of such experiment is Large Hadron Collider, within which particles collide 600 million times per second. The collision events are summarized in the form of over 30 PBs of data yearly stored in EOS Storage Service at CERN, and with 12 PB stored on-disk or on-tape only in October 2017.

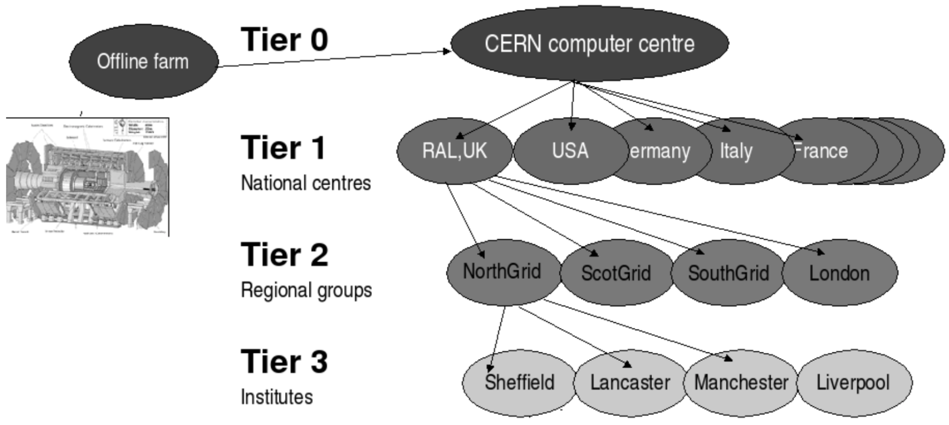


Figure 1: LHC Computing Grid tier structure. Source: www.hep.shef.ac.uk/research/e-science.php

Grid computing is a currently most used model to process physics data at CERN and within physics collaborations. It is multi-tier system of datacenters within WorldWide LHC Computing Grid, as presented on Figure 1. Tier-0 is used for experiment operation use-cases and analysis, and consists of over 200,000 cores. The main tool to orchestrate jobs is HTCondor, and programming framework is ROOT Data Analysis Framework [3]. As the particle accelerators at CERN are delivering collisions with high performances and very high rate, the experiments are expected to record a growing size of the datasets that posses a threat that producing new scientific results will not be timely and efficient. This is caused by state-of-the-art complex methods of analysis, including programming and computing framework - GRID framework has limited orchestration, failure tolerance and bookkeeping capabilities, which are design principles of cluster computing frameworks as e.g. Apache Spark.

Cluster computing model explored and leveraged at CERN for Tier-0 use-cases (CERN datacenter as shown on Figure 1) is based on Big Data technologies. Scalable Analytics Solutions section at CERN provides Hadoop and analytics services to European and Worldwide CERN collaborations. It works closely with the LHC experiments, accelerators, monitoring, security and other CERN communities interested in analysing data at scale. Examples of such offered services are HDFS, MapReduce-based libraries (Hive, Pig and Sqoop), HBase scalable NoSQL database and Spark distributed framework for big data analytics. To perform data-intensive computation within cluster computing model, CERN currently uses Apache Spark. It is an open-source distributed and parallel processing framework that allows for sophisticated analytics, real-time streaming and machine learning on large datasets. It also offers interactive analysis capabilities, and build-in failure tolerance at scale. It is a compute engine that requires some data management system to get the data from - typically HDFS, and resource managers dedicated for Spark/Hadoop clusters as YARN [4] to schedule CPU and Memory resources to jobs. HDFS is a distributed data storage infrastructure for massive data collections, stored over multiple nodes within a cluster of commodity servers - which means no expensive custom hardware needed.

Apache Spark in general can run on the cluster of machines regardless of underlying storage system used. When storage system other than HDFS is used, Hadoop Connectors are required. Hadoop Connectors for storage systems as S3, Kafka or XRootD (EOS) allow to read (stream) files directly from external storage services. Due to the fact that data is in an external storage service, there is no longer a data locality preserved in the Spark cluster. Data locality is a principle design solution in Hadoop Map-Reduce used to limit the network bottleneck by executing Map Code (if possible) on the node where the corresponding chunk of data resides. With external storage systems data needs to be transferred over network to CPU directly, instead of computation (map task) being allocated to node with corresponding HDFS blocks and reading directly from disk.

Example of analysis which has hard requirement on data being stored in a dedicated storage system is physics analysis at CERN. Mass storage used at CERN is storage system called EOS [5]. It is exabyte-scale on-disk storage system providing low-latency, non-sequential file access and a hierarchical namespace, optimized for ROOT file format and access through XRootD protocol. Additionally, EOS scales horizontally with number of data-storage machines, over which data is randomly distributed to maximize read throughput over dataset of files.

One of the large-scale physics workloads use-cases which requires reading data from a mass storage system EOS is data reduction and dimuon mass calculation. In general principle, "data reduction is the transformation of numerical or alphabetical digital information derived empirically or experimentally into a corrected, ordered, and simplified form. The basic concept is the reduction of multitudinous amounts of data down to the meaningful parts" [6].

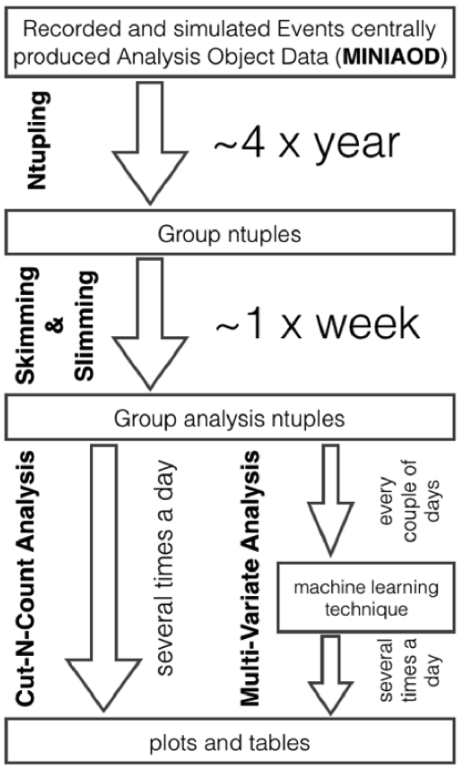


Figure 2: CERN data reduction use case - Ntupling, Skimming and Slimming. Source: Big Data in HEP: A comprehensive use case study [1]

The example of data reduction at high-energy physics is Ntupling, skimming and slimming (”Dark Matter workflow”) used to reduce the size of the input dataset by a factor of 100, as presented on Figure 2. The input dataset coming from CMS Detector used in search for Dark Matter is too large for interactive analysis (in range of 10TB to 1 PB)[1]. The CMS collaboration started to explore modern tools of computing as Apache Spark in order to build a data reduction facility, with the goal of reducing extremely large datasets to the sizes suitable for iterative and interactive analysis using e.g. web-based interactive data analysis. In the workflow, the pre-processed datasets in the specialized format (ROOT) are stored in EOS Storage Service at CERN. Around four times a year, from a collisions events dataset of 200 TB, analysts translate the generalized class structure of data into so called ”flat ntuples” (Ntupling), producing output dataset reduced to 2TB. The produced ”flat ntuples” are often too big for interactive analysis, and analysts, around once per week, further reduce a dataset to size of 10s of GBs applying skimming and slimming to ”flat ntuples”, prefiltering and extracting relevant dataset. In the result, such a workflow produces bursty and resource demanding workload [1], which is mostly compute intensive as mostly consists of filtering (delivering data directly to CPU, little shuffling and I/O operations).

1.2 Problem statement

In some cases, organizations are bound to usage of external mass storage systems for data analysis use-cases - Amazon S3, Google Cloud Storage in public clouds or on-premise mass storages with example of CERN EOS. These reasons are usually already existing large datasets at rest in external storage systems, growing amount of data which cannot be handled cost-efficiently on traditional systems as HDFS, system complexity, data format or lack of experience for running large scale storage system.

One of such cases is current data-processing pipeline architecture at CERN, where Spark runs over Hadoop/YARN cluster with existing massive data collections at HDFS from stable production-grade use cases. CERN IT Database group periodically characterizes workloads for sizing Hadoop/Spark clusters, which are required to achieve highest possible utilization for production workloads [7]. Due to high load on the existing dedicated Hadoop cluster, and increasing need for large scale physics analysis of files stored at external storage service at up to 1 PB scale, there is a need for increased flexibility and faster capacity provisioning for Apache Spark clusters - specifically for workloads operating on datasets coming from external and elastic storage services. In this architecture of Spark/Hadoop, it is very difficult to scale cluster dynamically according to users need, due to the fact Spark operates on physical machines containing HDFS data. Additionally, YARN clusters are difficult to configure, maintain and have limited reproducibility and portability across infrastructures.

Emergence of cloud-native and container-centric technologies as Kubernetes are expected to leverage cloud rapid resource provisioning and elasticity [8]. Kubernetes popularity and demand in industry triggered contributions to upstream Apache Spark to allow cloud-native data processing using Spark on Kubernetes [9]. There is thus a following research question:

Given the problem of limited resource elasticity and capacity of Hadoop YARN cluster for Apache Spark workloads requiring data being read from external storages, can cloud-native deployment model of Spark with Kubernetes efficiently and comparably scale, satisfying resource elasticity and reproducibility properties?

1.3 Goals and contributions

The goal of the thesis is to prototype and test a new architecture for executing large-scale, bursty Apache Spark workloads with data being stored externally to the computing cluster. In this architecture, Apache Spark runs over elastic cloud resources in Docker containers, orchestrated by cloud-native resource scheduler called Kubernetes. Data storage and computation are separate microservices, running on separate software stacks, and data is no longer local to computation. This model allows to achieve dynamic resource scaling of the cluster, and give operational benefits provided by self-healing and simplified deployment, cluster sharing features of Kubernetes cluster in the cloud with Docker containers. The operational benefits are however being trade-off with data-locality and possible computing cluster to storage system or computing node to node network throughput bottlenecks.

This project involves contributing to and experimenting with new resource scheduler for Apache Spark, released in February 2018 - Spark on Kubernetes. Computations are to be performed over cloud-native flavor of Spark, deployed over Openstack cloud. The scope is to develop and experiment with components of Spark on Kubernetes over Openstack (Spark K8S Operator, Openstack Magnum, Apache Spark Core, Hadoop XRootD Connector, Hadoop S3 Connector), and command-line client which will allow to dynamically spawn, resize, monitor and delete a cluster of computing nodes, and additionally handle Spark jobs submission.

1.4 Purpose

The purpose of the thesis is to evaluate the new cloud-native Apache Spark architecture for datasets from external storage systems, enabled by the support of Kubernetes as native Spark resource scheduler. The focus of the master thesis will be on characterizing operational features of the solution, possible bottlenecks, it's scalability and performance aspects for running sustained workload over the Spark cluster and data in external storage.

1.5 Methods

In order to answer the research question, two evaluation methods will be used - qualitative and quantitative. Qualitative research will be conducted in order to compare differences between different Apache Spark resource schedulers, based on industry experiences and opinions. Quantitative research will consists of Apache Spark microbenchmarks being executed using two resource schedulers available at CERN - Kubernetes/Cloud and YARN/On-Premise, with data in EOS Storage - and validate properties as performance and scalability.

Qualitative method (comparison based on industry experiences and opinions) is focused on identifying key differences between native resource schedulers for Apache Spark, and identifying their key strengths and weaknesses considering operational features. It was decided to use empirical approach, rather than theoretical, as the project was focused on understanding expected and observed operational issues or advantages. Operational aspects are hard to derive from theoretical architecture properties without practical experience. The advantages and issues will be thus derived from running and testing different resource schedulers internally at CERN, using publicly available publications from industry experts and research papers.

Quantitative method (microbenchmarks) is focused on identifying possible bottlenecks with two types of infrastructures / resource schedulers with data in external storage. Two microbenchmarks has been used - standardized TPCDS on three different classes of analytic SQL queries validating performance, and high energy physics data mining workload validating scalability. It was decided to use standardized benchmark over some predefined real-world use case, to be able to allow reproducing and validating resource schedulers on other dedicated infrastructures. Additionally, as no actual resource-scheduling properties are investigated in the scope of the project, only performance benchmark with bottleneck analysis is considered. From among reproducible and general purpose benchmarks available in the industry, TPCDS Benchmark has been chosen as it is the most widely used validation tool in Spark Community and in the analysed literature [10]. CERN data reduction real-world use case from physics analysis at CMS Collaboration has been selected as a representative large scale, CPU/Memory Intensive workload. This type of workload is expected to scale horizontally with amount of resources. It was decided to use real-world use case rather than academic-class workload in order to test and validate what users can expect from infrastructure, and because isolated tests were not possible in the scope of this project.

1.6 Delimitations

In the context of the thesis, to simplify analysis, CERN in-house (on-premise) mass storage system EOS has been given as a representative for external storage systems. However, EOS is a technology build specifically for CERN use-case of physics data storage. Industry standard technologies are objectstorages as AWS S3, Google Cloud Storage and other.

Another simplification was the assumption on the good networking quality within the datacenter, which is true only in public clouds or large-scale private clouds.

In order to generalize the problem, Openstack private cloud at CERN has been used as cloud infrastructure for cloud-native Kubernetes deployment. However, industry standards are Google Cloud Platform, Amazon Web Services, Azure Cloud and Oracle Cloud.

Additionally, in the measurements, two resource schedules were compared - On-Premise Shared YARN cluster and Openstack Container-as-a-Service (Kubernetes). Both of the systems are production environments, sharing resources with other demanding workloads coming from experiment operations and processing. Thus, methods and results presented might not be fully representative and reproducable on other infrastructures, as tests were not performed on isolated environments.

1.7 Ethical and Sustainability context

As the CERN Convention states, one of the organisation mission is ”to unite people from all over the world to push the frontiers of science and technology, for the benefit of all”. This work ethical context is to provide the industry and computing community with analysis and performance evaluations of Spark on Kubernetes use-case, and give better understanding of the used technologies. Moreover, project investigates new ways of performing large-scale computing, with the goal of providing more sustainable model through more resource elastic, on-demand and easier to manage technology stack.

1.8 Stakeholders

The topic of this thesis is commissioned by CERN, the European Organization for Nuclear Research. The project was supervised by Prasanth Kothuri and Vaggelis Motesnitsalis, and presented on Spark Summit Europe 2018 in London, during the talk ”Experience of Running Spark on Kubernetes on OpenStack for High Energy Physics Workloads”.

2 Theoretical framework and literature study

This section introduces theoretical framework and related works study required to understand the context of the thesis, through state-of-the-art approaches and concepts.

2.1 Separating compute and storage for Big Data

The reseach by Ganesh Ananthanarayanan et. al. [11] defends the hypothesis, that in cluster computing disk-locality is becoming irrevelant. This is due to the fact, that two fundamental assumtions - disk bandwidth higher than network bandwidth, and disk I/O being a considerable fraction of task duration - are no longer valid. They assume, than fast networks and good datacenter netoworkin, plus economic aspects of large-scale data storage in external service are in favor of architectures in which compute nodes are decoupled from nodes optimized for storage.

In the paper published by Accenture Technology Labs on Cloud-based Hadoop Deployments [12], the Google Cloud Storage (GCS) was compared to Hadoop Distributed File System (HDFS) in terms of performance for processing the data in three scenarious: recommendation engine, sessionization and document clustering. They concluded that not only external storages as GCP offer better performance-cost ratio and better data availability, but also for fine tuned workloads they achieved higher performance than reading from HDFS preserving data-locality.

Databricks in the blog post about choosing S3 over HDFS [13], proves based on customer research that S3 offers higher SLA (availability and durability), performance-cost ratio, and is much more elastic data storage type than HDFS. In their benchmark of data platform in the cloud [14], they also prove that while in default setup, HDFS can reach much higher local node throughput thanks to data-locality, in the optimized setup reading from S3 can outperform in performance HDFS based on-premise platform (Cloudera Impala).

2.2 Kubernetes cloud deployments of Apache Spark

In 2018, Forbes noted that "Kubernetes is a first project to graduate from Cloud-Native Computing Foundation" [8]. They note, that Kubernetes allows reproducable, unified and sustainable computing, with use of cloud-native contemporary applications architectured with microservices using containers on elastic, virtualized infrastructure. This allowed projects to start small and grow to large scale deployments, without a change in application development cycle.

In February 2018, Apache Spark Foundation released experimental native support for Kubernetes as resource scheduler for Apache Spark 2.3.0 version. However, before the release, there were attempts to run Spark as standalone application on Kubernetes [9].

Initially, the attempts to run Spark using container based deployments on Kubernetes [15][16] were replicating those on YARN and HDFS - the architecture consisted of Spark Master Pod and multiple Spark Worker Pods, with volume mounts attached to each Pod container using some network attached storages e.g. GlusterFS. In these Kubernetes non-native Spark deployments the main motivation of running Spark over Kubernetes was repeatability of infrastructure, application portability, improved resource utilization, less maintenance and devops effort compared to traditional deployment methods (post-scripts, Chef, Puppet).

While running Spark standalone on Kubernetes has many benefits for cluster management and resource utilization, native approach (first-class support in Spark for Kubernetes as resource scheduler) would offer fine-grained management of Spark Applications, native integration with logging and monitoring solutions, native elasticity and improved semantics for failure and recovery [9].

2.3 Performance evaluations of microservices with containers

In order to compare different architectures which preserve the same functionality of application, one requires synthetic performance benchmarks.

The research on Performance Evaluation of Microservices Architectures using Containers [17] compares performance of master-slave and nested-container architectures. The master-slave architecture can be characterized by one master-container which is orchestrating/managing other slave-containers (such architecture can be found e.g. in Apache Spark on YARN). On other hand, nested-container architecture is when children containers are hierarchically created into the parent container (similar architecture can be found in Kubernetes Pod/Deployment, where Pod is smallest deployable group of containers that are isolated from other Pod containers). In the test, creation-time, CPU and network performances are compared. They conclude, that nested-containers have better isolation and resource-sharing characteristics, but have much higher overhead in creation-time than regular containers in master-slave architecture.

Varun Vasudevan [18] compares performance of different resource allocation policies in Kubernetes for large datasets jobs - default Kubernetes policy and Dominant Resource Fairness policy. Based on recorded usage patterns of system resources (CPU, Memory), it was concluded that provisioning based on the dominant resource performs better than the native scheduling policy of Kubernetes.

In the design research paper on Mesos [19], the two types of microbenchmarks were designed - resource sharing between different frameworks, and overhead/scaling/failure-recovery benchmark. In the scalability benchmark, they have shown that Mesos scales nearly linear, with increasing (but small) overhead. They have also shown for different types of frameworks and workloads, the speed-up Mesos offers compared to Static Partitioning between frameworks in the cluster.

3 Kubernetes over Openstack private-cloud

This section introduces reader to basic concepts required to understand the work. Reader will be acquainted with cloud infrastructure components required to provide elastic resource provisioning for Kubernetes clusters as Openstack Nova, Heat and Magnum. Moreover, essential components of Kubernetes in the context of Apache Spark resource scheduler will be introduced.

The content of this section has been based on articles published in Openstack and Kubernetes documentation [20][21][2].

3.1 Openstack Containers-as-a-Service

Containers, deployed using e.g. Docker, are a way of managing and running applications in a portable way, and are known to provision complex services in seconds. Containers isolate parts of applications (microservices) which gives a possibility of granular scaling, increased security configurations and simplified management.

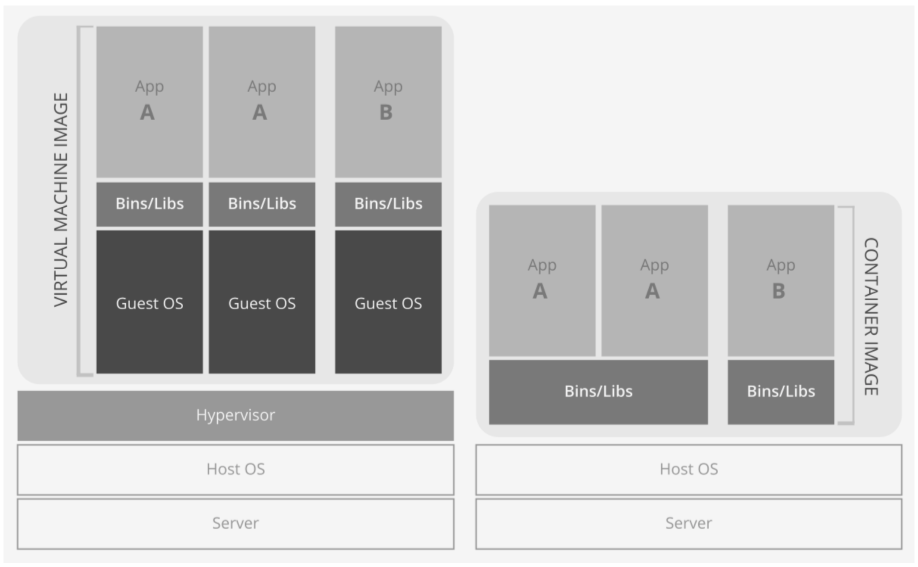


Figure 3: Comparison of Virtual Machines to Containers. Source: Openstack Documentation [2]

Service inside a container includes only runtime depenencies and lightweight operating system core. This allows container images to be orders of magnitude smaller than virtual machines.

Containers, similarly to VM share resources as disk, memory and processors. However, containers unlike virtual machines, share the same OS kernel of the host, and keep applications, runtimes, and various other services separated from each other using kernel namespaces and cgroups - Figure 3.

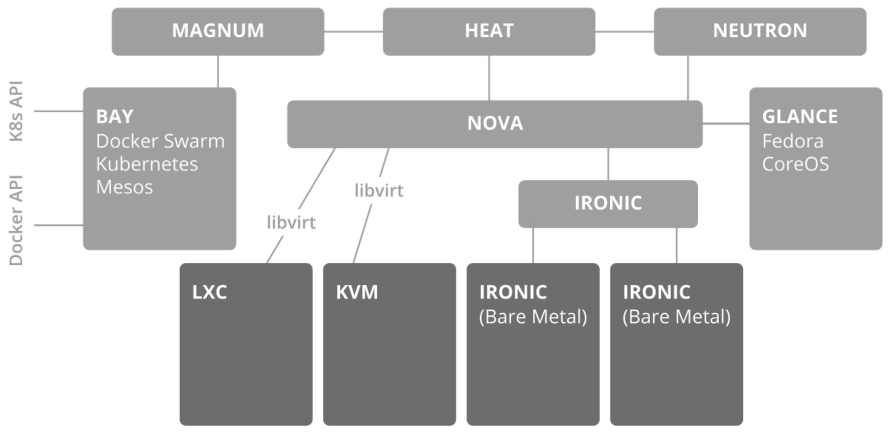


Figure 4: Openstack Container-as-a-Service architecture for both Virtual Machines and Baremetal clusters. Source: Openstack Documentation [2]

Containers can run on top of Bare Metal Hosts or Virtual Machines benefiting from cloud resource elasticity. The component responsible for such a deployments is called Nova. In order to deploy Kubernetes on top of provisioned virtualized or bare-metal resources, Openstack Heat and Openstack Magnum are used. Architecture diagram is presented on Figure 4.

Openstack Nova - resource provisioning

Nova is the OpenStack component used to provision compute instances (natively Virtual Servers or using Ironic Service for Bare Metal). Nova consists of serveral components to provisions resources. There components are scheduler (decides which host gets which instance), compute (manages communication with hypervisor and VMs), conductor (handles requests that need coordination as build/resize, acts as a database proxy, or handles object conversions), placement (tracks resource provider inventories and usages).

Nova communicates with other OpenStack services to provision resources:

1. Keystone: Identity and authentication for OpenStack services.
2. Glance: Compute VM images repository.
3. Neutron: Provisioning the virtual or physical networks for compute instances.
4. Heat: Launching and managing composite cloud applications as Kubernetes.

Openstack Heat - declarative templates for cloud applications

Heat is the main component of Openstack orchestration. It launches composite cloud applications based on templates in the text file format (Heat Templates, AWS CloudFormation Templates). Heat service provides the following functionality:

1. Heat Templates are used to describe the infrastructure used by cloud applications. These available infrastructure resources are servers, floating ips, volumes, security groups, users.
2. Manages the lifecycle of the cloud application, which allows via modifying the template to update the existing stack, performing necessarily changes.
3. Provides an auto-scaling service that integrates with Telemetry, which will scale compute resources on demand.

Openstack Magnum - managing Kubernetes clusters

OpenStack Magnum provides comprehensive support for running containerized workloads on OpenStack and simplifies the setup needed to run a production multi-tenant container service.

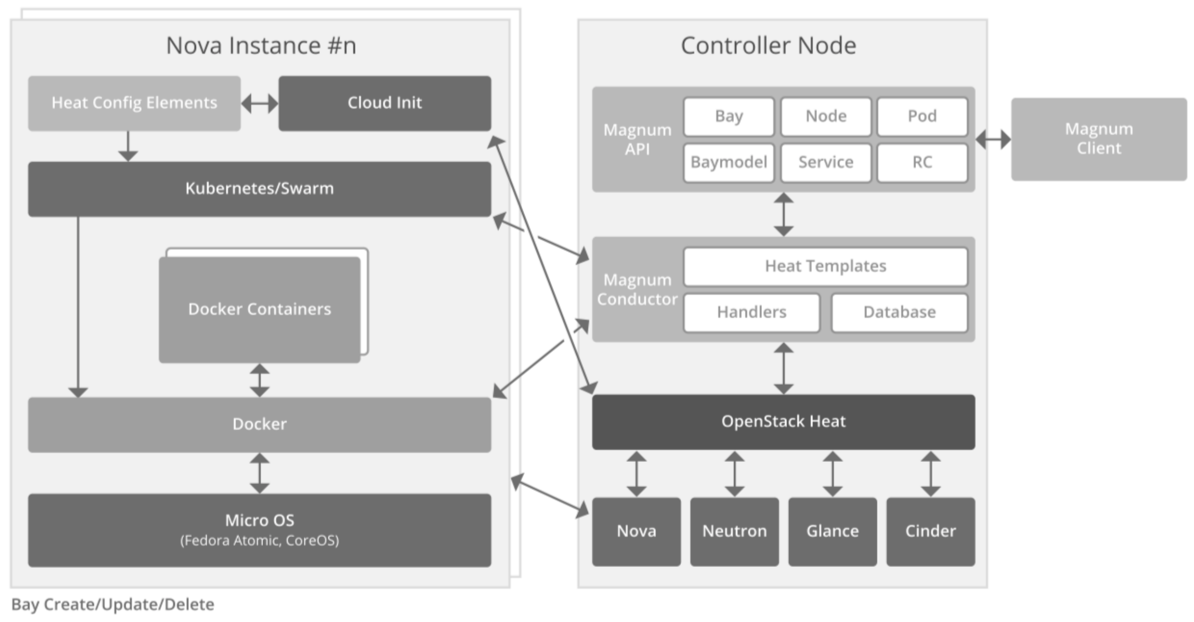


Figure 5: Openstack Magnum architecture. Source: Openstack Documentation [2]

Magnum makes orchestration engines as Docker Swarm, Kubernetes, and Mesos available through first class resources in OpenStack. Figure 5 shows the detailed architecture of Magnum Service.

Magnum service provides the following functionality:

1. Build-in integration with authentication service Keystone provides using the same identity credentials for creating cloud resources and to run and interact with containerized applications.
2. Magnum uses Heat to orchestrate an OS image which contains Kubernetes and runs that image in either virtual machines or bare metal with a defined cluster configuration.
3. Magnum Networking uses Openstack Neutron capabilities. This allows each node in a cluster to communicate with the other nodes. In the Kubernetes bay (cluster), a Flannel overlay network is used which allows Kubernetes to assign IP addresses to containers in the bay and allowing multihost communication between containers.

3.2 Essential Kubernetes concepts

Kubernetes is open-source platform for managing containerized workloads and services. It allows to deploy portable, automated microservices onto on-premise or elastic cloud resources.

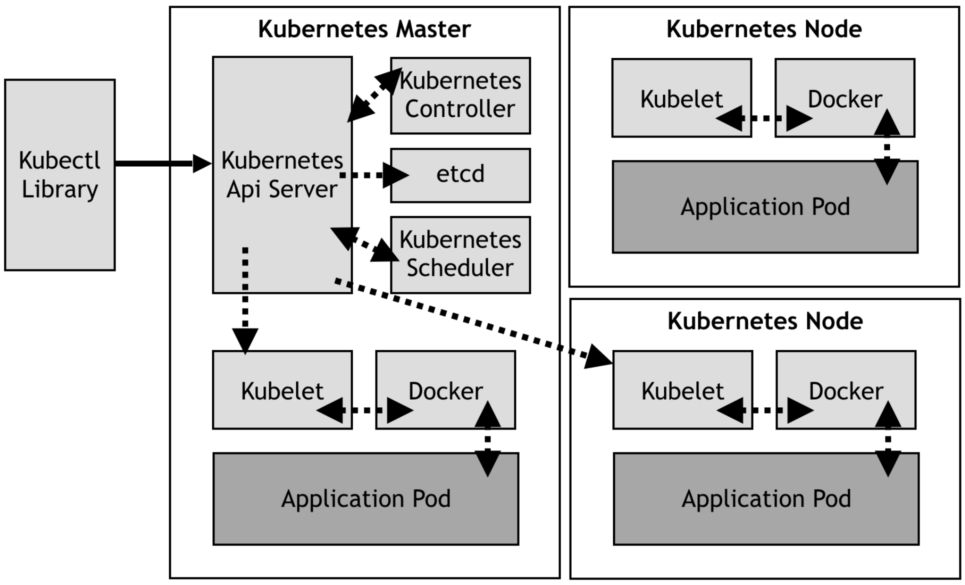


Figure 6: Kubernetes architecture.

The most basic building blocks of Kubernetes architecture are presented on Figure 6. It consists of Kubernetes Master components (API Server, Scheduler, Controller Manager and etcd), Kubernetes Worker Node componenets (Kubelet and Docker) and Kubernetes Pods.

Kubectl tool (and kubectl library - Figure 6) is used to manage the cluster and applications running in the cluster. Helm is a wrapper for Kubectl, and serves as package manager for Kubernetes.

Kubernetes Master and Node components

Kubernetes provides to run container-based primitives as Pods, Services and Ingress in the cluster nodes. It also provides lifecycle clouster functions as self-healing, scaling, updates and termination of workloads. In order to realize the features, many services in Kubernetes Master and Kubernetes Nodes have to cooperate with each other.

There are following Kubernetes Master components and their functions:

1. API Server - server Kubernetes API, providing access to business logic implemented as plugable components. server processes and validates REST requests, and updates the corresponding Kubernetes objects in etcd.
2. Etcd - distributed cluster state storage, which provides reliable configuration data storage, which can notify other components on updates (watch).
3. Controller Manager - application and composition manager, providing functions as self-healing, scaling, application lifecycle management, service discovery, garbage collection, routing, and service binding and provisioning.
4. Scheduler - watches for uncheduled pods and schedules to available nodes in the cluster. Scheduler takes into account availability of the requested resources, affinity and anti-affinity specifications, and quality of service requirements.

To provide very high-availability and resiliency agains failure of the master node, Kubernetes allows to replicate Master Nodes (in case of cloud, multiple availability zones).

Kubernetes Nodes have following components and their corresponding functions:

1. Kubelet - main component responsible for Pods and Node APIs that drive the container execution layer. It ensures isolation of application container, but also isolation from execution hosts. This component communicates with Schedulers and DaemonSets, and decides on execution or termination of Pods. It will also ensure stability of the Nodes regarding Node CPU and RAM resources.
2. Container Runtime (Docker) - responsible for downloading Docker Container images and running containers.
3. Kube Proxy - abstraction that provides common access policy as load-balancing to pods. It creates a virtual IP and transparently proxies to the pods in a Kubernetes Service.

Kubernetes Workloads (Pods and Controllers)

Pod is most fundamental component of Kubernetes, and smallest unit that can be deployed in the cluster. It running one or multiple same Docker containers, and encapsulates one application logic building up together containers, volumes, networking and other runtime settings.

Pod Controllers create and manage multiple Pods, handling replication, updates and ensuring self-healing capabilities such as rescheduling pods on other nodes in case of node failures. The most important controllers are:

1. Replica Set - ensures that a specified number of pods are running cluster-wide.
2. Deployment - is used to declaratively create, update and manipulate Replica Sets.
3. Stateful Set - unline stateless controllers as Replica Set and Deployment, Stateful Set provides guarantees about the ordering and uniqueness of the Pods, allowing to provide stable persistent storage and networking.
4. Job/CronJob - create one-time or on-schedule set of pods and controller will report success only if all the pods it tracks successfully complete, and in case of deletion to clean-up Pods.

Authentication, RBAC and Multitenancy

In order to control access to the kubernetes cluster, three phases of request pre-processing are performed: authentication, authorization (RBAC) and admission control (mutating or validating).

In order to authenticate HTTP request, modules as Client Certificates, Password, Plain Tokens, Bootstrap Token or Service Account Tokens are examined. One of the most common authentication methods are certificates, which can be obtained using cloud-provided tool e.g. *openstack coe cluster config*.

The authorization phase checks the request against existing policy that declares if the user has permissions to complete the requested action. RBAC (Role Based Access Control) autorization defines 3 top-level primitives: Namespaces, Roles and RoleBindings:

1. Namespace - Namespaces are a way to divide cluster resources between multiple users, allowing multi-tenancy. They are used to make some pods unique to some namespace and namespace users. Resource Quota can be used to allocate resources to the namespaces.
2. Role/ClusterRole - Roles are used to grant access to resources (pods, secrets etc.) within a single namespace. ClusterRole are the ones which might allow to give permissions in all namespaces, including to administrate nodes.
3. RoleBinding/ClusterRoleBinding - binding grants the defined permissions to a list of subjects such as users or service accounts.

Admission Control and Admission Webhooks

Admission control is the last step of providing an access to the cluster (after Authentication and Authorization). Admission Control modify or reject requests, deciding on object being created, deleted, updated or connected. It controls admission in two phases. It runs multiple admission controllers in parallel, and if any fails it rejects the request. First phase are Mutating Controllers which can modify the request accordingly, and in second phase Validation Controllers check if request can be passed. Example of Admission Controllers are AlwaysImagePull (ensures that image is always pulled in the Docker) and NamespaceLifecycle (ensures that when namespace is deleted, all resources are cleaned up and no other request is accepted for this namespace).

Special type of Admission Controllers are Dynamic Admission Controllers as Webhooks. These are user-provided Admission Controllers which intercept requests and processes them (mutating/modifying the resource before its creation, or validates the request according to custom requirement).

3.3 Kubernetes Controllers - Operator Pattern

Operator Pattern is a concept introduced in Kubernetes 1.7, which allows to write application-specific custom controllers (e.g. Spark Operator [22]). Due to the fact that controllers have implicit access to Kubernetes API, the custom controller can customize pods/services, scale application, provide failure tollerance by monitoring resources, call endpoints of the running applications, and define rules for application operation [23].

3.4 Scaling the cluster, Cluster Autoscaler and Pod Autoscaler

Kubernetes applications can request amount of Pods to run in the cluster, and amount of CPU and RAM to be allocated to pods. Kubernetes Scheduler based on request tries to find a node to run the pod. If there is no node that has enough free capacity or does not satisfy node affinity, then the pod has to wait until some pods are terminated or a new node is added.

Cloud Platforms as Openstack, Google Cloud, AWS and Azure allow to resize the Kubernetes cluster triggered by user request. Responsible service (e.g. Openstack Magnum) will take care of lunching, allocation or deletion of nodes. Kubelet will register/deregister its Node with API server to make it available/unavailable for scheduling.

Autoscaling cluster resources (nodes) is an example of automated resource provisioning. Since Kubernetes version 1.6, cloud providers provided support for cluster resources autoscaling (Google Cloud, AWS, Azure). Cluster Autoscaler monitors the pods that cannot be scheduled and conditionally adds additional node to the cluster to satisfy the requirement. It also scales down the cluster if some nodes are idle for a defined period of time.

Openstack Magnum as of 2018 does not support Kubernetes cluster autoscaling, but underlying components Heat can based on Ceilometer metrics build an auto scaling environments.

Horizontal Pod Autoscaler is a component of Kubernetes which allows to scale the application on existing cloud resources (nodes) according to the needs. It automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization or on custom application-provided metrics.

Vertical Pod Autoscalers is an Alpha Kubernetes 1.10 component which allows to scale the containers in a pod according to past usage (rescheduling with scaled RAM and CPU). Vertical Pod Autoscaler allows to specify which pods should be vertically autoscaled as well as if and how the recommendations for pod resources are applied.

3.5 Multi-cloud cluster federation

Kubernetes allows to federate multiple kubernetes clusters into one. It allows to run in high availability mode having two clusters in two availablity zones, and application portability between cloud providers and on-premises. The main Kubernetes primitives to be considered in federation are location affinity, cross-cluster scheduling, service discovery and application migration.

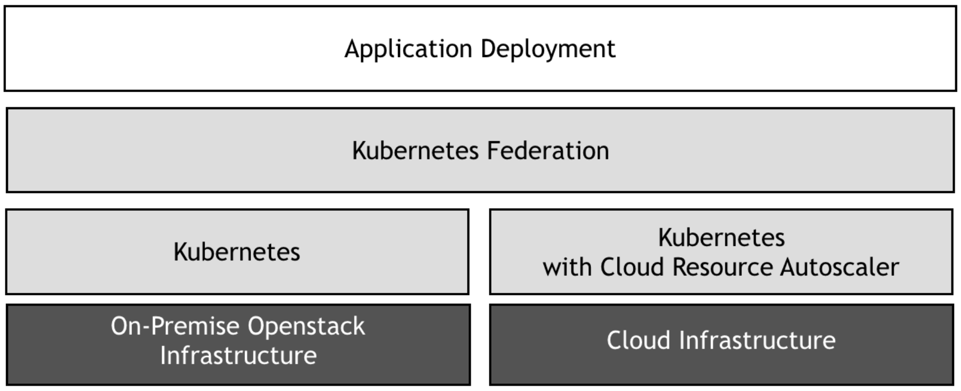

Application Deployment
Kubernetes Federation
Kubernetes
Kubernetes with Cloud Resource Autoscaler
On-Premise Openstack Infrastructure
Cloud Infrastructure

Figure 7: Kubernetes in a federation example.

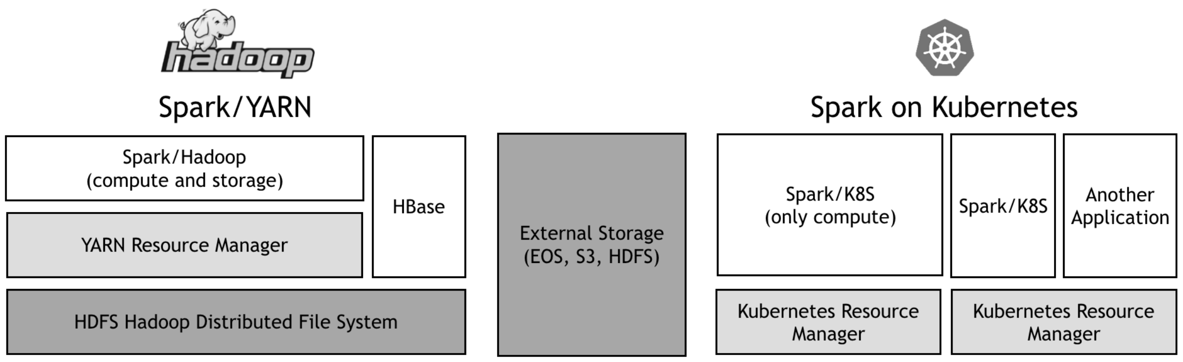
Kubernetes Federation allows for example to run main workloads on-premises, but scale-out with increased requirements in the cloud using autoscaling. Example of such architecture is presented on Figure 7.

4 Spark on Kubernetes Operator

This section gives an overview over components required to provide cloud-native Apache Spark deployment over Kubernetes. This also constitutes the implementation part of master thesis. Implementation and contribution to Kubernetes Operator for Spark and other related components was essential for the project.

4.1 Spark cluster operational features of Kubernetes compared to YARN

In this section, I will follow top-down approach of describing the Spark on Kubernetes and it's dedicated controller, Spark Operator.



hadoop
Spark/YARN
Spark/Hadoop (compute and storage)
HBase
YARN Resource Manager
HDFS Hadoop Distributed File System
External Storage (EOS, S3, HDFS)
Spark on Kubernetes
Spark/K8S (only compute)
Spark/K8S
Another Application
Kubernetes Resource Manager
Kubernetes Resource Manager

Figure 8: Comparison of Spark/YARN and Spark/K8S for data stored in external storage.

In a typical data warehouse and data analytics architecture, data-intensive computation is performed using Apache Spark. Spark is a distributed and parallel processing framework that requires some data management system to get the data from - typically HDFS with data being local to computation - and some resource scheduler to allocate resources as CPU and RAM available in the cluster of machines [4]. It is also possible in such architecture to stream data to processing in Apache Spark from external storage using dedicated connectors e.g. EOS over XRootD protocol, S3 over S3A protocol, HDFS over HDFS protocol and Kafka using Kafka subscription. Such an architecture is presented on Figure 8.

In Spark on Kubernetes architecture, compute cluster and storage clusters are separated by design. The sole role of Spark on Kubernetes is to stream required data from data storage, perform computations, and store over network protocol required state or output to external storage systems.

Spark on Kubernetes architecture has thus several properties which are lacking in traditional Spark/YARN (HDFS) clusters [9]:

1. Isolation of user environments realized by containers.
2. Allows users to provision and maintain isolated Spark cluster with minimal operational effort.
3. Logging, Monitoring and Storage are no longer part of the monolitic architecture as Spark/YARN and HDFS. Each of these are externaly managed services natively integrated with Kubernetes, usually offering certain SLA and durability.
4. Portability and reproducibility. It is easy to reproduce an exact same Spark environment on completely different infrastructure.
5. Compute cluster as sole service deployed with containers is easy to maintain, operate and control.
6. Separation from storage and native integration with public/private cloud providers allows resource provisioning elasticity and possibly infinite and cost-efficient provisioning capacity.
7. On-Premise YARN and Kubernetes have no build-in configuration and authentication methods. Openstack Cloud abstracts and provides configurations and authentication for provisioned Kubernetes clusters.

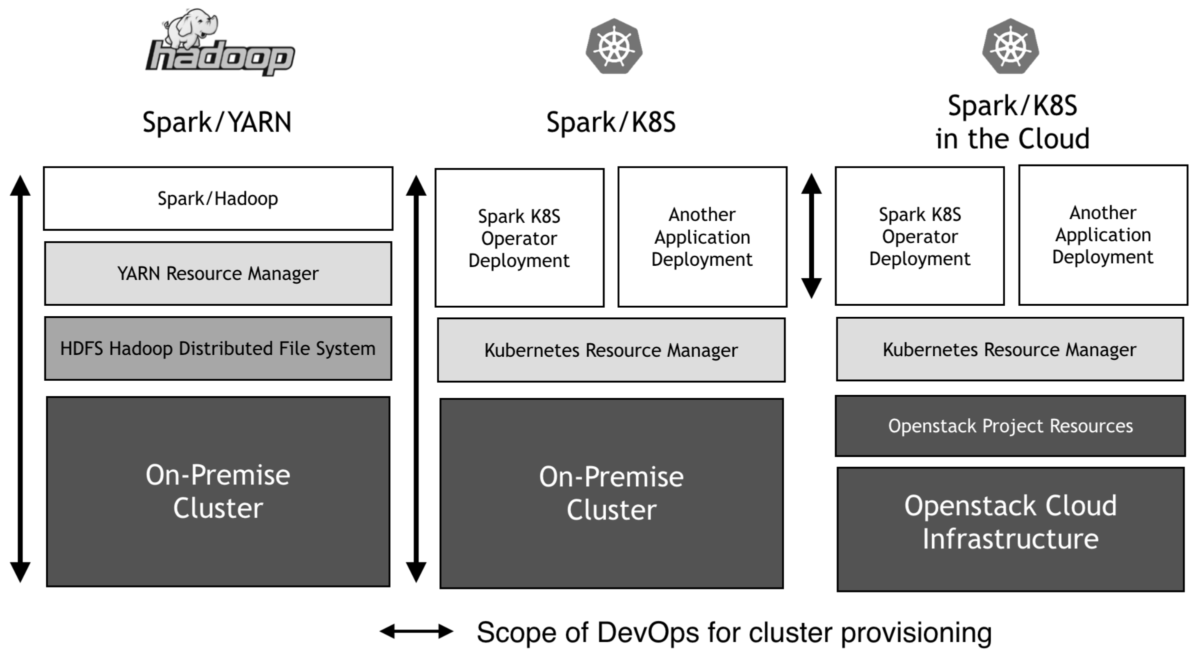

hadoop
Spark/YARN
Spark/Hadoop
YARN Resource Manager
HDFS Hadoop Distributed File System
On-Premise Cluster
Spark/K8S
Spark K8S Operator Deployment
Another Application Deployment
Kubernetes Resource Manager
On-Premise Cluster
Spark/K8S in the Cloud
Spark K8S Operator Deployment
Another Application Deployment
Kubernetes Resource Manager
Openstack Project Resources
Openstack Cloud Infrastructure
Scope of DevOps for cluster provisioning

Figure 9: Top-Level architecture comparison of Spark resource schedulers as on-premise YARN, on-premise Kubernetes and cloud-managed Kubernetes, according to DevOps and deployment effort.

Spark on Kubernetes can be deployed on-premise or as cloud-managed service over cloud resources. Example of cloud-managed Spark on Kubernetes cluster is deployment over Openstack private cloud. Kubernetes is being deployed using an *openstack coe create cluster* command, creating in minutes in an automated way a cluster using Openstack Magnum component. This allows to abstract from administrators DevOps effort of provisioning, running and maintaining clusters, as this is a case of Spark/YARN and Spark/K8S with on-premise. Comparison of architectures regarding DevOps scopes is shown on Figure 9.

4.2 Spark Kubernetes Operator as controller for Spark Applications

To control and manage Spark Applications running Kubernetes cluster, Spark Kubernetes Operator is used. In the basic concept, Spark Kubernetes Operator has the following roles in the cluster [22]:

1. Enables declarative application specification in YAML format and management of applications via dedicated API call (command line tool). Allows to customize driver and executors in a declarative way.
2. Deploys Spark Driver on random or selected node in the cluster (which later deploys on random/selected nodes executors) on behalf of the user via API call (command line tool).
3. Monitors driver and executors for specific application in granular way.
4. Resiliency and superivision of Spark Applications. Automated application restart with a configurable restart policy, automated retries of failed submissions with optional linear back-off, automated application re-submission for updated SparkApliation definition, allows to run applications on schedule.

Spark Operator has been developed (mainly by Google, with effort from CERN, Microsoft and RedHat) as an open-source project - reaching Alpha Release in May 2018 [9].

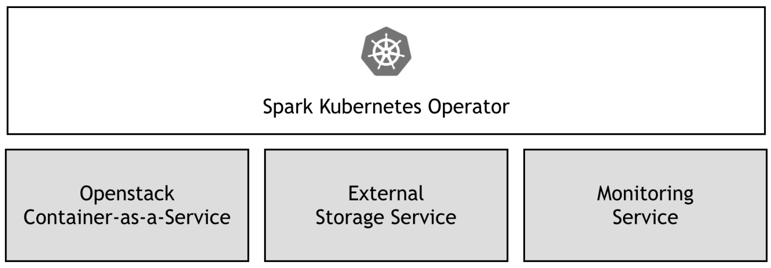
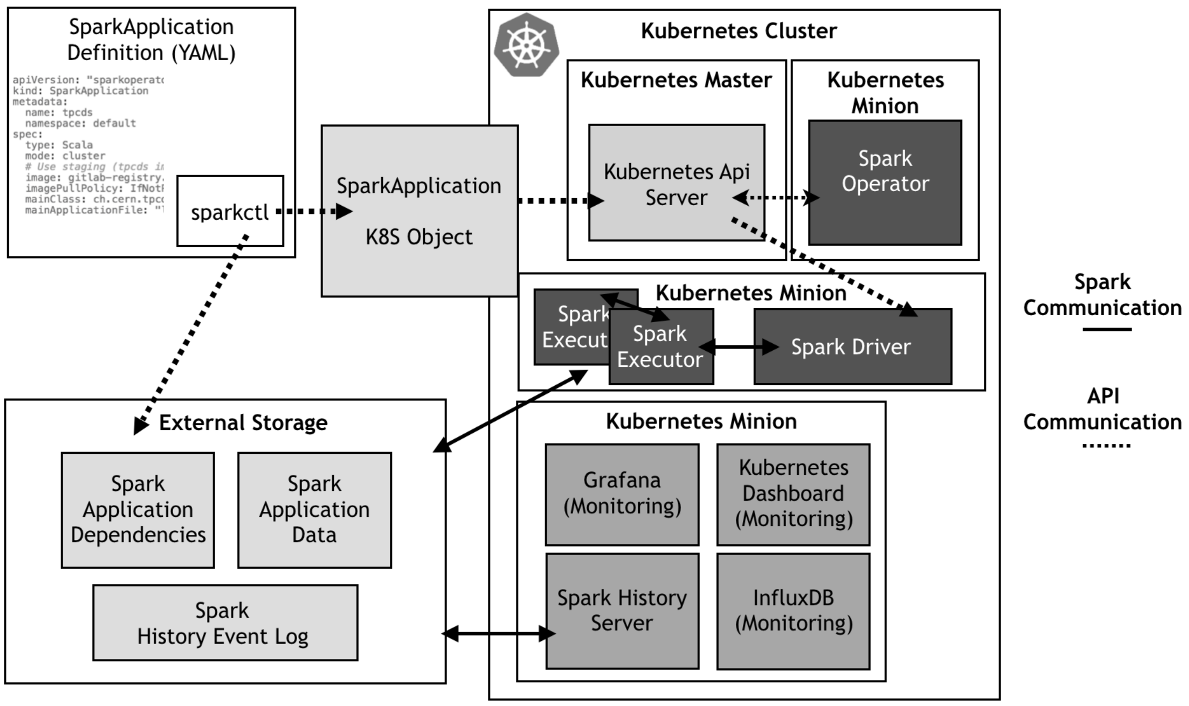


Figure 10: Spark-as-a-Service design based on Spark Kubernetes Operator

The design principle of Spark Kubernetes Operator is that it has to be easy to deploy, scale and maintain.

To realise it, Spark Operator and deployed Spark Applications are relying on other services (Figure 10) namely:

1. Container Service which provisions and maintains Kubernetes cluster
2. External Storage for preserving state, dependencies, logs and data
3. Monitoring Services running in/outside the cluster
4. Spark Operator which operates submitted spark applications



```
apiVersion: "sparkoperat
kind: SparkApplication
metadata:
  name: tpcds
  namespace: default
spec:
  type: Scala
  mode: cluster
  # Use staging (tpcds in
  image: gitlab-registry.
  imagePullPolicy: IfNotP
  mainClass: ch.cern.tpc
  mainApplicationFile: "
```

Figure 11: Spark Kubernetes Operator workflow and architecture. Interoperability of Kubernetes, Spark, Monitoring, Logging and External Storage.

Spark on Kubernetes Operator has multiple components which build workflow of execution as presented on Figure 10. The application definition written as text file in YAML format is interpreted using *sparkctl* command. Tool submits required dependecies to external storage and sends as HTTP request Custom Resource Definition representing Spark Application to Kubernetes API Server.

Kubernetes API Server, upon receiving the request, notifies through watch event the Spark Operator (specifically Custom Resource Definition Controller running in Spark Operator Pod) about the request to create Spark Application according to specification defined in YAML file. Executors and Driver then stream the required data, dependencies, update state and send logs to/from external storage as required by application logic.

Due to the fact that dependecies are in high-availability external storage, and Kubernetes is a self-healing system monitoring Pods running in cluster, cluster is resilient against node failures and allows fast recovery.

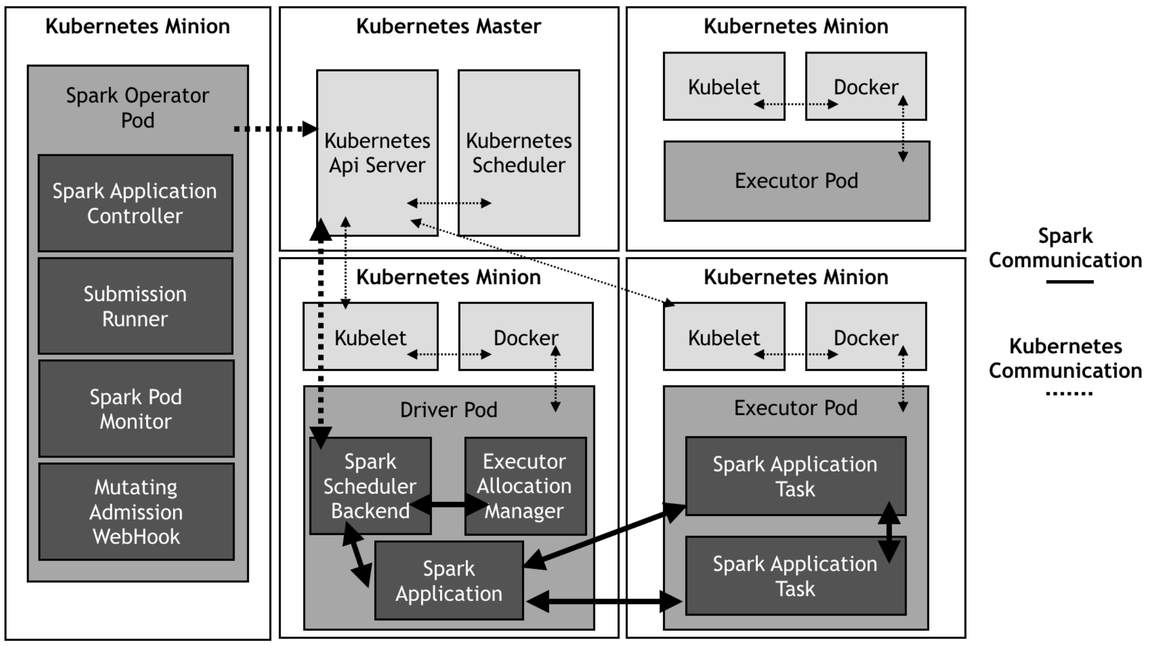


Figure 12: Spark Operator architecture

Spark on Kubernetes is build from components as Spark Operator, Spark Driver and Spark Executor. Each of these components interact with Kubernetes services as API Server, Scheduler (Kubernetes Master Node), and Kubelet (Kubernetes Minions). The whole architectrure overview is described in Figure 12.

Inside Spark Operator, Spark Application Controller (Custom Resource Definition Controller) listens to the watch events for new requests to create spark application. It then uses Submission Runner to create Spark Driver. Process inside Spark Operator called Spark Pod Monitor constantly watches the Spark pods (drivers and executors) and reports updates of the pods status to the controller. Controller in turn updates the status of the application, which allows to achieve resiliency (handles application restart policies, resubmissions). Component called Mutating Admission Webhook upon receiving request to create driver/executor can customize the resource on-demand according to user requirements e.g. mounting custom volume or applying pod affinity.

Inside deployed Spark Driver, there are processes responsible for scheduling and allocation of executors to available nodes (Executor Allocation Manager and Spark Scheduler Backend). Spark Application is a Custom Kubernetes Controller, which creates Docker containers (executors) in response to requests made by Spark Scheduler. Spark than can allocate tasks to executors.

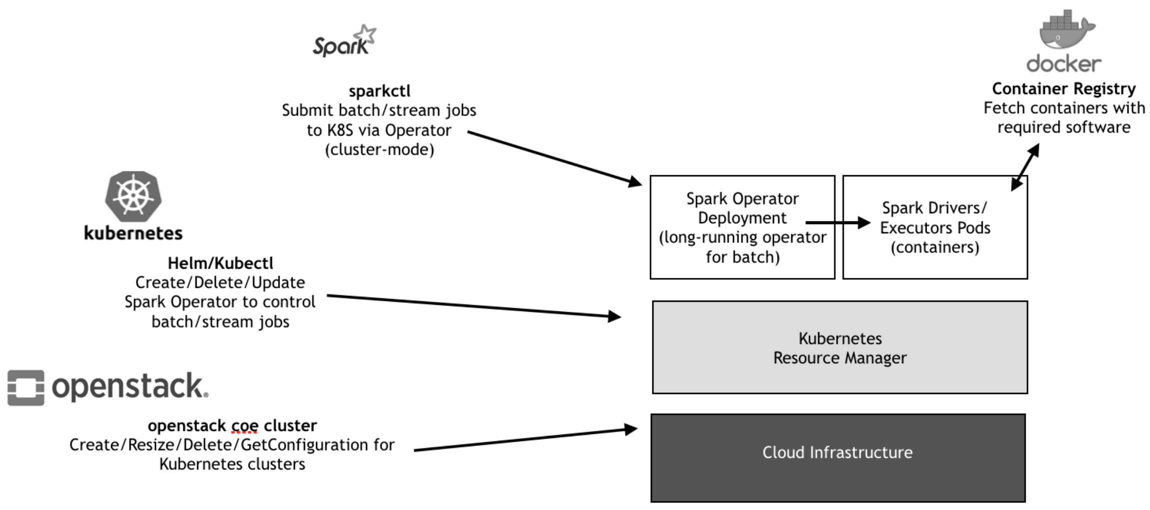

Spark
sparkctl
Submit batch/stream jobs to K8S via Operator (cluster-mode)
docker
Container Registry
Fetch containers with required software
kubernetes
Helm/Kubectl
Create/Delete/Update Spark Operator to control batch/stream jobs
Spark Operator Deployment (long-running operator for batch)
Spark Drivers/ Executors Pods (containers)
Kubernetes Resource Manager
openstack.
openstack coe cluster
Create/Resize/Delete/GetConfiguration for Kubernetes clusters
Cloud Infrastructure

Figure 13: Client tools used to manage Spark/Kubernetes clusters over Openstack.

In the cloud-native architecture of Spark on Kubernetes, there is a layered structure of software components, which are maintained using dedicated command line tools (Figure 13:

1. Openstack Magnum (*openstack coe cluster*) - used to create, update, resize, delete and access the cluster.
2. Kubectl (*kubectl*) - basic tool for managing Kubernetes. Used to maintain, deploy and configuration the cluster and cluster applications.
3. Helm (*helm*) - Package manager for Kubernetes. Used to deploy and update Spark Operator in the Kubernetes cluster
4. Sparkctl (*sparkctl*) - used to interact with Spark Operator in order to submit and monitor applications defined in YAML files. Uses libraries from Kubectl to operate.

The main tools for cluster administrator are *openstack coe cluster*, *helm* and *kubectl*. Clustera administrator is responsible for managing cloud resources for Kubernetes clusters and operating *openstack coe cluster* to fix any operational issues related to cloud resources. Higher level management of Kubernetes cluster can be done using *kubectl*, which allows to maintain and debug the Kubernetes Resources running in the cluster. In order to deploy and update the Spark Operator, *helm* package manager is used, which allows to customize the Spark Operator deployment for the users. As all the Kubernetes Resources, Spark Operator is being defined as YAML file - example of such deployment can be found in Appendix A.

The submission of spark applications by users can be done using *sparkctl* command, which issues submission requests to Spark Operator. The command line tool allows to create, delete, log and monitor spark applications running in the cluster. Advanced debugging of applications in case of issues can be performed using *kubectl* command.

The Spark Applications are defined using SparkApplication API, defined by Spark Operator and translated to *spark-submit* command during the submission [24].

Listing 1: Defining API, Namespace and Name

```
apiVersion: "sparkoperator.k8s.io/v1alpha1"
kind: SparkApplication
metadata:
  name: tpcds
  namespace: default
```

In the SparkApplication YAML file, user is required to specify apiVersion he wants to use, namespace in which to run the application, and the name of the application (which will later be used in other to manage application with *sparkctl*, as in the Listing 1.

Listing 2: Basic definition of Spark Application Specification

```
spec:
  image: gr.cern.ch/db/ss-dr/spark:v2.4.0-hadoop2.7
  mainClass: ch.cern.tpcds.BenchmarkSparkSQL
  mainApplicationFile: /opt/spark/examples/jars/sse.jar
  arguments:
    - ...
  deps:
    files:
      - ...
    jars:
      - ...
  sparkConf:
    ...
  restartPolicy: Never
  driver:
    cores: 4
    coreLimit: "4096m"
    memory: "6000m"
    serviceAccount: spark
  executor:
    instances: 50
    cores: 4
    memory: "5000m"
```

User has to specify also the base Docker image containing Apache Spark distribution with other base software packages included. If there are any local dependencies to be staged (main application file, files, jars), they will be uploaded to external storage by *sparkctl*. The parameters regarding specification of the driver and executor containers, as well as driver failure handling (restart policy of Never, OnFailure, Always) can be specified as in the Listing 2. User can additionaly specify other parameters which are usually passed to *spark-submit* command.

Listing 3: Running Spark Application in cron

```
spec:
  schedule: "@every 5m"
  concurrencyPolicy: Allow
  template:
    mainApplicationFile: /opt/spark/examples/jars/sse.jar
    ...
```

In order to schedule application in cron, code as shown in Listing 3 can be used. Full example of Spark Application or Scheduled Spark Application definition YAML file can be found in Appendix B and C.

4.3 Persistent Storage concerns in cloud-native Spark

The cloud-native Apache Spark architecture consists of multiple microservices as Storage, Compute and Monitoring. Spark requires persistent storage for use-cases as spark history events, spark and kubernetes statistics, spark streaming checkpoints or software packages. This in turn requires selection of proper services for persistent storage. Additionally, due to the fact that storage is not local to compute cluster, there are concerns about streaming data over network which need to be analyzed.

There are different types of persistent storages for Spark on Kubernetes, which allow simultaneous and parallel writes from multiple pods. Such storage services are Kubernetes Network Volume Mounts, Spark Native Filesystems accesible from within Spark Executors and Drivers, Objectstorages accesible from within Spark Executors and Drivers and Log Storage integrated with Spark. Comparison is shown in the Table 1.

Table 1: Comparison of available Persistent Storages for Spark on Kubernetes for parallel writes from multiple pods

Persistent Storage Feature	**Network Volume Mounts**	**Spark Native Filesystems**	**Object Storage**	**Log Storage**
Examples	CephFS, AzureDisk, EFS, CVMFS	HDFS, EOS	GCS, S3	InfluxDB, Stackdriver, Prometheus
Use-case	Logs, Checkpoints, Software Packages	Data processing	Data processing, Logs, Checkpoints	Logs
Spark Transactional Writes	No	Yes	Possible	Yes
Eventual Consistency Writes	No	No	Yes	No

Network Volume Mounts are characterised by the fact that these volumes are network attached as mounts to the Docker containers, in each of the pods. This offers data persistence, and in case of CephFS, AzureDisk, Amazon EFS, also parallel writes from multiple pods [20]. However due to lack of transactional writes in Spark it is not recommended for use-cases as large-scale data processing. It thus might be good solution for storing Spark History Events, Spark Streaming Checkpoints and Read-only Software Packages.

Spark Native Filesystems, implemented as extension of Hadoop Filesystem and supporting atomic renames, offer transactional writes. Transactional writes are required in cases that require efficient, highly-parallel and fault-tolerant writes to the same directory - such as distributed data processing and streaming. Examples of such solutions are HDFS, EOS (Hadoop-XRootD Connector) and Kafka. Regarding storage elasticity, HDFS and Kafka offer limited elasticity, while EOS provides variable storage service levels based on changing needs, without need of re-balancing or other operational issues [5][25][13]. Distributed Filesystem are thus best for storage of data for large-scale data processing.

Spark supports integrations with Objectstorages, implemented as extensions of Hadoop Filesystem, building abstraction of filesystem on top of objectstorage. However, in the default configuration these cannot offer transactional writes through Spark. This means, that in presence of failures data written in parallel to some directory might not be in consistent state presented in the storage. However, additional Directory Commiters in Hadoop3.1 might allow to ensure some level of consistency [25][13]. Additionaly due to eventual consistency of write they might cause operational problems while using Spark Streaming Checkpointing in large scale. Due to these properties, Objectstorage might be cheap and efficient for data processing (specifically reading in large scale), logs and spark streaming checkpoints, however might cause operational problems.

Platforms as Spark and Kubernetes log information about usage of resources for the deployed resources, and provide information essential to optimize the workflows. These logs can be streamed to Log Storages as InfluxDB Database, Google Stackdriver Cloud Service and Prometheus Monitoring Service.

Due to the fact that data-processing in Spark/K8S happens over the data which has to be streamed to compute (instead of disk local computation as in case of HDFS), there are several factors which has to be taken into account:

1. If other services share the same network in the cluster of machines, data-intensive applications can cause network saturation in the routers and in the hypervisors.
2. While reading data over network, it is important to find a balance between number of requests to the storage and size of the request (ReadAhead parameters for selective access vs data scans)

4.4 Handling Out of Memory in Spark on Kubernetes

Spark is a computing framework which is designed in maximizing the resource utilization. Very often, due to nature of Java Virtual Machine, Resilient Distrubuted Dataset and Shuffling, Spark applications are memory intensive.

Kubernetes as resource scheduler will attempt to schedule as much Docker containers as it is allowed by their Quality Of Service and Resource Request specification. Kubernetes Minion (Node) Kubelet process constantly monitors Memory usage on the node. If the Kubelet is notified about node experiencing System Out Of Memory, it will use scoring mechanism to kill the containers with highest score and thus reclaim memory required for system operation on the node (reference - Configure Out Of Resource Handling [20]).

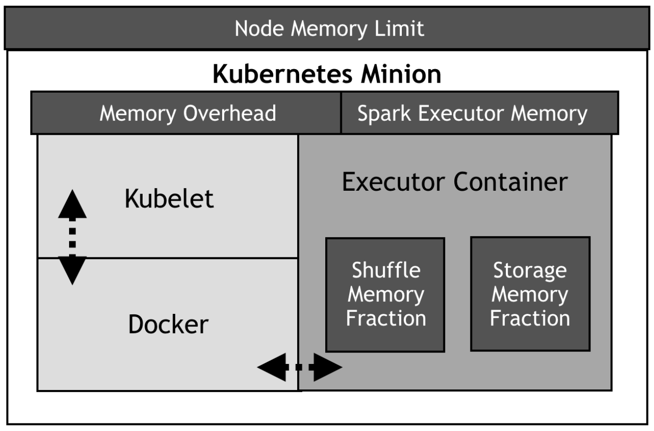


Figure 14: Kubernetes Minion and Spark Executor memory structure

In order to avoid Spark Tasks being killed with *OOMKilled* due to Node Out Of Memory or Kubelet observing MemoryPressure, proper parameter values for Memory Overhead

Fraction, Spark Executor Allocated Memory, Shuffle Memory Fraction and Storage Memory Fraction has to be set - as presented on Figure 14 representing memory structure of Spark on Kubernetes Executor.

Following actions can be taken to avoid Node Out Of Memory and Kubelet MemoryPressure, following the experiences of running Spark workloads on Kubernetes:

1. Spark on Kubernetes allows to set *spark.kubernetes.memoryOverheadFactor* parameter, which sets fraction of memory to Non-JVM Memory in Docker container. Non-JVM memory is usually allocated to off-heap memory allocations and system processes (e.g. Python uses off-heap and starts both Python and Java processes, Docker Deamon requires memory to maintain the container). In some cases Non-JVM tasks require more off-heap space and result in container being killed by Kubelet with *OOMKilled* error. This behaviour can be observed running TPC-DS Benchmark in larger scale for queries requiring extensive shuffling.
2. There needs to be balance in amount of memory allocated to shuffle data - *Shuffle Memory Fraction* - too small value will case performance drop of spill to disk, but too large will cause *OOMKilled* errors.
3. There needs to be balance between number of executors and the amount of Memory and CPU per executor (many small executors offer high-throughput from distribution but memory overhead vs few big executors offer high parallelism of many node CPUs but extensive garbage collection).
4. Too small amount of memory allocated to Spark Driver may cause performance drop in operations which require driver intervention.

5 Evaluation and analysis

This section is focused on evaluation of Spark on Kubernetes over Openstack. Firstly, comparison of different Spark resource schedulers will be presented in the context of required properties for physics data analysis. Next, the comparison of persistent storages use-cases, and handling memory for Spark on Kubernetes will be presented. The final part of the section will be focused on analysis of synthetic benchmark of state-of-the-art Spark/YARN with data at HDFS and Spark/K8S with data in EOS, and scalability tests performed on the data mining use-case of LHC data reduction.

The benchmarks and scalability test were performed on two types of infrastructures (resource managers), with the same version and setting of Apache Spark 2.3.0. One of the resource managers were on-premise deployment of YARN cluster. It is a shared, general purpose cluster available to researchers at CERN. Another resource scheduler used was deployment of Kubernetes over Openstack private cloud. Both infrastructures are within the datacenter of CERN in Mayrin, Switzerland.

5.1 Apache Spark with external storage: resource schedulers comparison

The main motivation for using architecture based on cloud-native Spark on Kubernetes, was to simplify resource provisioning, automate deployment, and minimize the operating burden of managing Spark Clusters.

Table 2: Comparison of different Apache Spark resource schedulers in the context of cluster management

Spark resource scheduler feature	**K8S/Cloud**	**Mesos**	**YARN**
Resource scheduling features	native scheduler implementation		
Solution/architecture complexity	easy	complex	moderate
Resource manager use-case	applications	data-center	Hadoop
Spawning/installing cluster	cloud-managed	complex	complex
Resizing cluster	elastic	complex	complex
Multi-cloud native support	yes	yes	no
Users/Developers community	strong	weak	strong
Cloud vendors	multiple	single	multiple
Reproducible Spark cluster	containers	containers	manual
Installing/managing Spark	containers	containers	manual
Spark deployment method	CRD Operator	DCOS	manual
Spark application definition	idiomatically	command	command
Build-in cron support	CRD Operator	external tool	external tool
Resubmission on driver-failure	CRD Operator	external tool	external tool
Type of user	ad-hoc	predictable	predictable

In order to understand the main differences between cloud-native Spark/K8S, Spark/Mesos and Spark/YARN, the comparison of the operational features of each resource schedulers for Spark is presented in the Table 2. The qualitative research is result of experience of running Kubernetes and YARN at CERN, and through experience and observations of other industry experts [26][27][28]. In all cases, it was assumed that the data required for computation is already external to the cluster (in storage system as S3, EOS or Kafka).

Firstly, all resource schedulers offer similar set of features, being integrated natively in Spark framework. The main difference between them, is their architecture and implementation details (which impact scalability and performance). Apache Mesos has been designed to be scalable and resilient enough to manage resources for the entire data center. Mesos+Marathon has very complex to understand and manage 2 tier architecture, which made it very scalable [26]. YARN on the other hand, has simpler architecture and has been designed to schedule resources for Hadoop [27]. Kubernetes on the other hand has very simple architecture, which has been designed for handling microservices workloads and containers [28].

Regarding spawning and maintaining of the cluster, both Apache Mesos and YARN are very complex to install and resize (as they build up an entire data-center cloud). In case of Kubernetes, most cloud providers (e.g. Openstack Magnum) allow to spawn on-demand customizable cluster on elastic cloud resources. Additionaly, Mesos and Kubernetes allow natively to be deployed in multi-cloud and multi-region architectures.

The important factor for decision making in choosing Big Data products is large developer community, which ensures that project will be constantly developed and supported in the future. Kubernetes is the fastest growing and largest in terms of community Open Source project, mostly with focus on web services. In the domain of Big Data, YARN has very strong community of users, and became mature product. Mesos however, unlike Kubernetes and YARN, is single-vendor product, and therefore has smaller community of both developers and users.

Regarding installing applications on the cluster, both Kubernetes and Mesos share similar model based on containers, which allows to reproduce the environment in the other cluster. These frameworks deploy Apache Spark using Kubernetes Operator deployment managed by Helm (Kubernetes) or using Mesosphere DC/OS which operates Spark (Mesos). In case of YARN, each node on the cluster has to have proper configuration installed, using e.g. network attached storages or package managers as Puppet.

Spark applications on Kubernetes are being submitted via YAML files, which allow to idiomatically specify the job configuration, while in YARN and Mesos jobs are being submitted directly using command like tool.

Regarding application cronjobs and restarts on driver failures, only Kubernetes (using Spark Operator) has these build-in features by default. Other frameworks as Mesos and YARN need external systems e.g. Nomad.

Summarizing, Kubernetes appears to be more suitable operationaly for ad-hoc users, requiring elastic resource provisioning or dedicated on-demand clusters which they could easily manage themselves benefiting from microservices (being users of multiple services as compute, storage and monitoring). On the other hand, YARN and Mesos appears to be better-off in multi-tenant and externally managed environment, with users running interactive analysis over smaller datasets, or sustained, production workloads with predictable requirements.

5.2 Synthetic TPC-DS benchmark for Spark/YARN and Spark/K8S

Decision support benchmarks as TPC-DS provide set of repeatable, controlled and highly comparable tests which evaluate upward boundries of systems in aspects as CPU, Memory and I/O utilization, and ability of systems to compute and examine large volumes of data [29]. Benchmark consists of over 100 queries in 4 categories: Reporting (periodical queries to answer predefined business questions), Ad-hoc (queries are not known beforehand, usually CPU or I/O intensive), Iterative OLAP (include a sequence of simple and complex statements that lead from one to the other), and data mining queries (these queries analyse large sets of data using joins and aggregations, producing large result sets) [10]. Benchmark has been performed in order to demonstrate differences between the architectures in terms of networking and temporary storage (I/O intensive queries), processing power and virtualization overhead (CPU intensive queries) and overall performance of Spark and shuffling (Iterative, Reporting and Data Mining Queries).

Table 3: Executor configurations used in benchmarking test for different Apache Spark cluster infrastructures (resource schedulers)

Resource Manager	**Job Executor Containers**	**Cluster Nodes**
YARN / On-Premise	8 x 2 CPU, 7GB RAM	42 Physical Machines (1200 CPU, 3.5 TB of RAM)
K8S / Openstack	8 x 2 CPU, 7GB RAM	9 Virtual Machines (76 CPU, 270GB RAM)

TPCDS Benchmark has been performed for Apache Spark with two resource managers and infrastructures (YARN/On-Premise and Kubernetes/Cloud) in order to ensure comparable and efficient execution for different types of workloads - the configuration is shown in the Table 3. To ensure that each executor will be allocated to 1 server, there were 8 executors allocation to TPCSD Spark Job - Figure 3. In all cases, 8 iterations of the query have been performed, in batches of 2 executions in different times of the day (31.07.2018).

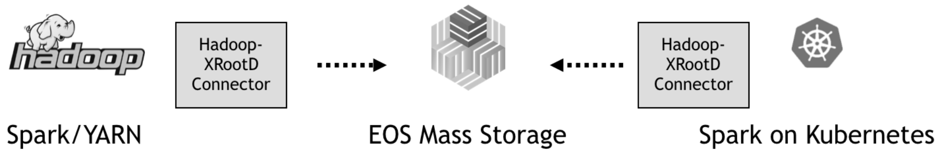


Figure 15: Overview of benchmarking test for different Apache Spark cluster infrastructures (resource schedulers)

In both cases the same configuration has been applied to Apache Spark 2.3 (Spark/K8S had included commits related to resource manager targeted at Spark 2.4.0), and dataset stored in EOS has been used - Figure 15. Query execution time has been measured, with median, min and max values being extracted. Additionaly, data access related metrics has been collected (total shuffle read/write, total input data read, shuffle fetch and write, memory garbage collection time, total cluster executor deserialize and compute CPU time).

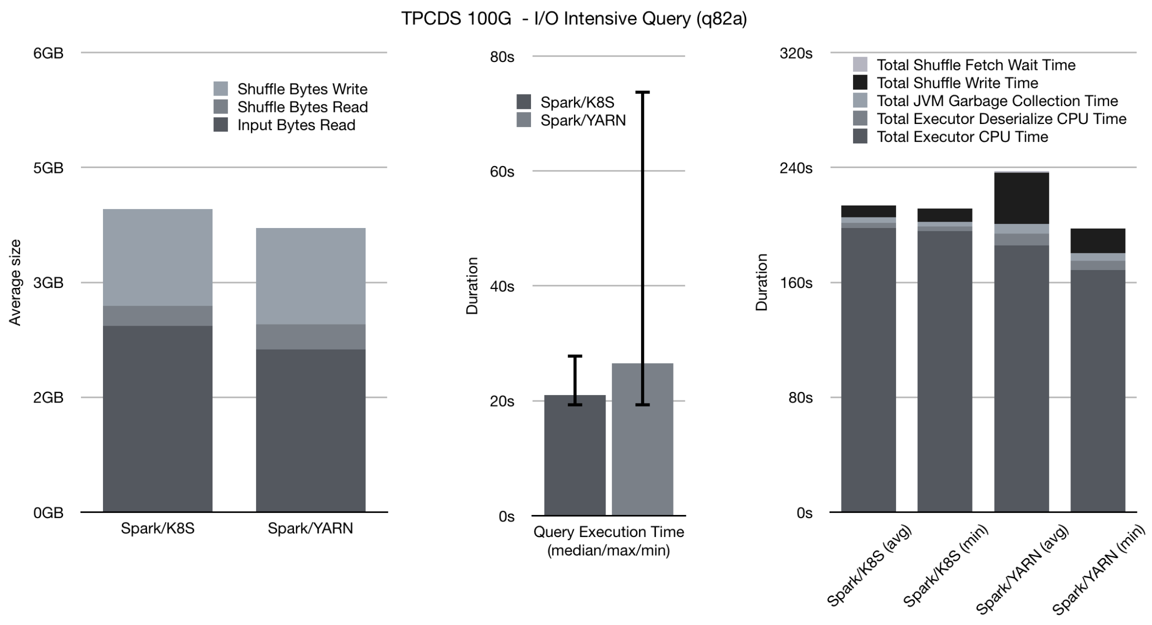


Figure 16: I/O Intensive Query. TPCDS Benchmark for Spark SQL 2.3 with 100GB (EOS-UAT) for YARN and Kubernetes resource schedulers. Results for 6 iterations.

Analysis for I/O Intensive Query based on results in Figure 16:

1. I/O Intensive Query on Spark/Kubernetes and Spark/YARN could reach similar I/O Performance (same minimal query duration)
2. Probability of lower I/O Performance is higher on YARN due to noisy neighbors in general purpose cluster (longer executor cpu, shuffle fetch wait and write times).
3. Spark/YARN can reach higher performance on Executor CPU Time, which indicated average 10% CPU loss in Spark/Kubernetes.

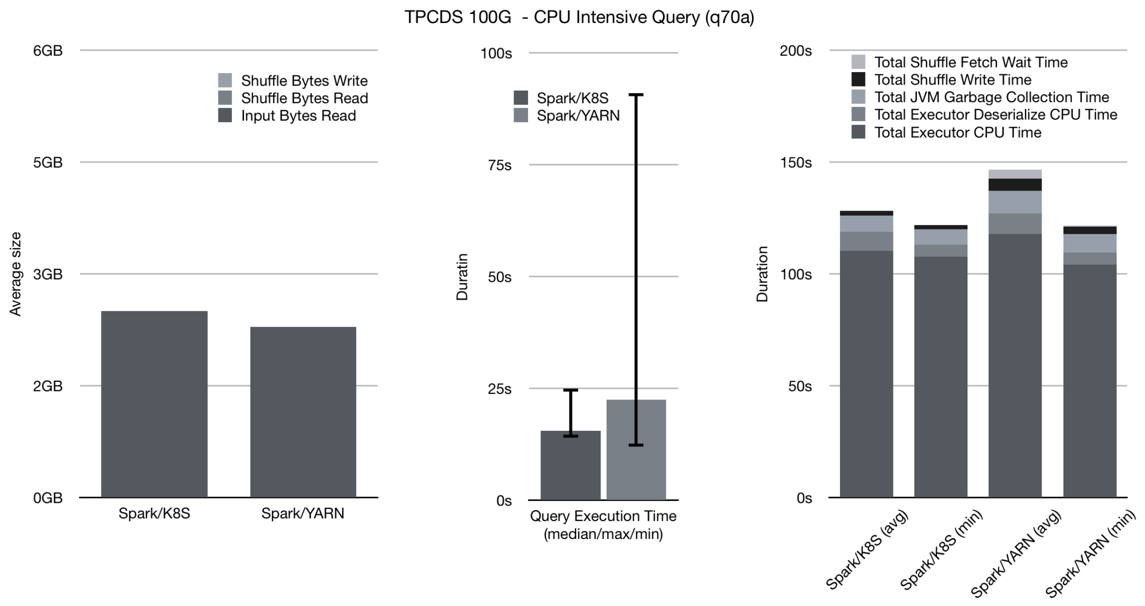


Figure 17: CPU Intensive Query. TPCDS Benchmark for Spark SQL 2.3 with 100GB (EOS-UAT) for YARN and Kubernetes resource schedulers. Results for 6 iterations.

Analysis for CPU Intensive Query based on results in Figure 17:

1. Spark/YARN can reach lower execution time for CPU Intensive Query than Spark/Kubernetes (lower minimal query duration)
2. Due to noisy neighbors, Spark/YARN in average performs CPU intensive queries with lower average performance and higher average execution time.
3. Spark/YARN can reach higher performance on Executor CPU Time, which indicated average <5% CPU loss in Spark/Kubernetes.

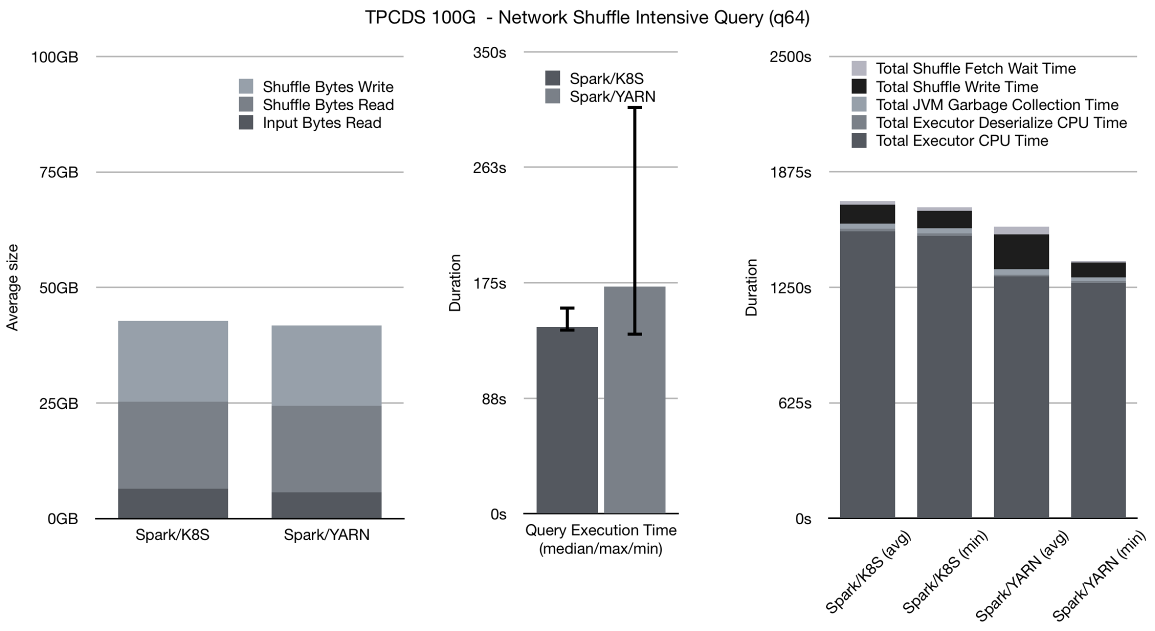


Figure 18: Shuffle Intensive Query. TPCDS Benchmark for Spark SQL 2.3 with 100GB (EOS-UAT) for YARN and Kubernetes resource schedulers. Results for 6 iterations.

Analysis for Network Shuffle Intensive Query based on results in Figure 18:

1. Spark/YARN can reach lower execution time for Shuffle Intensive Query than Spark/Kubernetes (lower minimal query duration)
2. Due to noisy neighbors, Spark/YARN in average performs CPU intensive queries with higher average execution time, and increased shuffle write time.
3. Spark/YARN can reach higher performance (10-15%) in average executor CPU time for computing SQL JOINS.

5.3 Scalability tests for large-scale LHC data reduction

The CMS collaboration with support of CERN IT-DB Group performed series of scalability tests on very large datasets, primarily on Spark/YARN due to large impact on network, affecting production services.

The data used for scalability test for data reduction is stored in external storage service called EOS. In order to allow reading data from EOS Storage in Spark, XRootD-Hadoop Connector has been used - Figure 15. The data used for the tests is 100TB of ROOT files, with each ROOT file being around 4GB, and all in total keeping information about millions of collision events.

Initial large scale scalability tests

Table 4: Executor configurations used in initial scalability test for data reduction for different Apache Spark cluster infrastructures (resource schedulers)

Resource Manager	Cluster executors	Executor memory	Executor vcores	Read Ahead
YARN / On-Premise	407	7GB	2	N.A (32MB buffer)
K8S / Openstack	1000	7GB	2	N.A (32MB buffer)

The initial test was performed using both YARN and Kubernetes resource schedulers, with the same configuration for Apache Spark (ref. Table 4). Number of executors has been set to maximal available in the particular computing cluster. The resources allocated for spark application were fixed by dedicated queue. ReadAhead parameter has not been used, and data were fetched in batches of 32MB per request.

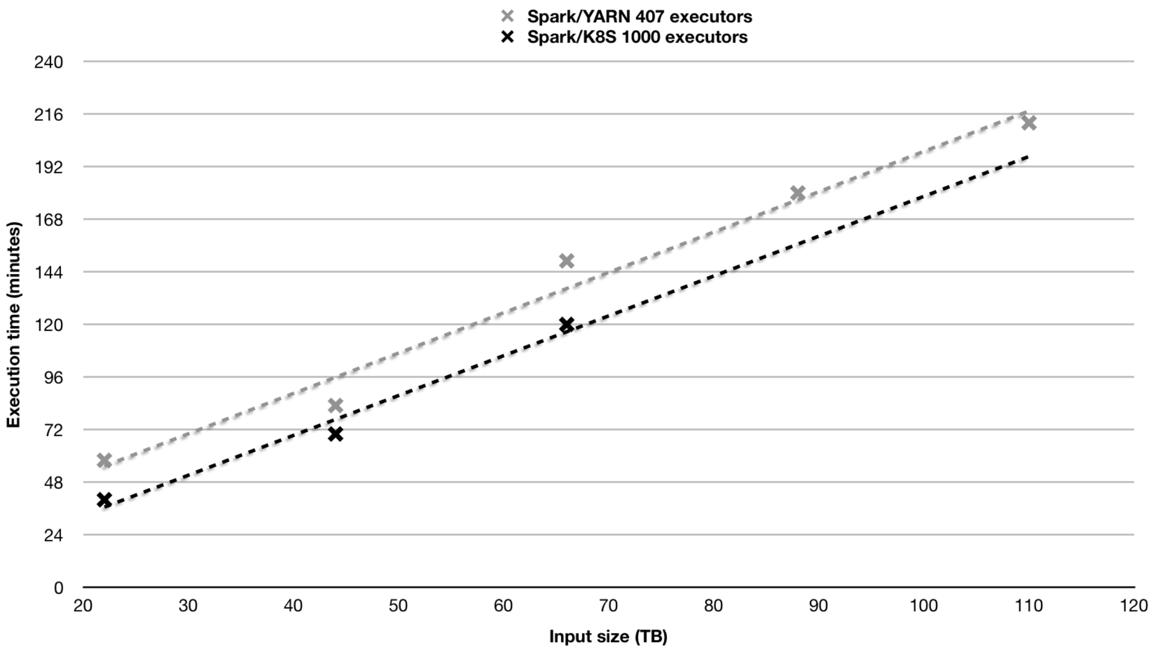


Figure 19: Input workload size scaling for Kubernetes and YARN resource managers.

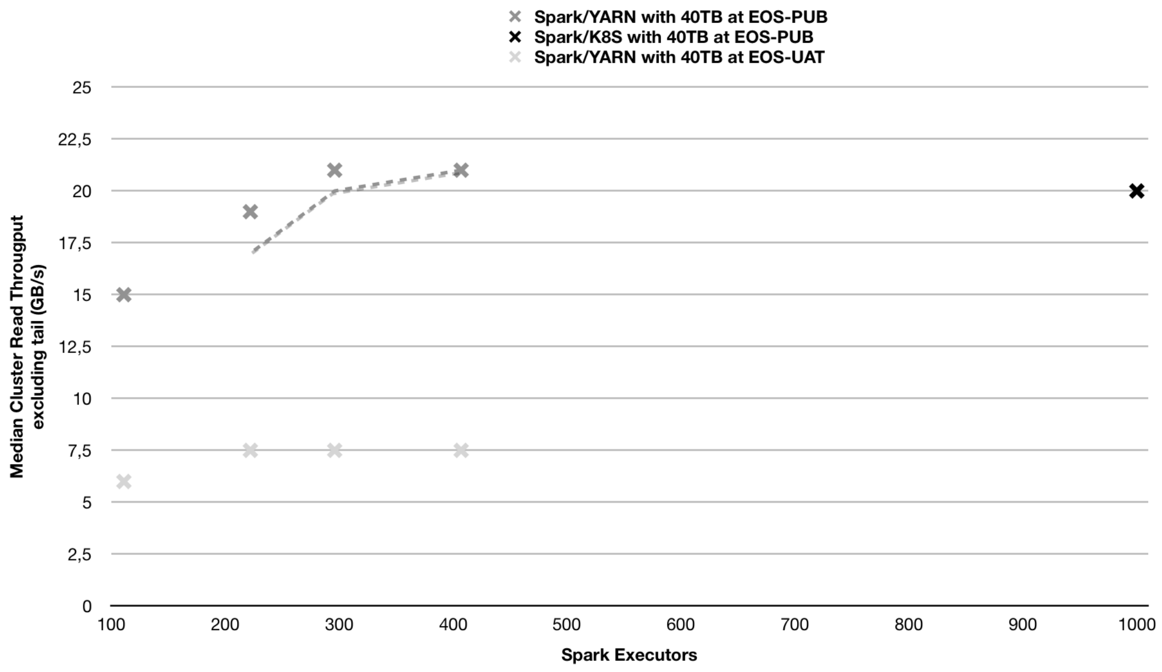


Figure 20: Cluster read throughput with scaling number of executors for Kubernetes and YARN resource managers. Data reduction of 40TB stored at EOS-UAT and EOS-PUBLIC instances.

In the first test, the scaling of workload size has been characterized in order to understand if and how system scales with increasing workload size, and what are possible bottlenecks. Figure 19 shows an execution time in seconds versus size of the dataset being reduced. Both tests were performed with files being streamed directly from EOS-Public Storage Instance. It was found, that both K8S and YARN scale linearly with increasing workload size, and at similar rate of 18 seconds per 10TB. Despite the fact Spark/K8S had nearly twice more executors, the trend is similar in both systems. This indicates possible bottleneck not being in amount of RAM nor CPU allocated to computation.

Median cluster read throughput were measured against number of executors used to perform data reduction with Apache Spark. Two storage instances has been tested, larger instance of EOS-PUBLIC, and smaller instance of EOS-UAT. The execution tail of struggler tasks were excluded from the measurements.

Figure 20 shows that scaling number of executors on YARN saturated the cluster read throughput at 250 executors in case of EOS-UAT storage instance with 7.5GB/s, and at 300 executors in case of EOS-PUBLIC storage instance at 21GB/s. It was also confirmed performing a test with Kubernetes (K8S) at 1000 executors with data being stored at EOS-PUBLIC, at 20GB/s. This indicates two possible bottlenecks: the network becomes saturated, or storage service cannot handle more reads per second. While network in cloud environment is difficult to scale, on-premise storage like EOS scales horizontally with number of machines.

Optimizing ReadAhead parameter and Spark executor CPU and RAM ratio

In typical data reduction scenario for particle collisions data, the goal is to summarize portion of parameters and to reduce complex object representation to simpler one with only the data required for computation (reduce the dimensions). Thus, similarly to Spark SQL queries over Parquet format, schema of the ROOT file is being extracted, and only specific columns are extracted from the files in the dataset.

After careful diagnostic of the scaling results with network saturation, it was found that job was requesting from storage much more data than it should. It was expected, that from 10TB dataset of particle collision objects, only around 2TB of essential data will be extracted and analyzed. It was found, that due to very large default value for read-ahead for request of data from storage, more than 12TB data has been read (ref. Section 4.3 - storage concerns in cloud-native model).

Table 5: Comparison of different read ahead configurations. Data reduction Spark job parameters and results for 500GB input dataset stored at EOS-PUBLIC instance.

RunID	**RA32MB**	**RA1MB**	**RA100KB**
ReadAhead	32MB	1MB	100kB
Elapsed Time	12min	6.3min	2.7min
Data Input	500GB	500GB	500GB
Executor CPU Time	30min	31min	31min
JVM Garbage Collection Time	2.2min	2.2min	2.0min
Tasks	369	369	369
Total Cluster Read Throughput	5GBps	1.8GBps	130MBps
Total Cluster CPU Used/Requested	20/90 CP	40/90 CPU	50/90 CPU
Bytes Read	851GB	231GB	57GB
Total Cluster RAM Used/Requested	380/380 GB	380/380 GB	380/380 GB

Table 6: Comparison of different CPU/RAM configurations. Data reduction Spark job parameters and results for 500GB input dataset stored at EOS-PUBLIC instance, with 128kB read-ahead.

RunID	**RA-3CPU-12GB**	**RA-3CPU-18GB**	**RA-3CPU-21GB**
Elapsed Time	3.7min	3.3min	3.0min
Data Input	500GB	500GB	500GB
Executor CPU Time	30min	30min	32min
JVM Garbage Collection Time	1.8min	1.7min	1.8min
Tasks	369	369	369
Total Cluster Read Throughput	130MBps	130MBps	130MBps
Total Cluster CPU Used/Requested	50/90 CP	50/90 CPU	55/90 CPU
Bytes Read (with speculation)	61GB	53GB	60GB
Total Cluster RAM Used/Requested	380/380 GB	540/540 GB	630/630 GB

Running optimization runs over smaller dataset of 500GB - Table 5 and Table 6 - it was found that reducing read-ahead data request size from 32MB to 100kB, cluster read throughput from storage has been reduced by a factor of 50, while maintaining constant Number of Tasks and Executor CPU Time (actual time to compute the data). It was also found, that ReadAhead optimization also lowered RAM requirements for the job - more CPUs were used maintaining the same amount of RAM. Additionally, increasing amount of RAM per CPU (while maintaining amount of CPU) was found to improve execution time of the run and better utilize the resources removing RAM bottleneck.

Scaling data reduction horizontally for Spark/K8S

After the optimization of the workload on the small dataset, the scaling the cluster resources horizontally with number of executors was performed. Due to possible impact of data reduction job test, it was decided not to exceed 2GB/s cluster read throughput, and take smaller representative dataset of 20TB.

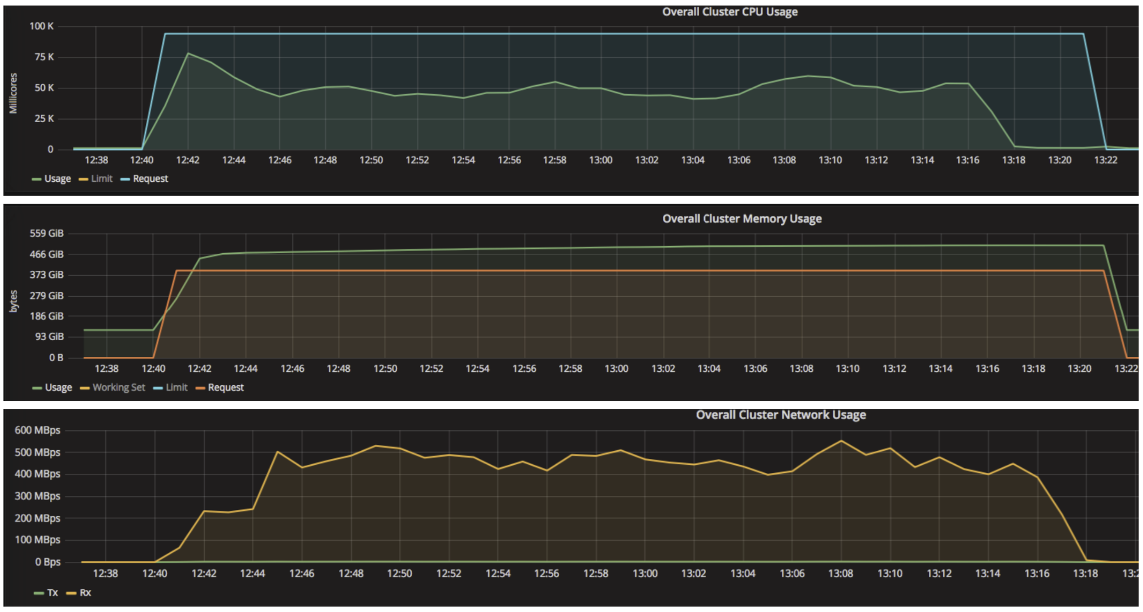


Figure 21: Job profile for physics data reduction of 20TB of ROOT data, taken from Graphana Monitoring Dashboard for Kubernetes cluster. Run with 30 executors, 3 CPU and 12GB RAM each running on Spark on Kubernetes.

It was found (ref. Figure 21) that each run characterizes by short time of high CPU utilization (until memory reaches its limits), long period of moderate CPU utilization, and a tail.

Table 7: Comparison of different executors number configurations. Data reduction Spark job parameters and results for 20TB input dataset stored at EOS-PUBLIC instance, with 128kB read-ahead.

Executors	**60**	**90**	**180**
Total Cluster CPU Used/Requested	80-120 / 120 CPU	110-180 / 180 CPU	205-290 / 360 CPU
Elapsed Time	28min	22min	13min
Executor CPU Time	19.1h	19.2h	19.6h
JVM Garbage Collection Time	55min	60min	60min
Tasks	12463	12523	12703
Total Cluster Read Throughput	600MBps	700MBps	1450MBps
Bytes Read (with speculation)	1488GB	1573GB	1490GB
Total Cluster RAM Used/Requested	800/800 GB	1160/1160 GB	2320/2320 GB

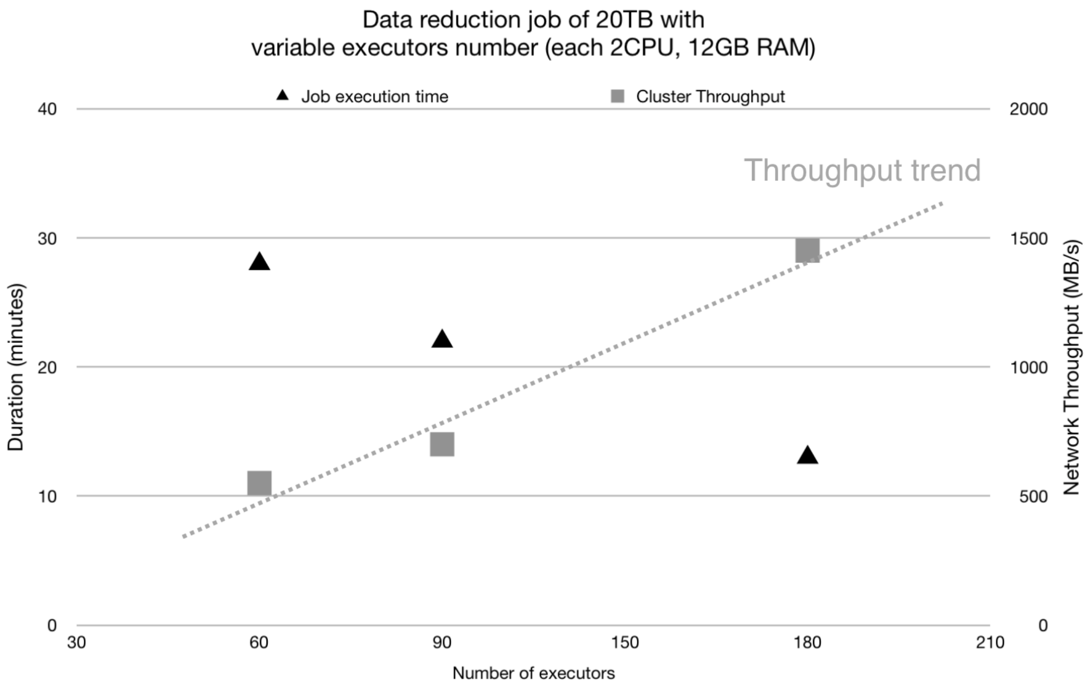


Figure 22: Executors scaling for physics data reduction of 20TB of ROOT data. Fixed executor parameters of 2 CPU and 12GB RAM running on Spark on Kubernetes. Job execution time and corresponding network traffic observed.

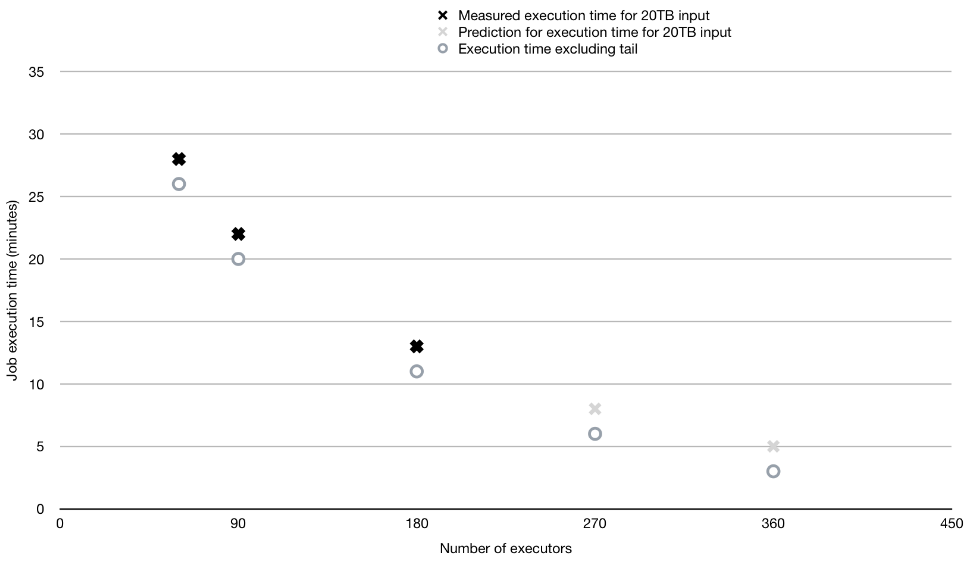


Figure 23: Executors scaling for physics data reduction of 20TB of ROOT data. Fixed executor parameters of 2 CPU and 12GB RAM running on Spark on Kubernetes. Job execution time with prediction for further scaling

The scalability tests achieved the goal of 2GBps read throughput from storage with 180 executors, performing the data reduction of 20TB dataset in 13 minutes (ref. Table 7). Figure 22 shows the relation of number of executors to job execution time. It was found that increasing number of executors static dataset size of 6000 ROOT files (20TB dataset, around 4GB each) decreases nearly linearly execution time. It also shows that network traffic (amount of data delivered to CPU per second plus processing speed) is scaling linearly with linearly increasing amount of executors. There is however an overhead associated with too small size of the dataset for increasing number of executors. Multiple runs for physics data reduction have shown that there is a trend to non-linearly scale further with number of executors keeping dataset size (number of files) stable - results are shown on Figure 23.

5.4 Discussion on use cases for Spark Kubernetes and Kubernetes Operator

Data locality is a concept introduced in Hadoop (HDFS) design, which is used to limit the network bottleneck while performing data-intensive computation by allocationg map tasks to datanodes containing required file blocks. This allows to avoid one of the major bottlenecks in data-intensive computing - cross-switch network traffic between storage and compute nodes during the map operations. Data shuffling (reduce phase) however uses memory or spills shuffle data to local disk storage on executors.

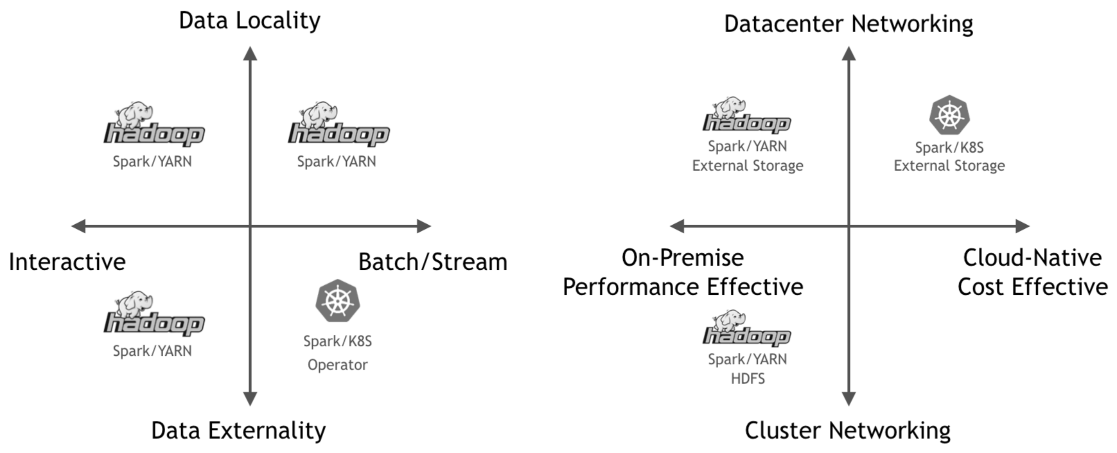


Figure 24: Use-case analysis for Spark on YARN/Kubernetes in terms of cost/performance effectiveness, networking, data locality and type of workload.

Spark/Kubernetes does not allow by design to achieve data locality for the map tasks, as it relies on separation of compute and storage clusters (Kubernetes cluster requires persistent storage to be realised over network for workloads running in container). Spark/Kubernetes is thus a good fit for batch/stream workloads which by-design require data to be stored externally to computing cluster. Additionaly, as of version of Spark 2.3, Spark on

Kubernetes only allows batch/stream analysis, as interactive client mode is not yet available. Interactive mode is thus production-ready supported currently in Spark/YARN for users at CERN.

Another important aspect to consider comparing Spark/YARN and Spark/Kubernetes for workloads from local/external storages is the type of networking between compute nodes and storage nodes. If there is good datacenter networking in-place with good QoS promises for services, both YARN and Kubernetes can well operate in case of data in external storages. The tests for data reduction of physics data (ref. Section 5.3) shown that both systems can comparably perform reading data from storage system within the datacenter network. However, if there is only cluster network available between the compute nodes (and storage nodes), and clusters are poorly connected, Spark/YARN with data in HDFS might be preferable as reading data from external storages might cause cross-switch network bottlenecks. If datacenter networking is available, and data is stored in externally managed service, Spark/Kubernetes might be thus a better fit in terms of cost-effectiveness, as this allows to use resource-sharing and resource-elastic cloud model in a cloud-native architecture, additionaly reducing operational effort of operating storage cluster. Spark/-YARN might however be better-off in cases where workloads are stable and predictable, using dedicated on-premise infrastructure. Figure 24 illustrates the aforementioned properties.

Table 8: Analysis of workload type use-cases for Spark on Kubernetes

Matching use-case	**Not suitable use-case**
CPU / Map-tasks Intensive	I/O intensive (except external storage data)
Shuffle intensive (many reduce phases, limited in size)	Large shuffle writes (assumes compute dedicated hardware)
Memory Intensive (in-memory processing, clusters with GPU)	

There are four types of workloads in Apache Spark, and are classified by the bottleneck they are bound by - CPU Intensive (data filtering, advanced mathematical operations), I/O Intensive (applications reading and writing large amounts of data, reading/writing data is largest fraction of task duration), Memory and Shuffle Intensive (in-memory processing of large amounts of data requiring full view on data, joining data tables, machine learning, deep learning). Based on experiences running workloads on Spark/Kubernetes (ref. Section 5.3 and Section 5.2), workloads types were assigned as matching or not suitable usacase for Kubernetes as resource manager for Apache Spark and presented in the Table 8.

The most intuitive Spark workload to be run on Kubernetes as resource scheduler are CPU and Memory Intensive workloads (data mining and machine learning workloads). As data is stored in external storage, data is being delivered over network directly to CPU for processing. In cloud-native architecture, users can provision virtual machines optimized for required profiles of jobs (CPU optimized clusters, GPU clusters, Memory-Intensive VMs), or use general purpose VMs and provision and downsize cluster as required. An example of CPU and Memory intensive workload is physics data reduction (ref. Section 5.3), and has been proven to scale efficiently on Kubernetes with scalability tests. Shuffle Intensive workload is also a matching use case, as in both Spark/YARN and Spark/Kubernetes shuffle data is transered over network, processed in memory, and spilled to disk if amount of memory is limited.

In the architecture that relies on data being transfered over network (reading/writing data in map phase, or shuffling data in reduce phase) I/O intensive queries might become bottleneck. In cases where network throughput or storage cluster read throughput is limited, resources as CPU and Memory might be underutilized. Thus such workloads with data stored in HDFS might not be a matching use-case, as these won't benefit from data-locality as in case of Spark/YARN and might be inefficient. Workloads with data in external storages which rely on data transfer over network, should have clusters that are well connected and isolated from the rest of the network. The example of workload which became I/O Intensive was physics data reduction with overtuned read-ahead parameter (ref. Section 5.3), and caused significant traffic in datacenter network impacting other production services. Another workload type, which might not be suitable on Spark on Kubernetes is one which requires large shuffle writes, which might exceed storage space allocated to virtual machines (which are compute optimized, as they are not used to run persistent storage system). In case of Spark/YARN running on top of HDFS, shuffle data is written to HDFS datanodes, which in principle are designed to store massive amount of data (as compute is bound to storage).

6 Conclusions

Large-scale data analysis use-cases as data mining and machine learning, relying on massive data-sets being stored in external storage systems as CERN EOS, impose operational challanges to provide reproducible and elastic compute infrastructure for Apache Spark, which cannot be currently addressed by existing Hadoop/YARN infrastructure at CERN. To address these challenges for data being read/written over network, cloud-native technologies as Kubernetes are investigated.

In a cloud-native architecture of Spark deployed on Openstack Kubernetes, user is offered externaly managed services as compute resource provisioning with containers-as-a-service (Kubernetes on Openstack), monitoring (InfluxDB, Grafana) and persistent storage service (EOS, S3). This limits operational effort to maintaining Spark Applications. Due to the fact that there is no storage cluster bound to computing as in case of Spark with HDFS, Spark on Kubernetes can scale number of cluster nodes dynamically using first-class support for Kubernetes in the cloud (Openstack Magnum). Kubernetes introduces Operator Pattern to provide custom controllers for applications. Spark Operator on Kubernetes allows automated deployment of Spark 2.3+ cluster with Docker containers on existing resource elastic Openstack Kubernetes cluster. It proven to deploy within seconds fully functional Spark cluster on top of existing Kubernetes cluster, with integrated services as monitoring (InfluxDB and Grafana), history server and logging (Ceph S3 Storage integration) , processing data stored in resource elastic external storages as Ceph S3 and EOS. Spark on Kubernetes imposes separation of compute and storage. This in turn requires that persistent storage has to be realised over network. Four types of systems have been evaluated (Network Volume Mounts, Spark Native Filesystems, Object Storage and Log Storages) according to requirements for persistent storage as data elasticity, Spark Transactional Writes and Eventual Consistency. Network Volume Mounts as CephFS might be best for use case as log and checkpoints storage as they mount distributed filesystem to the Docker containers, however they might be suboptimal for data processing. Spark Native Filesystems are best option for distributed and parallel data processing, as they offer Spark transactional writes, fast data access and no eventual consistency. Objectstorages could be universal for both logs, checkpoints and data processing as they offer good price-operability ratio, and can be extended with special write commiters. Based on experiences of running workloads on Kubernetes, it was concluded that in order to avoid Spark Tasks being killed due to Kubelet observing node memory pressure, there needs to be careful tuning of Spark parameters related to internal executor memory structures as executor memory overhead, executor memory, shuffle memory fraction and storage memory fraction. Openstack cloud hypervisors and Kubernetes containers introduce additional overheads in terms of RAM required to maintain additional processes and CPU virtualization. In return, they offer networking abstraction, authentication mechanisms and resource elasticity. TPCDS Benchmark has been performed for cloud-managed Spark/Kubernetes and on-premise Spark/YARN, in order to ensure both resource schedulers offer similar levels of performance for reading data, computation and shuffling. Results have shown that Spark/Kubernetes has overhead of around 5-10%, compared to bare-metal YARN. On the other hand, due to shared and not isolated environment on

YARN, Spark has been experiencing longer execution time for the queries due to longer shuffle fetch and write, and lower performance of shared cores. In order to validate that Spark on Kubernetes can scale for workloads requiring mining massive datasets stored in external storages, scalability tests for LHC physics data reduction facility have been performed. Dataset of 20TB stored in EOS Storage Service was used. Workflow is designed to reduce dimensionality of the particle collision events dataset of 20TB to dataset in size of 200GB (which could be even further reduced by selecting only relevant datasets or preserving data for histograms). It was shown, that optimizing read requests parameters and selecting proper ratio of CPU and RAM for executor, very high utilization of resources can be achieved (ensuring that workload is CPU/Memory bound and scales horizontally). It was also shown, that due to the fact that data is read over network, simple overtuning of workload parameter can make CPU/Memory bound workload being I/O Intensive, and cause significant network traffic between compute nodes and storage.

Concluding, the project successfully shown that Spark on Kubernetes can scale efficiently and comparably to Spark on YARN for workloads with datasets stored in external storage, while providing additional operational features. However, it imposes trade-off of operational features with additional networking bottleneck and possibly performance. Spark/Kubernetes is thus good in environments with good datanceter networking, and for use-cases which require by design to read data over network from storage services. The resource elasticity, flexibility and possible network bottleneck makes Spark on Kubernetes good for workloads as machine learning, data mining, in-memory processing, and not suitable for workloads stored already in HDFS, requiring massive I/O throughputs or large local persistent storage on executors. Finally, Spark on Kubernetes offers similar operational benefits as Spark on Mesos in terms of reproducable and isolated user environments with use of containers, unlike on-premise YARN. Kubernetes Operator for Apache Spark additionaly adds usability features known from Spark/YARN and Spark/Mesos.

7 Future work

The scope of this thesis has been limited to single workload in the cluster, with overview on main principles of the architecture. Future work in the context of Spark on Kubernetes should define and experiment with Multi-Tenancy in the Kubernetes cluster (including mixed workloads of different types). Additionally, constrains related to Spark Streaming, Interactive Spark Analysis and Monitoring of Spark Applications should be expanded and investigated. Performance and operational considerations of federated and multi-cloud Kubernetes clusters would be valuable extension of this work.

References

[1] O. Gutsche, M. Cremonesi, *et al.*, "Big Data in HEP: A comprehensive use case study," *Journal of Physics: Conference Series*, vol. 898, no. 7, 2017.

[2] K. Cacciatore, P. Czarkowski, *et al.*, "Exploring opportunities: containers and Openstack," 2015. https://www.openstack.org/assets/marketing/OpenStack-Containers-A4.pdf.

[3] "Computing at CERN." CERN CDS, 2012. http://cds.cern.ch/record/1997391.

[4] K. Noyes, "Five things you need to know about Hadoop v. Apache Spark," 2015. https://www.infoworld.com/article/3014440/big-data/five-things-you-need-to-know-about-hadoop-v-apache-spark.html.

[5] A. Peters and L. Janyst, "Exabyte scale storage at CERN," *Journal of Physics: Conference Series*, vol. 331, 2011.

[6] "Data reduction." Wikipedia, 2017. https://en.wikipedia.org/wiki/Data_reduction.

[7] L. Canali, "Performance analysis of a CPU-Intensive Workload in Apache Spark." CERN IT Blog, 2017. https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-cpu-intensive-workload-apache-spark.

[8] J. MSV, "Kubernetes becomes the first project to graduate from the cloud native computing foundation." https://www.forbes.com/sites/janakirammsv/2018/03/07/kubernetes-becomes-the-first-project-to-graduate-from-the-cloud-native-computing-foundation.

[9] A. Ramanathan and P. Bhatia, "Apache Spark 2.3 with Native Kubernetes Support," 2018.

[10] M. Poess, R. Nambiar, and D. Walrath, "Why you should run TPC-DS: A Workload Analysis," pp. 1138–1149, 2007.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[12] A. Wendt, "Cloud-based Hadoop deployments: benefits and considerations." Accenture Technology Labs, 2017. https://goo.gl/ZhNbVy.

[13] R. Xin, J. Rosen, *et al.*, "Top 5 reasons for choosing S3 over HDFS." Databricks Blog, 2017. https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html.

[14] R. Xin, "Benchmarking Big Data SQL Platforms in the cloud." Databricks Blog, 2017. https://databricks.com/blog/2017/07/12/benchmarking-big-data-sql-platforms-in-the-cloud.html.

[15] S. Watt, “Running Apache Spark in Kubernetes.” Proceedings of the Global Big Data Conference. http://www.globalbigdataconference.com/austin/big-data-bootcamp-55/speaker-details/stephen-watt-30816.html.

[16] R. Arora, “Why Kubernetes as a container orchestrator is a right choice for running spark clusters on cloud?,” 2018. http://www.globalbigdataconference.com/austin/big-data-bootcamp-55/speaker-details/stephen-watt-30816.html.

[17] M. Amaral, J. Polo, *et al.*, “Performance evaluation of microservices architectures using containers,” *IEEE 14th International Symposium on Network Computing and Applications*, 2015.

[18] V. Vasudevan, “Performance evaluation of resource allocation policies on the Kubernetes container management framework,” *Computer Science - Undergraduate Final Year Projects. City University Hong Kong*, 2016.

[19] B. Hindman, A. Konwinski, M. Zaharia, and A. Ghodsi, “Mesos: A Platform for Fine-grained Resource Sharing in the Data Center,” *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pp. 295–308, 2011.

[20] S. Perry, A. Chen, *et al.*, “Kubernetes reference documentation,” 2018. https://kubernetes.io/docs/reference.

[21] T. Fifield, A. Jaeger, *et al.*, “Openstack API documentation,” 2018. https://-docs.openstack.org/queens/api.

[22] A. Ramanathan and S. Suchter, “Apache Spark on Kubernetes Clusters.” Databricks Spark+AI Summit, 2018.

[23] D. Rosa, “Why Kubernetes Operators are game changer.”

[24] Y. Li, P. Mrowczynski, *et al.*, “Spark K8S Operator - Spark Application API,” 2018. https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/docs/api.md.

[25] E. Liang, S. Shankar, and B. Chambers, “Transactional writes to cloud storage on Databricks,” 2017. https://databricks.com/blog/2017/05/31/transactional-writes-cloud-storage.html.

[26] C. Wright, “Kubernetes vs Mesos + Marathon,” 2017. https://platform9.com/blog/kubernetes-vs-mesos-marathon.

[27] J. Scott, “A tale of two clusters: Mesos and YARN,” 2015. https://www.oreilly.com/ideas/a-tale-of-two-clusters-mesos-and-yarn.

[28] D. Norris, “Kubernetes vs. Mesos – an Architect’s Perspective,” 2017. https://www.stratoscale.com/blog/kubernetes/kubernetes-vs-mesos-architects-perspective.

[29] M. Barata, J. Bernardino, and P. Furtado, “An overview of decision support benchmarks: TPC-DS, TPC-H and SSB,” vol. 353, pp. 619–628, 2015.

Appendix A - Full example of Spark Operator Deployment definition file in YAML format

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: sparkoperator
  namespace: sparkoperator
  labels:
    app.kubernetes.io/name: sparkoperator
    app.kubernetes.io/version: v2.3.0-v1alpha1
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: sparkoperator
      app.kubernetes.io/version: v2.3.0-v1alpha1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app.kubernetes.io/name: sparkoperator
        app.kubernetes.io/version: v2.3.0-v1alpha1
      initializers:
        pending: []
    spec:
      serviceAccountName: sparkoperator
      containers:
      - name: sparkoperator
        image: gcr.io/spark-operator/spark-operator:v2.3.0-v1alpha1-latest
        imagePullPolicy: Always
        command: ["/usr/bin/spark-operator"]
        args:
        - -logtostderr
```

Appendix B - Full example of Spark Application definition file in YAML format

```
apiVersion: "sparkoperator.k8s.io/v1alpha1"
kind: SparkApplication
metadata:
  name: tpcds
  namespace: default
spec:
  type: Scala
  mode: cluster
  # Use staging (tpcds image)
  image: gitlab-registry.cern.ch/db/spark-service/docker-registry/spark:v2.4
  imagePullPolicy: IfNotPresent
  mainClass: ch.cern.tpcds.BenchmarkSparkSQL
  mainApplicationFile: "{{ path-to-examples }}/libs/spark-service-examples_2
  mode: cluster
  # By default, using cern provided spark-operator,
  # you are authenticated to use bucket of your cluster {{ cluster-name }}
  arguments:
  # working directory where data table reside (must exists and have tables
    - "s3a:///{{ cluster-name }}/TPCDS-TEST"
  # location to store results
    - "s3a:///{{ cluster-name }}/TPCDS-TEST-RESULT"
  # Path to kit in the docker image
    - "/opt/tpcds-kit/tools"
  # Scale factor (in GB)
    - "1"
  # Number of iterations
    - "1"
  # Optimize queries
    - "false"
  # Filter queries, will run all if empty - "q23a-v2.4,q23b-v2.4"
    - ""
  # Logging set to WARN
    - "true"
  deps:
    jars:
      - {{ path-to-examples }}/libs/scala-logging_2.11-3.9.0.jar
      - {{ path-to-examples }}/libs/spark-sql-perf_2.11-0.5.0-SNAPSHOT.jar
  sparkConf:
    # Enable event log
    "spark.eventLog.enabled": "true"
    "spark.eventLog.dir": s3a://{{ cluster-name }}/spark-events
    # Cloud specific - need to run with speculation to avoid strugglers
    "spark.speculation": "true"
    "spark.speculation.multiplier": "3"
    "spark.speculation.quantile": "0.9"
    # TPCDs Specific
    "spark.sql.broadcastTimeout": "7200"
    "spark.sql.crossJoin.enabled": "true"
    # S3 Specific config (remove if s3 not used)
    # We need it to speed up uploads, and outputcommiter/parquet to have co
    "spark.hadoop.fs.s3a.connection.timeout": "1200000"
    "spark.hadoop.fs.s3a.path.style.access": "true"
    "spark.hadoop.fs.s3a.connection.maximum": "200"
    "spark.hadoop.fs.s3a.fast.upload": "true"
    "spark.sql.parquet.mergeSchema": "false"
    "spark.sql.parquet.filterPushdown": "true"
    "spark.hadoop.fs.s3a.committer.name": "directory"
    "spark.hadoop.fs.s3a.committer.staging.conflict-mode": "append"
    "spark.hadoop.mapreduce.outputcommitter.factory.scheme.s3a": "org.apach
  driver:
    cores: 4
    coreLimit: "4096m"
    memory: "6000m"
    labels:
      version: 2.4.0
    serviceAccount: spark
  executor:
    instances: 50
    cores: 4
    memory: "5000m"
    labels:
      version: 2.4.0
  restartPolicy: Never
```

Appendix C - Full example of Scheduled Spark Application definition file in YAML format

```
apiVersion: ”sparkoperator.k8s.io/v1alpha1”
kind: ScheduledSparkApplication
metadata:
  name: spark-pi-schedule
  namespace: default
spec:
  schedule: ”@every 5m”
  concurrencyPolicy: Allow
  template:
    type: Scala
    mode: cluster
    image: gitlab-registry.cern.ch/db/spark-service/docker-registry/spark:v
    imagePullPolicy: IfNotPresent
    mainClass: ch.cern.sparkrootapplications.examples.SparkPi
    mainApplicationFile: ”local:///opt/spark/examples/jars/spark-service-exa
    mode: cluster
    driver:
      cores: 0.1
      coreLimit: ”200m”
      memory: ”512m”
      labels:
        version: 2.4.0
      serviceAccount: spark
    executor:
      cores: 1
      instances: 1
      memory: ”512m”
      labels:
        version: 2.4.0
    restartPolicy: Never
```

TRITA TRITA-EECS-EX-2018:609