Spark on Hadoop YARN & Kubernetes for Scalable Physics Analysis In collaboration with CERN Openlab, Intel, CMS Experiment, Fermilab

Vassilis Dimakopoulos
CERN
3 PUBLICATIONS **2** CITATIONS

Apache Spark on Hadoop YARN & Kubernetes for Scalable Physics Analysis

IT Department - Database Group - Scalable Analytics Solutions

Thursday 4th October, 2018

Author:
Vasileios S. Dimakopoulos

Supervisor:

Evangelos Motesnitsalis

Abstract

Big Data Technologies popularity continues to increase each year. The vast amount of data produced at the LHC experiments, which will increase further after the upgrade to HL-LHC, makes the exploration of new ways to perform physics data analysis a very important challenge. As part of the openlab project on Big Data Analytics in collaboration with the CMS Big Data Project and Intel, this project aims at exploring the possibility to utilize Apache Spark for data analysis and reduction in High Energy Physics. To do this, we decided to focus on the scalability aspect of Apache Spark. We scaled our data input between 20 and 110 TBs and our available resources between 150 and 4000 virtual cores and monitored Sparks performance. Our ultimate goal is to be able to reduce 1 PB of data in less than 5 hours. The datasets used in these scalability tests are stored in ROOT format within the EOS Storage Service at CERN. We used two different resource managers for Apache Spark: Hadoop/YARN and Kubernetes. Furthermore, we leveraged three important libraries: Hadoop-XRootD Connector to fetch the files from EOS, spark-root to import the ROOT files in Spark dataframes, and sparkMeasure to monitor the performance of our workload in terms of execution time. The detailed results and conclusions are presented in this report.

Contents

List of Figures

Chapter 1

Introduction

CERN Experiments data production rates will face a substantial increase in the following years due to the upgrade to HL-LHC. While current technologies perform rather well, the need to investigate alternative scalable ways to perform physics analysis in an easier, more efficient, and more scalable way is becoming imperative. The use of different approaches and tools has the potential to provide a fresh look at analysis of very large datasets that could potentially reduce the time-to-physics with increased interactivity.

1.1 Apache Spark

Apache Spark is a cluster-computing framework for large-scale data processing. It is designed for fast computation and it extends the MapReduce model to efficiently use it for more types of computations, which include interactive queries and stream processing. The main characteristic of Apache Spark is its in-memory computation model that increases the processing speed of applications. It offers a variety of components and multiple features for big data processing.
Such features are:

- **Supports of multiple languages** - Spark provides built-in APIs in Java, Scala, or Python and comes up with 80 high-level operators for interactive querying.
- **Advanced Analytics** - It supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.
- **DataFrames** - Spark can process structured data using Dataframes. Dataframes are conceptually an equivalent to tables in relational databases and can be constructed from a variety of sources such as structured data files, tables in Hive or existing RDDS.

It offers the possibility to be deployed in different modes using various resource managers(Hadoop YARN, Kubernetes, Apache Mesos). Spark components are illustrated in figure 1.1 [1]

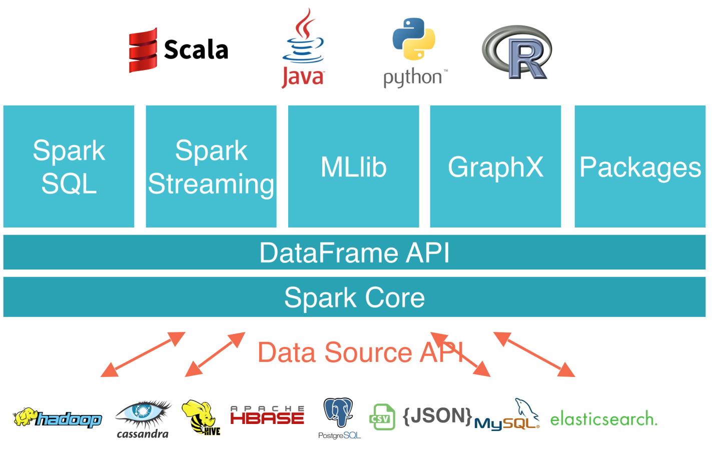


Figure 1.1: Deployment of Apache Spark

[1]Source: databricks Documentation

- **Standalone** - Spark Standalone deployment means Spark occupies the place on top of HDFS (Hadoop Distributed File System) and space is allocated for HDFS, explicitly. In this mode, Spark and MapReduce will run side by side to cover all park jobs on cluster.
- **Hadoop YARN** - Hadoop Yarn deployment means, simply, Spark runs on YARN without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack and it allows other components to run on top of stack.
- **Apache Mesos** - Apache Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications or frameworks. When Spark is deployed using Mesos, Mesos replaces Spark master as Cluster Manager.
- **Kubernetes** - Kubernetes is a containerized resource manager and when Spark is deployed using it, it uses Kubernetes scheduler for the resource management.

1.1.1 Architecture

Spark architecture is based on 2 main abstractions: **RDD,DAG** (Resilient Distributed Datasets, Directed Acyclic Graphs). However, **Apache Spark 2.x** is using DataFrames as well.

- **Resilient Distributed Datasets (RDD)**: RDDs are collection of data items that are split into partitions and can be stored in-memory on workers nodes of the Spark cluster. In terms of datasets, Apache Spark supports two types of RDDs Hadoop Datasets which are created from the files stored on HDFS and parallelized collections which are based on existing Scala collections. Spark RDDs support two different types of operations: Transformations and Actions.
- **Directed Acyclic Graphs (DAGs)**: DAG is a sequence of computations performed on data where each node is an RDD partition and each edge is a transformation on top of data. The DAG abstraction helps eliminate the Hadoop MapReduce multistage execution model and provides performance enhancements over Hadoop.

It is worth mentioning that it follows a master/slave architecture with two main daemons and a cluster manager. A standalone Spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal Spark cluster or on separate machines i.e. in a vertical Spark cluster or in mixed machine configuration.

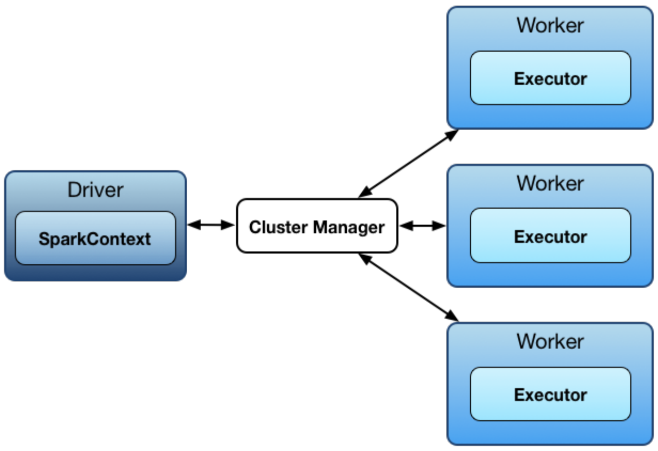


Figure 1.2: Apache Spark Architecture

Spark Driver runs the main function of each application and it is responsible for Spark Context Creation. It contains various components like *DAG Scheduler, Task Scheduler etc.*

1. The Driver program that runs on the master node of the Spark cluster schedules the job execution and negotiates with the cluster manager.

2. For Spark SQL, Dataframes are exposed to SQL optimizer (Catalyst) which leverages advanced programming language features in a novel way to build an extensible query optimizer. Catalyst's transformation framework performs 4 levels of analysis: **Analysis of Logical Plan, Logical Plan Optimization, Physical Plan Creation & Code Generation to compile the parts of the query to Java bytecode**
3. It translates the RDDs into the execution graph and splits the graph into multiple stages.
4. Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
5. Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
6. The tasks executions are handled using Spark's Memory Management which has 2 different memory regions. **Executor Memory** is mainly used for joints, shuffles or aggregations while **Storage Memory used to cache partitions of data**
7. Driver exposes the information about the running Spark application through a Web UI at port 4040.

Executor is a distributed agent responsible for the execution of tasks. Every Spark application has its own set of executor processes. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as Static Allocation of Executors. However, users can also opt for dynamic allocations of executors wherein they can add or remove Spark executors dynamically to match with the overall workload. This type of allocation is very common on shared environment. The executor is responsible for :

1. Performing all the data processing.
2. Reading from and writing data to external sources.
3. Storing the computation results data in-memory, cache or on hard disk drives.
4. Interacting with the storage systems.

Last but not least, **Cluster Manager** is an external service responsible for acquiring resources on the Spark cluster and allocating them to a Spark job. There are 3 different types of cluster managers which a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client Spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos, Kubernetes.

1.2 Hadoop YARN

In our use case Hadoop YARN is used as cluster manager.For the first part of the tests YARN is the Hadoop framework which is responsible for assigning computational resources for application execution.

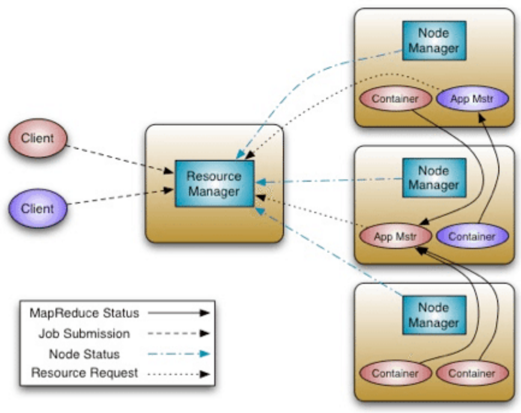


Figure 1.3: Hadoop YARN Architecture

The **ResourceManager** is the rack-aware master node in YARN. It is responsible for taking inventory of available resources and runs several critical services, the most important of which is the Scheduler, which allocates resources for applications to be executed.

Each application running on Hadoop has its own dedicated **ApplicationMaster** instance. This instance lives in its own, separate container on one of the nodes in the cluster. Each applications ApplicationMaster periodically sends heartbeat messages to the ResourceManager, as well as requests for additional resources, if needed.

The **NodeManager** is a per-node agent tasked with overseeing containers throughout their lifecycles, monitoring container resource usage, and periodically communicating with the ResourceManager.

1.3 Kubernetes

Kubernetes is an open source container management platform designed to run cloud-enabled and scalable workloads. It groups containers that make up an application into logical units for easy management and discovery. Whether testing locally or running a global enterprise, Kubernetes flexibility grows so that it can deliver the applications consistently and easily no matter their complexity.
Some of its features are:

- **Horizontal Scaling** - Scales the application up and down with simple commands or automatically based on CPU Usage.
- **Self Healing** - Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to user-defined health check
- **Configuration Management** - Deploys application configuration without rebuilding the images
- **Automatic Binpackings** - Automatically places containers based on their resource requirements and other constraints, while not sacrificing availability

The architecture of Kubernetes provides a flexible mechanism for service discovery. Like most distributed computing platforms, a Kubernetes cluster consists of at least one master and multiple compute nodes. The master is responsible for exposing the application program interface (API), scheduling the deployments and managing the overall cluster. Each node runs a container runtime along with an agent that communicates with the master. The node also runs additional components for logging, monitoring, service discovery and optional add-ons. Nodes are the "slave components" of this master-slave architecture. They expose compute, networking and storage resources to applications. Nodes can be virtual machines (VMs) running in a cloud or physical servers running within the data center.

1.4 Tools, Frameworks and Protocols

In this project the files are stored in EOS. EOS Storage Service is an open source distributed disk storage system in production since 2011 at CERN. Development focus has been on low-latency analysis use cases for LHC and nonLHC experiments and life-cycle management using JBOD(Just a Bunch of Disks) hardware for multi PB storage installations. The EOS design implies a split of hot and cold storage and introduced a change of the traditional HSM(Hierarchical Storage Management) functionality based workflows at CERN. Today, EOS provides storage for both physics and user use cases. Instances of EOS include EOSUSER, EOSPUBLIC, EOSATLAS, EOSCMS. For user authentication EOS supports Kerberos while it is also compatible with XRootD copy mechanism from/to other storage services using it. [2]

For this project we mostly focus on the scalability and performance of the data processing but in order to process our data we need specific tools. One such tool is **Hadoop-XRootD Connector** which allows the communication between the EOS storage service and Hadoop. Specifically, it is a Java library that reads files directly without the need to import/export to Hadoop File System (HDFS). Another useful tool used to analyse physics data in Apache Spark is **spark-root**. This is a SCALA library which offers the ability to translate ROOT data structures into Spark dataframes and RDDs. Furthermore, **sparkMeasure** is being used to monitor the scalability-tests thus offering performance analysis. It can provide further useful insight about a specific job by collecting performance metrics using Spark Listeners. Last but not least, we used Intel CoFluent System Modelling & Simulation Technology in order to configure our tests in an optimal way.

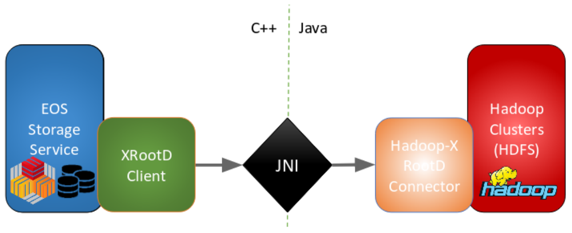


Figure 1.4: Hadoop-XRootD Connector Architecture

[2]Source: AJ Peters, EA Sindrilaru, G Adde

Chapter 2

Scalability Tests

2.1 Description & Prerequisites

For this work, our main concerns are performance, efficiency and scalability. In this chapter, we specify the optimal configuration parameters for our scalability tests as well as the prerequisites that will help us to achieve reproducible results.
Our workload consists of a single Spark job that receives multiple parameters. These parameters include:

- The Maven dependencies such as the spark-root library and the sparkMeasure tool
- The location of the Kerberos tokens in order to authenticate via Kerberos
- The required executor memory
- The desired number of executors
- The required ratio of executors and cores
- The Spark deploy mode
- The location of the input files
- The location of the output files
- The location of the output of the metrics produced by sparkMeasure

A typical example of a Spark job submission can be seen on Figure (2.5).

Our optimal goals for these tests are displayed in Figures 2.1 to 2.4. In Figure 2.1, we see the optimal time and input size behavior, which scales linearly as we increase the input size. Similarly, in Figure 2.2 the increase in processing resources leads to decrease in total execution time. Another apparent goal that we want to observe in our results is to keep the task parallelization factor as high as possible, as we see in Figure 2.3. Finally, we want to keep the network throughput as high as possible, without saturating the network, as it displayed in Figure 2.4.

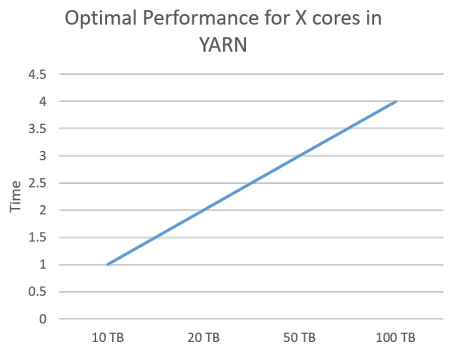


Figure 2.1: Input Size - Time Plot

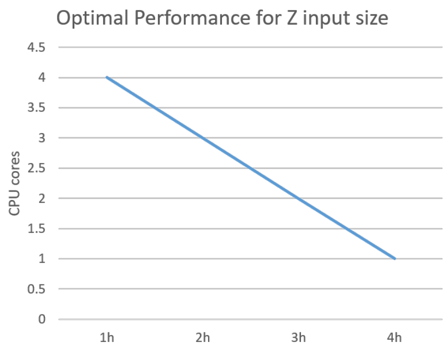


Figure 2.2: CPU Cores - Time Plot

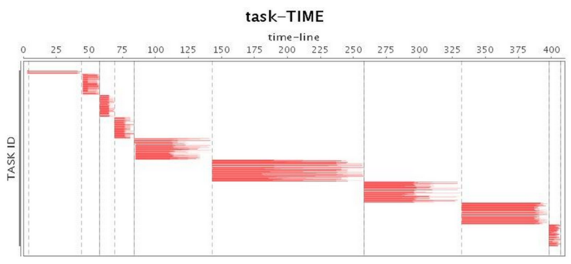


Figure 2.3: Tasks Straggling Effect

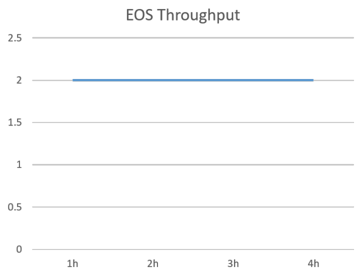
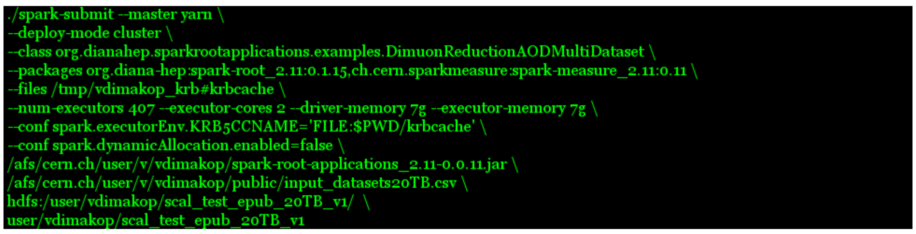


Figure 2.4: Optimal Network Throughput

To avoid some possible bottlenecks in tests performance it is recommended to follow some prerequisites. First of all, we need to create a dedicated queue in YARN with fixed resources that we could use in each iteration of our experiments. Furthermore, we need an EOS dedicated instance so that no other user's access to the same data impacts the performance of these tests. Additionally, a dedicated OpenStack cluster was necessary to have isolated environment for our tests on Kubernetes.

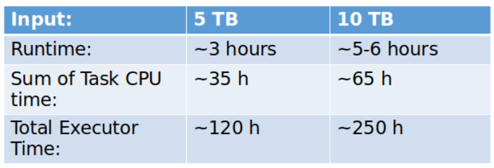
After setting up the prerequisites, the configuration of the job should follow. To be more specific, we modified the jar of the physics processing example that we are using in order to be able to use the **sparkMeasure tool**. sparkMeasure is a tool for performance troubleshooting of Spark workloads. It simplifies the collection and analysis of Spark task metrics data with the utilization of Spark listeners. It provided useful and detailed metrics results for our scalability tests such as executorRunTime, executorCPUTime, bytesRead, stageDuration and many others.

The next step needed to perform was the resource configuration. In order to define the optimal configuration it was recommended to analyze our jobs in small workloads and observe their behaviour with different parameterization. This steps could be divided into two sub steps, one for each resource manager (Hadoop YARN, Kubernetes). Starting with Hadoop YARN configuration we should take into consideration that the infrastructure is comprised of almost 1300 cores and 7TB of RAM. Since Spark demands at least 3 different resource configurations(executors, core per executor, memory per executor) we decided to use: **418 executors, 2 cores per executor and 7GB memory per executor**. The decision derives from the execution time. Specifically the 2 cores offered the best performance compared to 1 or 4 in terms of execution time, minimum stragglers and network throughput. Additionally, 418 executors mean that every node will use 11 executors (since in this way we achieve best parallelization factor) and we selected the maximum allowed memory per executor (7 GB for analytix) since these tests are memory hungy. We disabled also, dynamic allocation and enabled speculation to avoid the unbalanced allocation of resources and long execution times in specific tasks respectively.
Regarding Kubernetes we created a virtual cluster on OpenStack with 500 nodes each of which consists of 8 cores and 16GB of RAM. For these tests we used 1000 executors with 2 cores each and 6GB of memory. At this point it is also important to highlight that before running a job on the private instance of custom storage service it is necessary to authenticate using a Kerberos token. An example of the Spark job that is submitted is illustrated in figure (2.5)



```
./spark-submit --master yarn \
--deploy-mode cluster \
--class org.dianahep.sparkrootapplications.examples.DimuonReductionAODMultiDataset \
--packages org.diana-hep:spark-root_2.11:0.1.15,ch.cern.sparkmeasure:spark-measure_2.11:0.11 \
--files /tmp/vdimakop_krb#krbcache \
--num-executors 407 --executor-cores 2 --driver-memory 7g --executor-memory 7g \
--conf spark.executorEnv.KRB5CCNAME='FILE:$PWD/krbcache' \
--conf spark.dynamicAllocation.enabled=false \
/afs/cern.ch/user/v/vdimakop/spark-root-applications_2.11-0.0.11.jar \
/afs/cern.ch/user/v/vdimakop/public/input_datasets20TB.csv \
hdfs:/user/vdimakop/scal_test_epub_20TB_v1/  \
user/vdimakop/scal_test_epub_20TB_v1
```

Figure 2.5: Typical Example of Spark Job



Input:	5 TB	10 TB
Runtime:	~3 hours	~5-6 hours
Sum of Task CPU time:	~35 h	~65 h
Total Executor Time:	~120 h	~250 h

Figure 2.6: Small workloads with dynamic allocation enabled

2.2 Initial Tests

It is worth mentioning that the initial tests were performed on small scale to understand their performance and the behaviour of the workload. After executing tests on 22TB with different resource configuration, we obtained the results of the table (2.1). From these results it is apparent that more executors and cores don't affect the performance in a reasonable way. The strange behaviour observed in these specific tests lead us to further monitor the process using **ganglia**. The reason behind this originated from the fact that all of our straggler tasks were coming from a single node. Particularly, when observing the network usage of every node, it was detected that this node didn't use the network adequately. Therefore, the tasks assigned to this node were straggling thus, resulting in higher execution times. In order to improve the performance we decided to **exclude** this node from the analytix cluster and modify the resource configuration to be proportional to the new cluster architecture. The new resource configuration affected only the number of executors which would be 407 ((Previous nodes(thirty eight)-Struggler Node(one))*Executors per Node(eleven))

Input	Executors	Cores/Executor	Total Cores	Memory/Executor	Total Memory	Execution Time
22 TB	148	4	592	7 GB	1 TB	2 hrs
22 TB	148	2	296	7 GB	1 TB	1hr 50 min
22 TB	148	1	148	7 GB	1 TB	3 hrs
22 TB	296	1	296	7 GB	2 TB	3 hrs
22 TB	555	1	555	7 GB	4 TB	1 hr 50 min

Table 2.1: First Results with different resource configuration

2.3 Main Tests

Our first goal is to execute the workload for different input size of physics data in order to understand how the execution times scales with input size as well as obtain a first insight on how to reduce the 1PB data. The second goal is to execute the same workload again for a specific input size while scaling up the available resources(executors/cores). These two types of tests were repeated for 2 different storage instance, namely EOS Public and EOS UAT. We do this to identify if the network throughput and storage infrastructure affect the performance of the tests. As illustrated in (table: 2.2, fig: 2.7) it could be easily understood that there is a linear behaviour between the input size and the execution time. Moreover, we can claim that these results are promising since we reduced 110 TBs in 212 minutes but they still need further optimization.

Input Size	Execution Time
22TB	58 min
44TB	83 min
66TB	149 min
88TB	180 min
110TB	212 min

Table 2.2: The results of scalability tests for different input size

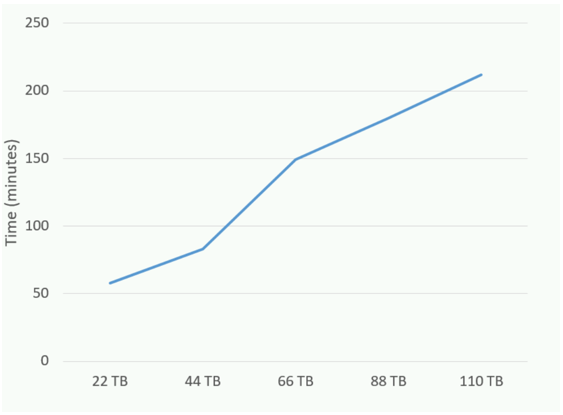


Figure 2.7: Performance for different input size

Coupled with the previous results the memory/cores scalability tests followed, where we scale up the amount of memory or the number of cores by adjusting the number of executors since the memory per executor is fixed to 7GB. The tables and the figures illustrate the results below. However, it seems that after a specific amount of memory/cores used the results converge to a constant performance .

Number of Executors/Cores	Total Memory	Execution Time
74/148	0.5TB	81 min
148/296	1TB	53 min
222/444	1.5TB	52 min
296/592	2TB	51 min
370/740	2.5TB	50 min
444/888	3TB	50 min

Table 2.3: The results from scaling up the memory

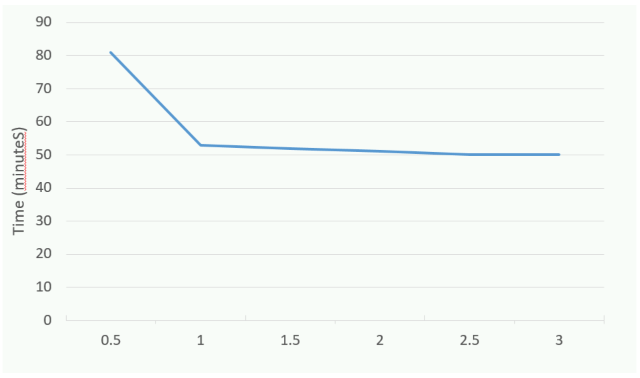


Figure 2.8: Performance for different memory size

Number of Executors/Cores	Execution Time for EOS public	Execution Time for EOS UAT	Execution Time for Shared Hadoop Cluster
111/222	81 min	153 min	41 min
222/444	52 min	146 min	35 min
296/592	51 min	144 min	33 min
407/814	50 min	140 min	29 min

Table 2.4: The results from scaling up the number of cores

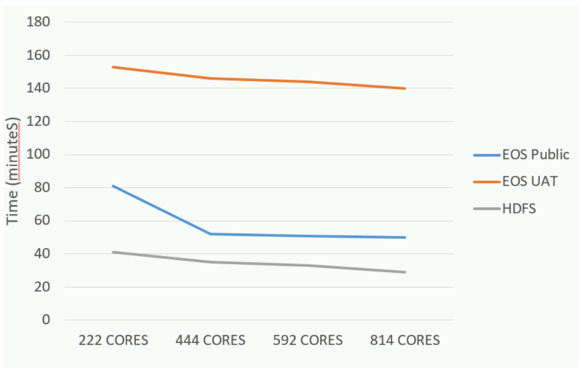


Figure 2.9: Performance for different number of cores

First of all, as far as EOS is concerned it seems that public instance behaves more efficient and the reason behind this is that it has more nodes to utilize and therefore more network capacity. Moreover, as in figure 2.8 as in figure 2.9 it could be highlighted that there is a convergence after scaling above a certain amount of memory or cores correspondingly. This led us to investigate why is this happening. Apparently, there is a fundamental factor that affects this behaviour and this is the the saturation of the available network bandwidth. With this in mind, we monitored and measured the total throughput of the network while running these jobs and the results are summarized below. Notwithstanding the gradual scaling of the resources the network reaches its upper bound in early stages of scaling resulting in freezing the improvement of the performance. Approaching this issue from another perpective we could claim that no matter how we scale up the resources the network could not fetch files faster to adapt to the availale computing resource. This means that the resources are "waiting" to be used when the files are ready for processing or in other words that our workload is **network bound** in our use case.

Cores	EOS public	EOS UAT
222	15GBytes/s	6GBytes/s
444	19GBytes/s	7.5GBytes/s
592	21GBytes/s	7.5GBytes/s
814	21GBytes/s	7.5GBytes/s

Table 2.5: Network throughput when scaling the resources

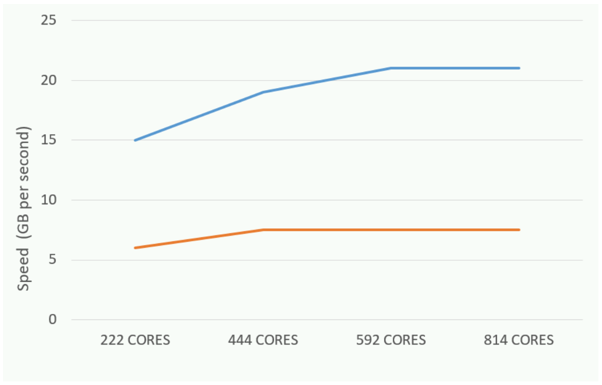


Figure 2.10: Network Throughput while scaling the resources

2.4 Spark on Kubernetes

At this point and after executing the scalability tests on Hadoop YARN we proceeded to execute the same Spark jobs for Kubernetes. The resource configuration is 1000 executors 2 cores per executor with 6GB memory for each executor. This parameterization was chosen since we needed to utilize all of the available Openstack resources to improve the performance of these tests. However, after running the first tests for three different input data sizes we observed a full **network saturation** which impacted some CERN production services (2.12). For this reason, we had to abort these tests since the infrastructure wasn't ready to support this network load without impacting the production services. The results obtained for those three tests that we managed to run before the termination are more than promising and could lead to the outcome that if the network infrastructure could handle this workload we would be able to achieve our ultimate goal to reduce 1PB in 5 hours with a limited amount of cores/memory.

Input Data	Runtime
22TB	40 min
44TB	70 min
66TB	120 min

Table 2.6: The results from Kubernetes scalability tests

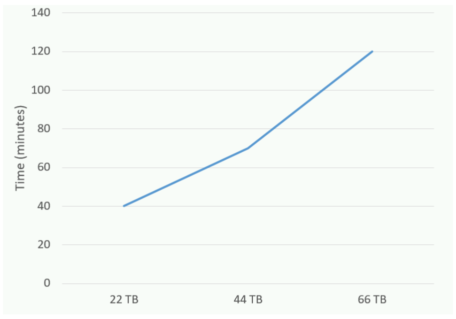
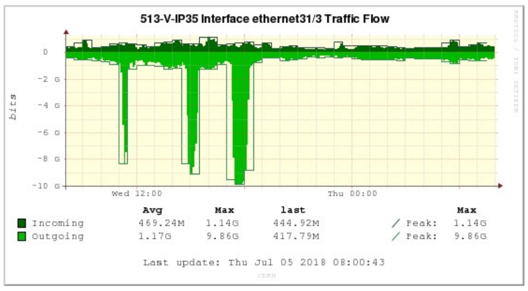


Figure 2.11: Kubernetes results for different input size



```
                Avg       Max      last                   Max
■ Incoming   469.24M   1.14G   444.92M       ⁄ Peak:   1.14G
■ Outgoing     1.17G   9.86G   417.79M       ⁄ Peak:   9.86G

           Last update: Thu Jul 05 2018 08:00:43
```

Figure 2.12: Full Network Saturation resulting in production services degradation

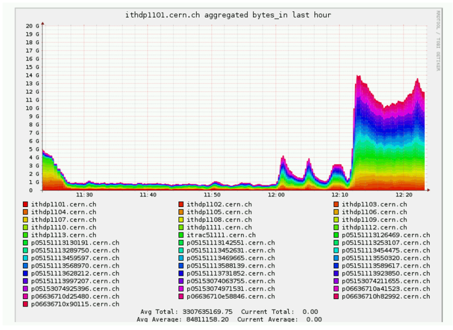


Figure 2.13: Low network saturation with 111 executors

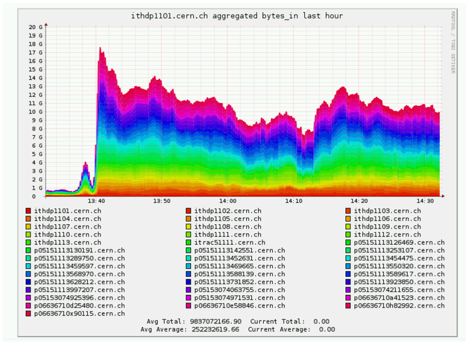


Figure 2.14: Intermediate Network saturation with 222 executors

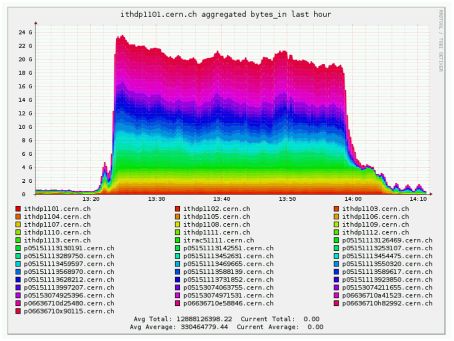


Figure 2.15: Network saturation with more than 300 executors

2.5 Validation and Final Results

An important step before confirming our findings is the validation step. Therefore, what we did is to repeat the same jobs multiple times in order to confirm our results and assumptions. However, after a closer look in the sparkMeasure results we noticed that some parameters, which were supposes to remain constant changed its values during the rerun of the same tests (e.g number of bytes read for 20TB job should be the same in all of the repeats of the job). This led us to investigate further and we noticed that there was a **defect in reading from EOS using Hadoop-XRootD connector**. This issue was correlated to the buffer size used to read the files from the storage service. Specifically the buffer size was **32MB** and along with the fact that requests for read is high this caused collecting unnecesary data from files thus, **increasing the network throughput**. Therefore, buffer size was decreased to 128 KB and all the tests re-executed. Keeping the configuration the same the results were performed again for both EOS public and EOS UAT instances. The final results could be summarized in the next tables and figures.

Input Size	Execution Time
22TB	17 min
44TB	30 min
66TB	38 min
88TB	48 min
110TB	56 min

Table 2.7: The results of re-executing scalability tests for different input size on EOS Public

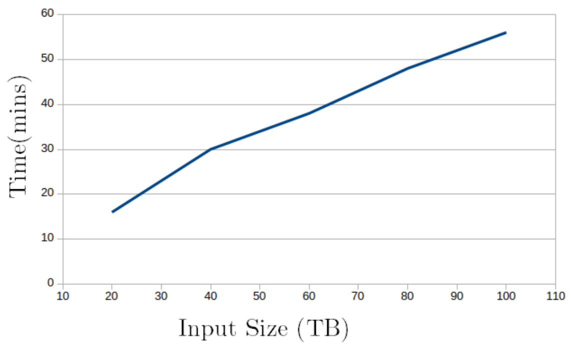


Figure 2.16: Performance of the tests for different input size with new Hadoop-XRootD Connector configuration on EOS Public

After this configuration of new Hadoop-XRootD Connector the tests seem to have significant improvements since as observed from table 2.7 the **110TB job could be reduced in less than an hour** but at the same time the scaling keeps its linear behaviour. Compared to figure 2.7 the scaling is smoother now with smaller slope.

Number of Executors/Cores	Total Memory	Execution Time
74/148	0.5TB	32 min
148/296	1TB	28 min
222/444	1.5TB	16 min
296/592	2TB	16 min
370/740	2.5TB	17 min

Table 2.8: The results of re-executing tests by scaling up the memory on EOS Public

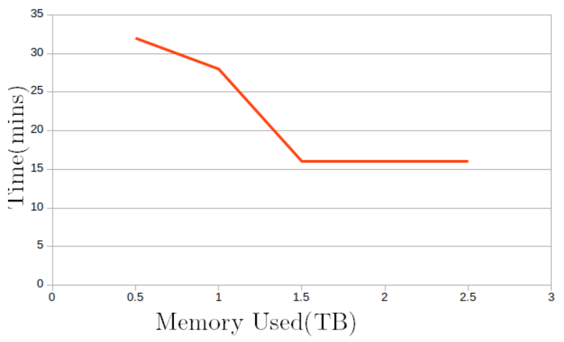


Figure 2.17: Performance of the tests for different memory size with new Hadoop-XRootD configuration on EOS Public

Coupled with the previous table and figure it is obvious that the tuning of Hadoop-XRootD Connector buffer offer improvements also in memory scaling for these tests. The reduction of the time is greater compared to the results before but we still could observe that when we scaled up the memory the results keep on improving thus meaning that physics application that we are using is maintaining its high memory requirements.

Input	Executors	Cores	Memory	Time
22 TB	111	222	0.7 TB	31 mins
22 TB	222	444	1.5 TB	19 mins
22 TB	296	592	2 TB	16 mins
22 TB	407	814	3 TB	16 mins

Table 2.9: The results of re-executing tests by scaling up the cores on EOS Public

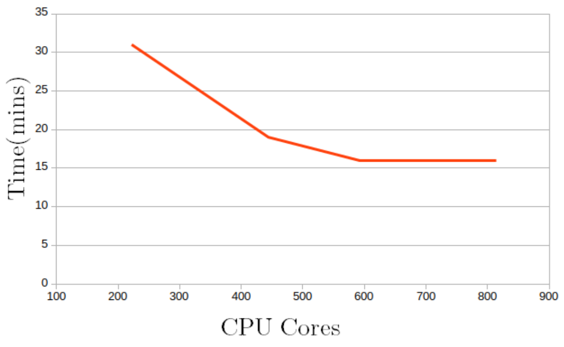


Figure 2.18: Performance of the tests for different number of cores with new Hadoop-XRootD Connector configuration on EOS Public

Similarly, we performed the tests for both EOS UAT & HDFS apart from EOS public and we compared the aggregate results to observe under which conditions our tests scale up better and in a more efficient way.

Input	EOS PUBLIC	EOS UAT
22 TB	16 mins	13 mins
44 TB	30 mins	25 mins
66 TB	38 mins	38 mins
88 TB	48 mins	57 mins
110 TB	56 mins	59 mins

Table 2.10: Aggregate results of re-executing tests by scaling up the input data size on EOS public, EOS UAT

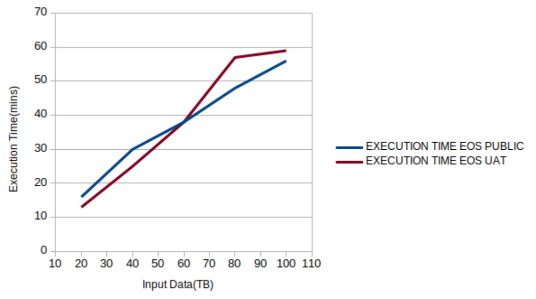


Figure 2.19: Performance of the tests for different input size with new Hadoop-XRootD connector configuration on EOS public, UAT

Input	Memory	EOS PUBLIC	EOS UAT
22 TB	0.5 TB	32 mins	21 mins
22 TB	1 TB	28 mins	16 mins
22 TB	1.5 TB	16 mins	15 mins
22 TB	2 TB	16 mins	13 mins
22 TB	2.5 TB	16 mins	13 mins

Table 2.11: Aggregate results of re-executing tests by scaling up the memory on EOS public, EOS UAT

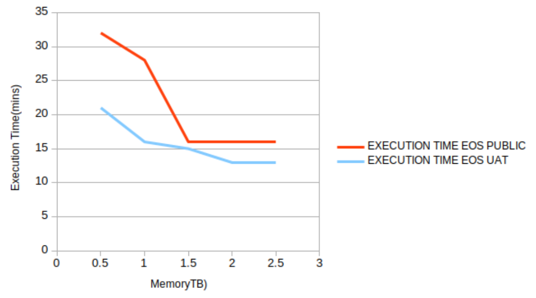


Figure 2.20: Performance of the tests for different memory with new Hadoop-XRootD Connector configuration on EOS public, UAT

Input	Cores	EOS PUBLIC	EOS UAT
22 TB	222	31 mins	17 mins
22 TB	444	19 mins	13 mins
22 TB	592	16 mins	13 mins
22 TB	814	16 mins	13 mins

Table 2.12: Aggregate results of re-executing tests by scaling up the cores on EOS public, EOS UAT

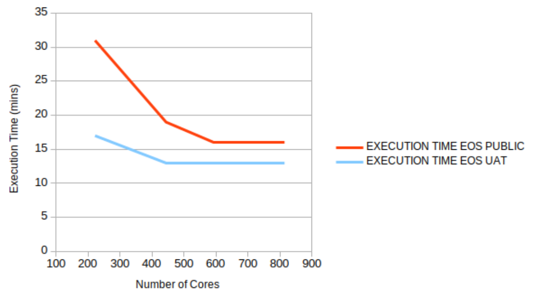


Figure 2.21: Performance of the tests for different number of cores with new Hadoop-XRootD Connector configuration on EOS public, UAT

2.6 Tests on Higher Workload - 1PetaByte

After the execution of all our tests, one of the next steps was to submit a higher workload, namely 1PB, while keeping constant the previous resource and parameter configuration. However, this job failed on the first runs since the output which was produced couldn't be handled by the current Driver size. To resolve this issue we tried with different Driver sizes between 2 and 8 GBs while changing **spark.driver.maxResultSize**. Apparently, the one that resolved the issue was the 8GBs. Moreover, due to multiple restarts of Application Master, one other parameter that we configured was **spark.yarn.maxAppAttempts**, which is responsible for maximum number of attempts that will be made to submit the application, to 1.

The execution time using these configurations was **9 hours & 56 minutes** which is an expected number compared to the performance of the previous tests. From another perspective this indicates that our tests, tools and services are scaling well and linearly even for higher workloads.

2.7 Tests on different input structure

This part of the project goes beyond the initial goals and it has as aim to investigate how Apache Spark behaves for different input structure. Specifically, it aims at testing the same workload but giving as input either one folder that contains all the input files or multiple folders each of which contains some of the input data. For EOS UAT we tested **110TB workload** and we noticed the following execution times:

- For one folder: 72 minutes
- For multiple folders: 58 minutes

From these results we could say that Apache Spark performs better when the files are distributed among multiple folders. Last but not least this could give us further insight on how higher workloads should be configured in terms of input structure to achieve better performance.

Chapter 3

Conclusions

Summarizing, in this project we performed scalability tests for physics data analysis using Apache Spark on top of Hadoop YARN and Kubernetes. The results obtained from the executed task jobs conform to most of our goals. Specifically, we managed to observe:

- Linear behaviour between input size & runtime
- Efficient utilization of resources
- Reduction of 110 TB in less than an hour
- Reduction of 1 PB in almost 10 hours

However, we faced also some bottlenecks in our use case that led to deviation from our primary goals. First of all, the network bound that we observed when we scale the cores or the memory didn't offer us the possibility to achieve better results initially as we reached the upper bound of network even in the early stages of the scaling (figures: 2.13, 2.14, 2.15). This issue though, was tackled since with the tuning of Hadoop-XRootD Connector readAhead buffer we managed to reduce the network throughput to almost the half. The results of the tests after these configuration were promising and they are indicating that we are getting closer to our initial goal of reducing 1PB in 5 hours.

Another key thing to remember is notwithstanding the uncompleted tests of Spark on Kubernetes tests they provided us with an insight of optimistic results for the future. However we need to repeat these tests in an isolated infrastructure in order to avoid impacting the production services.

3.1 Observations & Lessons Learnt

The Observations and the Lessons Learnt through this project are summarized below:

1. We found that it was necessary to tune the Hadoop-XRootD Connector buffers to achieve better network utilization.
2. The workload was memory hungry.
3. Hadoop, Spark, Openstack and Kubernetes scaled rather well.
4. Since slow nodes can create problems with straggler tasks, which can badly affect the performance, Apache Spark Task Speculation feature is a good mitigation to this issue.
5. EOS also scaled rather well but we noticed a maximum throughput.
6. The ROOT files cannot be split into blocks, by the current version of the spark-root library and therefore we have a file to task mapping which results in single tail at the end of the job taking 10-15 minutes of the total execution time.
7. With the new Hadoop-XRootD Connector buffer size and the same infrastructure we need **560 minutes** to reduce 1PB which approaches the initial aim of 300 minutes.
8. After executing 1PB job the execution time was 596 minutes which is close to 560 minutes which we were expected with the current configuration.
9. Apache Spark performs better when the input data are distributed in multiple folders.

3.2 Issues

Throughout our tests we faced several problems such as:

- Requesting many resources from the analytix Hadoop cluster led many times to resources denial and to problematic results that needed to be re-executed
- Network errors during our jobs resulting in re-submissions of the jobs
- Performance Degradation of production services on Openstack since they were co-hosted with network switches that we overloaded during the tests on Kubernetes.
- Kubernetes cluster took more than 1 week to be created due to a bug in the VM Scheduler
- Hadoop-XrootD Connector buffer resulted in excessive reading of data and thus high network throughput

3.3 Future Steps

As future steps of this work we would like to perform the tests on a **scaled infrastructure**. As a result, we aim at repeating the Kubernetes tests in an **isolated infrastructure and in a not impacting way**. Last but not least it is worth investigating if we are **CPU or I/O Bound** for our workload, once the network bottleneck is resolved. To be able to answer to this question we need to perform the tests for our workload with the new upgrades of the network infrastructure and also the newest Hadoop-XRootD Connector.