Apache Cassandra and Apache Spark Integration
A detailed implementation

Integrated Media Systems Center
USC
Spring 2015

Supervisor
Dr. Cyrus Shahabi

Student Name
Stripelis Dimitrios

Contents

1. Introduction

This paper can be used as a reference guide for a detailed technical implementation of Apache Spark **v. 1.2.1** and Apache Cassandra **v. 2.0.13**. The integration of both systems was deployed on Google Cloud servers using the RHEL operating system. The same guidelines can be easily applied to other operating systems (Linux based) as well with insignificant changes.

Cluster Requirements:

Software
Java 1.7+ installed
Python 2.7+ installed

Ports
A number of at least 7 ports in each node of the cluster must be constantly opened. For Apache Cassandra the following ports are the default ones and must be opened securely:
9042 - Cassandra native transport for clients
9160 - Cassandra Port for listening for clients
7000 - Cassandra TCP port for commands and data
7199 - JMX Port Cassandra

For Apache Spark any 4 random ports should be also opened and secured, excluding ports 8080 and 4040 which are used by default from apache Spark for creating the Web UI of each application. It is highly advisable that one of the four random ports should be the port 7077, because it is the default port used by the Spark Master listening service.

2. Apache Cassandra Overview

Apache Cassandra is an open-source Distributed Database Management System (DDBMS) developed entirely in JAVA. Cassandra adopts its architectural principals from Amazon Dynamo DB and its data model – Sorted String Table (SST) – design from Google's Big Table.

3. Apache Cassandra Production Development

For production development it is recommended to

- Install the Java Native Access (JNA)
- User Resource Limits
- Swap Disable

The next steps should be applied to each node separately. The same work can also be performed if we configure all the settings in one node exclusively and then we clone the image of that node to the rest of the nodes inside the cluster.

Download Cassandra 2.0.13

Create a new installation directory for Cassandra inside the node and navigate to the following page (http://www.apache.org/dyn/closer.cgi?path=/cassandra/2.0.13/apache-cassandra-2.0.13-bin.tar.gz) to start downloading the Cassandra package. Once you browse to the respective page you can download Apache Cassandra from a working mirror inside the preferred installation directory of the node by executing:

```
$wget http://mirrors.ibiblio.org/apache/cassandra/2.0.13/apache-cassandra-2.0.13-bin.tar.gz
```

Hint: if we need to install alternative versions we just simply change the Cassandra version in the URL path of the browsing page (e.g. 2.0.13 becomes 2.0.15, etc...).

Once we have downloaded the appropriate Cassandra version in the desired directory we untar the package. The path which shows to the Cassandra installation will be referred to as the ${CASSANDRA_HOME} in all the following sections.

Install JNA – (Java Native Access)

Inside the shell of the node we type:

```
$sudo yum install jna
```

If the jna cannot be installed using yum or the jna version that is installed is not greater than 3.2.7 then we have to download the jna.jar file from the github source (https://github.com/twall/jna). We can store this package to an arbitrary directory or if we prefer inside (*/usr/share/java/*) and then create a symbolic link to the *jna.jar* file by typing:

```
$ln -s /usr/share/java/jna.jar ${CASSANDRA_HOME}/lib
```

where ${CASSANDRA_HOME} is the directory that we have extracted the Cassandra package, as we have mentioned above.

User Resource Limits

For tarball installations we have to ensure that the following settings are included inside the */etc/security/limits.conf* file of each server/node:

```
*  – memlock unlimited
*  – nofile 100000
*  – nproc 32768
```

```
*  – as unlimited
```

Moreover, we have to set the nproc limits in the */etc/security/limits.d/90-nproc.conf* file of each server:

```
*  – nproc 32768
```

And lastly we have to add the following line inside the */etc/sysctl.conf*:

```
vm.max_map_count = 131072
```

In order all of the above settings to take effect we have to either reboot the server or run the following command:

```
$sudo sysctl -p
```

To verify that the limits have been correctly applied we can type:

```
$cat /proc/<pid>/limits
```

where <pid> is the process id of the currently running Cassandra instance.
The Cassandra pid can be easily found once the instance is up and running, by typing:

```
$ps –auwx | grep cassandra
```

Swap Disable

If the swap is not disabled entirely then we may have significant performance degradation. Since Cassandra stores multiple replicas and has transparent failover, it is preferable for a replica to be killed immediately when the memory is low rather than go into swap. This allows traffic to be immediately redirected to a functioning replica instead of continuing to hit the replica that has high latency due to swapping.

```
$sudo swapoff –all
```

To make this change permanent, remove all swap file entries from */etc/fstab*

4. Apache Cassandra Running Requirements

Each node must be correctly configured before starting the cluster. The following configurations are essential if we want to ensure the effectiveness of our Cassandra cluster:

1. **The .yaml file**
   Firstly, we have to navigate to the *${CASSANDRA_HOME}/conf/* directory. Inside the directory exists the *cassandra.yaml* file. **All the configurations of the node must be explicitly declared inside this file.** A very good guideline on what each property represents can be found in this link (http://docs.datastax.com/en/cassandra/2.0/cassandra/configuration/configCassandra_yaml_r.html)

2. **Get the IP address of the node**
   For setting the nodes intercommunication the required IP is the internal IP of each node. Type *ifconfig* on the shell of each node and store the respective Internal IP.

   For all the following steps (3-11) all the changes should be applied inside the *${CASSANDRA_HOME}/conf/cassandra.yaml* file.

3. **Choose a name for the cluster**
   Each Cassandra application has a unique cluster name. Inside the *conf/cassandra.yaml* file we have to change the parameter *cluster_name* to the appropriate one of our application (e.g. change the name from Test Cluster to MyProject).

4. **Determine which nodes will be seed nodes**
   Cassandra nodes use the seed node list for finding each other and learning the topology of the ring. Every seed node is responsible for bootstrapping the newly incoming nodes inside the ring/cluster. Without declaring any seed nodes the Cassandra cluster will not be able to operate. Depending on the magnitude of the under development cluster one must explicitly recognize which nodes will be used as seed nodes. An easy way to place the seed nodes is by setting the first node of the cluster as the seed node and then every 3 Cassandra nodes define a new seed node. Every node of the cluster must see the same seed nodes all the time. Inside the *conf/cassandra.yaml* file we must set the seeds nodes lists as:
   - seeds: "10.240.160.14,10.240.57.177,10.240.144.122"

5. **Determine the snitch**
   Snitch is used to declare the topology of the cluster, meaning the nodes' place in Racks and Data Centers. For a single data center deployment using the SimpleSnitch is recommended. To alter the snitch firstly we go to the *conf/cassandra.yaml* file and then we change the *endpoint_snitch* parameter by assigning any value that satisfies our deployment requirements. All the possible snitch values can be found in the following link (http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html)

6. **Listen Address**
   Every node must be associated with a listening address, which in this case is the Internal IP of the node. This address is responsible for the intercommunication of the cluster servers and should be changed from *`localhost`* to the appropriate value.

7. **Broadcast Address**
   Every node inside the cluster is also associated with a broadcast address. This address should be the external IP when a Cassandra cluster is being deployed across multiple areas/regions *or* the internal IP of the node if the cluster will be deployed solely inside a single Data Center. Similarly, the value of the broadcast address can be altered inside the *`/conf/cassandra.yaml`* file of the node.

8. **Rpc Address**
   This address holds for the Remote Procedure Call address. This is where the Cassandra node listens for the clients' connections. Its value can be set as the hostname or the internal IP address of the node.

9. **Commitlog Directory**
   This is the directory where Cassandra stores information when a new write is made so that it can be replayed if a crash occurs. It is responsible for storing information related to the uncommitted data. Moreover, the Cassandra node uses the directory to repopulate the memtables after any expected or unexpected shutdown. A memtable is a write-back in-memory cache of data rows that can be looked up key. For optimal performance this directory should have a path, if possible, different from the Data File Directories in a production environment because of the sequential I/O in disk.

10. **Data File Directories**
    This is the directory where the written data are stored for each node individually. The contents of this directory relate to information regarding the Sorted String Tables. For optimal performance this directory should be different from the commitlog directory.

11. **Saved Caches Directory**
    This is the directory where the tables' keys and row caches are stored. It can be assigned any possible directory inside the node.

12. **System Logs**
    Once the changes inside the *`cassandra.yaml`* file have been made we proceed with the declaration of the log directory inside the *`conf/log4j-server.properties`* file and specifically the line of `log4j.appender.R.File`. This is where the Cassandra nodes see where the system logs are stored.

13. Configuration File Location

Since we have installed Cassandra in a different directory than the default one we have to declare the new location of the Cassandra package and specifically the location of the configuration directory (`i.e. ${CASSANDRA_HOME}/config`). Therefore, we set the respective location inside the */bin/cassandra.in.sh file* on the line which contains the variable *CASSANDRA_CONF=*.*

All the above steps (1-13) as well as the following step must be equally applied to all the nodes of the cluster. In order to perform this in a dynamic and flexible way, we can create an executable bash script (*chmod +x <script_name>*) in any location/directory of the node's environment; for naming convention we call this script *cassandra_node.sh.*

The script's contents based on steps (1-13) can have the following content:

Create the appropriate directories

```
#! /bin/bash
mkdir -p ${CASSANDRA_DIR}/cassandra_logs
mkdir -p ${CASSANDRA_DIR}/cassandra_logs/data
mkdir -p /tmp/cassandra_commitlog
mkdir -p ${CASSANDRA_DIR}/cassandra_logs/saved_caches
```

Make the required configurations for the *conf/cassandra.yaml* file of each node

```
sed -ri  -e  "s/cluster_name:.*/cluster_name: 'BDTS'/" \
         -e  "s/listen_address:.*/listen_address: \""${MACHINE_IP}"\"/" \
         -e  "s/broadcast_address:.*/broadcast_address: \""${MACHINE_IP}"\"/" \
         -e  "s/rpc_address:.*/rpc_address: \""${MACHINE_IP}"\"/" \
        -e  "s|seeds:.*|seeds:  \""${SEEDS}"\"|" \
         -e  "s|commitlog_directory:.*|commitlog_directory:
"/tmp/cassandra_commitlog"|" \
         -e  "s|saved_caches_directory:.*|saved_caches_directory:
"${CASSANDRA_DIR}/cassandra_logs/saved_caches"|" \
   -e  "/^data_file_directories:$/!b;n;c\    - ${CASSANDRA_DIR}/cassandra_logs/data"
\
         ${CASSANDRA_HOME}/conf/cassandra.yaml
```

Change the system logs location inside the *conf/log4j-server.properties* file of each node

```
sed -i  -e
"s|log4j.appender.R.File=.*|log4j.appender.R.File="${CASSANDRA_DIR}/cassandra_logs/s
ystem.log"|" \
         ${CASSANDRA_HOME}/conf/log4j-server.properties
```

Change the conf location inside the *bin/cassandra.in.sh* file of each node

```
sed -i  -e "s|CASSANDRA_CONF=.*|CASSANDRA_CONF=\""${CASSANDRA_HOME}/conf"\"|" \
        ${CASSANDRA_HOME}/bin/cassandra.in.sh
```

As we can see from the commands we have to declare two different directories as our environmental variables.

- CASSANDRA_HOME: the location of the extracted Cassandra package

- CASSANDRA_DIR: the location of the external/secondary disk where the Cassandra logs (data, system logs) will be saved. This directory is optional if we do not deploy the cluster in a production level, we can use the CASSANDRA_HOME directory for testing purposes in its place.

NOTICE: the location of the commitlog_directory (`/tmp/cassandra_commitlog`) is different from the data_file_directories (`${CASSANDRA_DIR}/cassandra_logs/data`); *`${MACHINE_IP}`* is an environmental variable containing the internal IP of the Cassandra node and *`${SEEDS}`* is an environmental variable which refers to the Internal IPs of the seed nodes, separated by comma.

Following, we provide a detailed description on how one can employ correctly the environmental variables.

Apache Cassandra Environmental Variables Declaration
In order the environmental variables to be accessible from any script, as for instance the *`cassandra_node.sh`*, we can create a new script which will contain all the necessary variables and their associated values and will be sourced by other scripts which require their existence. For example, we could name such a file as *`remote_settings.env`* and type inside the following commands:

```
export SEEDS=<IPs of the seeds>;[e.g 10.240.160.14,10.240.57.177,10.240.144.122;]
export CASSANDRA_HOME=/home/{username}/cassandra-2.0.13;
export CASSANDRA_DIR=<external hard disk, if applicable>;
```

How to start & stop a Cassandra Node
Establishing a Cassandra Node in the cluster requires the creation of a bash script (e.g. *`run.sh`*), where we source both the *`remote_settings.env`* and the *`cassandra_node.sh`*. Once all the requirements and configurations are met we can add inside the *`run.sh`* the following:

```
${CASSANDRA_HOME}/bin/cassandra
```

Thereinafter, we make the *`run.sh`* script executable and we call it (*`sh run.sh`*).

A possible structure of the `run.sh` script could be:

```
#! /bin/bash
source ./remote_settings.env
source ./cassandra_node.sh
${CASSANDRA_HOME}/bin/cassandra
```

As soon as the execution takes place the screen will be filled with log files containing information of the Cassandra node initialization. If something went wrong during the start-up procedure then we have to pay close attention to the ERROR messages that will appear; if nothing else then the Cassandra node is now up and running. In addition, because of the long log messages that will appear during the initialization of every

Cassandra node, it is preferable to redirect the execution stream to another file (e.g. *sh run.sh >> output.txt*).

If we need to stop an already running Cassandra node we have to kill the associated Java Process. First we find the process id of the Cassandra Node and then we terminate it:

```
//find process
ps –auwx | grep cassandra

//kill process
sudo kill <pid>
```

Finally, assuming that all the installation configurations and settings have been applied correctly to all the servers of the cluster, we can verify that our Cassandra cluster is up and running by executing the *nodetool* script inside the *${CASSANDRA_HOME}/bin.* Specifically, we can login to any server of the cluster and perform the following command using the *status* argument:

```
${CASSANDRA_HOME}/bin/nodetool   status
```

If everything is operating correctly then next to the internal IP of every node we should see that the state's label is shown as Normal (UN). If one of the nodes is shown as being in the Joining state (UJ) then we can wait for a couple of seconds and execute the above command once more to confirm that now the state has changed to Normal. However, if one of the nodes has a Down state (DN) then something went wrong during the configuration process for that specific node. In this case, we will have to kill the corresponding Java Process and start the Cassandra node again with the correct configurations.

5. Apache Cassandra Read/Write Requests using the Python API

Since every Cassandra node inside the cluster can act as a Master node there are no restrictions into executing client requests (write/ read data) from any random node of the cluster. In this report we are using the Python Driver (python version 2.7+) to install the Cassandra Driver and execute some basic queries. One is free is to use any other language API to implement the driver requests (e.g. JAVA, C++, PHP)

In this section we present how one can start running queries on a Cassandra cluster using three python scripts. The first script can be used as an interface in order to call the appropriate functions to connect to the Cassandra cluster. The second script shows how we can create the keyspace and the column-families or "tables" of the database and how we can push some data inside the tables. Finally, the third script illustrates the data querying procedure.

The required steps are as follows:

Steps:

1. Install Cassandra Driver

```
sudo yum install gcc python-devel
pip install cassandra-driver
```

A more detailed description of the python driver with more dependencies and configuration settings can be found at (https://datastax.github.io/python-driver/installation.html)

2. Client API based on Python

Following we present a python script, which can be used as an interface to connect to a Cassandra cluster, create tables, indexes and exit the cluster

Script 1: cassandra_driver.py

```
# import the Cassandra Cluster module
from cassandra.cluster import Cluster

#create a Driver class, with each object's attributes being the nodes that represent the object. These
#nodes will be used as a pivot to connect to the cluster. The metadata are responsible for storing
#useful information for the cluster, such as the cluster's name. The session attribute is used to
#connect to the cluster and execute the queries
class Driver:

    ClusterNodes = []
    cluster = None
    metadata = None
    session = None

    def __init__(self,cluster_nodes):
        self.ClusterNodes = cluster_nodes

    #call this function to connect to the cluster – Cassandra finds itself which cluster each node is
    #connected to
    def Connect(self):
        self.cluster = Cluster(self.ClusterNodes)
        self.session = self.cluster.connect()
        self.metadata = self.cluster.metadata
        print('Connected to cluster: ' + self.metadata.cluster_name)

    #get the metadata of the cluster
    def GetMetadata(self):
        return self.metadata

    #use this function to create the required keyspace with its name as a parameter
    def CreateKeyspace(self,keyspace):
```

```
        cluster_keyspaces = self.metadata.keyspaces
        if keyspace not in cluster_keyspaces.keys():

        #when creating the keyspace we have to declare the replication strategy. Simple Strategy is
        #for a Single Data Center deployment and NetworkTopologyStrategy is for a multi-data
        #center deployment. Moreover at this point we have to declare the number of replicas that
        #we will use for each data stream by assigning an integer value to the replication_factor
        #attribute
                self.session.execute(""" CREATE KEYSPACE """ +  keyspace + """ WITH replication =
{'class':'SimpleStrategy', 'replication_factor':3}; """)

    # if you want to delete/erase the keyspace and all of its contents (column families, data) call
    # this function; the only required parameter is the keyspace name
    def DropKeyspace(self,keyspace):
        cluster_keyspaces = self.metadata.keyspaces
        print(cluster_keyspaces.keys())
        if keyspace in cluster_keyspaces.keys():
                self.session.execute(""" DROP KEYSPACE """ + keyspace + ";")

    #use this function to create a table; it is required to pass the whole command of table creation
    #in triple quotes ('''...''')
    def CreateTable(self,string):
        self.session.execute(string)

    #use this function to delete an existing table; the required parameters are the keyspace name
    #and table name in String format
    def ClearTable(self,keysapce,table_name):
        self.session.execute("TRUNCATE " + keysapce + "." + table_name)

    #use this function to create index on a specific attribute(attribute_name) of a specific
    #table(table_name)
    def CreateIndex(self,keyspace,table_name,attribute_name):
        self.session.execute("CREATE INDEX IF NOT EXISTS ON " + keyspace + "." + table_name + " ("
+ attribute_name +");")

    #use this function to delete an already existing index
    def DropIndex(self,keyspace_dot_table,index_name):
        print(keyspace_dot_table + "_" + index_name)
        self.session.execute("DROP INDEX IF EXISTS " + index_name + ";")


    #use this function to terminate the connection with the cluster when the requests have finished
    def CloseConnection(self):
        self.session.cluster.shutdown()
        self.session.shutdown()
```

3. Execute CQL queries using the Python API

Script 2: create_schema.py

```
#import the above created script
import cassandra_driver

#Use the internal IP of the node on which the client will run. Cassandra automatically discovers the
#rest of the nodes inside the cluster; therefore there is no need to assign more IPs other than the one
#of the coordinator. Assign the value as expressed below so to create the respective
#cassandra_driver object.
client = cassandra_driver.Driver(["10.240.57.177"])

#store the cluster's session
session = client.session

#create the keyspace of the cluster by assigning its name
client.CreateKeyspace("highway")

#create tables inside the keyspace by calling <keyspace>.<table_name>, for instance here it is
#highway.highway_config

client.CreateTable("""
                CREATE TABLE IF NOT EXISTS highway.highway_config (
                        CONFIG_ID int,
                        AGENCY varchar,
                        CITY varchar,
                        EVENT_TIME timestamp,
                        LINK_ID int,
                        PRIMARY KEY (CONFIG_ID, AGENCY, LINK_ID)
                );
""")

#create indexes for attributes of the table by using the following #syntax, <keyspace name>
#<table name>, <attribute name>
client.CreateIndex("highway","highway_config","DIRECTION")

#insert data inside the highway_config table – first we create a prepared statement, on which we will
#bind the values we want to store inside the table
prepared_statement1 = session.prepare("""
    INSERT INTO highway.highway_config (CONFIG_ID, AGENCY, CITY, EVENT_TIME, LINK_ID)
                        VALUES (?, ?, ?, ?, ?)
                """)

#whenever we want to bind values to a prepared statement, we have to assign them using a tuple
#(val1,val2,…) or a list [val1,val2,…].
#For example:
values=(92,"Caltrans","Los Angeles","2014-02-05 11:30:00",17656)
insertion = prepared_statement1.bind(values)
session.execute(insertion)
```

Script 3: cassandra_cql_queries.py

```
#import the above created script
import cassandra_driver

client = cassandra_driver.Driver(["10.240.57.177"])

#connect to the cluster
client.Connect()

#store the cluster's session
session = client.session

#write the desired query
query = "SELECT speed FROM highway.street_monitoring WHERE onstreet='47' AND year=2014
AND month=2 AND day=21 AND time>=360 AND time<=7560 AND speed>30"

#retrieve the results
results = session.execute(query)

#results will be returned in the form of an array – print the first retrieved data
print results[0]
```

6. Types of Cassandra Queries

The Cassandra Query Language (CQL) resembles the syntax of the PL/SQL. However, there is no correlation between them since Cassandra's Data Model does not support table joins (foreign integrity is absent) and sub queries. One must first understand the principals of the Sorted String Table { Map [String, Map [String, Data] } representation model and most importantly the operations that can be performed over the Partition Keys and the Clustering keys of the column families. Following we present a thorough guideline that everyone should be aware of in order to effectively and correctly apply CQL queries along with some common errors.

Guidelines:

- Partition key columns support the = operator
- The last column in the partition key supports the IN operator
- Clustering columns support the =, >, >=, <, and <= operators
- Secondary index columns support the = operator

Queries Examples:
Assuming that we have created our table with the following attributes and Primary Key,

```
client.CreateTable ("""
        CREATE TABLE IF NOT EXISTS highway.street_monitoring (
                ONSTREET varchar,
                YEAR int,
                MONTH int,
                DAY int,
                TIME int,
                POSTMILE float,
                DIRECTION int,
                SPEED int,
PRIMARY KEY   ( ( ONSTREET,YEAR,MONTH ),DAY,TIME,POSTMILE,DIRECTION )
            );
""")
```

client.CreateIndex("highway","regional_monitoring",”SPEED")

where the Partition key are the attributes (ONSTREET, YEAR, MONTH), clustering keys are the attributes (DAY, TIME, POSTMILE, DIRECTION), and the secondary index is on top of the SPEED attribute. Some of the queries that may result in an error message could be:

Query1 – Full partition key
SELECT * FROM highway.street_monitoring WHERE onstreet=‘I-10’ AND year=2015 AND month IN(2,4)
Error Message: None

No error message is displayed since we query based on the full partition key and we are allowed to perform a range – IN – query on the last component of the partition key

Query2 – Some parts of Partition key
SELECT * FROM highway.street_monitoring WHERE onstreet=‘I-10’ AND month=3
Error Message: cassandra.InvalidRequest: code=2200 [Invalid query] message="Partition key part year must be restricted since preceding part is”

The reason that the above error occurs relates to the fact that we have not restricted the year value, which is required since it is a component of the partition key

Query3 – Range on Partition Key
SELECT * FROM highway.street_monitoring WHERE onstreet=‘I-10’ AND year=2014 AND month<=1
Error Message: cassandra.InvalidRequest: code=2200 [Invalid query] message="Only EQ and IN relation are supported on the partition key (unless you use the token() function)"

Even though the partition key is correctly restricted with equalities we cannot perform range queries on the last component of the partition key without declaring a lower and an upper bound; the inequality month>= is absent from the query

Query4 – Only secondary index

SELECT * FROM highway.street_monitoring WHERE day>=2 AND day<=3

Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING

The above query even if it is not entirely correct it can be executed as long as we append the ALLOW FILTERING keyword at the end of the WHERE clause. However, we must be confident that we know exactly which data we want to extract. It is advisable to use the ALLOW FILTERING keyword on a secondary index after we have selected the partition and clustering key and not solely using a single attribute. The underlying reason is that in this case Cassandra will fetch all the data from the database that correspond to this specific single attribute range value.

Query5 – Range on Secondary Index

SELECT * FROM highway.street_monitoring WHERE onstreet='47' AND year=2014 AND month=2 AND day=21 AND time>=360 AND time<=7560 AND speed>30

Error Message: cassandra.InvalidRequest: code=2200 [Invalid query] message="No indexed columns present in by-columns clause with Equal operator"

The error arises due to the fact that a secondary index can support only operations of equalities and inequalities, no range.

7. Apache Spark Overview

Apache Spark is an open-source cluster-computing platform for fast and general-purpose large-scale data analytics. All the computations run in-memory and it is built entirely in Scala. Spark uses master-slave architecture and can access any Hadoop data source, including Cassandra.

8. Building the Spark Project

In this section we will present how we can install the Spark Framework using the Apache Spark 1.2.1 stable version. It is required to use this version since it is the latest version being compatible with Cassandra 2.0.13. First, we have to navigate to the Spark download page (https://spark.apache.org/downloads.html) and select the relative package that we want to

install. Afterwards, we select any of the available mirrors that will be shown and we download (wget <mirror page>) the package inside the cluster node in a similar way like we did during the Cassandra installation. The same procedure has to be performed to all the servers of the cluster. However, it is recommended that we make Spark fully operational to a single machine and we clone that machine to the rest of the servers, as we did during the Cassandra set up.

There are two ways that one can build the Spark Framework; using either Apache Maven or Scala Built Tools (SBT). Both ways can be deployed with ease but Apache Maven requires some further modifications that have to be met. In this section we will present how to build Apache Spark using Maven.

Build with Apache Maven

Assuming that Apache Maven is not pre-installed inside the node, we have to execute the following steps in a glance:

Step 1: Download Spark 1.2.1
Step 2: Download Spark 1.3.1
Step 3: Copy */build* folder from Spark 1.3.1 inside Spark 1.2.1
Step 4: Create a custom script
Step 5: Change pom.xml
Step 6: Change mvn script (optional)
Step 7: execute the custom script

Steps 1 – 4

The reason that we have to download both Spark versions can be justified as of the fact that Spark version 1.3.1 comes with a very well written bash script, which we are going to use for our installation process. Most importantly, the script can handle all the required dependencies during the Spark installation procedure. The script resides in the *${SPARK_1.3.1_HOME}/build/* directory with the name mvn. We have to copy the entire directory *(/build/*)* inside the *${SPARK_1.2.1_HOME}/* directory and navigate to it. Now that we are inside the newly created directory *${SPARK_1.2.1_HOME}/build/* we create our own bash script with any arbitrary name that we like. For our purposes we can use the name *spark_init.sh.* We then populate the script with the following commands:

```
export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:ReservedCodeCacheSize=512m";

./mvn -e -f ../pom.xml -Dscalastyle.failOnViolation=false -
Dcloud.spark.directory=/home/{username}/spark-1.2.1 \
             -DHPC-Dhadoop.version=1.2.1 -Dscala-2.10 -DskipTests clean
package;
```

The first export command relates to the memory that will be used for the Maven installation and the second one executes the mvn script, which we acquired when we copied the build folder from Spark 1.3.1 to Spark 1.2.1. As we can see from the parameters we execute the

maven building tools against the Spark's pre-existed pom.xml file with the following arguments:

- Dscalastyle.failOnViolation=false
This parameter refers to the plugin with artifactId scalastyle-maven-plugin. We need to assign the false value to the failOnViolation parameter because we want the Spark installation to continue even if this plugin violates some of the dependencies.
- Dcloud.spark.directory=<spark installation directory>
With this parameter we assign to the cloud.spark.directory attribute the installation path of the Spark project.
- DHPC-Dhadoop.version=<Hadoop version>
This parameter specifies the Hadoop version that the Spark project will be built on top.
-Dscala-2.10<scala version>
This argument specifies the scala version that we want the Spark project to be built against; the recommended Scala version is 2.10 and not Scala v. 2.11 since there are some minor Spark functionalities which are not fully supported using Scala 2.11.
-DskipTests clean package
Recommended for first-time installations.

Step 5
All of the above attribute assignments cannot be effective if we do not apply the required changes inside the pre-build pom.xml file of the Spark Project (*path: ${SPARK_1.2.1_HOME}/pom.xml*). Specifically, we search inside the pom.xml in order to find the plugin with `artifactId` equal to the `scalastyle-maven-plugin` and then we change the tags

```
<failOnViolation>...</>
<configLocation>...</>
<outputFile>...</>
```

so to have a different content/value:

```
<failOnViolation>${scalastyle.failOnViolation}</failOnViolation>
<configLocation>${cloud.spark.directory}/scalastyle-config.xml</configLocation>
<outputFile>${cloud.spark.directory}/scalastyle-output.xml</outputFile>
```

In this way we can assign to each dependency/plugin attribute a different tag value with respect to our building purposes.

Step 6
One last configuration that has to be made before executing the custom created script (*spark_init.sh*) is to modify the mvn script (which now resides inside the `${SPARK_1.2.1_HOME}/build/` directory). Towards the end of the script we will find the

${MVN_BIN} "$@" command. Before this command we create a new variable named as MAVEN_LOG_DIR, which will store the path to the maven artifacts.

```
e.g. MAVEN_LOG_DIR="\/${SPARK_1.2.1_HOME}\/build\/maven_artifacts"
```

Even though the maven_artifacts folder does not already exist it will be created once we start building Spark. What we want to avoid is the maven project to be installed in a non-secure or non-writable directory (e.g. .m2/repository). Therefore, in order to accomplish this we use the previously created MAVEN_LOG_DIR variable to replace the path of the local repository inside the apache-maven-3.2.5/conf/settings.xml file.

For instance, we can write the following command inside the mvn script after the declaration of the MAVEN_LOG_DIR variable and before the *${MVN_BIN} "$@"* command:

```
sed -ri -e "/^\s*<localRepository>.*<\/localRepository>/!b;n;n;c\
<localRepository>${MAVEN_LOG_DIR}<\/localRepository>" \
        ${SPARK_1.2.1_HOME}/build/apache-maven-3.2.5/conf/settings.xml
```

where the ${SPARK_1.2.1_HOME} variable with the actual path of the installation directory.

Step 7
By the time that all the steps are completed we can now proceed to convert the custom script into an executable one and run it (*sh spark_init.sh*) to start building Spark. The average building time will take around ~45 minutes.

9. Spark Nodes Configurations

Apache Spark uses Master-Worker architecture. In order to develop an Apache Spark cluster we have to firstly create a Spark Master in one of the cluster's nodes and one or more Spark Workers on the rest of the nodes. The Spark Workers in the Spark Framework can also be also defined as Executors. Spark Workers are the nodes, which will communicate with the Spark Master and they will create the Executors. To be more accurate, the Executors are the JVM processes which will handle the Spark jobs execution. Spark Master on the contrary is the node that will act as the coordinator between the Workers. Moreover, in order to submit and execute jobs in the Spark cluster a Cluster Manager is required to distribute the cluster resources. In this report we use the default built-in cluster manager, named Spark Standalone.

Spark Application

There are two ways that one can run jobs on the Spark Framework. One way is to initiate a driver using the spark-shell for interactive executions and the other one is to use the spark-submit script to run a job in the cluster without the need to refer to the interactive shell. In the second case we have to create a jar file, which will store all the dependencies of the work that needs to be done while in the first case we can run queries and experiment with the Spark workload interactively.

In order to start writing code in the Spark-shell we have to primarily understand how a Spark cluster works on the interactive mode. By starting the spark-shell we instantly create the Spark driver. The Spark driver is responsible for:

- coordinating the tasks across the Executors (inside the Spark Workers)
- delivering the tasks to the Executors in the appropriate location

The spark-driver altogether with the Executors connected to the spark-driver is called the Spark Application. Once we start a spark-shell, a Spark Application starts as well. The Spark Driver process communicates with the Spark Master in order to allocate the respective resources and thereafter distribute the tasks to the Executors.

Environmental Variables

Before start implementing the cluster we have to apply some configurations on each node separately as we did for the Cassandra nodes. Recall that at the end of section 4 we created the *remote_settings.env* file, which stored the environmental variables that needed to be accessible during the Cassandra installation. Similarly for Apache Spark we have to append some extra environmental variables inside the file, since they are required for the Spark nodes initialization. Following, we present what variables should be assign to every node with respect to its purpose. Specifically:

MASTER & WORKERS

- export MASTER_HOST_PORT=spark://<hostname of the spark master>:port
  e.g. export MASTER_HOST_PORT=spark://instance-trans1.c.bubbly-operator-90323.internal:7077

- export SPARK_HOME: <the location of the whole Spark package, it holds the path of the SPARK_1.2.1_HOME directory as we described earlier>

- export SPARK_DIR: the location of the external/secondary disk where Spark logs of Master and Workers will be saved. We are not obliged to refer to a secondary disk when using Spark for testing purposes we can simply assign any possible directory.

ONLY FOR MASTER

- export SPARK_DEFAULTS_FILE=<path to the file containing the required default values for deployment>

ONLY FOR WORKERS

- export MACHINE_IP=<the internal IP address of the node>
- export WORKER_CORES=<the number of cores that the Worker node is allowed to use on this machine>
- export WORKER_MEMORY=<total amount of memory that the Worker node is allowed to use for Executors>

The MASTER_HOST_PORT will be assigned the spark master's hostname followed by the port that we have reserved for the connection. Here, we use the default port 7077.

The SPARK_DEFAULTS_FILE is a variable that points to a text file, which contains crucial information regarding the settings that the Spark Cluster require to start and operate properly. The file has the following format:

<spark property 1> <value 1>
<spark property 2> <value 2>

This file (SPARK_DEFAULTS_FILE) will be used by the Spark Master to assign the respective properties to the spark configuration variable when the spark-shell begins. For our purposes, we will use a handful number of properties that are essential to set up the cluster and start our application across the cluster. More importantly, during the initialization of the Spark-Master we will copy the built-in *`${SPARK_HOME}/conf/spark-defaults.conf.template`* file to the file *`${SPARK_HOME}/conf/spark-defaults.conf`* where we will write all the necessary configuration settings that appear inside the SPARK_DEFAULTS_FILE. The higher goal of this modification is to load dynamically all the important settings for the spark config variable at the start of the spark-shell. We will explain shortly what is the content of this file. For now, we assume that the SPARK_DEFAULTS_FILE exists with no content.

A detailed description over which Spark properties can be used inside the SPARK_DEFAULTS_FILE can be found in the following link (https://spark.apache.org/docs/latest/configuration.html).

THE WORKER_CORES variable is the total number of cores that can be used by the Executors of each Worker machine.

The WORKER_MEMORY variable is the size of the system memory that can be used by the Worker node to create Executors on the machine.

Spark Master & Spark Worker

In this section we will define the bash scripts that are required for each Spark machine. We need to separate the script of the Spark Master from the script of the Spark Workers.

Spark Master script

The most interesting point is that the Spark project has all the pre-existing necessary development files with a *.template* extension. Therefore, we have to copy paste the ones that we need without the extension. We start by copying the *spark-env.sh.template* file since it is required to define essential properties for the Spark machines and then we continue by copying the *spark-defaults.conf.template* file, inside which we will write the contents of the SPARK_DEFAULTS_FILE. In particular, the contents of the Spark Master script, we will refer to it as *spark_master.sh*, could be as follows:

```
#! /bin/bash

#copy the necessary files from template extension to one without the extension
cp ${SPARK_HOME}/conf/spark-env.sh.template ${SPARK_HOME}/conf/spark-env.sh
cp ${SPARK_HOME}/conf/spark-defaults.conf.template ${SPARK_HOME}/spark-defaults.conf

#clear the logs directory – if we create multiple times the Master in the node and we want
#to trace any troubleshooting that may occur; it is sensible to erase the previously written log files
rm -rf ${SPARK_HOME}/logs/*

#we store the hostname of the specific Master Node
host=$(hostname -f)

#the SPARK_LOCAL_DIRS will be used by the Spark Driver for the RDD storage
#for sanity we change the Spark Master_IP and the Local IP for the Master Machine to resemble the
#Master's hostname. Moreover, we change the path of the default SPARK_WORKER_DIR in order to
#store the log files related to the running Spark applications
sed -i  -e "/^.*SPARK_LOCAL_DIRS.*/a export
SPARK_LOCAL_DIRS=${SPARK_DIR}\/node_${MACHINE_IP}_storage/" \
        -e "/^.*SPARK_MASTER_IP.*/a export SPARK_MASTER_IP=${host}" \
        -e "/^.*SPARK_LOCAL_IP.*/a export SPARK_LOCAL_IP=${host}" \
        -e "/^.*SPARK_WORKER_DIR.*/a export
SPARK_WORKER_DIR=${SPARK_DIR}/worker_dir/" \
        ${SPARK_HOME}/conf/spark-env.sh

#we write the SPARK_DEFAULTS_FILE properties inside the spark-defualts.conf file
cat ${SPARK_DEFAULTS_FILE}>${SPARK_HOME}/conf/spark-defaults.conf
```

Spark Worker script

Similarly as we did for the Spark Master script, here we will apply some identical configurations for the Spark Worker. Specifically, the content of the Spark Worker script (*spark_worker.sh*) could be as follows:

```
#! /bin/bash

#copy the spark-env.sh.template to spark-env.sh
cp ${SPARK_HOME}/conf/spark-env.sh.template ${SPARK_HOME}/conf/spark-env.sh
```

```
#create the necessary directories – If they do not exist. Storage_logs directory will be used for storing
#log stack trace information; worker_dir will be used as the working directory of the worker
#processes.
mkdir -p ${SPARK_DIR}/storage_logs
mkdir -p ${SPARK_DIR}/worker_dir


#clear the directories content every time that a Spark Worker starts, similarly as we did with the
#Spark Master
rm -rf ${SPARK_DIR}/storage_logs/*
rm -rf ${SPARK_DIR}/worker_dir/*

#now we change the content of the spark-env.sh file and we declare the properties for every Worker
#the SPARK_LOCAL_DIRS is used by the Spark Worker for the RDD data
#the SPARK_LOG_DIR is used by the Spark Worker for storing the log messages
sed -i  -e "/^.*SPARK_LOCAL_DIRS.*/a
SPARK_LOCAL_DIRS=${SPARK_DIR}\/worker_dir\/node_${MACHINE_IP}_storage/" \
        -e "/^.*SPARK_LOCAL_IP.*/a SPARK_LOCAL_IP=${MACHINE_IP}" \
        -e "/^.*SPARK_WORKER_DIR.*/a SPARK_WORKER_DIR=${SPARK_DIR}/worker_dir/"
\
        -e "/^.*SPARK_WORKER_CORES.*/a SPARK_WORKER_CORES=${WORKER_CORES}" \
        -e "/^.*SPARK_WORKER_MEMORY.*/a SPARK_WORKER_MEMORY=${WORKER_MEMORY}" \
        -e "/^.*SPARK_LOG_DIR.*/a SPARK_LOG_DIR=${SPARK_DIR}/storage_logs/" \
        ${SPARK_HOME}/conf/spark-env.sh
```

CLARIFICATION: The SPARK_WORKER_DIR in the Master node will be used by the Spark Driver; in future steps we will initialize the driver on this machine. The SPARK_WORKER_DIR in the Worker nodes will be used explicitly by the working machines.

Contents of the SPARK_DEFAULTS_FILE

Assuming that we have created and have written all the pre-mentioned variables inside the *remote_settings.env* file and we have created the scripts for the Spark Master and Workers, now we proceed with the actual creation of the SPARK_DEFAULTS_FILE. The contents of this file could be similar to:

spark.driver.cores 4
spark.driver.memory 2156m
spark.driver.host instance-trans1.c.bubbly-operator-90323.internal
spark.driver.port 3389
spark.broadcast.port 3456
spark.blockManager.port 7080
spark.executor.port 8080
spark.executor.memory 9024m
spark.eventLog.enabled true
spark.master spark://instance-trans1.c.bubbly-operator-90323.internal:7077
spark.cassandra.connection.host 10.240.160.14

We see that we assign the three random ports (spark.driver.port, spark.broadcast.port, spark.blockManager.port), which we have reserved during the initialization of the cluster. Moreover, we can see the maximum number of cores that the driver is able to use, as well as the memory that the driver and every single Executor can freely use. Cumulatively, we define the spark.cassandra.connection.host property, which refers to the Internal IP of the node that the Spark driver is running. This IP will be used by the driver to establish the connection between the Apache Spark and the Apache Cassandra cluster.

IMPORTANT: In the implementation we provide in this report we deploy the Spark Framework using one single Spark Executor per Spark Worker and starting the Spark Driver from the Spark Master machine. Moreover, inside the Spark Master machine we do not deploy any Spark Workers.

Running the Spark Master

Once we have created the needed scripts – `remote_settings.env, spark_master.sh -` we can create another script (*`master.sh`*) in which we source both scripts:

```
#! /bin/bash
source remote_settings.env;
source spark_master.sh
${SPARK_HOME}/sbin/ start-master.sh
```

The last line calls the built-in script of the Spark Project so to start the Spark Master. Afterwards, we alter the script's mode to executable and we execute it (*`sh master.sh`*) to initialize the Spark Master on the machine. If we want to avoid the log messages that will be generated during the Master initialization we can redirect the execution of the script to another file (e.g. *`sh master.sh>>master.output`*).

Running the Spark Worker

Equivalently for the nodes that will be used as Spark Worker nodes we can create a new bash script (`worker.sh`), where we will source the *`spark_worker.sh`* and *`remote_settings.env`* :

```
#! /bin/bash
source remote_settings.env;
source spark_worker.sh
${SPARK_HOME}/bin/spark-class org.apache.spark.deploy.worker.Worker ${MASTER_HOST_PORT}
```

The last command is essential for setting up the Spark Worker process. In particular, we pass the environmental variable ${MASTER_HOST_PORT} to the `spark-class` script in order to to establish the connection between the newly spawned Spark Worker and the existing Spark Master.

Thereinafter, we make the script executable and we call it (`sh worker.sh`) to initialize the Spark Worker in the node. It is recommended to redirect the log messages to another file as we did during the Spark Master initialization.

Cluster Verification

Steps to verify that the Spark Cluster is up and running:

1. connect to the Spark Master machine
2. go to the *${SPARK_HOME}/bin* directory
3. execute: `./spark-shell`

Once the shell starts initializing, the Spark logo should appear and explanatory messages coming from the Spark Driver and the Spark Executors should be displayed. The messages will contain the servers/machines internal IP along with the ports that were used to establish the connection. When the initialization of the spark-shell finishes the Scala shell (`scala> …`) should start as well. To terminate the shell we can type `exit` or press `CTRL+D`.

10. Building the Spark Cassandra Integration

Apache Spark and Cassandra integration requires that we download and we build the respective jar package, which will be used when we start the Spark-shell in the spark-driver. In order to perform so, we can build the jar package either by using the SBT assembly or the Apache Maven. In this section we provide a detailed description of the jar creation using the Apache Maven project build tool.

Firstly, we have to create a new directory inside which we will store the required mvn script and a new pom.xml file. We can utilize the mvn script that we have already used during the construction of the Apache Spark Project with some minor configurations. To be more precise, we have to reassign the MAVEN_LOG_DIR to the new directory we have created for the dependency jar and the replace statement (sed –ri –e …) should be operated over the setings.xml file that resides inside this new directory.

In detail, if we assume that the new directory is called *spark-cassandra-connector* we can assign the MAVEN_LOG_DIR as:

```
MAVEN_LOG_DIR="<path>\/spark-cassandra-connector\/maven_artifacts"
```

And the replace statement as:

```
sed –ri –e "…" <path>/spark-cassandra-connector/apache-maven-
3.2.5/conf/settings.xml
```

The pom.xml file that we can use for our Maven building purposes can be similar to the following; acknowledgment for this file (https://gist.github.com/maasg/8f5cbbb4cd62e90412c7). Given the above file we should change the first `<dependencies>…</dependencies>` tag to the version of the jar package we want to run the integration.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
```

```
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.acme</groupId>
    <artifactId>spark-cassandra-assembly</artifactId>
    <version>1.2.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>Cassandra-Spark Uber-jar driver for spark-shell </name>

    <properties>
        <maven-shade-plugin.version>2.2</maven-shade-plugin.version>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

      <!-- a more detailed list with all the available spark-cassandra integration jars can be
      found here: http://mvnrepository.com/artifact/com.datastax.spark/spark-cassandra-
      connector_2.10 -->
    <dependencies>
        <dependency>
            <groupId>com.datastax.spark</groupId>
            <artifactId>spark-cassandra-connector_2.10</artifactId>
            <version>1.2.0-rc3</version>
        </dependency>
    </dependencies>


    <build>
        <outputDirectory>target/classes</outputDirectory>
        <testOutputDirectory>target/test-classes</testOutputDirectory>

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>${maven-shade-plugin.version}</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <shadedArtifactAttached>true</shadedArtifactAttached>
                            <shadedClassifierName>jar-with-dependencies</shadedClassifierName>
                            <filters>
                                <filter>
                                    <artifact>*:*</artifact>
                                    <excludes>
                                        <exclude>META-INF/*.SF</exclude>
                                        <exclude>META-INF/*.DSA</exclude>
                                        <exclude>META-INF/*.RSA</exclude>
                                    </excludes>
                                </filter>
                            </filters>
                            <transformers>
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>com.datastax.spark.connector.SparkContextFunctions</mainClass>
                                </transformer>
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                                    <resource>reference.conf</resource>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

```
        </plugins>
    </build>
</project>
```

By the time that we have the pom.xml file and the mvn script in the same directory we can execute the following command inside the directory in order to start the building procedure:

```
./mvn -e -f ./pom.xml clean package;
```

After the building process is completed we will navigate to the */spark-cassandra-connector/target/* directory and we will use the *spark-cassandra-assembly-1.2.0-SNAPSHOT-jar-with-dependencies.jar* package during the initialization of the Spark-Shell. It is highly advisable the Spark Cassandra jar file building procedure to be performed inside the machine that we will launch the Spark Driver; in our case this machine should be the Spark Master node.

11. Running the Spark-Cassandra Shell

Having followed carefully all the previous steps we should have established by now a Cassandra Cluster with seed and simple nodes and a Spark Cluster with as many Spark machines as Cassandra nodes.

Now, we will connect to the Spark Master machine as we did during the Spark cluster verification steps and we will start the driver (`spark-shell`). The spark-shell relies to the usage of the Scala Programming Language. It is recommended to write queries and perform RDDs operations (action, transformations) using Scala due to its expressiveness and flexibility. By the time that we are inside the directory, we execute the following command to start the interactive shell:

```
./spark-shell --jars spark-cassandra-connector/target/spark-cassandra-assembly-1.2.0-SNAPSHOT-jar-with-dependencies.jar
```

We can see that by using the –-jars option we can add the previously created spark-cassandra uber jar to the classpath of the spark-shell in order to establish the connection between Cassandra and Spark. When the shell stops loading, we execute the following commands in order to import the required libraries, which will enable us to start the interactive mode between Spark and Cassandra:

```
import com.datastax.spark.connector._
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.sql.cassandra.CassandraSQLContext
```

As soon as the libraries have been loaded we create a new CassandraSQLContext variable:

val cc = new CassandraSQLContext(sc)

which will be used to execute SQL queries with respect to the principals of the correctness of the CQL queries. There are two major advantages over this integration. One is the ease of use of aggregated functions (MAX, COUNT, SUM, AVG etc..) on top of the CQL queries and the other is the capability of joining two Cassandra tables by leveraging the transformation operations of the RDDs (map, join, union).

An illustrative query that can be applied using the running spark-shell, which is based on the keyspace (highway) and the column family (highway_config) that we created with the Cassandra Python driver, is:

val rdd = **cc.sql**("SELECT count(*) from highway.highway_config")
rdd.collect()

From the above commands we have to stress out:

- every CassandraSQLContext variable must be used in the form of <variable_name>.sql("…") in order to perform queries over Cassandra
- every query returns an RDD (think of it as a transformation); in order to execute the query we have to perform an action over the returned RDD. Here, we use the `.collect()` action

The documentation for all the possible queries and integration between Apache Spark and Cassandra can be found at https://github.com/datastax/spark-cassandra-connector.

12. Summary

In this report we have presented thoroughly the fundamental steps that one requires to perform in order to easily set-up and deploy the Apache Cassandra No-SQL Database and the Apache Spark Framework on Linux based machines. Most importantly, we have presented how the communication between both the platforms can be established and how one can start using the Spark interactive shell to execute queries on top of the Cassandra Cluster.