Andrey Verbovskiy

Big Data Processing Cluster for Time Series Modelling

Metropolia University of Applied Sciences

Bachelor of Engineering Degree

Information Technology

Bachelor Thesis

12 April 2023

Abstract

Author:	Andrey Verbovskiy
Title:	Big Data Processing cluster for time series modelling
Number of Pages:	61 pages + 3 appendices
Date:	12 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Specialisation option:	Smart IoT Systems
Instructor(s):	Peter Hjort, Senior Lecturer
	Mayank Darbari, Data Scientist

The goal of this project was to build a Big Data Processing cluster, consisting of three main components: Apache Spark for data processing, a Big Data distributed database called Apache Cassandra and a distributed event streaming platform called Apache Kafka.

The main research question was how to build and maintain a Big Data processing cluster with open-source technologies on virtual machines. Moreover, the thesis studies and discusses the core technologies and their architecture.

The end product of this study was a cluster of three virtual machines, with all three technology instances installed on each of them. The system is stable and has a functional web interface. In conclusion, companies such as Nokia can use open-source technologies for big data processing instead of purchasing expensive subscriptions of similar technologies provided by other companies.

Keywords: Big Data, Apache Spark, Apache Cassandra, Apache Kafka, Linux, Cluster, Data Processing, Machine Learning Engineering

Contents

1. Introduction 1

2. Goal of this thesis 1

3. Time series data 2

4. Apache Cassandra 6

4.1. Background 6

4.2. Cassandra and CAP theorem 9

4.3. Distribution and replication of data in Cassandra 10

4.4. Cassandra architecture 13

5. Apache Spark 15

5.1. Background 15

5.2. Resilient Distributed Datasets 17

5.3. Spark architecture 18

5.4. RDD lineage 19

6. Apache Kafka 21

6.1. Background 21

6.2. Kafka architecture 22

6.3. Kafka applicability in this project and beyond 25

7. Cluster set up 26

7.1. Initial resources 26

7.2. Cassandra setup 26
7.3. Spark setup 32
7.4. Kafka setup 37
8. Final changes and experiments 40
8.1. Cassandra data replication and partitioning 40
8.2. Spark-Cassandra connector and Jupyter server 45
8.3. Kafka functionality and Spark Streaming-Kafka connector 49
9. Reflection 53
10. Conclusion 54
References 56
Appendix A: Cassandra
Appendix B: Spark
Appendix C: Kafka

1. Introduction

In the current state of technologies in the world, developers and companies have to deal with enormous amounts of data, which is often a very complicated process to manage. Even the biggest titans of the IT industry such as Microsoft, Google and others invest enormous amounts of money in structuring their data analysis and forecasting in a sufficient and fast pipeline. Nokia is not an exception, and in order to address this issue, this study was initiated. The objective of the project was to create a cluster of virtual machines that can process and store big amounts of data to later apply machine learning modelling on the data inside the cluster, and to provide a detailed documentation of the process. This can be achieved by combining three different software programs called Apache Spark, Apache Cassandra and Apache Kafka.

2. Goal of this project

Why is such technology required and what are the main benefits of this project to Nokia? Nokia, like any other company has big amounts of data circling around, whether it is financial or telecommunication data or any other kind of data related to Nokia business. As any other company, Nokia has many Data Scientists that would love to run experiments on the data, yet this leads to several challenges. The main struggle is that amounts of data are incredibly big. Usually, a programmer would use Python libraries such as Pandas to work with and pre-process the data which is not heavier than 10 gigabytes, but what if the data volume is 10 or even 100 terabytes? To address this issue, data management companies like DataStax provide systems and services to process big amounts of data for data analysis and machine learning modelling. However, this leads to the second challenge. These services cost a lot, which puts companies in a place of thinking whether it is worth it to spend such a large amount of money for often quite questionable experiments. The final challenge is that some companies like LinkedIn, Facebook, Netflix and many others use these systems, yet most of the

guides and documentations were published around 2008 and are outdated as all of the applications have been updated.

Solving both of the listed challenges is the main purpose of the project. Building a cluster of virtual machines that can easily distribute and process big amounts of data is important. Moreover, being built on open-source software, which can be accessible by anyone anywhere, would save a great amount of money to Nokia.

Another important benefit for the Nokia team called ATG (Advanced Technology Group) that this thesis is made for is that the team is carrying out several big data studies and that it takes too much time - usually more than ten hours - to train the models on the data. Also, the ATG team has a lot of resources when it comes to virtual machines that are not used anywhere but could be implemented in this project and be actively used. Thus, when completed, this thesis will become the most suitable guide for developers that would like to explore the mentioned technologies and create a suitable platform for Big Data processing.

3. Time series data

There are many data types, but the specific one that the thesis is focused on is time series data. Time series data is a type of data that includes timestamps, time periods or other time data. There are several methods and processes that take advantage of time being present in the dataset, one of such is forecasting, which, in simple words, is a prediction of future values based on past ones. The most obvious examples of time series data are financial stock market data, web traffic or user activity.[1.]

Time series data can be defined as being stationary or non-stationary. Stationary data is the data type in which mean and variance are not changing future values,

while they change for non-stationary type. The following Figure 1 demonstrates the difference between stationary and non-stationary data.

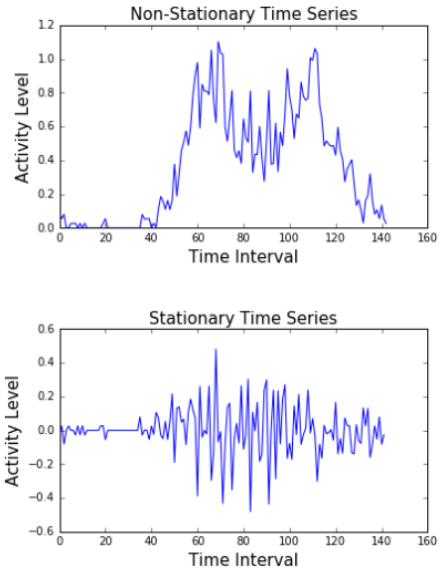


Figure 1. Stationary and non-stationary data graphs comparison [2].

A time series dataset can be of two types, univariate or multivariate based on its structure. A univariate time series dataset consists of only one data column besides the timestep one, which is illustrated in figure 2.

	Datetime	AEP_MW
0	2004-12-31 01:00:00	13478.0
1	2004-12-31 02:00:00	12865.0
2	2004-12-31 03:00:00	12577.0
3	2004-12-31 04:00:00	12517.0
4	2004-12-31 05:00:00	12670.0
...	...	...
121268	2018-01-01 20:00:00	21089.0
121269	2018-01-01 21:00:00	20999.0
121270	2018-01-01 22:00:00	20820.0
121271	2018-01-01 23:00:00	20415.0
121272	2018-01-02 00:00:00	19993.0

Figure 2. Univariate dataset example. [3]

While multivariate time series dataset consists of multiple columns besides the timestamp:

	date	Usage_kWh	Lagging_Current_Reactive.Power_kVarh	Leading_Current_Reactive_Power_kVarh	CO2(tCO2)
0	01/01/2018 00:15	3.17	2.95	0.00	0.0
1	01/01/2018 00:30	4.00	4.46	0.00	0.0
2	01/01/2018 00:45	3.24	3.28	0.00	0.0
3	01/01/2018 01:00	3.31	3.56	0.00	0.0
4	01/01/2018 01:15	3.82	4.50	0.00	0.0
...	...	...	...	...	...
35035	31/12/2018 23:00	3.85	4.86	0.00	0.0
35036	31/12/2018 23:15	3.74	3.74	0.00	0.0
35037	31/12/2018 23:30	3.78	3.17	0.07	0.0
35038	31/12/2018 23:45	3.78	3.06	0.11	0.0
35039	31/12/2018 00:00	3.67	3.02	0.07	0.0

Figure 3. Multivariate time series dataset. [4]

There are many Machine Learning algorithms and models designed for forecasting time series, yet not all of them are suitable for both multivariate and univariate data. The following table includes time series models and general models like XGboost that are suitable for time series related experiments:

Table 1. ML models for time series forecasting and their compatibility for multivariate and univariate data.

Name of the algorithm/model	Univariate	Multivariate
Exponential smoothing	✓	✗
Theta forecaster	✓	✗
AutoARIMA	✓	✗
BATS	✓	✗
Facebook Prophet	✓	✗
Neural Prophet	✓	✗
Greykite: Silverkite	✓	✗
NBEATS	✓	✓
Temporal Fusion Transformer	✓	✓
XGboost	✓	✓

Why is studying time series data important? The importance of studying the time series data is emphasized by gaining additional valuable correlations of data with time. Time series models take into account many extra variables that usual models do not. Examples of such variables are trend and seasonality.

Trend is an increase or decrease of a time series level. This change is not periodic and occurs in every subset of time series.

Seasonality is a presence of variations that occur in a regular period. For example, data can have yearly seasonality when the price of gifts increases before Christmas.[1.]

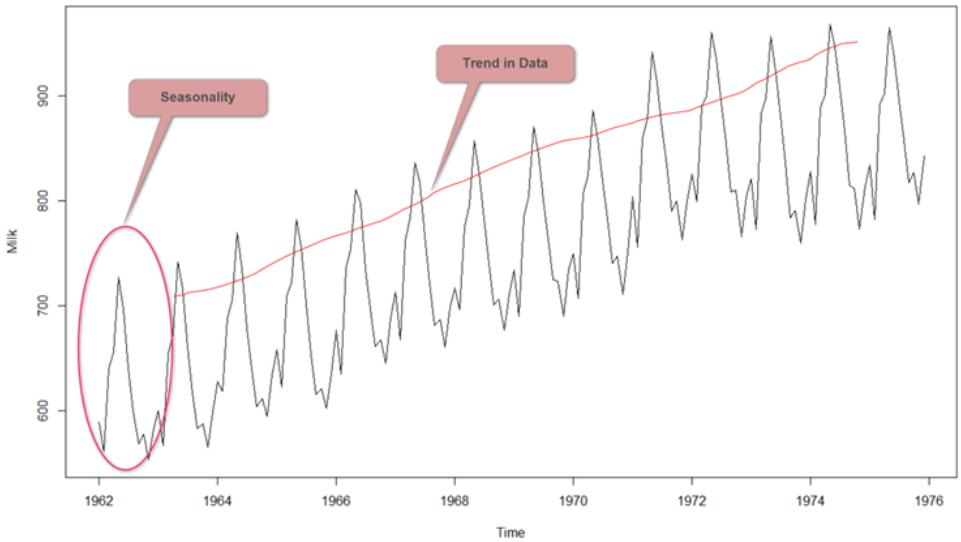


Figure 4. Trend and seasonality on a time-series graph [5].

Considering these variables improves accuracy in the predictions and forecasting of models significantly, which enables time series forecasting modelling to be a hot topic in the Data Science field.

4. Apache Cassandra

4.1. Background

The main three software used in this project were Apache Spark, Apache Cassandra and Apache Kafka. Each one of them serves a specific mission in the cluster and each of the mentioned technologies will be co-located on all three nodes.

Firstly, in order to store big data, a scalable database needs to be selected. Cassandra suits well for it. Apache Cassandra is an open-source decentralised NoSQL database management system. It is widely used when it comes to preparing/accessing large datasets, mostly in combination with Apache Spark. The main purpose of Cassandra is to split a big dataset between multiple nodes (computers, virtual machines) which form an ecosystem, which is called a cluster. The process of splitting is called partitioning. One of the key features of

Cassandra is its communication protocol between nodes in the cluster which is called Gossiping. In the cluster all nodes have equal rights and there is no master node. The nodes form a ring in which the replication of data happens. Replication ensures that whatever happens to the node, it is data will not be lost. The node sends a copy of its data to a few other nodes (to three by default, but this can be changed by a user). The nodes actively communicate with each other about themselves and who they gossip about, so if something happens to a node, others will immediately identify it. Every exchange of gossip has a version attached to it, so during gossiping old information gets rewritten with the newest information about nodes' state.

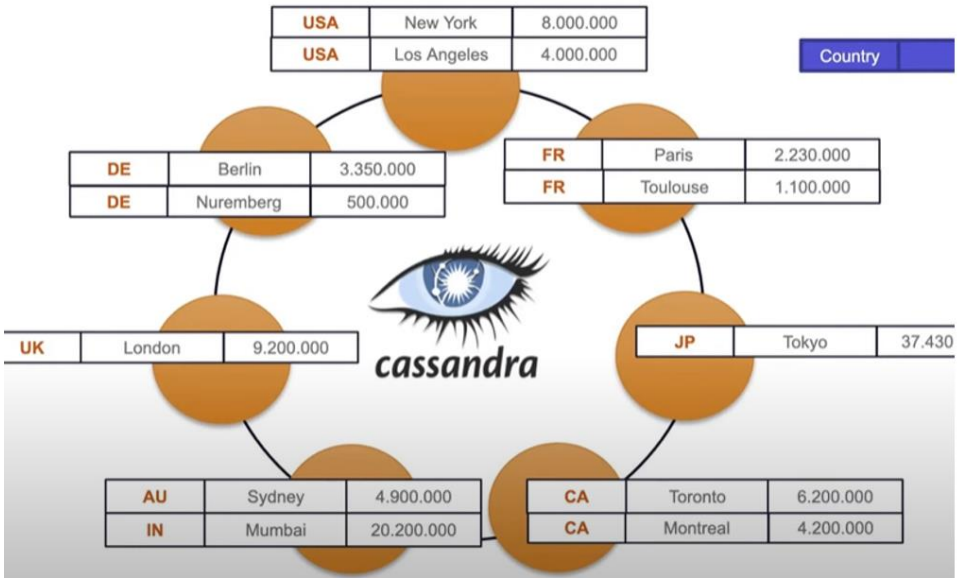


Figure 5. Example of the data split across the nodes [6].

Compared to alternatives, such as Azure Cosmos DB or ScyllaDB, Cassandra has the following several advantages:

- **Horizontal scalability.** With the inevitable increase in data size after some time of use, the cluster will need to increase its capacity. Horizontal scalability implies increasing the cluster by adding new nodes (virtual machines, laptops...) to the system without interrupting the work of the cluster. Unlike the vertical scalability, in which the only way to increase the capacity is by updating the hardware of the existing nodes. Horizontal scalability loses in terms of initial cost, as it costs much more to buy multiple devices instead of a few. However, in the long run, when amounts

of data increase many times, horizontal scalability wins in terms of the cost, as it is much cheaper to just add many not expensive laptops or virtual machines to the system rather than buy a more expensive hardware setup. Moreover, horizontal scalability is more fault tolerant as losing one node of a hundred is not as crucial as losing one of a few. Yet, it does have some disadvantages like complex maintenance of the system with that big number of nodes. [7.]

- **Opensource.** Apache Cassandra is opensource technology, which is available for anyone to download and use. There are also many contributors, who develop additional features or files for it, all accessible on the internet.
- **Data replication.** After getting split, the data of each node still gets replicated to another neighbouring node to prevent the loss of data if critical situations like the destruction of a node happen. If a particular node gets destroyed, the data and the user will be automatically relocated to the nearest working node in the cluster. The system will continue to work as it is supposed to without interruption, with applications always available, and data always accessible. Users will not know if there was an outage as the work will continue smoothly thanks to relocation of data or connected users. Moreover, Cassandra's built-in repair service fixes problems immediately when they occur, without any additional human assistance. This is incredibly significant for companies which cannot afford to ever have their database go offline or to lose any data.
- **Decentralized data distribution**. In order to ensure proper work of machine learning, dataflow in the database must be kept at proper pace, which means that it is essential for a database to be decentralized and fault-tolerant. Because Cassandra is a completely decentralized database, its network is structured so that every node in the cluster is identical in terms of its responsibilities while each is responsible for different partitions (data is partitioned into multiple groups).

In this project, Cassandra will serve the role of data storage. The Big Data will be split and distributed along the virtual machines for further processing.

4.2. Cassandra and CAP theorem

Whenever a developer needs to decide on the best suiting database for the system, one should refer to the famous CAP theorem. [6]

To implement machine learning, a database must follow best-read and write practices to minimize data inconsistency. According to the CAP theorem, a distributed database cannot guarantee consistency, availability, and partition tolerance at the same time.

- **Data consistency** refers to whether the same data kept at various places do or do not match.
- **Partition tolerance** means a database can still operate even if network connections between nodes are down.
- **Data availability** means that all working nodes in the distributed system return a valid response for any request.

Partition tolerance is highly necessary for any Machine Learning project, so it must be included. Distributed databases are usually either "CP," prioritizing consistency over availability, or "AP," prioritizing availability over consistency.

Cassandra is an "AP" system as it prioritises availability of the data even if it means sacrificing its consistency but it can be configured to behave like a "CP" database, providing the best possible performance for machine learning by balancing high data availability and consistency.

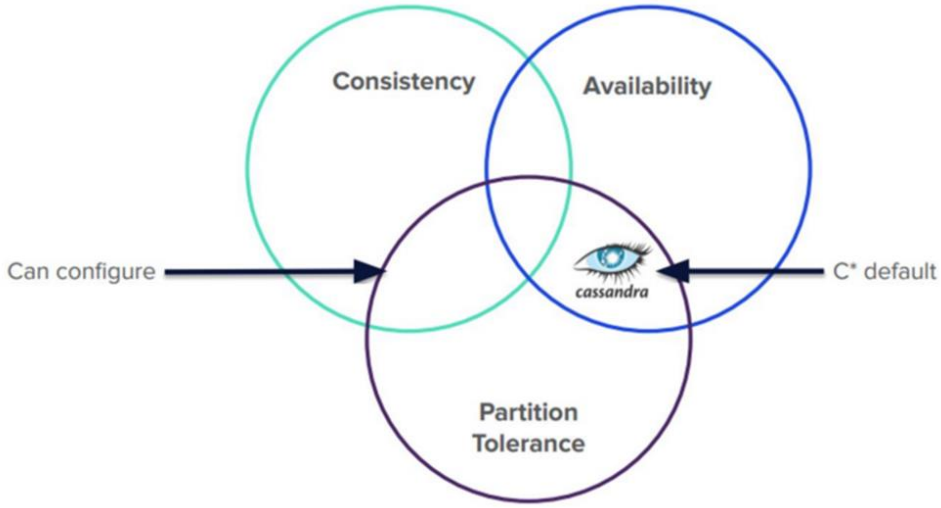


Figure 6. Cassandra in CAP theory diagram. [6]

4.3. Distribution and replication of data in Cassandra

To get a better understanding of how Cassandra works, consider this example. Given the following dataset which could be a million times larger than shown, Cassandra takes the partition key, a data column that the split of data will be based on and splits the dataset as evenly as possible.

Country	City	Population
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

Figure 7. Dataset before partitioning example. [6]

As it can be seen in the figure below, all identical partition keys (countries) are put together. Yet, there can be some unique partition keys like JP or AU, which are partitioned into a separate node or relocated to nodes with other unique partition keys. The main objective for Cassandra while partitioning is to distribute data samples as evenly as possible and to make data distribution as logical as possible, which means that there is some correlation in partition keys between the data samples that were assigned to the same node, like belonging to the same time period or a country:

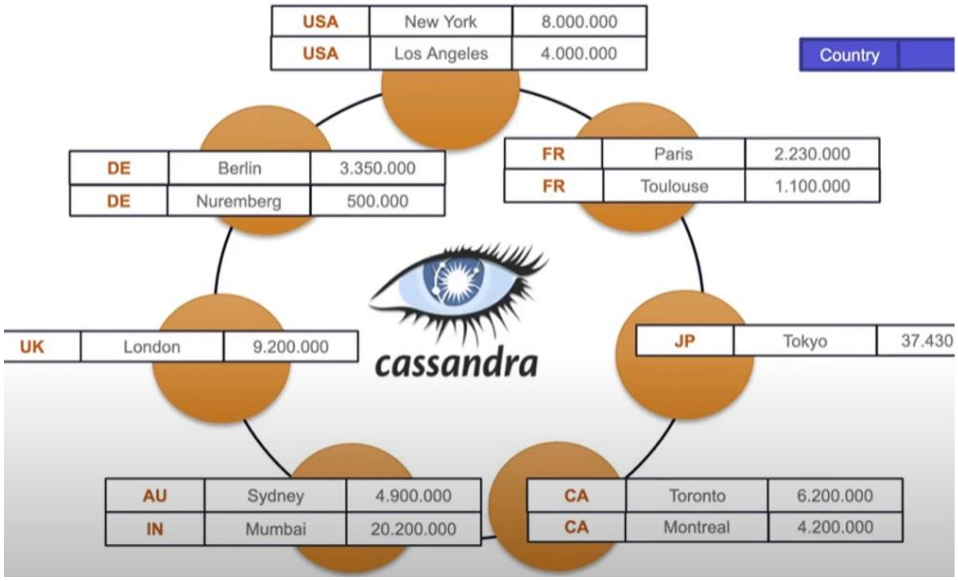


Figure 8. Data distribution between Cassandra nodes. [6]

When it comes to replication strategies, there are a few types to consider:

Table 2. Replication strategies available in Cassandra. [6]

Strategy name	Description
Simple Strategy'	Specifies a simple replication factor for the cluster.
Network Topology Strategy	Using this option, you can set the replication factor for each data-center independently.
Old Network Topology Strategy	This is a legacy replication strategy.

The main difference to consider is that Network Topology Strategy involves multiple datacentres and racks, it is a bit complicated, yet a safer option as if one datacentre or rack gets damaged, the entire work process does not stop, while Simple Strategy replicates data inside one Datacentre and one rack, which is a much riskier strategy.[6]

When it comes to working with Cassandra, the main tasks would be to create a KEYSPACE (storage of tables) and tables inside them:

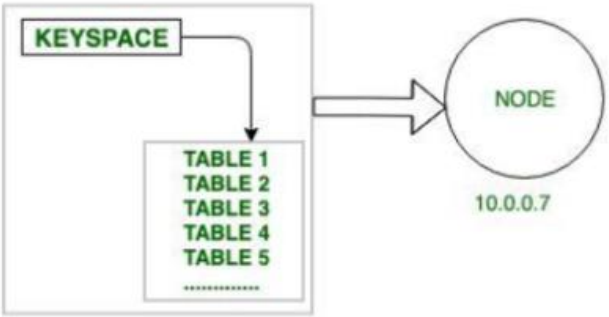


Figure 9. KEYSPACE and tables inside Cassandra node. [8]

In Cassandra, data replication is the process of duplicating data across nodes in the cluster to ensure redundancy, fault tolerance, and high availability. Instead of having a single point of failure, Cassandra treats all nodes equally and uses a replication factor (RF) configuration to specify the number of data copies to be stored across the cluster. By default, the replication factor is set to three replicas, but can be manually changed while defining the KEYSPACE.

When data is written to a node, it is first saved to a commit log on the local disk for durability and then stored in memory. The data is then written to one or more replicas according to the RF configuration. A consistent hashing algorithm is used to determine which nodes should store the data replicas based on the partition key. [9]

When a read request is made, the data is returned from the local replica. If the data is not available on the local replica, Cassandra coordinates a read repair to ensure that the data is up-to-date across all replicas. If a node fails, Cassandra uses a hinted handoff to temporarily store writes on other nodes until the failed node is recovered, and then the writes are replayed to the failed node.

4.4. Cassandra architecture

In order to fully understand how Apache Cassandra works, its architecture and main components must be overviewed.

Cassandra consists of the following key components: [9.]

- **Token** is a hashed value of a partition key. In other words, the partition key gets converted into a token value by the MurMur3 function (the default function can be changed by the user). Each node by default has 256 token ranges assigned to them and each node informs others what the ranges are. For example, a cluster of two nodes has the number of tokens values set to 3 and the token range 0 to 20. Node 1 randomly picks three values from the range: 4, 11, 17 and so does node 2: 2, 6, 14. Then Node 2 is responsible for token ranges 2-4, 6-11 and 14-17, while Node 1 is responsible for the rest.
- **Rack** is a logically grouped set of servers or nodes. The main purpose of the racks' existence in the Cassandra architecture is to make sure no replica is stored inside a singular rack as it would be risky if a rack goes down. The cluster sends replicas to different racks to ensure replicas are distributed among different logical groupings. A datacentre can have multiple racks.
- **Datacenter** is a logical set of racks. A/The datacenter must contain at least one rack. Datacenters reduce latency and save transactions from being influenced by other workloads.
- **Virtual node** is a data storage layer located inside a server with Cassandra installed. Typically, there are 256 virtual nodes on each server.

However, if tokens are unevenly distributed or a user wants a specific node to have more tokens than the others, then the number of virtual nodes can be increased or decreased on each of the nodes. In other words, a/the virtual node number can be used to manipulate how evenly the data should be distributed.

To illustrate how Cassandra is architectured in the case of this project, the following diagram was designed:

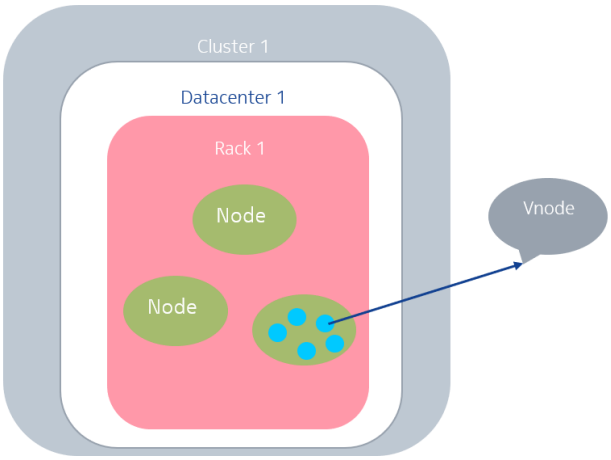


Figure 10. Cassandra architecture in the cluster.

This project's cluster has one datacenter with a single rack containing three nodes with the default 256 vnodes in each.

5. Apache Spark

5.1. Background

Apache Spark is an open-source platform for running data engineering, data science, and machine learning applications on either multi-node or single-node servers. It is designed to provide fast computation, scalability, and programming flexibility for large amounts of data. Spark has many functionalities and components, including GraphX that works with graph data, Spark Streaming, that works with streaming data, MLib that works with machine learning models and algorithms and finally Spark SQL that works with querying data. Originally, it was developed in 2009 by UC Berkeley and is now maintained by the Apache Software Foundation. Currently, Spark has a large open-source community with over 1,000 contributors. [10.]

Spark is often compared to Apache Hadoop, specifically to its MapReduce data-processing component. However, the main difference between Spark and MapReduce is that Spark stores data in memory, resulting in faster processing speeds without needing to read from or write to disks, unlike MapReduce. [10.]

In this thesis, Apache Spark serves the role of a data processing tool. Through Spark, data will be accessed and queried, and machine learning models will be applied to produce some forecasting or metrics that Spark will save to Cassandra or display. Spark is crucial to this cluster as just saving data is not enough, and there is a need for a toll that can easily access and manipulate the data.

Compared to alternatives, like Apache Hadoop, Splunk or Lumify, Spark has several advantages and was chosen based on them: [10.]

- **Speed**: The main value of Spark is its high speed when it comes to processing Big Data in real-time, whether it is static or streaming data. Spark was designed to be fast, mainly, due to its ability to store data in memory and process it in parallel across multiple nodes in a cluster.
- **Ease of use**: Spark is easy to use, with a simple and intuitive API that supports multiple programming languages including Python, Java, and Scala. Especially in comparison with its main alternative – Hadoop, which developers consider not friendly software for beginners due to its complicated interface and usability. This makes it accessible to a wide range of users with various levels of programming expertise.
- **Flexibility**: As mentioned before, Spark is flexible and can be used for a wide range of diverse tasks, including batch processing, streaming, machine learning, and graph processing, using its various components.
- **Fault-tolerant**: Spark is fault-tolerant, meaning it can handle failures in the cluster without losing data or causing downtime. This is achieved through RDDs, which are immutable and can be recreated in the event of a failure. This feature is similar to that of Cassandra and is one of the reasons these technologies were combined in the project. Both programs ensure that data or processes are distributed and in the case of a single failure the system will survive and keep doing its job.
- **Integration**: Spark integrates with a wide range of data sources and storage systems, including Hadoop Distributed File System (HDFS), Cassandra, and Amazon S3. This makes it easy to work with data stored in various locations.
- **Community**: Spark has a large and active community of developers, which means it is constantly evolving and improving. This community provides support, resources, and a wide range of libraries and tools that can be used with Spark.

In this project, Apache Spark serves a crucial role. Its main responsibility is to communicate with and connect a data source and data storage. Moreover, it will be used to access and process the data from Cassandra in order to execute different applications on it.

5.2. Resilient Distributed Datasets

RDDs (Resilient Distributed Datasets) are vital components in Apache Spark that enable the parallel processing of fault-tolerant collections of elements distributed across multiple nodes in a cluster. RDDs are static sets of items that store Python, Java, Scala, or user-defined objects. They were designed to overcome the limitations of MapReduce in data sharing by reducing the reliance on external storage systems such as HDFS, HBase, and Cassandra. RDDs leverage in-memory computing to improve data exchange speeds between tasks, leading to 10 to 100 times faster performance compared to MapReduce's read and write processes. By providing faster data processing, Spark RDDs simplify the handling of large data volumes for analytics and machine learning applications.[11]

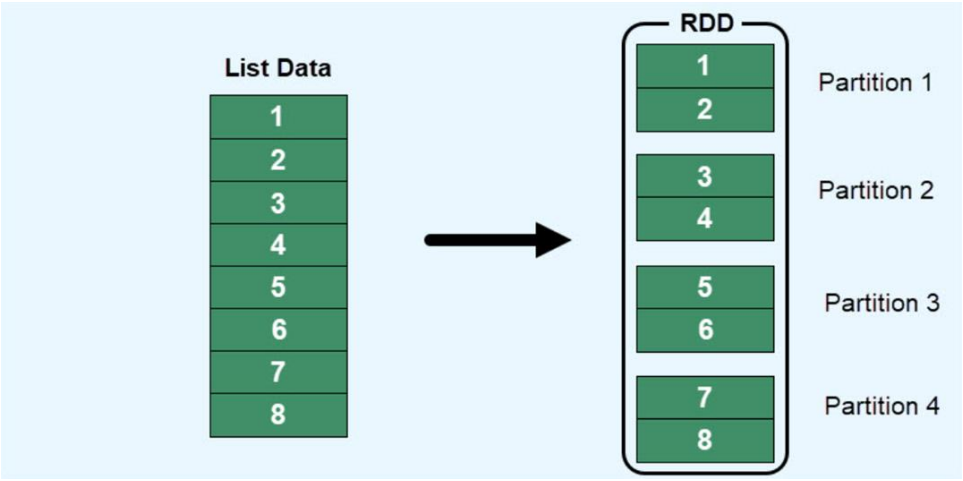


Figure 11. Highlight of the way RDDs work. [11]

5.3. Spark architecture

Spark is built on a cluster computing framework and has a master-worker architecture. The master node is responsible for coordinating tasks across worker nodes in the cluster. Each worker node runs multiple executors, which are responsible for running tasks on partitions of data.

The main components of Spark's architecture are:

- **Driver**: The driver program is the main entry point of Spark applications. It runs the main function and creates a SparkContext, which is used to coordinate tasks across the cluster.
- **Cluster Manager**: Spark supports multiple cluster managers such as Apache Mesos, Hadoop YARN, and Spark's built-in standalone cluster manager. The cluster manager’s main responsibility is to manage worker nodes and scheduling tasks.
- **Executors**: Executors are worker nodes that run tasks and store data on either memory or disk. Each executor runs in a separate JVM and can run multiple tasks concurrently. An executor can be co/located with the driver on the same node.
- **Tasks**: Tasks are units of work that are executed on data partitions by executors. Tasks can be shuffled between executors to optimize data locality and reduce network overhead.
- **Data Sources**: Spark supports various data sources such as HDFS, local file systems, Apache Cassandra, Apache HBase, and more. Data sources are used to read data into Spark or write data out of Spark.
- **APIs**: Spark provides APIs in various programming languages such as Scala, Java, Python, and R. These APIs are used to interact with Spark's core functionality, such as RDDs (Resilient Distributed Datasets), DataFrames, and Datasets.

Overall, Spark's architecture is designed to be scalable, fault-tolerant, and flexible to handle several types of workloads. The master-worker architecture allows Spark to distribute tasks across nodes in the cluster and process large datasets in parallel.

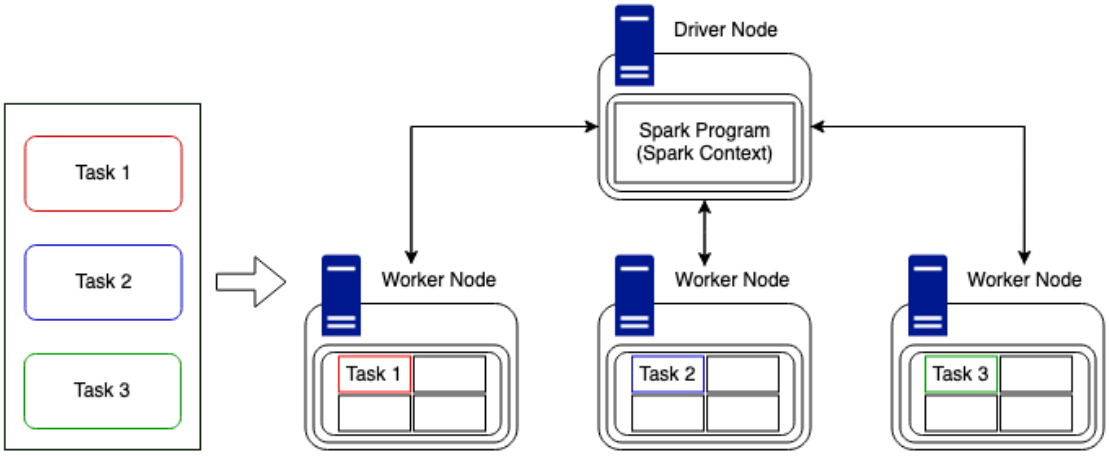


Figure 12. Example of Spark architecture with three executor and one driver nodes. [12]

5.4. RDD lineage

As previously mentioned, the Driver's primary responsibility is to convert user code into Tasks for Spark to execute. The Driver determines the number of tasks required based on the user code.

The lineage is the primary method that the Driver uses to define tasks. When a new RDD is created from an existing RDD using a transformation, it contains a reference to its parent RDD. The lineage keeps track of all these dependencies between RDDs. If data is lost, the Lineage is used to reconstruct the data. Spark internally converts DataFrames, datasets, and SQL queries to RDDs to perform computations as RDDs are the lowest level of abstraction in Spark.

Therefore, by converting them to RDDs, all the transformations involved in a DataFrame, Dataset, or SQL query can be observed.[13]

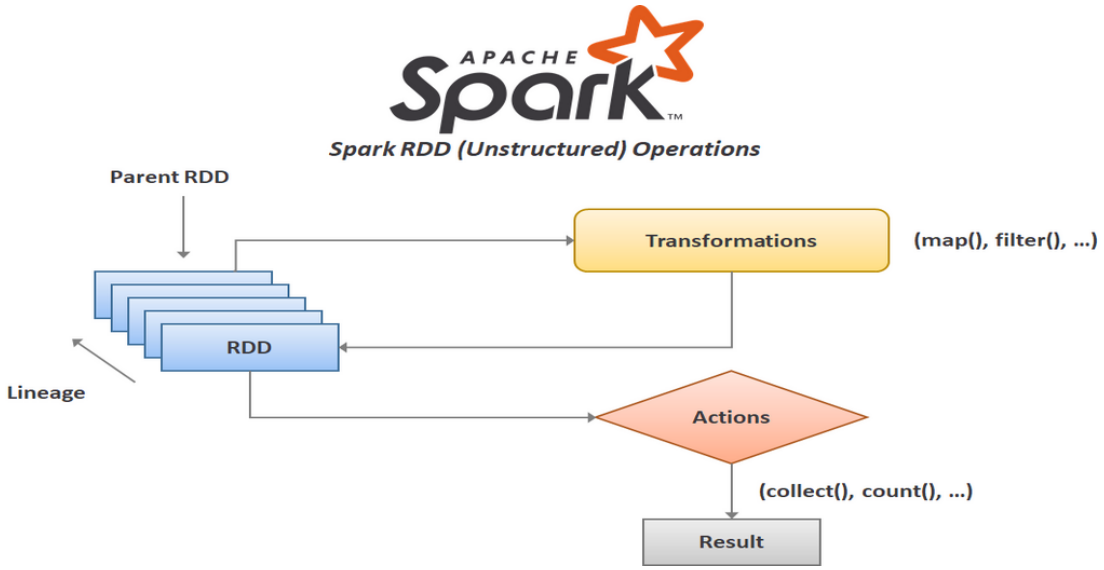


Figure 13. RDD lineage illustration. [13]

6. Apache Kafka

6.1. Background

Apache Kafka is a widely-used open-source distributed event streaming platform that was initially developed by LinkedIn in 2010. The Apache Software Foundation later took the ownership of it. It is designed to handle large volumes of real-time data streams, making it an ideal solution for applications that require large-scale, real-time data processing. Kafka has a fault-tolerant architecture, low latency, high throughput, and scalability. It is used for a wide range of purposes, such as real-time data processing, message queuing, log aggregation, and stream processing. It has become an essential element of many modern data architectures, particularly those dealing with big data and microservices. [14.]

Apache Kafka has several advantages, including:

- **High scalability**: Kafka is designed to handle enormous amounts of data and can scale horizontally by adding more servers to the cluster.

- **High throughput**: Kafka can handle high message throughput rates and is designed for efficient message delivery.

- **Fault tolerance**: Kafka is highly fault-tolerant and can continue to operate even if some nodes fail or go offline.

- **Durability**: Kafka can store messages for a long time, making it suitable for use in data retention and analytics applications.

- **Low latency**: Kafka can deliver messages with low latency, making it suitable for real-time data streaming applications.

- **Stream processing**: Kafka integrates well with stream processing frameworks like Apache Spark and Apache Flink, enabling real-time processing of data streams.

- **Flexibility**: Kafka is a versatile platform that can be used for a wide range of applications, including log aggregation, data integration, and messaging.

In this project, Apache Kafka has a direct role of receiving data from any streaming sources to pass it to Cassandra through Spark. [14.]

6.2. Kafka architecture

The Kafka messaging system is designed to handle high-volume data streams with low latency and high throughput. It consists of four main components: producers, brokers, topics, and consumers.

Producers publish data to Kafka topics, which are distributed log streams partitioned across Kafka brokers. Each partition is replicated for fault tolerance and availability. Brokers handle read and write requests from producers and consumers and manage partition replication and failover.

Consumers subscribe to Kafka topics and read data from partitions. They can be part of a consumer group that shares the load of consuming messages from the topic. Kafka also supports stream processing through the use of the Kafka Streams API.

Overall, Kafka's architecture is designed to provide scalable and fault-tolerant message processing for real-time applications, analytics, and event-driven systems.

Another important part of the Kafka architecture is the Zookeeper program. Apache Kafka ZooKeeper is a distributed coordination service used by Kafka to maintain configuration information and provide distributed synchronization. It is an integral part of Kafka's architecture and responsible for maintaining the state of the Kafka cluster.

Some of the tasks that ZooKeeper performs in Kafka are: [14.]

- **Cluster Management**: ZooKeeper keeps track of the state of all Kafka brokers, topics, and partitions.
- **Leader Election**: ZooKeeper is used to elect a leader for each partition in a Kafka cluster.
- **Configuration Management**: Kafka uses ZooKeeper to store configuration information, such as the location of Kafka brokers, topics, and partitions.
- **Synchronization**: ZooKeeper provides synchronization services to ensure that all Kafka brokers in a cluster are coordinated with each other.

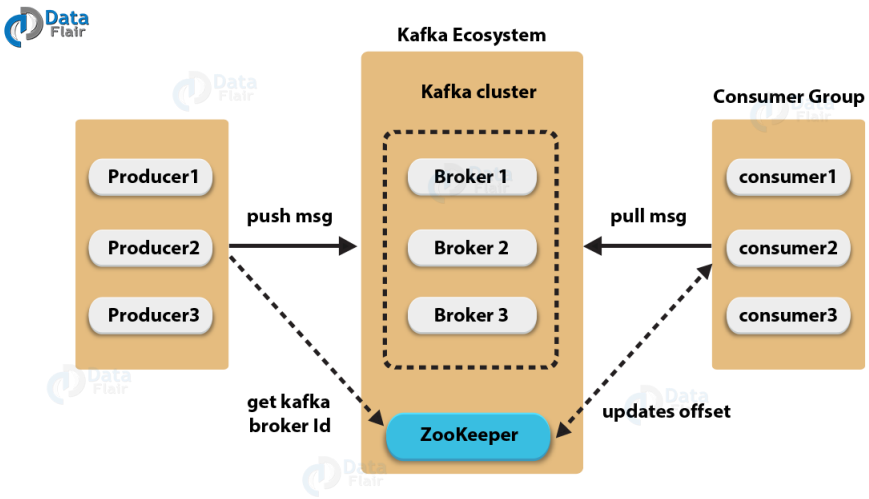


Figure 14. Example of Kafka architecture. [15]

In this project, Kafka will be installed onto all three virtual machines. Kafka will be receiving streaming data. Then Spark´s component, called Spark Streaming, will move the data to the Cassandra storage for later processing with Spark and Spark MLib. The producer could be any website or sensor or a script, while the consumer is the Cassandra database and each of the node serves as a broker.

There are several benefits of implementing Spark-Kafka Integration:

- By setting up the Spark Streaming and Kafka Integration, minimum data loss through Spark Streaming is ensured, while all the received Kafka data gets saved synchronously for the sake of an easy recovery.
- Users can access messages from a single topic or multiple Kafka topics.
- Along with this level of flexibility, high scalability, throughput and fault-tolerance, and a range of other benefits by using the combination of Spark and Kafka.

6.3. Kafka applicability in this project and beyond

Apache Kafka can take data input from many different types of sources. This includes sensors, websites and other systems. That is why installing Kafka is significant, as realistically, the developer does not always have an access to a large dataset; however, there are many sources available anywhere. In the case of Nokia, a great example would be Nokia Arena, located in Tampere. Nokia Arena has a lot of sensors placed around the stadium, which are a reliable source of data. According to Nokia´s plan, this system will be connected to some of Arena´s sensors in order to collect data. That is why it is important to add Kafka to this project and not just keep Spark and Cassandra, which only operate with static data.

7. Cluster set up

In this section, a step-by-step explanation of how to set up the cluster is presented. Each of three programs has its own section with detailed information and figures describing the installation. As the goal of this thesis is not only to study the technologies, but also to provide an updated guide on how to properly set up and utilize the systems, all the challenges and benefits will be described as well.

7.1. Initial resources

For this project, Nokia provided three virtual machines, two with 3 TB of storage and one with 5 TB. All machines have the Ubuntu server 22.04 LTS with unique IP addresses and hardware located in the Tampere office.

7.2. Cassandra setup

The first step is to access one of the virtual machines. As Cassandra does not have master-slave communication, the choice of the first machine does not make any difference. This is usually done with a similar command in the terminal:

ssh <username>@<ip address>

After inputting the password, the next step would be installing Java, the newest Cassandra versions 4.0+ only work with Java 11, while Cassandra 3 works with Java 8.

As the purpose of this thesis is to explore the newest versions of the software, and their compatibility, the obvious choice is Java 11 and Cassandra4:

sudo apt update

sudo apt install default-jdk

java --version

```
ubuntu@s1697andrey01:~$ java --version
openjdk 11.0.16 2022-07-19
OpenJDK Runtime Environment (build 11.0.16+8-post-Ubuntu-0ubuntu122.04)
OpenJDK 64-Bit Server VM (build 11.0.16+8-post-Ubuntu-0ubuntu122.04, mixed mode, sharing)
```

Figure 15. Checking Java version after installing.

Then, in order to ensure that the node can access repositories through the HTTPS protocol, the following command need to be implemented:

sudo apt install apt-transport-https

After that, it is required to export the GPG public key for the Cassandra package repository to convert it later into binary format and to save it:

```
ubuntu@s1697andrey02:~$ gpg --keyserver keyserver.ubuntu.com --recv-keys 7E3E87CB
gpg: directory '/home/ubuntu/.gnupg' created
gpg: keybox '/home/ubuntu/.gnupg/pubring.kbx' created
gpg: /home/ubuntu/.gnupg/trustdb.gpg: trustdb created
gpg: key E91335D77E3E87CB: public key "Michael Semb Wever <mick@thelastpickle.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
ubuntu@s1697andrey02:~$ gpg --export --armor 7E3E87CB | sudo gpg --dearmour -o /etc/apt/trusted.gpg.d/cassandra-key.gpg
```

Figure 16. Preparations for Cassandra installation.

Now, when the preparations are over, it is time to install Cassandra itself:

sudo sh -c 'echo "deb http://www.apache.org/dist/cassandra/debian 40x main" > /etc/apt/sources.list.d/cassandra.list'

Then, the following command should be run:

```
ubuntu@sl697andrey02:~$ sudo apt install cassandra
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  cassandra-tools
The following NEW packages will be installed:
  cassandra
0 upgraded, 1 newly installed, 0 to remove and 17 not upgraded.
Need to get 47.5 MB of archives.
After this operation, 58.8 MB of additional disk space will be used.
Get:1 https://downloads.apache.org/cassandra/debian 40x/main amd64 cassandra all 4.0.5 [47.5 MB]
Fetched 47.5 MB in 10s (4,875 kB/s)
Selecting previously unselected package cassandra.
(Reading database ... 125555 files and directories currently installed.)
Preparing to unpack .../cassandra_4.0.5_all.deb ...
Unpacking cassandra (4.0.5) ...
Setting up cassandra (4.0.5) ...
vm.max_map_count = 1048575
net.ipv4.tcp_keepalive_time = 300
update-rc.d: warning: start and stop actions are no longer supported; falling back to defaults
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
```

Figure 17. Cassandra installation

Additionally, Cassandra's activation can be enabled automatically with a system boot in the following way:

sudo systemctl enable cassandra

To manually start Cassandra, the following command can be used:

sudo systemctl start cassandra

```
ubuntu@sl697andrey01:~$ sudo systemctl enable cassandra
cassandra.service is not a native service, redirecting to systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable cassandra
ubuntu@sl697andrey01:~$ sudo systemctl start cassandra
```

Figure 18. Automated initialization of Cassandra with system boot

In order to make sure that Cassandra operates properly, its status can be checked:

```
ubuntu@sl697andrey01:~$ sudo systemctl status cassandra
● cassandra.service - LSB: distributed storage system for structured data
     Loaded: loaded (/etc/init.d/cassandra; generated)
     Active: active (running) since Tue 2022-08-30 12:42:11 UTC; 5min ago
       Docs: man:systemd-sysv-generator(8)
      Tasks: 57 (limit: 77041)
     Memory: 8.4G
        CPU: 19.783s
     CGroup: /system.slice/cassandra.service
             └─12797 /usr/bin/java -ea -da:net.openhft... -XX:+UseThreadPriorities -XX:+HeapDumpOnOutOfMemoryError

Aug 30 12:42:11 sl697andrey01 systemd[1]: Starting LSB: distributed storage system for structured data...
Aug 30 12:42:11 sl697andrey01 systemd[1]: Started LSB: distributed storage system for structured data.
lines 1-12/12 (END)
```

Figure 19. Cassandra status.

If everything was set up properly, then the status should be **active(running).**

In order to check the list of nodes and the status of each, the following command should be run:

```
ubuntu@sl697andrey02:~$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns (effective)  Host ID                               Rack
UN  127.0.0.1  79.53 KiB  16      100.0%            a4741b91-6ac4-45f7-b0a4-908e298dc31c  rack1
```

Figure 20. List of current Cassandra nodes.

Currently, there is only one node installed, and later more will appear. Moreover, in order to work with Cassandra data, the user needs to use Cassandra query language (cql), with the following command:

```
ubuntu@sl697andrey01:~$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.0.0 | Cassandra 4.0.5 | CQL spec 3.4.5 | Native protocol v5]
Use HELP for help.
cqlsh> exit
```

Figure 21. Entering Cassandra terminal.

Inside this terminal, the user can create KEYSPACES and modify the replication and the partition strategies of tables inside the KEYSPACES. These features and the cql language will be overviewed later in the thesis.

Also, launching the Cassandra shell is a good way to check if Cassandra nodes are properly working, as if there are some issues, the terminal will not be opened.

Now, after the previous steps were successfully implemented, the **same steps need to be repeated on the remaining two virtual machines**.

After all the virtual machines have Cassandra nodes installed, the following steps must be completed on each of them:

A backup copy of the main configuration file must be made to make sure that if anything goes wrong with the configurations, the default version is still available in another file:

sudo cp /etc/cassandra/cassandra.yaml /etc/cassandra/cassandra.yaml.backup

Then, the next step would be to modify the configuration file:

sudo nano /etc/cassandra/cassandra.yaml

The configuration file is quite massive, yet the most significant part of it is called **seed_provider**, where all the IP addresses of all nodes must be inputted to ensure the connection between them.

```
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and learn
    # the topology of the ring.  You must change this if you are running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "10.10.[redacted],10.10.[redacted],10.10.[redacted]"
```

Figure 22. Modifying the seed_provider section in the configuration file.

The next section that needs to be modified in the configuration file is **listen_address**, where the default localhost needs to be replaced with the unique IP (Internet Protocol) address of the single machine that the file belongs to. If it were a single node cluster, this step would not be required, yet in the case of a multi-node standalone cluster, it is a crucial step.

```
listen_address: localhost
```

Figure 23. Default listen_address setting.

After the steps are completed, it is necessary to restart the system on each virtual machine:

sudo systemctl restart cassandra

sudo systemctl start cassandra

Now, by running the **nodetool status** command, all nodes must appear on the list!

```
ubuntu@sl697andrey03:~$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load        Tokens  Owns (effective)  Host ID                               Rack
UN  10.10.[redacted]  222.3 KiB   16      66.5%             e680ca04-ef3a-4839-af9d-0e6dbedc002c  rack1
UN  10.10.[redacted]  112.23 KiB  16      75.5%             2fa6eac8-8abe-4067-8f44-d7dcf8af503d  rack1
UN  10.10.[redacted]  155.09 KiB  16      58.0%             a4741b91-6ac4-45f7-b0a4-908e298dc31c  rack1
```

Figure 24. Updated nodetool status with all three nodes connected.

In the end, there is now a cluster of three Cassandra nodes. Later, using **cqlsh**, a KEYSPACE can be created with tables, where such parameters as replication factor and partitioning strategies will be configured. At the moment, the cluster can be illustrated in the following way, with a simple example of how Cassandra operates:

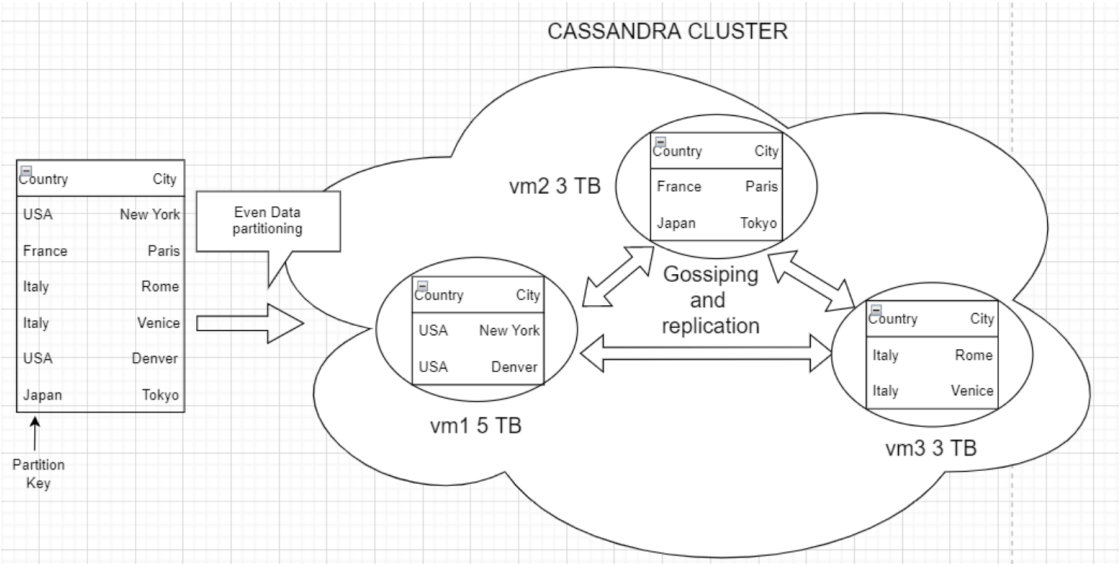


Figure 25. Current cluster diagram.

7.3. Spark setup

The following steps need to be repeated on **each virtual machine**. When it comes to Spark, besides the already installed Java, Scala is required as well. It can be easily installed with the following command:

sudo apt install scala

Then, the archive with Spark can be downloaded and the necessary files can be extracted and moved to another directory:

wget https://dlcdn.apache.org/spark/spark-3.3.0/spark-3.3.0-bin-hadoop3.tgz

tar xvf spark-*

sudo mv spark-3.3.0-bin-hadoop3 /opt/spark

cd /opt/spark

Then, the **bashrc** file needs to be changed in the following way:

echo "export SPARK_HOME=/opt/spark" >> ~/.bashrc

echo "export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin" >> ~/.bashrc

echo "export PYSPARK_PYTHON=/usr/bin/python3" >> ~/.bashrc

The next step is to reload the file:

source ~/.bashrc

To test, if Spark was configured properly, spark-shell can be launched and simple calculations can be run:

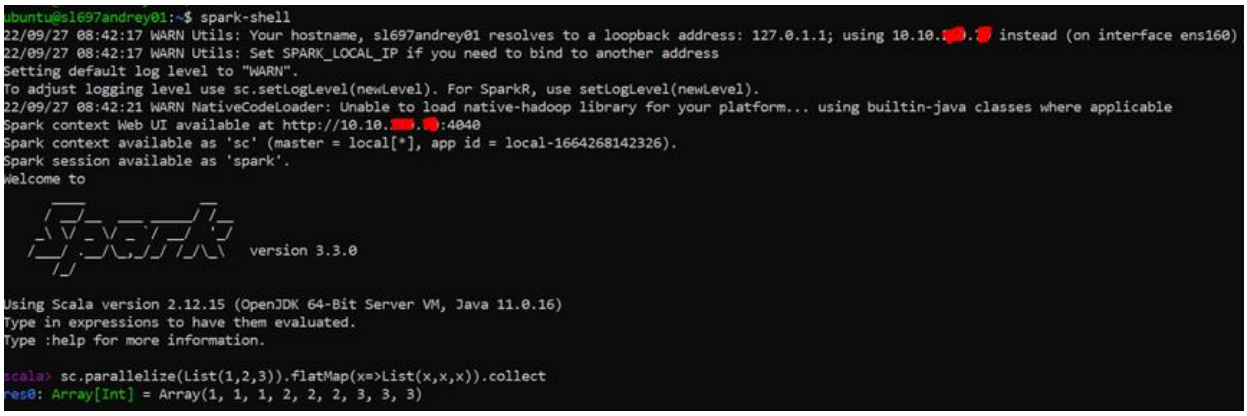


Figure 26. Spark shell on Scala.

Spark has another alternative to its original shell, which is the PySpark shell that runs on Python and can be used for the same functions as the original one.

```
ubuntu@sl697andrey01:~$ pyspark
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
22/09/19 13:30:28 WARN Utils: Your hostname, sl697andrey01 resolves to a loopback address: 127.0.1.1; using 10.10.[redacted] instead (on interface ens160)
22/09/19 13:30:28 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/09/19 13:30:29 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.3.0
      /_/

Using Python version 3.10.4 (main, Jun 29 2022 12:14:53)
Spark context Web UI available at http://10.10.[redacted]:4040
Spark context available as 'sc' (master = local[*], app id = local-1663594229867).
SparkSession available as 'spark'.
>>> x = 10
>>> print(x)
10
>>> quit()
```

Figure 27. PySpark shell.

Then, it is important to decide, which node will be the master. According to the Spark architecture, there are executors and driver nodes, where executors are workers and the driver is the master. In this case, the virtual machine with the biggest storage space (5TB) will have both a driver drivers and an/the executor, while machines with 3TB of space will only have executors on them. Spark allows having both executor and driver programs on the same machine. This will make the studied cluster considerably more efficient as three executors will be present.

After deciding on the master node, firstly, **on all nodes** the hosts file needs to be modified:

sudo nano /etc/hosts

Inside the file, all ip addresses of nodes must be added with the according name (master, slave01, slave02):

```
127.0.0.1 localhost
127.0.1.1 sl697andrey02
10.10.[redacted] master
10.10.[redacted] slave01
10.10.[redacted] slave02
```

Picure 28. Hosts file modifications.

In order to ensure, that all nodes can communicate and recognise virtual machine 1 as the host, the environment file needs to be configured, by firstly moving to the **conf** directory and then opening the environment file:

cd /opt/spark/conf

sudo nano spark-env.sh

Inside the file, only two variables need to be changed. Instead of the ip address and port number, the actual values need to be written.

```
SPARK_MASTER_HOST='10.10.[redacted]'
SPARK_MASTER_PORT='7077'
```

Figure 29. Master ip and port configuration.

Port 7077 is the default communication port that Spark uses. It is recommended to keep it the same; however, it could be changed if needed.

Now, the nodes can finally be connected! To achieve that, the master node needs to start the master process:

start-master.sh

In order to stop the master process, the following command can be used:

stop-master.sh

```
ubuntu@sl697andrey01:~$ start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /opt/spark/logs/spark-ubuntu-org.apache.spark.deploy.master.Master-1-sl697andrey01.out
ubuntu@sl697andrey01:~$ stop-master.sh
```

Figure 30. Starting and stopping master process.

After starting the master process, the worker nodes can be connected. After opening the terminals of two other virtual machines, the following command can be run:

start-worker.sh spark://<ip>:<port>

Further, it can be stopped in this way:

stop-worker.sh

```
ubuntu@sl697andrey02:~$ start-worker.sh spark://10.10.[redacted]:7077
starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/logs/spark-ubuntu-org.apache.spark.deploy.worker.Worker-1-sl697andrey02.out
ubuntu@sl697andrey02:~$ stop-worker.sh
```

Figure 31. Starting and connecting the worker node.

At this moment, the master node is initialized and the workers should be connected. In order to confirm the connection, Spark WebUI can be used. Whenever a master process is initialized, a local web UI initializes as well. It can be accessed by opening a web browser and forwarding the following URL address:

spark://<ip address>:<port>

This URL will open up a web interface that lists all nodes in the cluster and recently executed applications:

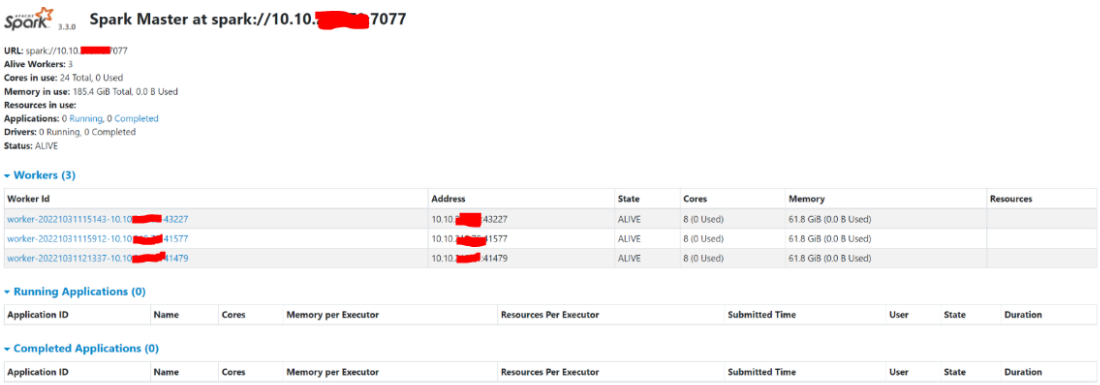


Figure 32. Spark Web Interface.

Now it is confirmed that all three nodes are communicating. In the case of the Nokia network, there was an issue with accessing the web interface due to numerous firewalls and proxies. In order to solve the issue, the reverse proxy from ngnix was used, to connect the web browser of the laptop with the Web UI of virtual machines, as the original port was blocked. However, in a domestic environment, there should be no such issues.

Figure 33 shows what the updated cluster diagram looks like after installing and connecting the Spark nodes on virtual machines:

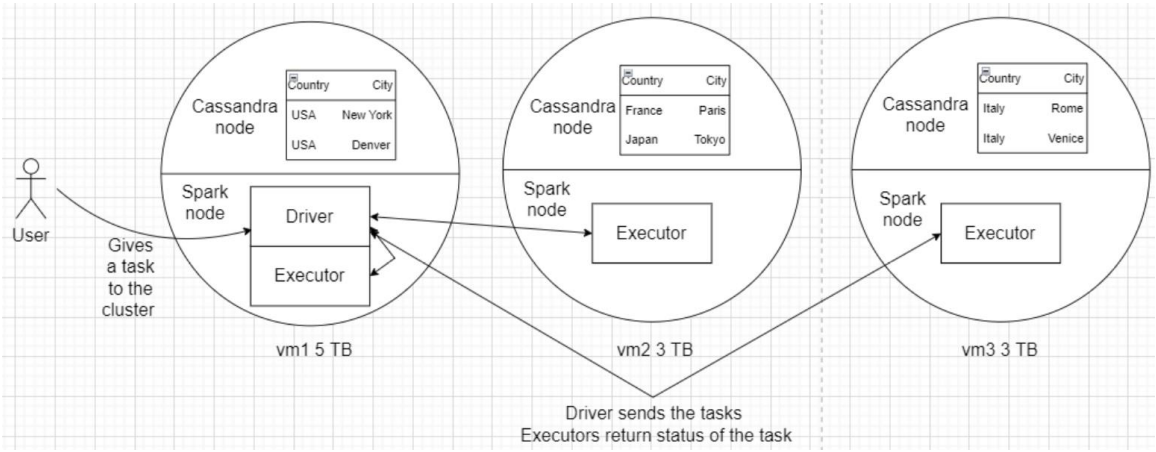


Figure 33. Cluster diagram after adding Spark nodes.

7.4. Kafka setup

Last, but not least, Kafka needs to be added to the system. **For each virtual machine**, firstly, a tar file needs to be downloaded:

wget http://mirror.cogentco.com/pub/apache/kafka/2.8.2/kafka_2.12-2.8.2.tgz

Secondly, the files need to be extracted and moved to the opt directory:

tar -xvf kafka_2.12-2.8.2.tgz
sudo mv kafka_2.12-2.8.2 /opt/kafka

Thirdly, after moving to the **conf** directory, the **server.properties** file needs to be modified:

cd /opt/kafka/config
sudo nano server.properties

Inside the file, a few things need to be changed. Firstly, the unique broker ID needs to be set for each virtual machine.

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1
```

Figure 34. Unique broker ID for virtual machine 1.

Then, the location of the log directory (preferably an empty directory somewhere on machine) must be inputted:

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/opt/kafka/logs
```

Figure 35. The location of logs directory.

The next step is to list all IP addresses of nodes with port 2181, which is a default port for such connection:

```
############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.10.[redacted]:2181,10.10.[redacted]:2181,10.10.[redacted]:2181
```

Figure 36. IP addresses in server properties file.

This was the last step for this file. The next one that needs to be modified is located in the same folder and is called **zookeeper.properties**. This file sets zookeeper properties. Inside it a few things need to be changed.

Firstly, **dataDir** needs to be pointed to any empty directory to store the zookeeper data.

Secondly, the **clientPort** needs to be assigned to 2181 as it was in the server properties file to ensure connection.

Another two values that are required to be set are **initLimit** and **syncLimit**. InitLimit is responsible for the amount of time ticks (2 seconds periods) that Zookeeper servers let followers to connect with the leader when they were just added to the ensemble. While syncLimit specifies the amount of time (in ticks) that the Zookeeper servers allow followers to synchronise with the leader, after they already joined the ensemble. For such a system, good values would be 10 ticks (20 seconds) for initLimit and 5 ticks (10 seconds) for syncLimit.

Last, but not least, IP addresses need to be written down for zookeeper servers, with the IP of each machine that user inputs this information from being 0.0.0.0. Also the ports are recommended to be default 2888:3888. The rest of the file can be kept as it is.

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# the directory where the snapshot is stored.
dataDir=/opt/kafka/zookeeper_data
# the port at which the clients will connect
clientPort=2181
# disable the per-ip limit on the number of connections since this is a non-production config
maxClientCnxns=0

initLimit=10
syncLimit=5
# Disable the adminserver by default to avoid port conflicts.
# Set the port to something non-conflicting if choosing to enable this
admin.enableServer=false
# admin.serverPort=8080

server.1=0.0.0.0:2888:3888
server.2=10.10.[redacted]:2888:3888
server.3=10.10.[redacted]:2888:3888
```

Figure 37. File configuration of zookeeper properties.

Now, on any of the nodes, the zookeeper can be started:

sudo bin/zookeeper-server-start.sh /opt/kafka/kafka_2.12-2.8.2/config/zookeeper.properties

This command will launch the zookeeper instance. Then, another terminal of the same machine needs to be opened and the Kafka server can be started:

sudo bin/kafka-server-start.sh config/server.properties

Now Kafka and zookeeper are added to the system. The next step would be to design the data pipeline and check if Kafka functionality works. These observations and experiments are described in later chapters of this thesis.

8. Final changes and experiments

8.1. Cassandra data replication and partitioning

After installing Cassandra, its functionality can be tested on static datasets. After entering the cql terminal, a new KEYSPACE can be created. In Cassandra the KEYSPACE is a storage of the column families of related data or in this case tables. When defining the KEYSPACE, it is highly recommended to set its replication and to set the properties, which are the class and the replication factor:

- The class variable is responsible for the replica placement strategy. By default it is recommended to use 'SimpleStrategy', but depending on the task, other options can be chosen.
- Replication factors specify the number of replica nodes. In this case there are three nodes in the cluster, and it would be wise to replicate data to each of them.

```
cqlsh> CREATE KEYSPACE Atg_test
   ... WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor' : 3};
cqlsh> desc KEYSPACES;

atg_test  system       system_distributed  system_traces  system_virtual_schema
library   system_auth  system_schema       system_views
```

Figure 38. KEYSPACE gets created.

The command **desc KEYSPACES** lists all keyspaces which include the default ones and the newly created one.

The cql language is very similar to SQL and it is quite intuitive, so there is no real need in learning anything extra. The next step is to enter the KEYSPACE, which is not a necessary step, but this way the user specifies that the next commands will be related to this specific keyspace, which will save the user from extra writing and some potential errors:

USE <KEYSPACE name>

```
cqlsh> USE atg_test;
```

Figure 39. Entering the test KEYSPACE.

Then, a table can be created inside the KEYSPACE. The user can assign any number of columns in the table, but it has to specify the input format (int, text...). Moreover, the primary key needs to be specified as well. The primary key is the main column based on which the data will be organized in the table. An example of the primary key could be an index (1,2,3,4,5...) or any of the data columns. As an option multiple primary keys can be set:

```
cqlsh:atg_test> CREATE TABLE IF NOT EXIST Atg_test.atg_projects
( id int, title text, number_of_interns int, PRIMARY KEY (field)
 );
```

Figure 40. Creating a table inside a KEYSPACE.

In this example, a simple table was created. The table includes an ID, the title of a project inside the Nokia group, a number of interns assigned to the project

and the field. Then, a user can manually insert data rows inside the table, in the following format:

```
cqlsh:atg_test> INSERT INTO ATG_test.atg_projects (id, title, number_of_interns, field)
 VALUES(1, 'time series analysis', 3, 'ML')
```

Figure 41. Manually inserting data into a table.

After inserting the required data, an entire table can be overviewed:

select * from <KEYSPACE name>.<table name>

```
cqlsh> select * from atg_test.atg_projects;

 field  | id | number_of_interns | title
--------+----+-------------------+-----------------------
     UX |  3 |                 4 |               Graph UX
     UX |  4 |                 1 |         Augmented Maps
     ML |  1 |                 3 |   time series analysis
     ML |  2 |                 2 |  time series synthesis
     ML |  6 |                 2 |       Causal discovery
 DevOps |  5 |                 1 | Infra & tools projects
```

Figure 42. Table content display.

As ‘field’ was chosen to be a primary key, the table is structured accordingly, where similar fields are close to each other.

Now, it would be a good idea to check that the table is partitioned (split) among all three nodes. This can be checked with the following tokens:

select token(<column name>), <column name> from <table name>;

The tokens get displayed the following way:

```
cqlsh:atg_test> select token(field), field from atg_projects;

 system.token(field)  | field
----------------------+--------
 -7184959072001002467 |     UX
 -7184959072001002467 |     UX
  2484577390804984672 |     ML
  2484577390804984672 |     ML
  2484577390804984672 |     ML
  4855069840617214231 | DevOps

(6 rows)
```

Figure 43. Listing tokens assigned to each row.

As it can be seen in figure 43 above, data is indeed split between three unique tokens, depending on the field value. This means that each of the three nodes has one of these tokens assigned to. The next thing to check would be if the data were replicated among the nodes. Firstly, cqlsh needs to be closed with the **exit** command, then in the terminal, the following command will help:

```
cqlsh:atg_test> exit
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects -7184959072001002467
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects 2484577390804984672
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects 4855069840617214231
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
```

Figure 44. List of the token endpoints.

In the figure above, it can be clearly seen that each of the tokens is present on all of the three nodes, with the original ones listed first. This means that, for example, UX project data rows were originally placed onto the very first virtual machine with 5 TB of space and then they are replicated to the other two nodes. Now, by opening another terminal and logging into another virtual machine, the KEYSPACE and the table with all the data will be displayed and will be accessible there.

Obviously, inserting a huge amount of data manually is not convenient at all, so data can be copied to Cassandra from other storages or files with just a few steps.

Inside a new KEYSPACE or in the existing one, it is necessary to create an empty table for the data. Then, the data can be copied from a file into the empty table:

```
cqlsh:testcsv> CREATE TABLE testcsv.oil_wti_yearly ( Date text PRIMARY KEY, Price text);
cqlsh:testcsv> desc tables

oil_wti_yearly

cqlsh:testcsv> COPY testcsv.oil_wti_yearly (Date, Price)  FROM 'datasets/oil-prices/data/wti-year.csv' WITH HEADER = TRUE;
Using 7 child processes

Starting copy of testcsv.oil_wti_yearly with columns [date, price].
Processed: 36 rows; Rate:      57 rows/s; Avg. rate:      86 rows/s
36 rows imported from 1 files in 0.416 seconds (0 skipped).
```

Figure 45. New table for CSV data created inside a new KEYSPACE.

The small dataset has a some time series data of oil prices. All the data was copied into the table and now can be accessed:

```
cqlsh:testcsv> select * from testcsv.oil_wti_yearly;

 date       | price
------------+-------
 1986-06-30 | 15.05
 2013-06-30 | 97.98
 2006-06-30 | 66.05
 1996-06-30 | 22.12
 2020-06-30 | 39.16
 2007-06-30 | 72.34
 2002-06-30 | 26.18
 2018-06-30 | 65.23
 2005-06-30 | 56.64
 2001-06-30 | 25.98
 1988-06-30 | 15.97
 1998-06-30 | 14.42
 2004-06-30 | 41.51
 2015-06-30 | 48.66
 1999-06-30 | 19.34
 2012-06-30 | 94.05
 1995-06-30 | 18.43
 2010-06-30 | 79.48
 2009-06-30 | 61.95
 1990-06-30 | 24.53
 1994-06-30 |  17.2
 2003-06-30 | 31.08
 1997-06-30 | 20.61
 2016-06-30 | 43.29
 2019-06-30 | 56.99
 2008-06-30 | 99.67
 1992-06-30 | 20.58
 1993-06-30 | 18.43
 1991-06-30 | 21.54
 2014-06-30 | 93.17
 1989-06-30 | 19.64
 2021-06-30 | 68.13
 2017-06-30 |  50.8
 2000-06-30 | 30.38
 1987-06-30 |  19.2
 2011-06-30 | 94.88

(36 rows)
```

Figure 46. Copied dataset inside the table.

Concluding this part, Cassandra was installed and configured perfectly, with all replications and partitions working. This means that Cassandra is ready for storing and modifying the future Big Data.

8.2. Spark-Cassandra connector and Jupyter server

After installing and setting the Spark nodes, they must be connected to the Cassandra database in order to ensure the processing of the data. The most reliable way of doing this is to install an open-source connector. Firstly, **only on the master node**, the GitHub repositorium of the connector needs to be cloned and entered into:

```
ubuntu@sl697andrey01:~$ git clone https://github.com/datastax/spark-cassandra-connector
Cloning into 'spark-cassandra-connector'...
remote: Enumerating objects: 45759, done.
remote: Counting objects: 100% (1805/1805), done.
remote: Compressing objects: 100% (588/588), done.
remote: Total 45759 (delta 1041), reused 1741 (delta 992), pack-reused 43954
Receiving objects: 100% (45759/45759), 17.65 MiB | 2.50 MiB/s, done.
Resolving deltas: 100% (21832/21832), done.
ubuntu@sl697andrey01:~$ ls
datasets  import_testcsv_oil_wti_yearly.err.20220913_135451  spark-3.3.0-bin-hadoop3.tgz  spark-cassandra-connector
ubuntu@sl697andrey01:~$ cd spark-cassandra-connector
```

Figure 47. Clonning repositorium of spark-cassandra connector.

For the future steps, the jar files need to be compiled with the help of the following command:

./sbt/sbt assembly

Now, there are many jar files in different folders, mainly Java and Scala ones. For this case, the Scala one is needed. It can be a challenge to find the exact location of the jar file as the documentations available online are that of old versions. In the end, the needed jar file was found in the scala-2.21 folder:

```
ubuntu@sl697andrey01:~/spark-cassandra-connector/connector/target$ cd scala-2.12/
ubuntu@sl697andrey01:~/spark-cassandra-connector/connector/target/scala-2.12$ ls
classes  resolution-cache  spark-cassandra-connector-assembly-3.2.0-2-g92ba7ba0.jar
```

Figure 48. Scala jar file location.

After locating the jar file, it needs to be relocated to the jars folder in Spark:

sudo mv spark-cassandra-connector-assembly-3.2.0-2-g92ba7ba0.jar /opt/spark/jars

The next step is to test the connection through spark-shell. Firstly, the spark shell must be opened:

spark-shell –jars ~/jars/spark-cassandra-connector-assembly-3.2.0-2-g92ba7ba0.jar

Then, inside the shell, the default spark context must be stopped and necessary libraries imported:

sc.stop

import com.datastax.spark.connector._, org.apache.spark.SparkContext, org.apache.spark.SparkContext._, org.apache.spark.SparkConf

```
scala> sc.stop

scala> import com.datastax.spark.connector._, org.apache.spark.SparkContext, org.apache.spark.SparkContext._, org.apache.spark.SparkConf
import com.datastax.spark.connector._
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

Figure 49. Spark context stopped and libraries installed.

Secondly, a new Spark Context should be started:

```
scala> val conf = new SparkConf(true).set("spark.cassandra.connection.host", "localhost")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@41366811

scala> val sc = new SparkContext(conf)
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@7969e0cb
```

Figure 50. New Spark context created.

At this point, it is necessary to check if Spark has access to Cassandra tables. To test it, a table variable can be created to store the Cassandra table data:

```
scala> val table = sc.cassandraTable("atg_test", "atg_projects")
```

Figure 51. Accessing the Cassandra table through Spark.

If this gives no errors, Spark has successfully been able to access the data! To confirm this, some metrics can be gathered from the data, for example the table length (number of rows):

```
scala> println(table.count)
6
```

Figure 52. Outputting the table length.

This means that Spark has access to all Cassandra data. This means that any script can be run on the Cassandra data using the Spark shell or the spark-submit command.

However, what if a Data Scientist wants to run some more complicated experiments, visualizations and model training on the data? The Spark shell is not a convenient option due to a poor interface. The perfect solution would be to connect Spark and Cassandra to the Jupyter server. Connecting can be easily implemented through the **bashrc** file. The plan is to install Jupyter Notebook and start a local server that can be accessible through a URL just like WebUI. To make it easier, the **pyspark** command will be modified to launch the server instead of opening the Python shell that is not necessary for the system.

Firstly, the notebook must be installed:

pip install notebook

Secondly, new lines must be added to the **bashrc** file, replacing the old ones:

sudo nano ~/.bashrc
export PYSPARK_DRIVER_PYTHON='jupyter'
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port=8889'

These lines point to Jupyter and specify the port number for the new Jupyter interface.

```
export PYSPARK_DRIVER_PYTHON='jupyter'
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port=8889'
```

Figure 53. Bottom part of the bashrc file.

Finally, by simply running the **pyspark** command in the terminal, the Jupyter Notebook will be launched and the URL will be given at the end:

```
ubuntu@sl697andrey01:~$ pyspark
[I 13:39:04.428 NotebookApp] Writing notebook server cookie secret to /home/ubuntu/.local/share/jupyter/runtime/notebook_cookie_secret
[I 13:39:04.623 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 13:39:04.623 NotebookApp] Jupyter Notebook 6.5.2 is running at:
[I 13:39:04.623 NotebookApp] http://localhost:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
[I 13:39:04.623 NotebookApp]  or http://127.0.0.1:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
[I 13:39:04.623 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 13:39:04.626 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/ubuntu/.local/share/jupyter/runtime/nbserver-2644405-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
     or http://127.0.0.1:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
```

Figure 54. Jupyter Notebook launched.

Now, the notebook can be connected to the cluster by simply inputting the following lines to the code:

```
import os
import pyspark
from pyspark.sql import SQLContext, SparkSession
from pyspark import SparkConf, SparkContext
import random as rnd
```

```
conf = SparkConf().setAppName("test")
sc.stop() #stop the previous/default spark context
sc = SparkContext('spark://10.10.219.79:7077', conf=conf)
```

Figure 55. Connecting to the master node.

Concluding this section, now Saprk and Cassandra are connected and communicating. Also, there is a Saprk Web UI along with Jupyter Notebook ready to be used by anyone anytime.

8.3. Kafka functionality and Spark Streaming-Kafka connector

After installing Kafka, it still needs to be tested to ensure that its functionality is intact. Firstly, a topic can be created, with the following command:

bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092

This command creates a topic named quickstart-events. Then, it would be a good idea to ask Kafka to describe the topic to observe the main settings with the following command:

bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092

```
ubuntu@s1697andrey01:/opt/kafka/kafka_2.12-2.8.2$ bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092
Topic: quickstart-events        TopicId: m_ovSZgiSMS5cs7KTQIfBg PartitionCount: 1       ReplicationFactor: 1    Configs: segment.bytes=1073741824
        Topic: quickstart-events        Partition: 0    Leader: 1       Replicas: 1     Isr: 1
```

Figure 56. Topic description.

The next step would be to simulate producer behaviour. This is not complicated as Kafka directly communicates with the producer. The simulation can be done by manually sending events to the topic:

bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092

This command launches an inside terminal in which messages can be sent to Kafka, simulating the behaviour of the producer:

```
ubuntu@s1697andrey01:/opt/kafka/kafka_2.12-2.8.2$ bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092
>hello
>hello 2
```

Figure 57. Sending messages to Kafka as a producer.

In order to ensure that the events were stored in Kafka's memory, the list of events can be displayed for the chosen topic in the following way:

bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092

In the console, the output looks the following way:

```
ubuntu@s1697andrey01:/opt/kafka/kafka_2.12-2.8.2$ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092
hello
hello 2
```

Figure 58. List of events inside the topic.

The obtained result means that Kafka is able to receive and store messages. However, storing messages in the Kafka memory is not efficient when it comes to big amounts of data. The data can be redirected to the Cassandra database. Yet, directly pushing data from Kafka to Cassandra is not efficient, as in certain scenarios the developers would like to firstly pre-process the upcoming data or even limit it if not all the data is required. For that reason, it would be a good

architecture decision to make data flow to Cassandra through Spark Streaming, which was designed to work with streaming data.

In order to make it happen, the Spark Streaming-Kafka connector must be installed. There are many open-source connectors available in maven repositories. The one suitable for Cassandra 4 and Spark 3.3.0 is **spark-streaming-kafka-0-10_2.12-3.3.0**. There are many other versions available for any combination of Spark and Kafka. The Installed connector will be in the format of a jar file, just like that of Spark-Cassandra connector. The reason behind choosing this format is that it perfectly worked the previous time and the process of adding it to the system is not complicated. Downloading can be accomplished with the following command:

```
wget https://repo1.maven.org/maven2/org/apache/spark/spark-streaming-kafka-0-10_2.12/3.3.0/spark-streaming-kafka-0-10_2.12-3.3.0.jar
```

The next step is to simply move the jar file to Spark's jars folder:

```
sudo mv spark-streaming-kafka-0-10_2.12-3.3.0.jar /opt/spark/jars
```

Just in these two steps, the connector should be installed and added to the system.

Now, the connector can be used in the Spark shell or Jupyter Notebook, by simply importing the connector libaray, along with Spark Streaming:

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka010._
import com.datastax.spark.connector.streaming._
```

Finally, a producer process can be simulated in Jupyter Notebook by feeding the dataset to the broker:

```python
from kafka import KafkaProducer
```

```python
import pandas as pd

df = pd.read_csv("datasets/oil-prices/data/brent-daily.csv")
```

```python
producer = KafkaProducer(
    bootstrap_servers=["localhost:9092"],
    value_serializer=lambda x: x.encode("utf-8")
)
```

```python
# Send each row as a message to Kafka
for index, row in df.iterrows():
    message = {"Date": row["Date"], "Price": row["Price"]}
    producer.send("testtopic", value=str(message))

# Close KafkaProducer object
producer.close()
```

Figure 59. Feeding dataset to Kafka from Notebook server.

Concluding this part, now the cluster has Kafka to accept streaming data, along with the connector that ensures data flow through Spark Streaming into Cassandra. The updated cluster diagram is shown below:

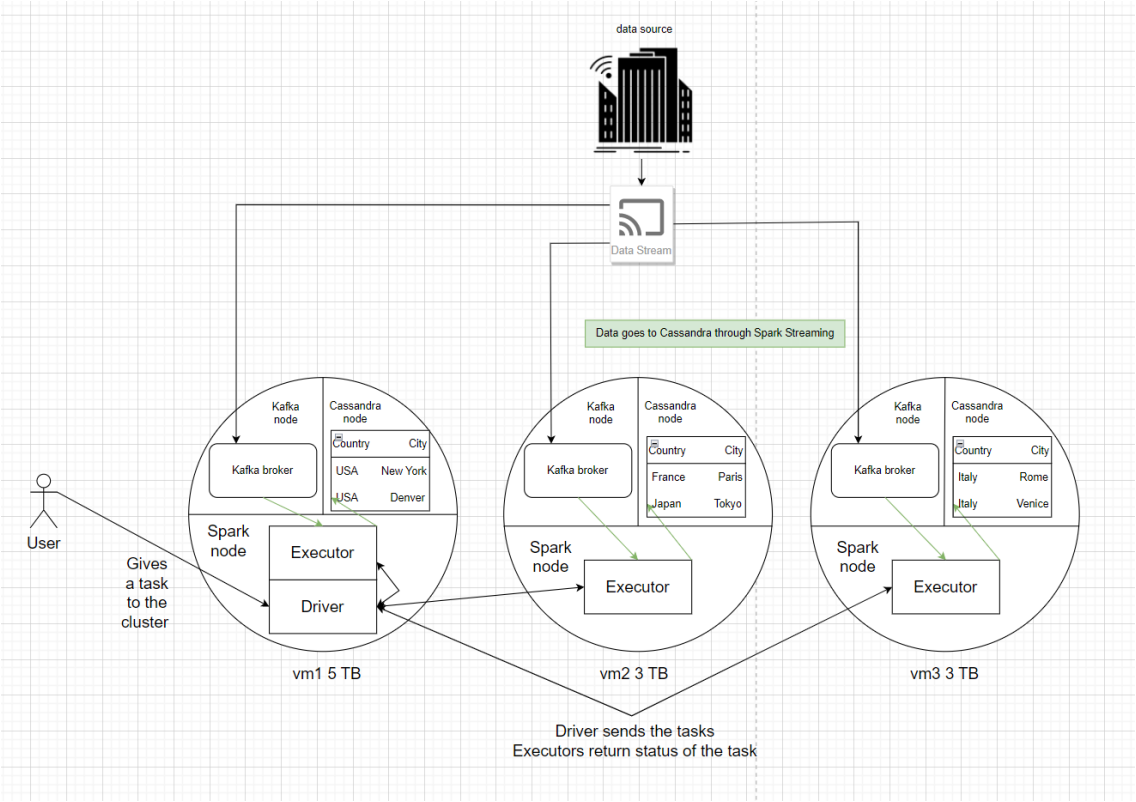


Figure 60. Final cluster diagram.

9. Reflection

This thesis studies the fundamentals of Big Data technologies, which is a broad topic full of content. Particularly, this work has been impactful for the Nokia team, as the project will later be implemented in most of the Machine Learning research projects. The thesis has given a better understanding of the studied technologies and data pipelines. Apache Cassandra was the easiest and the least troublesome technology to study and implement. It also gives a great insight of how effective the distributed databases are. Apache Kafka was not that easy to implement in the beginning due to the necessity of setting up Zookeeper and Brokers and making sure that the nodes fully communicate. However, Kafka is a necessary part of this project due to its ability to work with streaming data. It also gives useful information of how streaming data gets involved into similar systems. Apache Spark introduces the power of parallel data processing and was slightly more challenging to implement due to its need in connecting both Cassandra and Kafka, but also because of the Web UI and

Jupyter server components. Yet it was still a great experience in terms of understanding how connectivity of different services work and how to organize an efficient data pipeline.

10. Conclusion

This thesis required a lot of work, from designing the architecture to configuring files and writing scripts. The main goal of the study was to build a data processing cluster and to explore the core Big Data technologies. At the end, a cluster that can process a big amount of streaming and static data was constructed. All core technologies were fully implemented and connected into an ecosystem that can be freely used by data analysts and data scientists.

Apache Cassandra has three working nodes, with a total of 11 TB of space. All nodes are set to be gossiping and the main KEYSPACES are set to replication factor of three. The Cassandra environment assigns partition keys and distributes data as evenly as possible as the vnode value on each of the nodes was kept the same.

Apache Kafka has three brokers, one on each node with the Zookeeper being able to manage all of them. By using the command shell, a necessary topic can be created and data fed to it.

Apache Spark has three nodes with executor programs on each of them and a driver program on the node with the biggest storage. Spark has two connectors, one for Cassandra and one for Kafka, which enable the data transfer, extraction and processing whenever needed. Moreover, Web UI is accessible to keep track of the cluster. Finally, by running a pyspark command, the local Jupyter Notebook server can be launched, which enables a considerably more convenient Python environment to manage all three programs.

This cluster is a place of interest to multiple teams at Nokia that would love to improve their machine learning related work pipelines. This thesis provides the most updated guide on how to use the newest versions of Spark, Cassandra and Kafka, which will benefit many developers who would come across it. Many more projects and improvements are ahead for this cluster, as it currently solves all the listed challenges, having turned out to be a success.

References

1) Nielsen, A. (2019) *Practical time series analysis*, *O'Reilly Online Learning*. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/practical-time-series/9781492041641/ (Accessed: 2023).

2) Sultan, Kashif & Ali, Hazrat & Zhang, Zhongshan. (2018). Call Detail Records Driven Anomaly Detection and Traffic Prediction in Mobile Cellular Networks. IEEE Access. 6. 1-1. Available at: https://www.researchgate.net/publication/326619835_Call_Detail_Records_Driven_Anomaly_Detection_and_Traffic_Prediction_in_Mobile_Cellular_Networks (Accessed: December 2022).

3) Arezoodahesh (2022) *Hourly Energy Consumption: With prophet*, *Kaggle*. Available at: https://www.kaggle.com/code/arezoodahesh/hourly-energy-consumption-with-prophet/data (Accessed: 03 December 2022).

4) Csafrit (2021) *Steel Industry Energy Consumption*, *Kaggle*. Available at: https://www.kaggle.com/datasets/csafrit2/steel-industry-energy-consumption (Accessed: 03 December 2022).

5) Etaati, L. (2017) *New series of Time Series: Part 1*, *RADACAD*. Available at: https://radacad.com/new-series-of-time-series-part-1 (Accessed: January 2023).

6) *Cassandra Basics* (no date) *Apache Cassandra*. Available at: https://cassandra.apache.org/_/cassandra-basics.html (Accessed: January 2023).

7) CloudZero (2021) *Horizontal vs. vertical scaling: How do they compare?*, *CloudZero*. CloudZero. Available at:

https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling (Accessed: May 4, 2023).

8) *Architecture of Apache Cassandra* (2021) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/architecture-of-apache-cassandra/ (Accessed: January 2023).

9) Carpenter, J. and Hewitt, E. (2022) *Cassandra: The definitive guide, (revised) third edition, 3rd Edition*, *O'Reilly Online Learning*. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/cassandra-the-definitive/9781492097136/ (Accessed: January 10, 2023).

10) Damji, J.S. *et al.* (2020) *Learning spark, 2nd edition*, *O'Reilly Online Learning*. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/learning-spark-2nd/9781492050032/ (Accessed: February 3, 2023).

11) Dancuk, M. (2023) *Resilient Distributed Datasets (spark RDD): Phoenixnap KB*, *Knowledge Base by phoenixNAP*. Available at: https://phoenixnap.com/kb/resilient-distributed-datasets (Accessed: January 2023).

12) Thorp, L. (2022) *Apache spark-multi-part series: Spark Architecture*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/apache-spark-multi-part-series-spark-architecture-461d81e24010 (Accessed: January 2023).

13) Dhore, G. (2022) *Create RDD in Apache Spark using Pyspark*, *Analytics Vidhya*. Available at:

https://www.analyticsvidhya.com/blog/2022/08/create-rdd-in-apache-spark-using-pyspark/ (Accessed: January 2023).

14) Narkhede, N., Shapira, G. and Palino, T. (2017) *Kafka: The definitive guide*, *O'Reilly Online Learning*. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/kafka-the-definitive/9781491936153/ (Accessed: March 15, 2023).

15) Team, D.F. (2018) *Kafka architecture and its fundamental concepts*, *DataFlair*. Available at: https://data-flair.training/blogs/kafka-architecture/ (Accessed: January 2023).

Appendix A: Cassandra

```
sudo apt update

sudo apt install default-jdk

java --version

sudo apt install apt-transport-https

gpg --keyserver keyserver.ubuntu.com --recv-keys 7E3E87CB

gpg --export --armor 7E3E87CB | sudo gpg --dearmour -o
/etc/apt/trusted.gpg.d/cassandra-key.gpg

sudo sh -c 'echo "deb http://www.apache.org/dist/cassandra/debian 40x main" >
/etc/apt/sources.list.d/cassandra.list'

sudo apt update

sudo apt install cassandra
```

```
ubuntu@sl697andrey02:~$ sudo apt install cassandra
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  cassandra-tools
The following NEW packages will be installed:
  cassandra
0 upgraded, 1 newly installed, 0 to remove and 17 not upgraded.
Need to get 47.5 MB of archives.
After this operation, 58.8 MB of additional disk space will be used.
Get:1 https://downloads.apache.org/cassandra/debian 40x/main amd64 cassandra all 4.0.5 [47.5 MB]
Fetched 47.5 MB in 10s (4,875 kB/s)
Selecting previously unselected package cassandra.
(Reading database ... 125555 files and directories currently installed.)
Preparing to unpack .../cassandra_4.0.5_all.deb ...
Unpacking cassandra (4.0.5) ...
Setting up cassandra (4.0.5) ...
vm.max_map_count = 1048575
net.ipv4.tcp_keepalive_time = 300
update-rc.d: warning: start and stop actions are no longer supported; falling back to defaults
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
```

Cassandra's activation can be enabled automatically with system boot by the following way:

sudo systemctl enable cassandra

To manually start cassandra, the following command can be used:

sudo systemctl start cassandra

```
ubuntu@sl697andrey01:~$ sudo systemctl enable cassandra
cassandra.service is not a native service, redirecting to systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable cassandra
ubuntu@sl697andrey01:~$ sudo systemctl start cassandra
```

nodetool status

cqlsh

sudo cp /etc/cassandra/cassandra.yaml /etc/cassandra/cassandra.yaml.backup

sudo nano /etc/cassandra/cassandra.yaml

All the ip addresses of all nodes must be inputed, to ensure the connection between them.

```
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and learn
    # the topology of the ring.  You must change this if you are running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "10.10.[redacted],10.10.[redacted],10.10.[redacted]"
```

sudo systemctl restart cassandra

sudo systemctl start cassandra

nodetool status

```
cqlsh> CREATE KEYSPACE Atg_test
   ... WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor' : 3};
cqlsh> desc KEYSPACES;

atg_test  system       system_distributed  system_traces  system_virtual_schema
library   system_auth  system_schema       system_views
```

USE <KEYSPACE name>

```
cqlsh:atg_test> CREATE TABLE IF NOT EXIST Atg_test.atg_projects
( id int, title text, number_of_interns int, PRIMARY KEY (field)
 );
```

```
cqlsh:atg_test> INSERT INTO ATG_test.atg_projects (id, title, number_of_interns, field)
 VALUES(1, 'time series analysis', 3, 'ML')
```

```
cqlsh> select * from atg_test.atg_projects;

 field  | id | number_of_interns | title
--------+----+-------------------+-----------------------
     UX |  3 |                 4 |               Graph UX
     UX |  4 |                 1 |         Augmented Maps
     ML |  1 |                 3 |   time series analysis
     ML |  2 |                 2 |  time series synthesis
     ML |  6 |                 2 |       Causal discovery
 DevOps |  5 |                 1 | Infra & tools projects
```

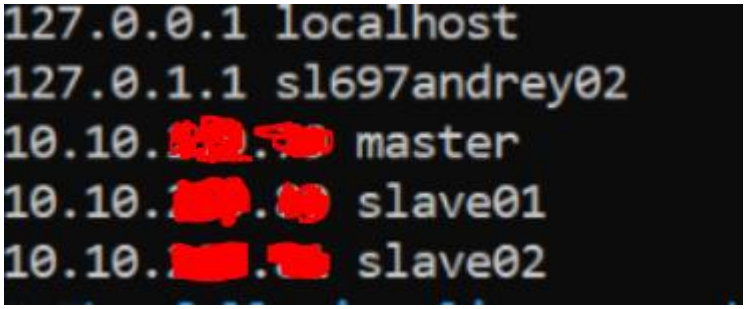
```
cqlsh:atg_test> select token(field), field from atg_projects;

 system.token(field)  | field
----------------------+--------
 -7184959072001002467 |     UX
 -7184959072001002467 |     UX
  2484577390804984672 |     ML
  2484577390804984672 |     ML
  2484577390804984672 |     ML
  4855069840617214231 | DevOps

(6 rows)
```

```
cqlsh:atg_test> exit
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects -7184959072001002467
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects 2484577390804984672
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
ubuntu@sl697andrey01:~$ nodetool getendpoints atg_test atg_projects 4855069840617214231
10.10.[redacted]
10.10.[redacted]
10.10.[redacted]
```

Appendix B: Spark

```
sudo apt install scala

wget https://dlcdn.apache.org/spark/spark-3.3.0/spark-3.3.0-bin-hadoop3.tgz

tar xvf spark-*

sudo mv spark-3.3.0-bin-hadoop3 /opt/spark

cd /opt/spark

echo "export SPARK_HOME=/opt/spark" >> ~/.bashrc

echo "export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin" >>
~/.bashrc

echo "export PYSPARK_PYTHON=/usr/bin/python3" >> ~/.bashrc

source ~/.bashrc

spark-shell

sudo nano /etc/hosts
```

Inside the file, all ip addresses of nodes must be added with the according name (master, slave01, slave02):



cd /opt/spark/conf

sudo nano spark-env.sh

Inside the file, only two variables need to be changed:

SPARK_MASTER_HOST='ip address'

SPARK_MASTER_PORT='port number'

Now the master node needs to start the master process:

start-master.sh

After starting the master process, worker nodes can be connected. After opening the terminals of two other virtual machines, the following command can be run:

start-worker.sh spark://<ip>:<port>

git clone https://github.com/datastax/spark-cassandra-connector

cd spark-cassandra-connector

Setting up the Spark-Cassandra connector:

git clone https://github.com/datastax/spark-cassandra-connector

cd spark-cassandra-connector

./sbt/sbt assembly

sudo mv spark-cassandra-connector-assembly-3.2.0-2-g92ba7ba0.jar /opt/spark/jars

Next step is to test the connection through spark-shell. Firstly, the spark shell must be opened:

spark-shell –jars ~/jars/spark-cassandra-connector-assembly-3.2.0-2-g92ba7ba0.jar

```
scala> sc.stop

scala> import com.datastax.spark.connector._, org.apache.spark.SparkContext, org.apache.spark.SparkContext._, org.apache.spark.SparkConf
import com.datastax.spark.connector._
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

```
scala> val conf = new SparkConf(true).set("spark.cassandra.connection.host", "localhost")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@41366811

scala> val sc = new SparkContext(conf)
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@7969e0cb
```

```
scala> val table = sc.cassandraTable("atg_test", "atg_projects")
```

```
scala> println(table.count)
6
```

Launching local Jupyter Notebook:

pip install notebook

Secondly, new lines must be added to the bashrc file, replacing the old ones:

sudo nano ~/.bashrc

export PYSPARK_DRIVER_PYTHON='jupyter'
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port=8889'

source ~/.bashrc

Jupyter Notebook server initialization:

```
ubuntu@s1697andrey01:~$ pyspark
[I 13:39:04.428 NotebookApp] Writing notebook server cookie secret to /home/ubuntu/.local/share/jupyter/runtime/notebook_cookie_secret
[I 13:39:04.623 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 13:39:04.623 NotebookApp] Jupyter Notebook 6.5.2 is running at:
[I 13:39:04.623 NotebookApp] http://localhost:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
[I 13:39:04.623 NotebookApp]  or http://127.0.0.1:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
[I 13:39:04.623 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 13:39:04.626 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/ubuntu/.local/share/jupyter/runtime/nbserver-2644405-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
     or http://127.0.0.1:8889/?token=60b468435b909efe56c06705ca440117424730a50e36c6f1
```

Connecting Notebook to the cluster:

```
import os
import pyspark
from pyspark.sql import SQLContext, SparkSession
from pyspark import SparkConf, SparkContext
import random as rnd
```

```
conf = SparkConf().setAppName("test")
sc.stop() #stop the previous/default spark context
sc = SparkContext('spark://10.10.219.79:7077', conf=conf)
```

Appendix C: Kafka

wget http://mirror.cogentco.com/pub/apache/kafka/2.8.2/kafka_2.12-2.8.2.tgz

tar -xvf kafka_2.12-2.8.2.tgz
sudo mv kafka_2.12-2.8.2 /opt/kafka

server.properties file needs to be modified:

cd /opt/kafka/config

sudo nano server.properties

```
############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=1
```

```
############################# Log Basics #############################

# A comma separated list of directories under which to store log files
log.dirs=/opt/kafka/logs
```

```
############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.10.[redacted]:2181,10.10.[redacted]:2181,10.10.[redacted]:2181
```

sudo nano zookeeper.properties

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# the directory where the snapshot is stored.
dataDir=/opt/kafka/zookeeper_data
# the port at which the clients will connect
clientPort=2181
# disable the per-ip limit on the number of connections since this is a non-production config
maxClientCnxns=0

initLimit=10
syncLimit=5
# Disable the adminserver by default to avoid port conflicts.
# Set the port to something non-conflicting if choosing to enable this
admin.enableServer=false
# admin.serverPort=8080

server.1=0.0.0.0:2888:3888
server.2=10.10.[redacted]:2888:3888
server.3=10.10.[redacted]:2888:3888
```

sudo bin/zookeeper-server-start.sh /opt/kafka/kafka_2.12-2.8.2/config/zookeeper.properties

sudo bin/kafka-server-start.sh config/server.properties

bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092

bin/kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092

bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092

bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092

```
ubuntu@s1697andrey01:/opt/kafka/kafka_2.12-2.8.2$ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092
hello
hello 2
```