

Master’s Thesis

Master in Data Science, Universidad Rey Juan Carlos

Academic Year: 2017-2018

Call Detail Records (CDR) Analysis

Miguel-Angel Monjas Llorente

[miguel-angel.monjas@ericsson.com](mailto:miguel-angel.monjas@ericsson.com)

[ma.monjas.2016@alumnos.urjc.es](mailto:ma.monjas.2016@alumnos.urjc.es)

Supervisor: Dr. Felipe Ortega, ETSIT, URJC

# Abstract

This project meets the need to analyze a large dataset containing six month’s Call Detail Records from a small European telecommunications operator, Operator X, when no infrastructure is previously available. Thus, before any analysis task is executed, a HDFS/Spark cluster is set up to enable data storage and processing, which is executed by means of Python notebooks. Elastic Stack is deployed and used for visualization. Finally, a Kafka message broker is deployed as well in order to enable real-time use cases.

Once the infrastructure is available, some exploratory data analysis operations are executed, mainly focused on finding and visualizing information about the subscribers and their terminals. Next, the load in the operator’s cells are computed and visualized. Visualization is carried out in two ways: inline, by means of matplotlib; and by using Kibana (part of the Elastic Stack). The cell graph is determined as well (the dataset does not contain information about the network topology). Finally, a streaming scenario is created and congestion is detected and visualized in real-time.

# Agradecimientos

A mis padres: hoy estoy aquí gracias a vosotros.

A Susana, que siempre ha confiado en mí y ha sido mi sustento incondicional. Este año no hubiese sido posible sin ti. A mis hijos, Raquel y Álvaro, Álvaro y Raquel. Todo lo hago por vosotros. A mi hermana. Siempre conmigo.

Nunca hubiese crecido, en lo personal y en lo profesional, sin mi grupo de trabajo de Ericsson. Ha sido un largo camino y, justo cuando leo esta tesis, la realidad, que llevaba tiempo persiguiéndonos, nos ha alcanzado. Ya no habrá más T&I. Alejandro, Carolina, Nacho, Miguel Ángel, José Miguel, Manuel… Mónica, Juan, Miguel Angel, David, Javi, Luis… siempre recordaré estos años con mucho cariño y con aún más nostalgia. Alejandro, no sé qué haría Rafa en este momento, pero Roger mordería el polvo. Seguro. Amigos como tú se encuentran pocas veces en la vida. Carolina, gracias por recordarme siempre que soy mejor profesional de lo que a menudo creo. Manuel, siempre luchaste por nuestro grupo, por innovar, y por dar valor a la compañía. Sin tu visión no me hubiese convertido en el profesional que soy hoy y posiblemente seguiría haciendo *paperware*. Gracias. A Mari Luz, gracias a la cual sigo, de verdad, dando guerra.

Román, seguimos compartiendo, desde la distancia, cafés virtuales. Y mucho más. Eres uno de los mejores profesionales que conozco, un amigo y un ejemplo.

Durante los dos últimos años, los becarios de la cátedra de Data Science lograron sacar de mí lo mejor (o, al menos, eso creo). Héktor, sé que fue divertido.

Tengo también que expresar mi agradecimiento al cuerpo docente del máster y, sobre todo a mi tutor, Felipe Ortega. Habéis hecho un programa coherente y muy enfocado. Al igual que esta tesis, podría haber tenido el doble (o el triple) de contenido pero, aun así, ha cumplido con creces las expectativas.

¡Gracias a todos!

Contents

[Abstract 2](#_Toc501532119)

[Agradecimientos 3](#_Toc501532120)

[1 Objectives 6](#_Toc501532121)

[2 Deliverables 8](#_Toc501532122)

[3 Deployment 9](#_Toc501532123)

[3.1 Technology selection 9](#_Toc501532124)

[3.2 Infrastructure deployment 10](#_Toc501532125)

[3.2.1 Instance creation 10](#_Toc501532126)

[3.2.2 HDFS cluster setup 12](#_Toc501532127)

[3.2.3 Spark Standalone cluster setup 14](#_Toc501532128)

[3.2.4 Python installation in the Spark Standalone cluster 16](#_Toc501532129)

[3.2.5 Jupyter notebook server installation on a Spark Standalone cluster 17](#_Toc501532130)

[3.2.6 Zeppelin notebook server installation on a Spark Standalone cluster 18](#_Toc501532131)

[3.2.7 Elastic Stack setup 19](#_Toc501532132)

[3.2.8 Kafka setup 22](#_Toc501532133)

[4 Execution 23](#_Toc501532134)

[4.1 Data understanding 23](#_Toc501532135)

[4.2 Data storage 24](#_Toc501532136)

[4.3 Data analysis and visualization 26](#_Toc501532137)

[4.3.1 Size of the dataset 28](#_Toc501532138)

[4.3.2 Terminal analysis 29](#_Toc501532139)

[4.3.3 Subscriber analysis 37](#_Toc501532140)

[4.4 Cell load computation and visualization 43](#_Toc501532141)

[4.4.1 Spark processing 43](#_Toc501532142)

[4.4.2 Elasticsearch management 44](#_Toc501532143)

[4.4.3 Kibana visualization 47](#_Toc501532144)

[4.5 Cell graph computation 50](#_Toc501532145)

[4.6 Congestion management 52](#_Toc501532146)

[4.6.1 Data source simulation 53](#_Toc501532147)

[4.6.2 Spark processing 53](#_Toc501532148)

[4.6.3 Elastic Stack management 55](#_Toc501532149)

[5 Open issues and further work 58](#_Toc501532150)

[6 Conclusions 59](#_Toc501532151)

[References 60](#_Toc501532152)

Images

[Figure 1: Project infrastructure components 10](#_Toc497214804)

[Figure 2: Project infrastructure 11](#_Toc497214805)

[Figure 3: Openstack UI 12](#_Toc497214806)

[Figure 4: HDFS Web Interface 13](#_Toc497214807)

[Figure 5: Spark Context UI. No available jobs 15](#_Toc497214808)

[Figure 6: Spark Standalone Cluster UI 15](#_Toc497214809)

[Figure 7: Spark Notebook 18](#_Toc497214810)

[Figure 8: Main Zeppelin UI home 19](#_Toc497214811)

[Figure 9: Main Kibana UI home 21](#_Toc497214812)

[Figure 10: HDFS Web Interface: it shows the December 2015 uploaded file 25](#_Toc497214813)

[Figure 11: Jupyter Notebook server home page 27](#_Toc497214814)

[Figure 12: CDR type query executed with Zeppelin 29](#_Toc497214815)

[Figure 13: Top terminals in Operator X network 31](#_Toc497214816)

[Figure 14: Top manufacturers in Operator X networks 32](#_Toc497214817)

[Figure 15: Top operating systems in Operator X network 32](#_Toc497214818)

[Figure 16: Top M2M vendors in Operator X network 33](#_Toc497214819)

[Figure 17: Monthly terminal share evolution among the Operator X subscribers 34](#_Toc497214820)

[Figure 18: Monthly vendor share evolution among the Operator X subscribers 34](#_Toc497214821)

[Figure 19: Terminal activation in Operator X 36](#_Toc497214822)

[Figure 20: Spark Context UI: Stages for All Jobs 36](#_Toc497214823)

[Figure 21: Top countries in the Operator X network 38](#_Toc497214824)

[Figure 22: Top daily countries in the Operator X network 39](#_Toc497214825)

[Figure 23: Evolution of country share (top three countries) over time 40](#_Toc497214826)

[Figure 24: Evolution of country share (top three countries) in the Christmas season 40](#_Toc497214827)

[Figure 25: Duration of roamers stay in Operator X network 41](#_Toc497214828)

[Figure 26: Short-term visitors (less than two days) in the Operator X network 42](#_Toc497214829)

[Figure 27: Operator split for top countries among visitors to Operator X network. 43](#_Toc497214830)

[Figure 28: cell\_info\* Index Pattern at Kibana 47](#_Toc497214831)

[Figure 29: Active cells in Operator X (Kibana) 49](#_Toc497214832)

[Figure 30: Top loaded cells in Operator X (Kibana) 49](#_Toc497214833)

[Figure 31: Traffic in Operator X network (Kibana) 50](#_Toc497214834)

[Figure 32: Spark Context UI: DAG for a count() action executed after dataframe transformations 52](#_Toc497214835)

[Figure 33: Congestion status (Kibana) 57](#_Toc497214836)

# Objectives

Data Science is a broad discipline that ranges from so-called Big Data to Machine Learning. A data scientist is expected to make sense of data and extract as much knowledge as possible. Thus, s/he must be able to select the best algorithms to handle data, but also to set up the necessary infrastructure to make the magic work, and to visually present the findings.

In an ideal life, data scientists would be given the means to quickly test their hypothesis with reduced datasets and provide the algorithms and the code that enable knowledge extraction. In real life, this is not always the case and many times, before starting to code, necessary infrastructure must be set up.

This master thesis describes all the phases of a simple data science project: a large dataset is provided, and some exploratory data analysis findings are requested. The dataset contains the six month’s Call Detail Records (CDR) of a telecom operator, Operator X,[[1]](#footnote-1) which serves a small country in Europe and is mainly interested in knowing about the roamers (i.e. subscribers from abroad served by Operator X when they are in the territory served by Operator X). No infrastructure exists for carrying out this data analysis task and therefore it is necessary to set it up, considering that it must be available for future Data Science projects.

There are several constraints upon the project. They lie on the actual data and its transformation and on the available infrastructure:

* Data has been provided only for carrying out the analysis and therefore, **cannot be transferred to third parties**. That means that it is not possibly to use the capabilities provided by Amazon Web Services or any other cloud provider and accordingly, the analysis will have to be executed in-house.
* A **general-purpose OpenStack cloud infrastructure**[[2]](#footnote-2) is available and consequently will be used to deploy the data analysis infrastructure in. However, although creation of new images and instances is available, it is not possible to administer the OpenStack cloud. That means that it is not possible to install new plug-ins or functionalities (let’s think, for example, of Sahara).[[3]](#footnote-3)
* **Additional privacy safeguards** have been built into the dataset by the data generator: on one hand, user and phone identifiers have been pseudonymized and hence it is not possible to track back the new identifiers to real identities in a straightforward way. On the other, although cell identifiers are provided, no cell location is given. Thus, it is not possible to take advantage of location information to track back user identities based on behavior patterns. This adds an additional drawback, as it is not possible to know which cells are contiguous to each other (this information is important in congestion-related situations, as alleviation measures to be adopted when a congestion situation is detected must be taken not only in congested cells, but also in adjoining areas).

No formal requirements specification is provided but a set of user stories,[[4]](#footnote-4) written from the perspective of users inside the Operator X:

As a responsible of roaming agreements, I want to get information about roamers into my network so that I can negotiate better agreements with other operators abroad.

As a network operations responsible, I want to have the means to detect the load of cells in real or almost real-time, so that I can apply alleviation measures.

This master thesis describes how the specific objectives of the project have been fulfilled. In particular, the following sections are available: Section 2 lists this project deliverables. A detailed view of how to deploy the infrastructure to execute the project is provided in section 3. Section 4 details the execution of the project and how the user stories have been implemented. Opportunities for improvement, suggested new technologies or approaches are listed in section 5. Finally, in section 6 some of the reached conclusions will be written down.

# Deliverables

The project has generated the following deliverables:

* This master’s thesis.
* A public GitHub repository, <https://github.com/miguel-angel-monjas/master-thesis.git>, with several folders:
  + The doc folder contains the documentation associated to the infrastructure deployment (summarized in this document in section 3.2).
  + The elastic folder stores all outcomes related to the Elastic Stack functionalities: Elasticsearch indices, Logstash filters, Kibana objects (visualizations and dashboards) … These deliverables are outcomes of sections 4.4.2, 4.4.3, and 4.6.3.
* A public GitHub repository, <https://github.com/miguel-angel-monjas/docker-elastic.git>, a submodule of the previous one, which provides the Docker-related files needed to launch the Elastic Stack components (see section 3.2.7).
* A private GitHub repository, <https://github.com/miguel-angel-monjas/master-thesis-notebooks.git>, a submodule of the first one, which contains the notebooks implementing the Spark processing and the generation of plots (sections 4.3, 4.4, 4.5, and 0). The repository is private as it contains confidential information.[[5]](#footnote-5)

# Deployment

Unlike regular Agile methodologies, no formal split of the existing user stories will be carried out. Instead, as many tasks as needed to provide the outcomes requested by the user stories will be described:

* Technology selection. In this task, the most suitable technologies to implement the project will be selected.
* Infrastructure deployment. Next, the analysis infrastructure will be deployed in the Openstack cloud, as automatically as possible.
* Data preparation and understanding. After verifying that the data dictionary is available and the datasets follow it, any other additional data source will be accessed or retrieved.
* Data storage so that it becomes available for subsequent analysis.
* Exploratory data analysis with focus on roaming information. Basic analysis of dataset, visualization generation and results storage will be accomplished.
* Computation of the load of each cell during the considered period and visualization of the load as time series.
* Generation of the graph of the operator cells, detecting contiguous cells.
* Simulation of congestion scenario and visualization.

## Technology selection

Technology selection is driven by requirements but also by budget, by competence, and by personal preferences. In this project, the following technologies are thoroughly used:

* **Apache Hadoop** (2.7.2).[[6]](#footnote-6) It enables massive storage (HDFS) on commodity hardware and provides also a resource manager for clusters: YARN. The project input datasets and outcomes will be persisted in the HDFS cluster.
* **Apache Spark** (2.0.2).[[7]](#footnote-7) Processing engine used for carrying out analysis task on the dataset. At least, one cluster configuration will be tested: Standalone. YARN will be considered if time is available. In any case, HSFS and Spark will run on the same cluster.
* Apache Spark applications are run interactively by means of **Python** 2.7 notebooks via **Jupyter**. Inline visualization is based on **matplotlib**. If time is available, **Zeppelin** (0.7.2)[[8]](#footnote-8) will be also tested. Python and its packages and associated functionalities are handled by means of the **Anaconda Distribution** 4.4 for Python 2.7[[9]](#footnote-9). Some additional Python packages are also used:
  + **findspark**[[10]](#footnote-10) (a Python module that allows to call PySpark from any Python script; as we plan to trigger notebook execution by running the pyspark command, it is not actually needed);
  + **seaborn**,[[11]](#footnote-11) to enhance visualization in notebooks;
  + the necessary packages to save notebooks as PDF files.
* The **Elastic Stack** (Elasticsearch/Kibana/Logstash) (5.6.1) [[12]](#footnote-12) will be used for the visualization of the cell loads in both static and dynamic scenarios.
* **Apache Kafka** (0.8).[[13]](#footnote-13) Message broker used in the simulation of the congestion scenario. On one hand, to dispatch communication events (start/stop) to the Spark Streaming applications. On the other, to deliver the results of the dynamic computation of cell load to the Elastic Stack (via Logstash).
* **Docker Community Edition** (CE) [[14]](#footnote-14) and **Docker Compose**[[15]](#footnote-15) will be used to deploy Apache Kafka and the Elastic Stack.

There are two architectural approaches: one of them focuses on batch processing; the other implements a streaming-based use case:

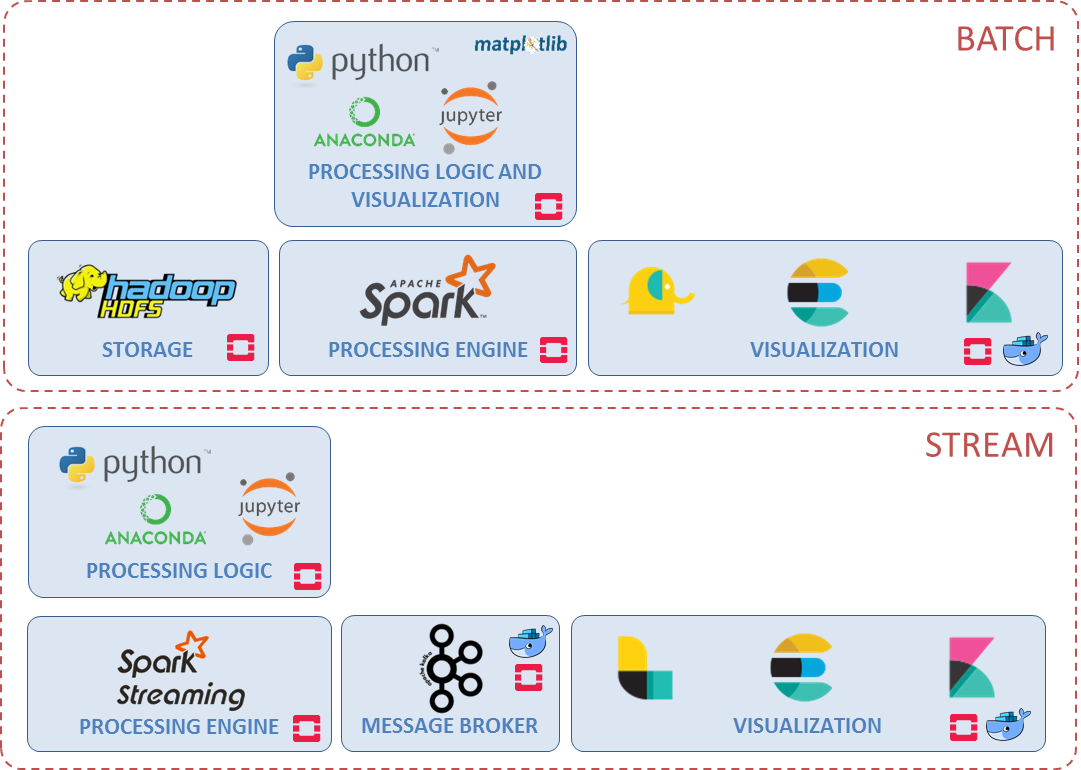


Figure 1: Project infrastructure components

## Infrastructure deployment

The project aims to analyze a large amount of data (about 300 GB with about 55 million records) and therefore will require the setup of an HDFS cluster (to handle storage) and a Spark cluster, for processing. Additionally, there will be dedicated instances for visualization (a node hosting an ELK Stack), and for the Kafka message broker. Applications will be run as Jupyter and Zeppelin notebooks from the master instance in the cluster.

### Instance creation

The following instances will be created:

* Four instances for the HDFS/Spark cluster: one master (for the HDFS and Spark cluster manager) and three slaves. The master instance will not host any HDFS *DataNode* or Spark *Worker*. The master will run also the Spark *Driver*, and a notebooks server as well.
* One instance for hosting a Dockerized ELK Stack.
* One instance for hosting a Dockerized Kafka message broker.

As with any other cloud-based solution, instances in Openstack are assigned an authentication key on creation so that no need to provide credentials at login is needed any more. Instead, ssh-based login by means of a private key available at the client is enabled. That key is assigned to the user ubuntu. This default user has sudo privileges as well.

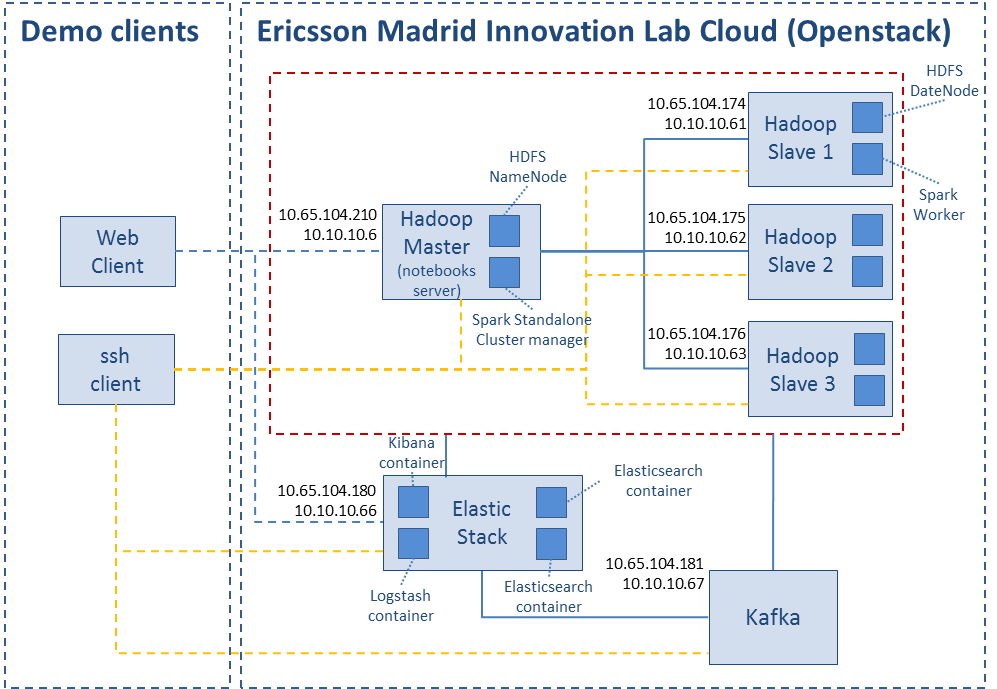


Figure 2: Project infrastructure

#### HDFS/Spark cluster instances

Four instances will be created, with the following hostnames cluster-master, cluster-slave-1, cluster-slave-2 and cluster-slave-3. The OS used is Ubuntu 16.04 and the flavor (spark-data-intensive) has the following features:

* Memory: 32 GB.
* vCPU: 16.

Each instance is given a private IP address, <master-ip-address>, <slave-1-ip-address>, <slave-2-ip-address> and <slave-3-ip-address>. A floating IP address must be manually assigned to each instance: <master-floating-ip-address> (slave instances are also assigned floating IP addresses, as it eases debugging; otherwise, access would be only possible through the master instance, by means of ssh connections).

#### Elastic Stack instance

A single instance will be created, with the following hostname: elk. The OS used is again Ubuntu 16:04 and the flavor (data-intensive) provides the following features:

* Memory: 16 GB.
* vCPU: 4.

This instance is given a private IP address: <elastic-ip-address> and a floating IP address, <elastic-floating-ip-address> as well. Otherwise, it will not be possible to access Kibana to visualize the information stored in Elasticsearch.

#### Kafka instance

An instance with similar configuration to that of the Elastic Stack instance is created, with the following hostname: kafka. This instance is given a private IP address: <kafka-ip-address> and a floating IP address, <kafka-floating-ip-address> as well.

To ease host name handling, the private IP addresses of all the instances is mapped to a meaningful host name in the /etc/hosts file in each instance: cluster-master, cluster-slave-1, cluster-slave-2, cluster-slave-3, elk and kafka.

The list of instances can be seen in the Openstack UI:

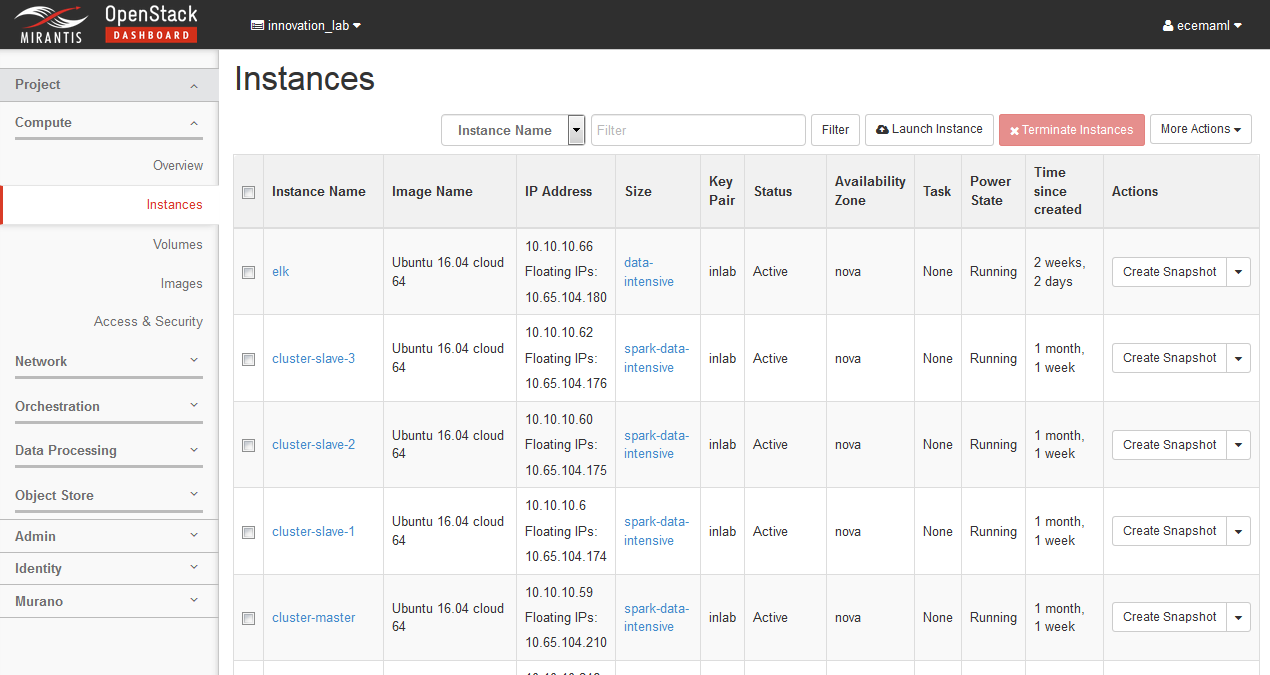


Figure 3: Openstack UI

### HDFS cluster setup

HDFS is the primary distributed storage used by Hadoop applications. An HDFS cluster consists of a *NameNode* that manages the file system metadata and *DataNodes* that store the actual data.[1]

The HDFS cluster is set up according to the official Hadoop documentation [2] and other resources found on the Internet.[3] It is made of four nodes: one master (cluster-master) and three slaves (cluster-slave-1, cluster-slave-2 and cluster-slave-3). Slaves run a *DataNode* each, while only a *NameNode* runs on the master node (initially, a configuration where a *DataNode* was also deployed on the master node was attempted, but it did not work, as the *NameNode* run out of resources frequently). Thus, the cluster manager will host only a *NameNode*.

To set up the HDFS cluster, and once the instances have been deployed, the following steps are taken:

* **Java installation** on all the instances: Oracle Java 8 is installed.
* **Hadoop** is needed to setup the HDFS cluster: as mentioned in 3.1, Hadoop 2.7.4 has been chosen. It is installed and configured on all the instances. Configuration involves the appropriate configuration of paths and environment variables. The location where Java is available is also configured (in hadoop\_env.sh).
* **Ssh installation** on all the instances. Ssh is required to enable password-less communication between master and the slave instances in the cluster.
* **Ssh password-less setup**. An ssh key-pain is created out of the box. The private key and the authorized\_keys file is uploaded to the master instance. Finally, the authorized\_keys file is uploaded to the slave instances.
* **Instance configuration**: Three configuration files must be updated on master and slave instances to have the cluster configured: core-site.xml, hdfs-site.xml, and slaves (it is important to consider that some variables become deprecated as new versions of Hadoop come out). Although there are some options that are only relevant for the master, it is simpler to copy the same configuration files to all the instances in the cluster. The configuration includes variables such as: the location of the HDFS filesystem (fs.defaultFS), the replication factor, the locations of the *DataNode* and *NameNode* folders, and the list of available slave instances.
* **HDFS filesystem format**: it is done via the *NameNode* by means of the HDFS File System shell.[4]

Once installed, the status of the HDFS cluster can be verified at http://<master-floating-ip-address>:50070/:

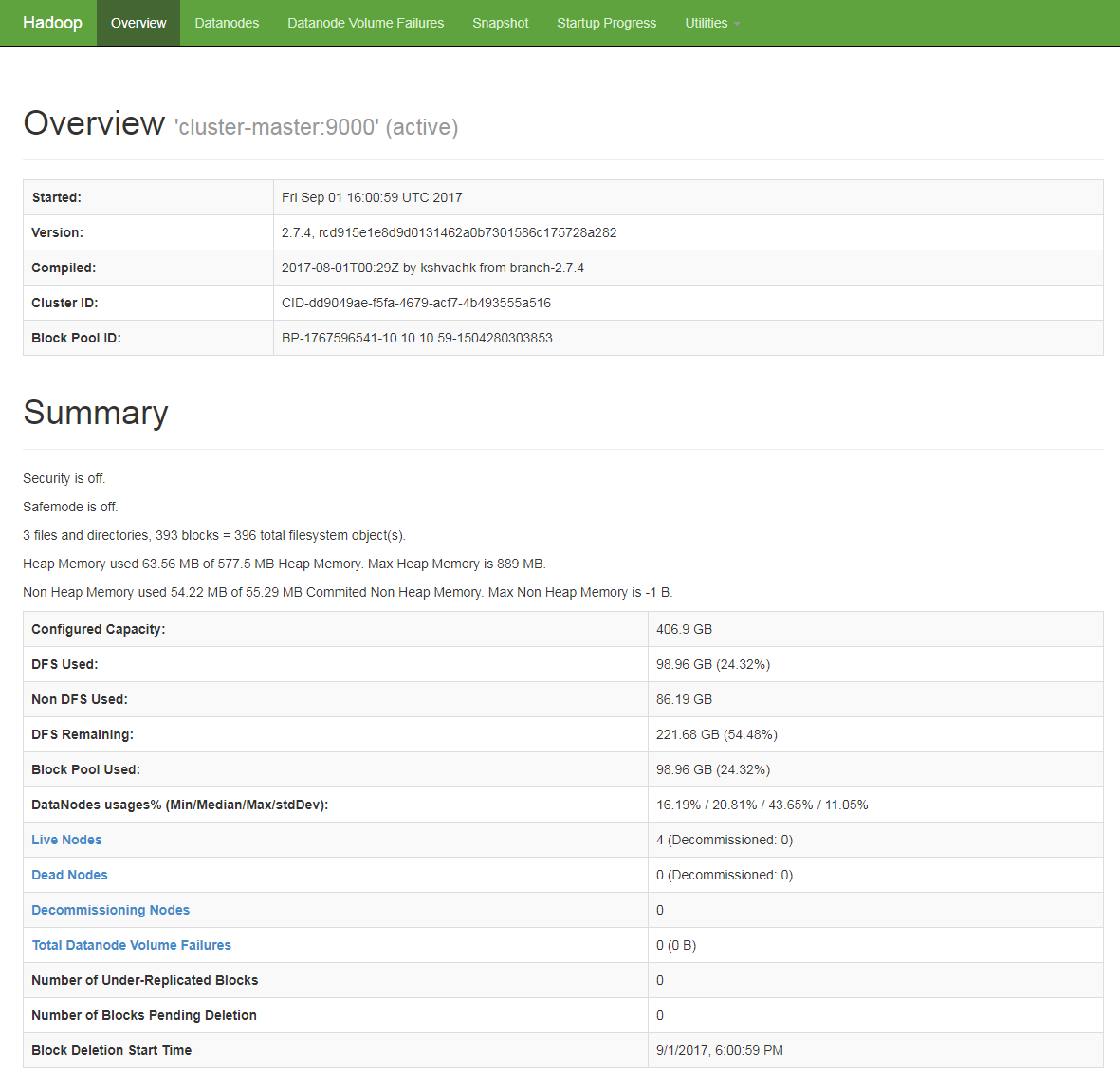


Figure 4: HDFS Web Interface

**Key takeaways**: Works such as *Spark in action* (Manning, 2017)[[16]](#footnote-16) state that "The installation [of YARN and Hadoop] is straightforward". However, depending on the environment it can be not true. The key issues addressed when setting up the HDFS cluster in the considered scenario (Openstack cloud with Ubuntu 16.04 instances) are the following ones:

* Private IP addresses must be used when referring to the cluster instances in the configuration files. If floating IP addresses are used, it will be not possible to connect any instance to each other. It is possible to override this behavior and use the floating IP addresses by setting the properties \*-bind-host in hdfs-site.xml to 0.0.0.0 (see the official guidelines for HDFS multihoming environments),[5] but Spark does not support this kind of configuration.
* Careless password-less ssh setup may lead to exposing the cloud private IP address: it is much more secure to use a key pair just for the cluster. It is created out of the box and the private key uploaded to the master instance. Slave instances do not even need the public key, but only the authorized\_keys file created as part of the key pair generation process.
* The *NameNode* is a single point of failure in the HDFS cluster. Hosting the *NameNode* and a *DataNode* in the same instance may translate into resource shortage in the *NameNode* and the usual Name node is in safe mode. Resources are low on NN. Please add or free up more resources then turn off safe mode manually error message. Although the deployment started initially with a *DataNode* on the master instance, it was finally removed.

The detailed description of the cluster setup can be found at <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/hadoop-cluster-setup.md>.

### Spark Standalone cluster setup

In the previous section, a description on how to set up an HDFS cluster has been provided: it is made of a master node (the HDFS cluster manager, hosting an HDFS *NameNode*) and three slaves (running each an HDFS *DataNode* each). Here, a concise description on how to deploy a Spark cluster in Standalone mode on top of it is provided as well (as YARN is not used, no dependencies between the HDFS and the Spark clusters will exist and HDFS will simply provide storage capabilities). The cluster deployment follows the official documentation.[6]

To set up the cluster, and provided that the HDFS cluster has been already deployed (for instance, it means that Java has been deployed on all the cluster instances), the following steps are taken:

* A **Spark** release compatible with Hadoop 2.7.4 and Zeppelin 0.7.2, Spark 2.0.2, is chosen. It is installed and configured on all the cluster instances. Configuration involves the appropriate configuration of paths and environment variables. The location where Java is available is also configured (in spark-env.sh).
* Appropriate **logging** options are configured. Following the suggestion from *Spark in action* (chapter 2) log4j.properties is updated on master and slaves.
* Finally, the **slaves configuration file** must be created, only on the master node, containing the hostnames of all the slaves in the cluster.

As a result, there will be a Spark Worker in each of the slaves and a Spark cluster manager in the master instance. It is possible to determine whether Spark and the Spark cluster have been properly installed by running the spark-shell command with the right master:

spark-shell --master spark://cluster-master:7077

Besides some warnings, the output should be something like this (to exit the Spark shell, type CTRL-D):

Spark context Web UI available at http://cluster-master:4040  
Spark context available as 'sc' (master = local[\*], app id = local-1503914008988).  
Spark session available as 'spark'.  
Welcome to  
 \_\_\_\_ \_\_  
 / \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_  
 \_\ \/ \_ \/ \_ \`/ \_\_/ '\_/  
 /\_\_\_/ .\_\_/\\_,\_/\_/ /\_/\\_\ version 2.0.2  
 /\_/  
  
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_144)  
Type in expressions to have them evaluated.  
Type :help for more information.

Before leaving the shell, the status of the Spark context created by the shell can be verified at http://<master-floating-ip-address>:4040/:

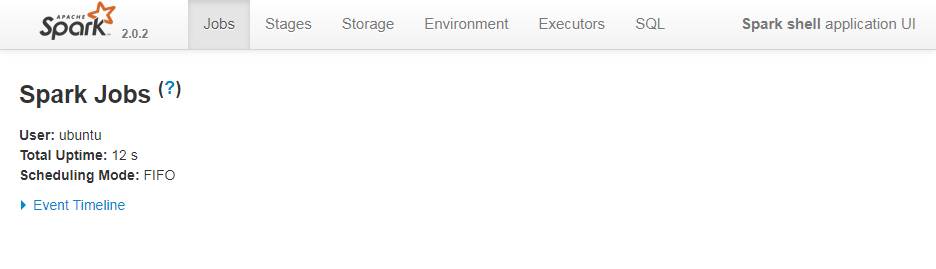


Figure 5: Spark Context UI. No available jobs

Once the cluster is started, its status can be verified at http://<master-floating-ip-address>:8080/. There, three workers must be listed:

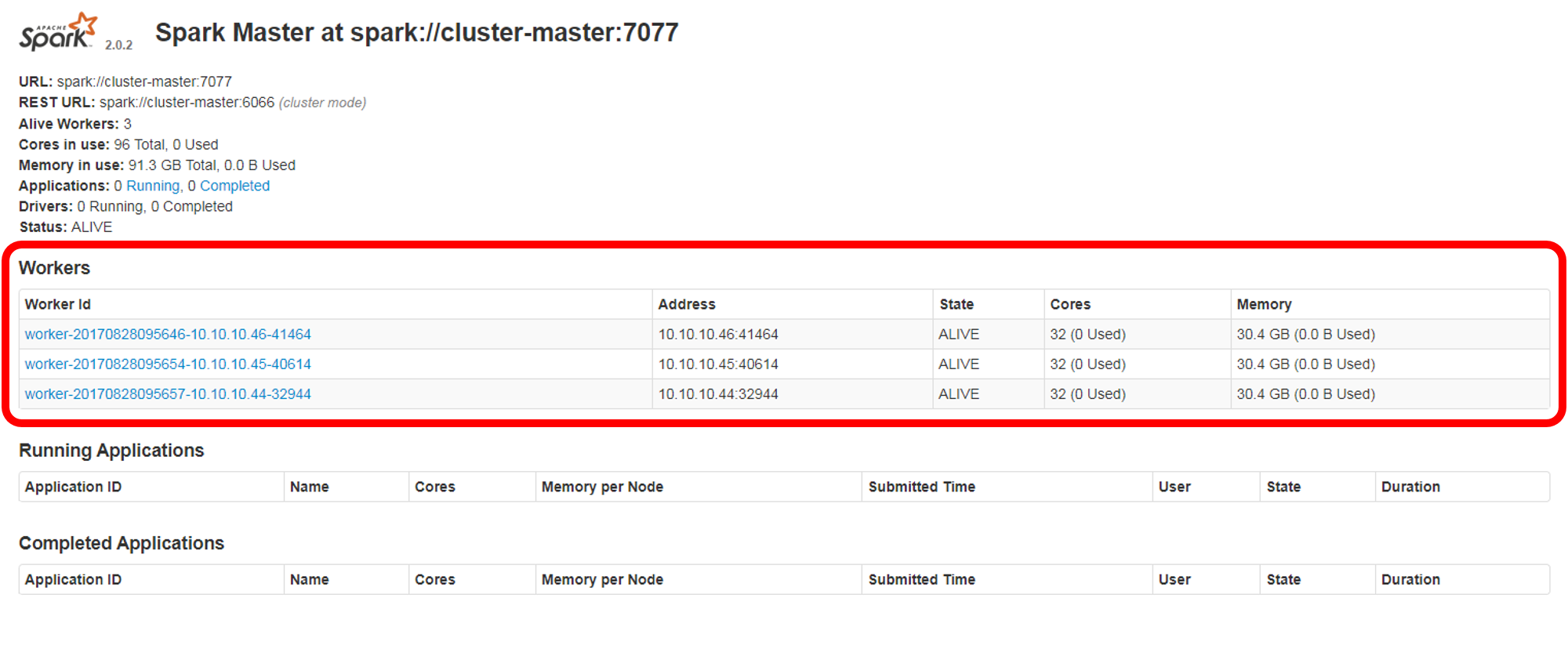


Figure 6: Spark Standalone Cluster UI

**Key takeaways**: as with Hadoop, installation and deployment of Spark is supposed to be simple and straightforward. Downloading, unpacking and minimal configuration would be enough for starting to work (tuning is necessary, but even with default options, Spark should offer the standard functionality). The key issues addressed when setting up the Spark cluster in the considered scenario (Openstack cloud with Ubuntu 16.04 instances) are the following ones:

* Interaction between Spark and other Hadoop packages must be carefully planned: From release 2.1.0 onwards, the Spark binaries have been created with built-in Hive support and that can mean specific problems when interacting with Hadoop if the HSFS cluster is configured but not started. In said situations, when running spark-shell, the following error pops up: java.lang.IllegalArgumentException: Error while instantiating 'org.apache.spark.sql.hive.HiveSessionState'. Starting the cluster or unsetting the environment variable $HADOOP\_CONF\_DIR would be enough.
* Hive support means that a derby.log file and a metastore\_db folder are always created in the current directory (where spark-shell is run, for instance). The best way to reduce the burden related to Hive is to appropriately configure Spark (in spark-defaults.conf) to make such files be created in a specific default folder, on the master node.[7]
* At the time of implementing this project, the last Zeppelin release, Zeppelin 0.7.2, does not support Spark releases from 2.2.0 onwards.

The detailed description of the Spark Standalone cluster setup can be found at <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/spark-standalone-cluster-setup.md>.

### Python installation in the Spark Standalone cluster

PySpark requires the same minor version of Python in both driver and workers. As, by personal preference, Python 2.7 has been selected, a Python 2.7 Anaconda distribution is used. It installs not only Python 2.7 but several valuable Python packages and Jupyter Notebook as well. The following steps are taken: The **Anaconda Distribution** 4.4 for Python 2.7 is downloaded, installed, and configured on all the cluster instances. Configuration involves the appropriate configuration of paths and environment variables. Additional **Python packages** (findspark, seaborn…) are installed from the Conda cloud.

Verification of a right Python 2.7 installation can be done by typing python. The output should be like this:

Python 2.7.12 |Anaconda custom (64-bit)| (default, Jul 2 2016, 17:42:40)  
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
Anaconda is brought to you by Continuum Analytics.  
Please check out: http://continuum.io/thanks and https://anaconda.org  
>>>

Next, verification of PySpark availability can be carried out by running pyspark (or pyspark --master spark://cluster-master:7077, if verification against the cluster is wished). The output should be similar to this:

Welcome to  
 \_\_\_\_ \_\_  
 / \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_  
 \_\ \/ \_ \/ \_ \`/ \_\_/ '\_/  
 /\_\_ / .\_\_/\\_,\_/\_/ /\_/\\_\ version 2.0.2  
 /\_/  
  
Using Python version 2.7.12 (default, Jul 2 2016 17:42:40)  
SparkSession available as 'spark'.

### Jupyter notebook server installation on a Spark Standalone cluster

Jupyter Notebook[[17]](#footnote-17) is a web-based application initially created for the Python language (but supporting right now many others) that allows creating *notebooks documents* by means of a web browser and executing it by means of a *kernel*. Jupyter Notebooks are ideal means for running PySpark applications and therefore will be one of the main choices when carrying out projects like the one being tackled. The instance where Jupyter is running, which plays the Spark Driver role, needn't be the master node or even one of the cluster instances. However, for the sake of simplicity, the notebooks will be run from the master instance in the cluster.

In the previous section, an Anaconda Distribution was installed and configured and therefore, Jupyter Notebook has become installed on the cluster as well. Just proper configuration is needed to enable the creation of execution of PySpark notebooks against the Spark cluster. The following steps are taken:

* **Generation of a configuration file for Jupyter Notebook** (jupyter\_notebook\_config.py) on the master instance.
* **Actual configuration** to: a) set the folder where the Notebook server starts from; b) enable access to the Notebook server from clients other than the localhost; and c) define the port the Notebook server should listen to (9999). This configuration sets up a public Notebook server. There is no built-in security and that is acceptable as a private Openstack cloud is being used. In open environments, the server must be secured with a password and TLS.[8]
* Finally, **Spark is configured** to run a notebook when pyspark is invoked (by setting the PYSPARK\_DRIVER\_PYTHON and PYSPARK\_DRIVER\_PYTHON\_OPTS environment variables in spark-env.sh).

The execution of Jupyter Notebook is triggered when pyspark is run. The Jupyter Notebook server becomes accessible at http://<master-floating-ip-address>:9999/:

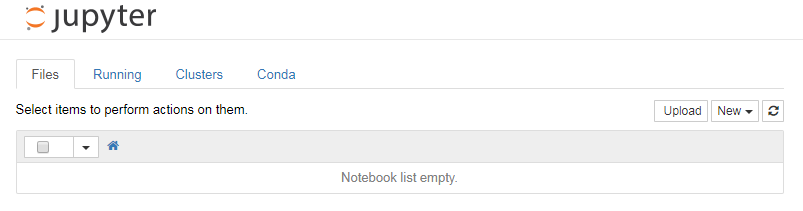


Figure 7: Spark Notebook

Once the *kernel* is loaded, it is possible to start to create and execute notebooks. The Jupyter Notebook server can be terminated by typing CTRL-C.

The detailed description of the Python and Jupyter Notebook on the cluster can be found in <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/jupyter-setup.md>.

### Zeppelin notebook server installation on a Spark Standalone cluster

Apache Zeppelin is another web-based Notebook that supports a variety of languages and back-end technologies (interpreters, according to the Zeppelin parlance). It is especially useful the possibility to combine several languages at the same notebook (for instance, shell to run HDFS commands and python to execute tasks on pyspark). However, the last available release (Zeppelin 0.7.2) [9] does not support Spark 2.2.\*. This is the main reason not to use the more up-to-date Spark version.

As with Jupyter, the instance where Zeppelin runs needn't be the master node or even one of the cluster instances. However, for the same reason, the Zeppelin notebooks will be run from the master instance in the cluster as well. The following steps are taken:

* **Zeppelin 0.7.2** is downloaded and installed.
* Besides paths and environment variables, **additional configuration** is needed to a) set the folder where the Zeppelin notebooks (notes in Zeppelin parlance) are stored (same as with Jupyter);[[18]](#footnote-18) b) set the Spark location (SPARK\_HOME) in the instance and cluster master location (spark://cluster-master:7077); and c) define the port the Notebook server should listen to (the default port where the Zeppelin notebook is exposed is 8080; it creates a conflict with the Spark cluster UI and therefore, a different port is used: 8180). The configuration file is zeppelin-env.sh.

Once started, the Notebook server is available at http://<master-floating-ip-address>:8180/. The result should be like the one in the screen below:

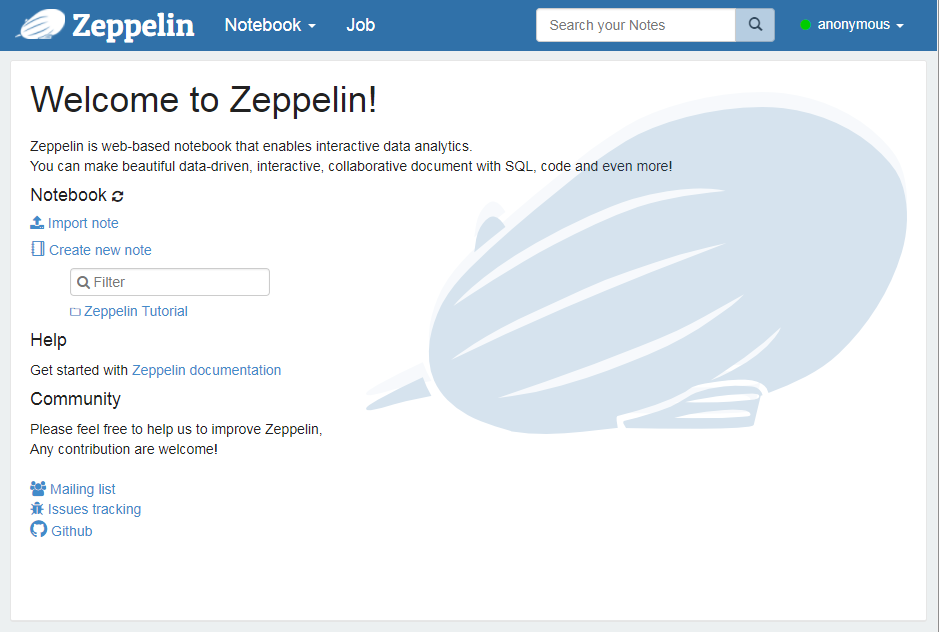


Figure 8: Main Zeppelin UI home

The detailed description of the Zeppelin notebook server setup can be found in <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/zeppelin-setup.md>.

### Elastic Stack setup

The Elastic Stack (formerly the ELK Stack) is a set of open source tools to search, analyze and visualize data in real time. Although the combination of Jupyter notebooks with matplotlib provides a compelling way of showing results, there are situations in which a more powerful tool is required. Thus, it is necessary to set up an Elastic Stack instance and make it accessible from a Spark Standalone cluster so that it is possible to write the results of the Spark computation to an Elasticsearch index.

The deployment of Elastic Stack on an Openstack instance will be carried out by means of Docker containers that run official Elastic images (release 5.6.1). Thus, Docker Community Edition (Docker CE) and Docker Compose must be installed before the containers can be created. Adapted from the official Docker documentation on Docker CE [10] and Docker Compose,[11] a shell script has been created. It is available in the docker folder within the master-thesis repository at GitHub (<https://github.com/miguel-angel-monjas/master-thesis/blob/master/docker/docker_install.sh>).

Deployment of the Elastic Stack environment (with two Elasticsearch instances, and one each for Kibana and Logstash) is based on the *Docker ELK stack* distribution by Anthony Lapenna (deviantony) [12] and on the official Elastic documentation.[13] Logstash is installed as well to enable integration scenarios with Spark Streaming (see section 4.6.3). All necessary files (with a specific Docker Compose file) are available as a submodule within the master-thesis repository at GitHub. Thus, after cloning the repository (<https://github.com/miguel-angel-monjas/docker-elastic.git>), Docker Compose can be run.

Some aspects to remark:

* A convenient feature of deviantony's distribution is the possibility of introducing specific configuration options in each service defined in the Docker Compose file. It is done by exposing folders in the host instance where Elasticsearch, Kibana, and Logstash configuration files can be set. The folder structure after cloning the repository is the following one:

.

├── docker-compose.yml

├── kibana

│   └── config

│   └── kibana.yml

├── LICENSE

└── logstash

├── config

│   └── logstash.yml

└── pipeline

└── logstash.conf

* Storage in the Elasticsearch cluster is persistent, as two volumes in the host machine, handled by Docker, are created: esdata1 and esdata2. That is, although the containers are stopped, storage is kept. The volumes contents can be erased by removing the volumes (docker volume rm <volume\_name>).
* Three ports are exposed in the Elastic Stack instance so that it is possible to interact with its components: 9200 for Elasticsearch, 5601 for Kibana and 5001 for Logstash (that is, <elk-floating-ip-address>:9200, <elk-floating-ip-address>:5601 and <elk-floating-ip-address>:5001).

It is possible to verify the right installation of the Elastic Stack by accessing http://<elk-floating-ip-address>:5601. The Kibana web UI should be shown. It is also possible to verify the right initialization of the containers by typing docker ps and checking that all the containers are running, or to inspects the logs by typing docker logs kibana, docker logs elasticsearch, etc.):

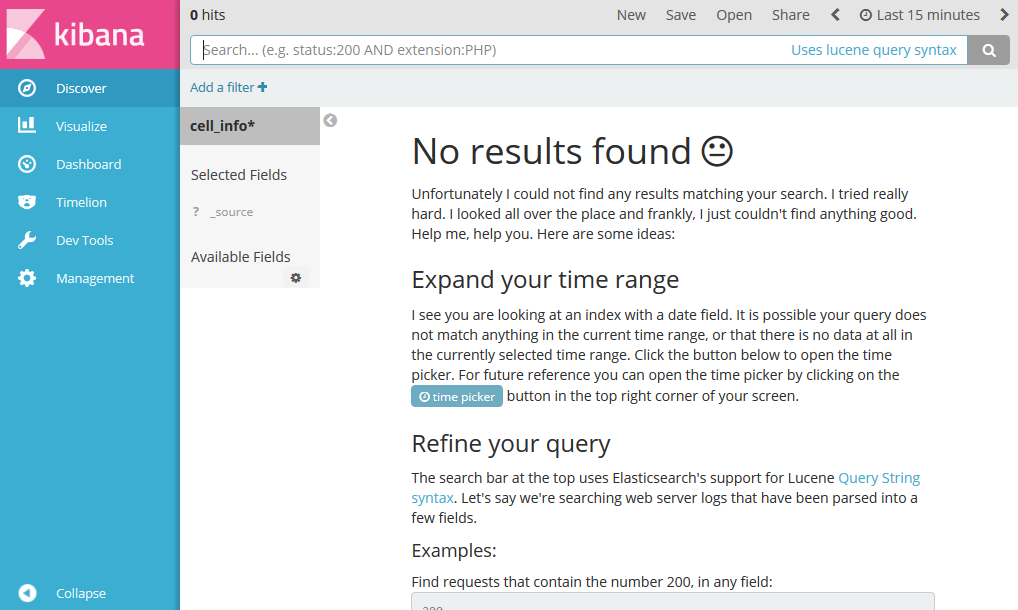


Figure 9: Main Kibana UI home

Once the Elastic Stack instance is deployed and configured, it is necessary to provide the means for interacting (reading/writing) from the Spark cluster. A connector enables this functionality: Elasticsearch for Apache Hadoop (elasticsearch-hadoop).[14] The connector binaries cover a variety of scenarios. “Minimalistic” jar files are also offered for specific integrations and therefore, the minimalistic jar file for Spark (elasticsearch-spark-20\_2.11-5.6.1.jar) is enough to meet the project requirements. Several options are available:

* Download the jar file from Maven[[19]](#footnote-19) and copy it to the $SPARK\_HOME/jars folder on all the cluster instances.
* Download the elasticsearch-hadoop *jar* file from the Elastic site,[[20]](#footnote-20) uncompress it (as it is a zip file), and copy it to the $SPARK\_HOME/jars folder on all the cluster instances.  
  With any of the previous options, the jar file can be placed anywhere and included on the driver and executor classpaths by means of the --jars option.
* Use Maven coordinates of jars (groupId:artifactId:version) to include on the driver and executor classpaths.[[21]](#footnote-21) It is done by means of the --packages option:

pyspark --master spark://<master-ip-address>:7077 \  
--packages org.elasticsearch:elasticsearch-spark-20\_2.11:5.6.1

The detailed description of the Elastic Stack setup can be found in <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/spark-elasticsearch-setup.md>.

### Kafka setup

Apache Kafka is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable. Kafka maintains feeds of messages in so-called topics. Producers write data to topics and consumers read from topics. In the project, it is used to provide inputs to Spark Streaming applications and to deliver the outputs of said applications to Elasticsearch, so that they can be visualized.

As with the previous component, the deployment of Kafka on an Openstack instance will be carried out by means of Docker containers that run the ubiquitous spotify/kafka image.[[22]](#footnote-22) It can be run by using the following command:

docker run -d -p 2181:2181 -p 9092:9092 \  
--name kafka\_container \  
--env ADVERTISED\_HOST=<kafka-ip-address> \  
--env ADVERTISED\_PORT=9092 spotify/kafka

Successful execution of Kafka can be verified by checking whether it is one of the listed running containers (docker ps) or by inspecting the container logs (docker logs kafka\_container). Once producers start pushing messages to Kafka under the topic\_name topic, it is possible to read them by executing, from the Kafka host, the following command:

docker run -it --rm --link kafka\_container spotify/kafka \  
/opt/kafka\_2.11-0.10.1.0/bin/kafka-console-consumer.sh \  
--bootstrap-server kafka:9092 \  
--topic <topic\_name> \  
--from-beginning

Once the Kafka instance is deployed and configured, it is necessary to provide the means for Spark Streaming applications to interact (consume/produce) from the Spark cluster. A connector enables this functionality: Spark Integration For Kafka 0.8.[[23]](#footnote-23) The connector binaries must be added to the driver and executor classpaths by means of the --packages option:

pyspark --master spark://<master-ip-address>:7077 \  
--packages org.apache.spark:spark-streaming-kafka-0-8\_2.11:2.0.2

It is also possible to download the jar file and copy it to the $SPARK\_HOME/jars folder.

# Execution

## Data understanding

The input dataset contains six-month Call Detail Records (CDR) from a small European telecom operator, from October 2015 to March 2016. Six compressed files, one per month, are provided. Total (compressed) size is 102 Gbytes.

A Call Detail Record is a data record generated by a telecommunications operator documenting the details of a telephone call or other communication transaction (a short message, a data connection…).[[24]](#footnote-24) It contains different transaction features, such as time, duration, source, and destination identifiers… Every file in the dataset has a CSV format with the same data dictionary, where every line contains a single CDR. A semicolon is used as separator and all the files have a header. The data dictionary can be seen in the table below:

|  |  |  |
| --- | --- | --- |
| Variable | Meaning | Type |
| DS\_CDMSISDN | Hash of the user's phone number | string |
| DS\_CDIMSI | Hash of the user's IMSI number | string |
| DS\_IMEI | Hash of the phone's IMEI | string |
| DT\_CDDATASTART | Start date of the call | date |
| DT\_CDDATAEND | End date of the call | date |
| NUM\_LENGTH | Duration of call in seconds, empty for text | integer |
| ID\_CELL\_SOURCE | Cell identifier | integer |
| ID\_CELL\_DEST | Destination cell identifier | integer |
| DS\_CALLIDENTIFICATIONNUMBER | Hash of the destination phone number | string |
| ID\_RECORDSEQUENCENUMBER | Empty | string |
| ID\_CDTIPUSCOM | Event type. It takes 5 values: S-CDR: Data Session; MOC (Mobile Originated Call): Outgoing Call; MOS (Mobile Originated SMS): Outgoing SMS; MTC (Mobile Terminated Call): Incoming Call; MTS (Mobile Terminated SMS): Incoming SMS | integer |
| DS\_CDCENTRAL | Name of the communications central (switch) who processes the event (not useful for analysis) | string |
| ACTUALIZATION\_DATE | Insertion date in the CDR database (not useful for analysis) | date |
| ID\_CLIENTSOURCE | Hash of the customer ID | string |
| ID\_CDOPERATORSOURCE | Home operator of the call | integer |
| ID\_CDCOUNTRYSOURCE | Home country of the call | integer |
| DS\_CDNUMDEST | Hash of the receiver's phone number | string |
| ID\_CDOPERATORDEST | Home operator of the receiver | integer |
| TAC\_IMEI | Phone's Type Allocation Code (TAC) | integer |

Some additional remarks about the dataset must be considered:

* Variables *ID\_CDOPERATORSOURCE* and *ID\_CDCOUNTRYSOURCE* contains respectively the Mobile Network Code (MNC) and the Mobile Country Code (MCC) of the source of the call or transaction, as specified by ITU-T Recommendation E.212.[15] The MCC consists of 3 decimal digits, while the MNC comprises 2 or 3 decimal digits. As the dataset only provides the codes, it is necessary to use an additional database to get the mapping between the code and the actual operator or country name. Public MCC-MNC databases can be used (<https://clients.txtnation.com/hc/en-us/articles/218719768-MCCMNC-mobile-country-code-and-mobile-network-code-list-> provides 2 300 network codes).
* Variable *TAC\_IMEI* passes on the phone's Type Allocation Code (TAC). The TAC is the initial eight-digit portion of the 15-digit IMEI (whose hash is passed on by variable DS\_IMEI), used to uniquely identify any mobile device. Again, additional databases are needed to map the TAC (provided in clear text, unlike the whole IMEI) to an actual terminal model. The database used in the project stores the features of about 43,000 different terminals: TAC code, manufacturer, model name (both internal and marketing name), type of device (mobile phone, tablet, router, M2M device…), radio technology, operating system, and release year. In the project, a private TAC database is used.

## Data storage

An essential prerequisite for carrying out data analysis tasks is to make the dataset available to the processing engine (i.e., to the Spark cluster). Given its size and the chosen processing engine, the dataset will be persisted in an HDFS cluster. To efficiently store data, the Parquet format will be used. Apache Parquet[[25]](#footnote-25) is a columnar storage format available to any project in the Hadoop ecosystem. Thus, it is compatible with Apache Spark and can be read and written by any Spark-based task. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. It supports several codecs for compressing data; in this project, the default option, gzip, will be used.

The procedure to store data in the HDFS cluster is as follows:

* A folder for the dataset is created in the HDFS namespace (/data). The HDFS File System shell from the master instance will be used.

hdfs dfs -mkdir /data

* In an analogous way, a new folder will be created to store summaries of the results (Pandas dataframes stored as CSV files). The main idea is to have them available for creating new visualizations without making Spark recompute the results.

hdfs dfs -mkdir /dataframes

* Each file in the dataset is uploaded to the cluster master instance and uncompressed (one of the files happens to be corrupted; gzrecover[[26]](#footnote-26) has been used to extract the uncorrupted part of the file). Other instances could have been used as well.
* The file is copied to the HDFS cluster. For instance, the December 2015 data file is copied to the HDFS cluster:

hdfs dfs -put DWFET\_CDR\_CELLID\_201512.csv /data

To verify that the files have been loaded, the HDFS Web Interface (http://<master-floating-ip-address>:50070/ > Utilities > Browse the file system) can be used:

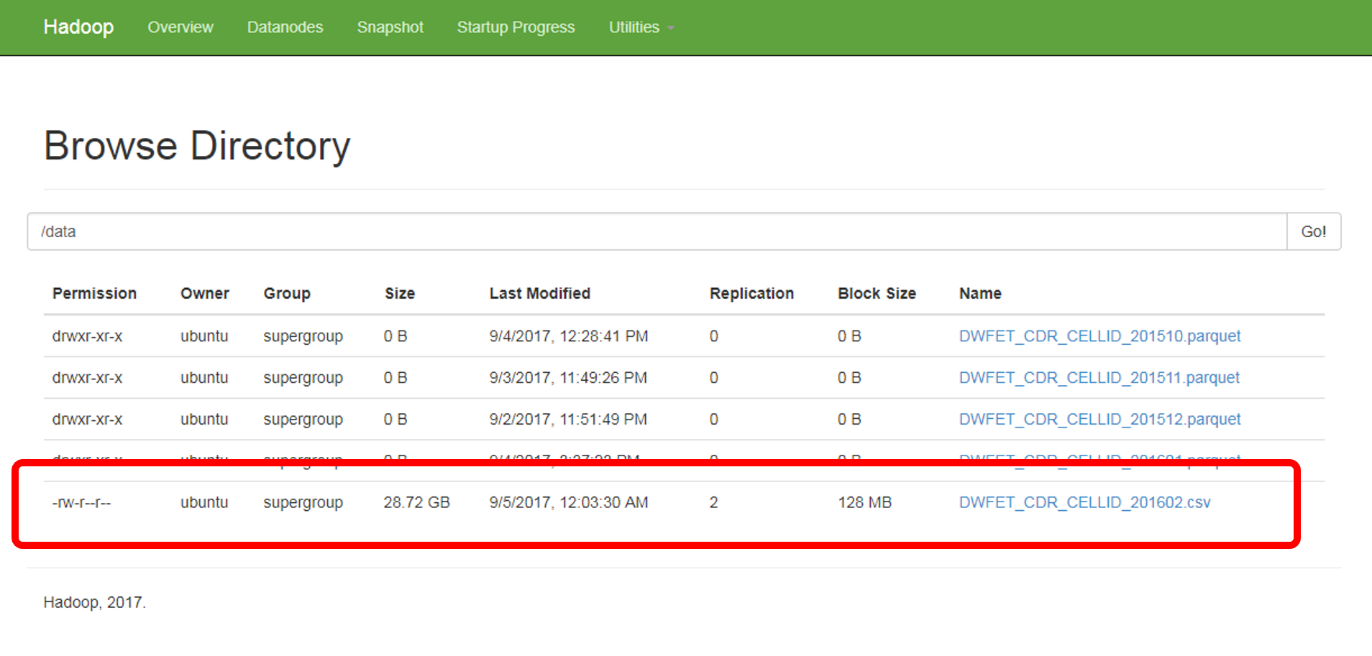


Figure 10: HDFS Web Interface: it shows the December 2015 uploaded file

* Once copied to the HDFS cluster, the CSV files are removed from the local filesystem.
* A Python Spark notebook (00\_save\_as\_parquet.ipynb within the master-thesis-notebooks repository) is used to retrieve the plain CSV file from the HDFS cluster, read it as a data frame in a Spark Session, appropriately apply a data type to each column, and save the data frame content as a Parquet file. The default format (StringType) is left for most of the columns (almost all numeric columns are handled as strings, as no numeric operations will be involved). IntegerType is assigned as the type of the column containing the duration of the Call Detail Record. The data type of two columns carrying dates (the start and end of Call Detail Records) is turned into TimestampType. Useless columns are removed. See code snippet below:

# In[1]

file\_name = u'DWFET\_CDR\_CELLID\_201603'

input\_file = u'hdfs://cluster-master:9000/data/%s.csv' %(file\_name)

output\_file = u'hdfs://cluster-master:9000/data/%s.parquet' %(file\_name)

# In[2]

# the data schema is defined

schema = StructType([StructField("DS\_CDMSISDN", StringType(), True),

               StructField("DS\_CDIMSI", StringType(), True),

               StructField("DS\_IMEI", StringType(), True),

               StructField("DT\_CDDATASTART", TimestampType(), True),

               StructField("DT\_CDDATAEND", TimestampType(), True),

               StructField("NUM\_LENGTH", IntegerType(), True),

               StructField("ID\_CELL\_SOURCE", StringType(), True),

               StructField("ID\_CELLA\_DEST", StringType(), True),

               StructField("DS\_CALLIDENTIFICATIONNUMBER", StringType(), True),

               StructField("ID\_RECORDSEQUENCENUMBER", StringType(), True),

               StructField("ID\_CDTIPUSCOM", StringType(), True),

               StructField("DS\_CDCENTRAL", StringType(), True),

               StructField("ACTUALIZATION\_FE", StringType(), True),

               StructField("ID\_CLIENTSOURCE", StringType(), True),

               StructField("ID\_CDOPERATORSOURCE", StringType(), True),

               StructField("ID\_CDCOUNTRYSOURCE", StringType(), True),

               StructField("DS\_CDNUMDEST", StringType(), True),

               StructField("ID\_CDOPERATORDEST", StringType(), True),

               StructField("TAC\_IMEI", StringType(), True),

              ])

# Useless columns are removed

df = spark.read.format('csv').\

                            load(input\_file,

                                sep=';',

                                header=True,

                                schema = schema,

                                timestampFormat = 'yyyy.MM.dd hh:mm:ss'

                            ).\

                            drop("DS\_CDIMSI").\

                            drop("ID\_RECORDSEQUENCENUMBER").\

                            drop("DS\_CDCENTRAL").\

                            drop("ACTUALIZATION\_FE")

# In[3]

df.write.parquet(output\_file)

* Finally, the CSV files at the HDFS cluster are removed as well by means of the HDFS File System shell:

hdfs dfs -rm /data/DWFET\_CDR\_CELLID\_201512.csv

hdfs dfs -expunge

As mentioned, one of the files in the dataset is corrupted and some records are missing. As data imputation is not actually needed, the amount of missing records will be estimated by making a simple proportion. After grouping records by day in the month, it is found out that records from the 18th day onwards are missing. Thus, values in the considered month, February, will be multiplied by 1.65 (28/17).

Additionally, two auxiliary datasets are also uploaded to the HDFS cluster and turned into parquet files. These files are needed to enrich the analysis of the dataset:

* TAC.csv: It contains the mapping between a phone's Type Allocation Code (TAC) and several terminal features such as model, manufacturer, operating system...
* MCCMNC.csv: It provides the mapping between a Mobile Country Code (MCC) and a country or territory, and between a Mobile Network Code (MNC) and a telecom operator.

The procedure is similar and can be reviewed in the notebook 00\_save\_parquet.ipynb within the master-thesis-notebooks repository. Thus, the data is ready for analysis in the HDFS cluster.

## Data analysis and visualization

This analysis is based on SparkSQL and is executed by means of Python notebooks (available at the master-thesis-notebooks repository). Visualization is done by means of matplotlib.

The following command is run in the master instance, which plays the role of Spark Driver:

pyspark --master spark://cluster-master:7077 \  
--executor-memory 24G \  
--driver-memory 10G

The Jupyter Notebook server becomes accessible at http://<master-floating-ip-address>:9999/:



Figure 11: Jupyter Notebook server home page

These notebooks run several tasks and create visualizations in order to carry out Exploratory Data Analysis. It aims to get, at least, the following information from the dataset:

1. Size of the dataset.
2. Analysis of terminals:
3. Terminal share for all the subscribers in Operator X network
4. Terminal share for Operator X subscribers
5. Terminal vendor share for all the subscribers in Operator X network
6. Terminal vendor share for Operator X subscribers
7. Terminal operating system share for all the subscribers in Operator X network
8. Terminal operating system share for Operator X subscribers
9. Monthly model evolution for Operator X subscribers
10. Monthly vendor evolution for Operator X subscribers
11. Analysis of subscribers, with focus on roamers:
12. Unique subscribers, both roamers and from Operator X
13. Daily evolution of subscribers, both roamers and from Operator X
14. Roamer split per country and operator

From these basic requirements, the analysis is open to carry out additional tasks and extract and visualize additional information. In the following sections, some code snippets and screenshots will be provided.

### Size of the dataset

Zeppelin is a convenient tool to get a quick view of the dataset. It supports notebooks with cells with different interpreters. It is possible to use, for instance, the Spark (pySpark in fact) and the SQL interpreters in order to get basic results and visualizations.

The following code snippet is used to load the dataset as a dataframe and register it as a table for subsequent query execution from the SQL interpreter (taken from 00\_basic\_analysis.json in the master-thesis-notebooks repository):

%spark.pyspark

input\_hdfs\_path = u'hdfs://cluster-master:9000/data/DWFET\_CDR\_CELLID\_\*.parquet'

df = spark.read.format('parquet').load(input\_hdfs\_path)

df.registerTempTable("df")

Next, different SQL queries can be run:

%sql

**select** count(\*) **as** Total\_CDR,

   (**select** count(\*) **from** df **where** ID\_CDOPERATORSOURCE != '21303') **as** Total\_CDR\_Roamers,

   ((**select** count(\*) **from** df **where** ID\_CDOPERATORSOURCE != '21303') \* 100 / (**select** count(\*) **from** df))

**as** Roamers\_CDR\_Percentage **from** df

%sql

**select** count(**distinct** DS\_CDMSISDN) **as** Total\_Distinct\_Subscribers,

   (**select** count(**distinct** DS\_CDMSISDN) **from** df **where** ID\_CDOPERATORSOURCE != '21303') **as** Total\_Roamers,

   ((**select** count(**distinct** DS\_CDMSISDN) **from** df **where** ID\_CDOPERATORSOURCE != '21303') \* 100 / (**select** count(**distinct** DS\_CDNUMORIGEN) **from** df)) **as** Roamers\_Percentage

**from** df

Thus obtaining the following figures:

|  |  |  |
| --- | --- | --- |
| Concept | Total | Total (from roamers) |
| CDR’s | 538,646,581 | 60,952,030 (11.32%) |
| Unique IMSI’s | 1,586,079 | 1,452,334 (91.57%) |
| Unique IMEI’s | 1,989,716 | 1,829,113 (91.93%) |

It is also possible to take advantage of the Zeppelin built-in chart capabilities and run queries that automatically generate charts:

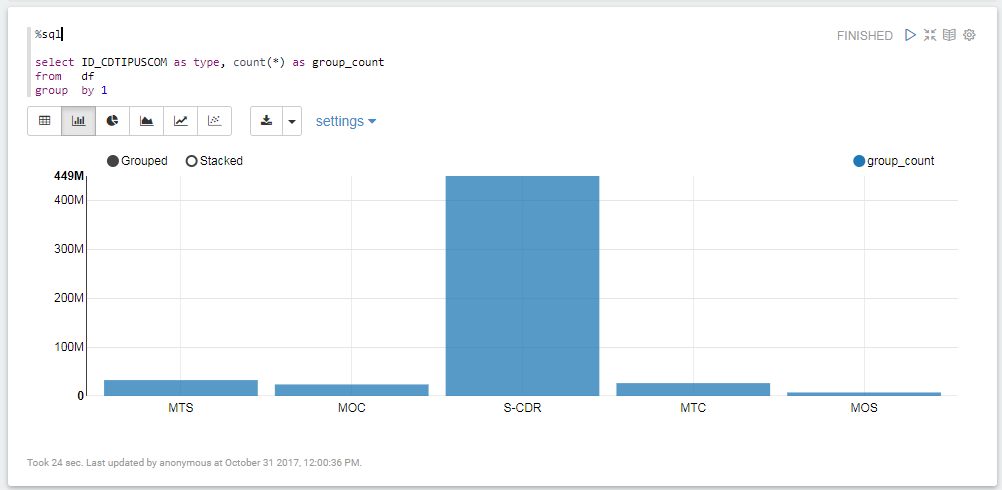


Figure 12: CDR type query executed with Zeppelin

### Terminal analysis

All the Exploratory Data Analysis tasks on terminal information follow a similar approach: processing and result extraction is carried out by means of Spark SQL. Next, Spark dataframes are turned into Pandas dataframes (and saved in the HDFS cluster as CSV files in order to enable subsequent creation of visualizations without requiring recomputation; mind that the dataframe save operation should be similar to df.repartition(1).write.format("csv").option("header", "true").save("u'hdfs://cluster-master:9000/dataframes/df.csv"),[[27]](#footnote-27) as several executors are involved and otherwise several files would be available). Finally, matplotlib is used to create the plots showing the information. Tasks are fulfilled with notebooks 01\_terminal\_queries\_i.ipynb and 02\_terminal\_queries\_ii.ipynb within the master-thesis-notebooks repository.

Terminal analysis relies on the TAC dataset and therefore, the first operation to run is the retrieval of the dataset as a Spark dataframe. Next, it is joined with the overall dataset:

# In[6]:

tac\_file = u'hdfs://cluster-master:9000/data/TAC.parquet'

tac\_df = spark.read.format('parquet').load(tac\_file)

# In[7]:

tac\_df.count()

# Out[7]:

43968

# In[8]:

tac\_df.printSchema()

# Out[8]:

root

 |-- TAC: string (nullable = true)

 |-- TERM\_MANUFACTURER: string (nullable = true)

 |-- TERM\_MODEL: string (nullable = true)

 |-- TERM\_NAME: string (nullable = true)

 |-- TERM\_TYPE: string (nullable = true)

 |-- TERM\_RADIO: string (nullable = true)

 |-- TERM\_OS: string (nullable = true)

 |-- TERM\_YEAR: integer (nullable = true)

# In[9]

dh = df.join(tac\_df, df\_term['TAC\_IMEI'] == tac\_df["TAC"]).drop('TAC\_IMEI')

As an example, the code snippet computing the global percentage of unique terminals is provided:

# In[9]

total\_terminals = dh.count()

# In[10]

models = dh.groupBy('TERM\_NAME')\

            .count()\

            .sort("count", ascending=False)\

            .withColumn('total', fn.lit(total\_terminals))\

            .withColumn('percentage', fn.expr('round(100\*count/total, 2)'))

models.show(5)

# Out[10]

+-----------------+------+-------+----------+

|        TERM\_NAME| count|  total|percentage|

+-----------------+------+-------+----------+

|         iPhone 6|186664|1655538|     11.28|

|        iPhone 5S|116711|1655538|      7.05|

|        iPhone 6S| 54415|1655538|      3.29|

|        iPhone 5C| 53953|1655538|      3.26|

|        Galaxy S4| 46685|1655538|      2.82|

+-----------------+------+-------+----------+

only showing top 5 rows

The top terminals for both the Operator X subscribers and subscribers roaming to the Operator X network is shown below:

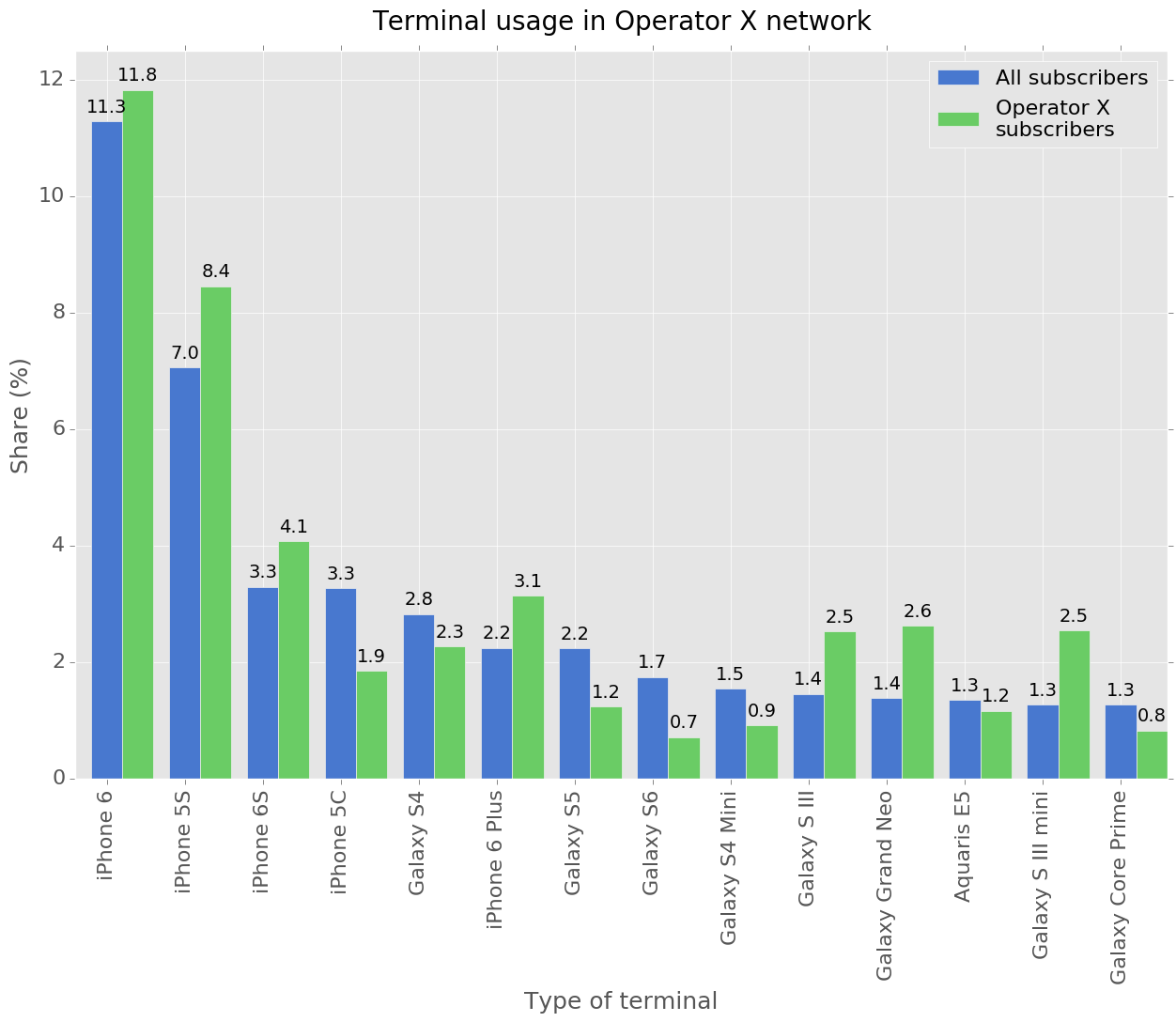


Figure 13: Top terminals in Operator X network

A similar visualization can be generated for terminal manufacturers:

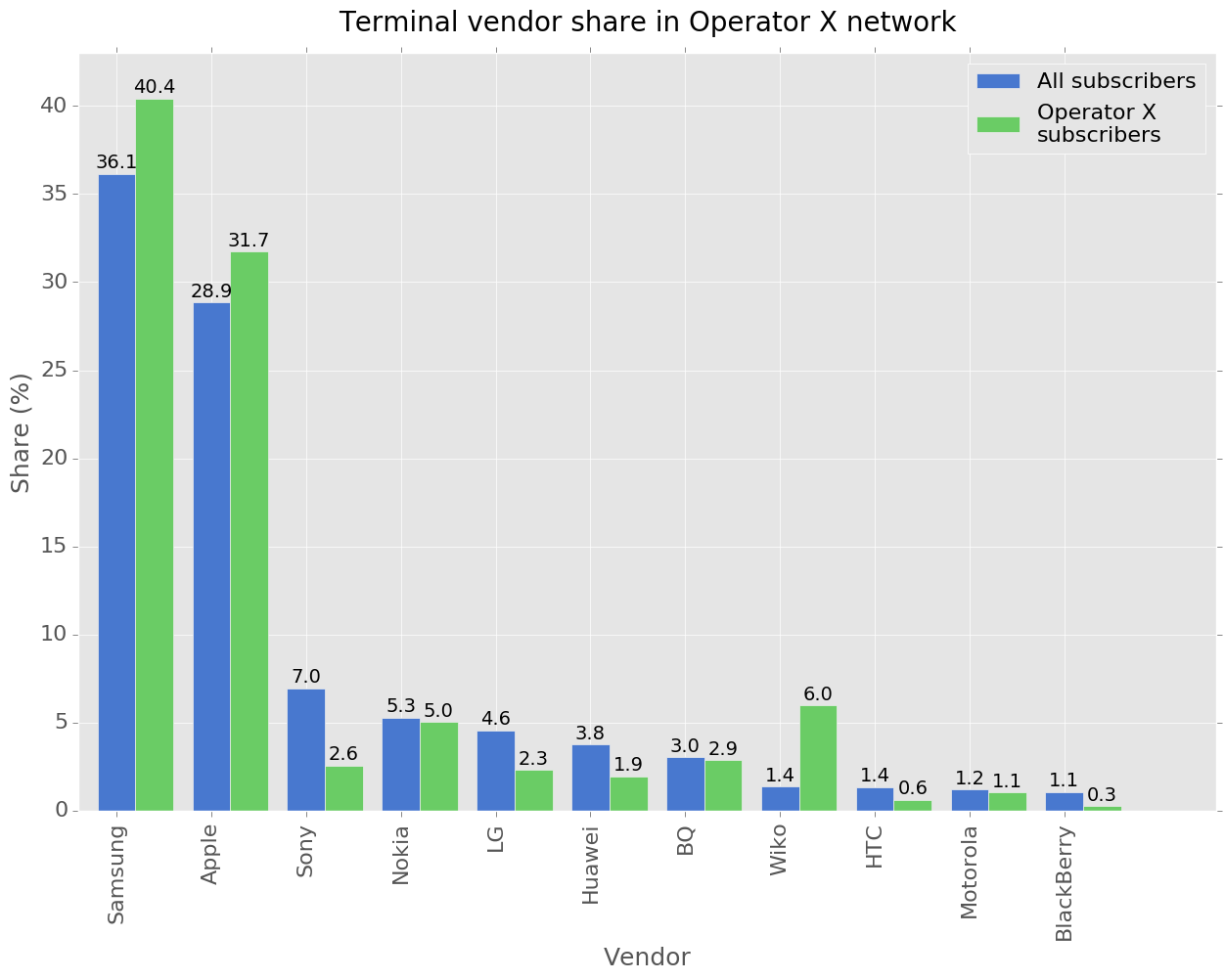


Figure 14: Top manufacturers in Operator X networks

Or for the operating system:

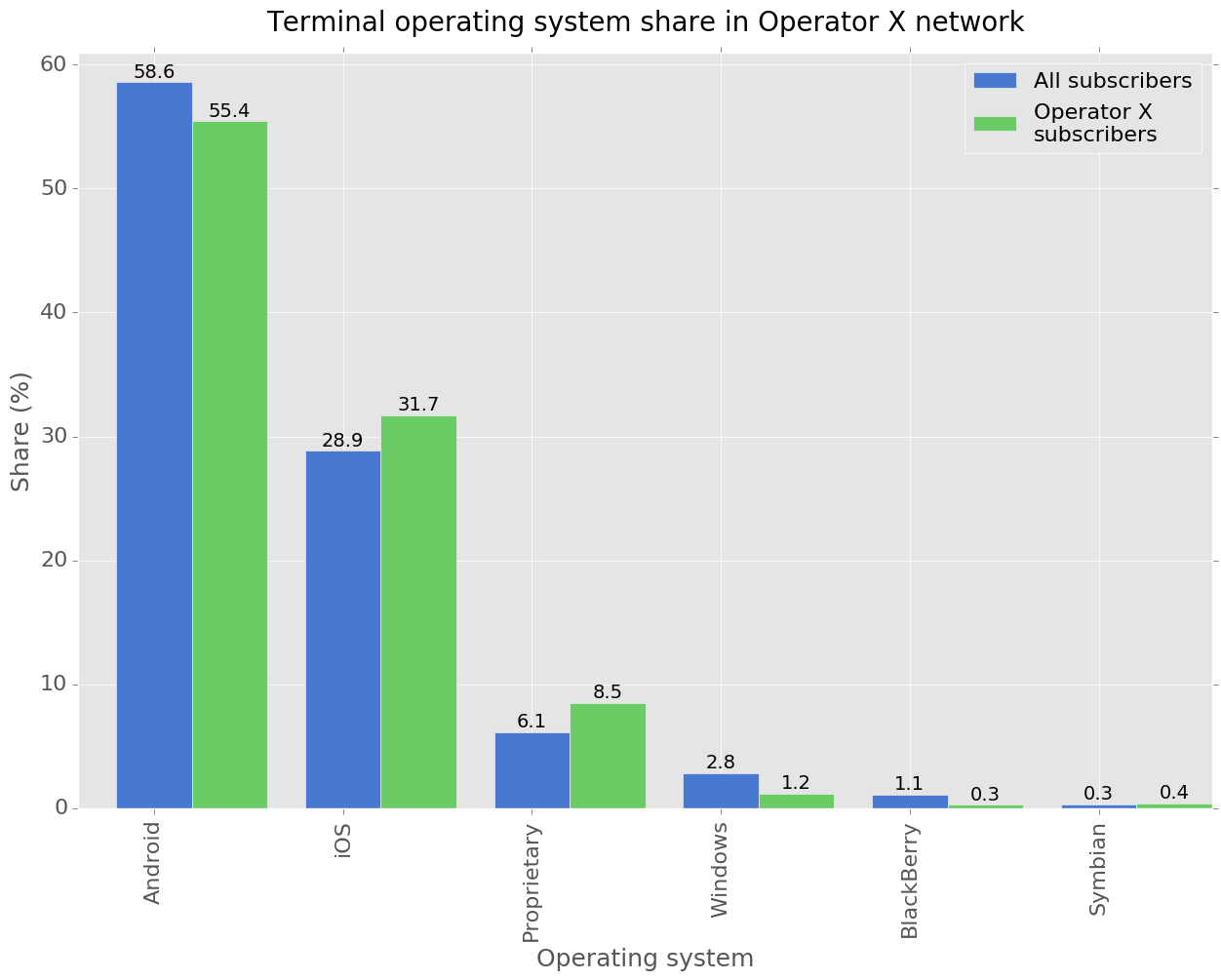


Figure 15: Top operating systems in Operator X network

It is also possible to determine the types of terminals used by the Operator X subscribers (96.85% happens to be a mobile phone) and some specific details about not so frequent terminals, such as M2M devices:

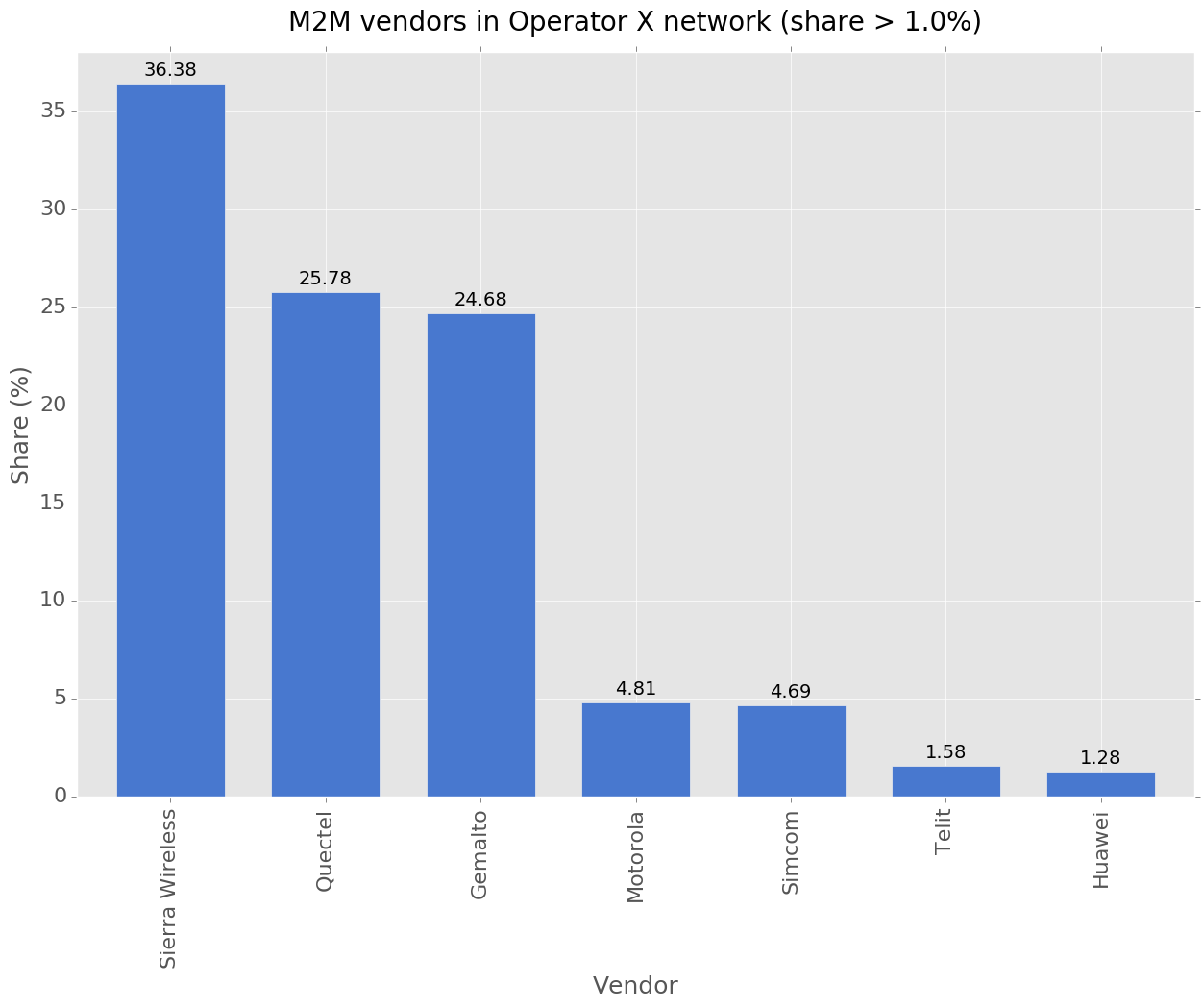


Figure 16: Top M2M vendors in Operator X network

The previous analyses are simple and straightforward. Next, the monthly evolution of terminal shares among Operator X subscribers is computed. A User Defined Function is employed to create a new column with the CDR month (here a lambda function would have been also possible), group by this new column, and count:

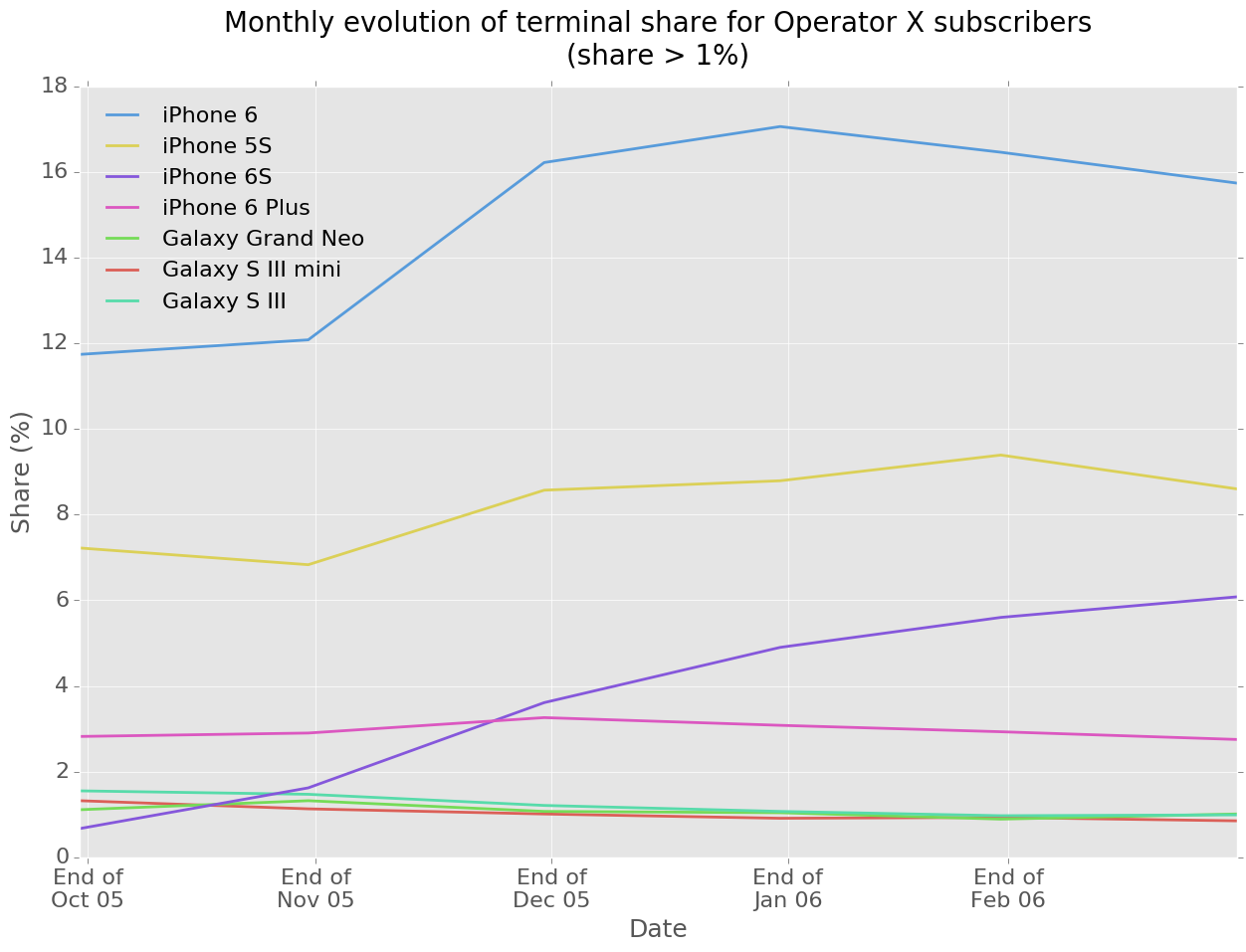


Figure 17: Monthly terminal share evolution among the Operator X subscribers

It is interesting to note than the surge in the iPhone 6 share in Christmas leads to Apple to take the manufacturer with the larger share:

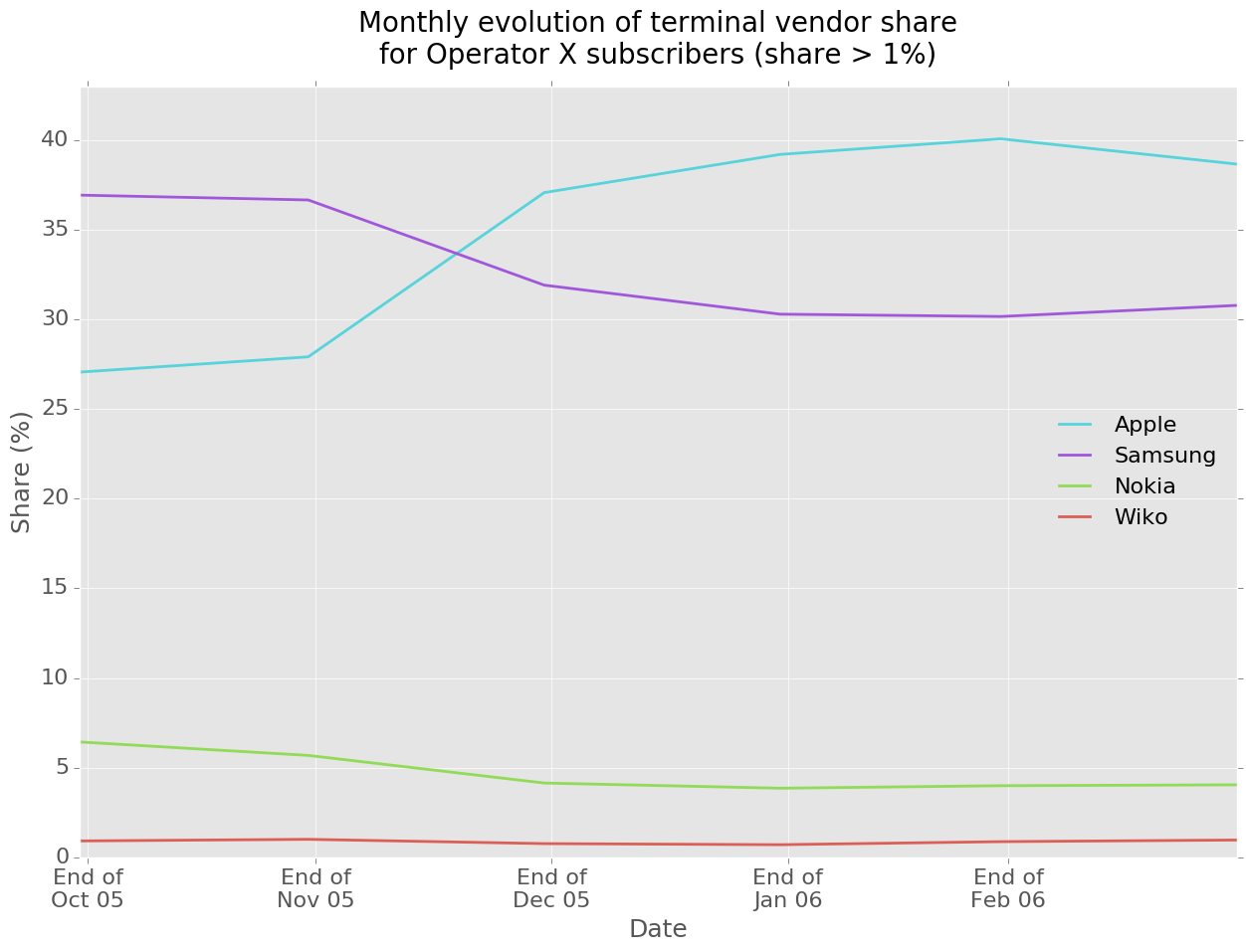


Figure 18: Monthly vendor share evolution among the Operator X subscribers

The next challenge is to determine the daily rate of terminal activations among the Operator X subscribers. That is, how many terminals are activated every day. To accomplish that task, some principles must be considered: only subscribers with several terminals are considered (it is assumed that the mobile penetration is so high that Operator X, the only operator in country X, does not captures a significant number of new customers) The procedure will be as follows:

* Analysis is done in hourly slots. It allows to drop all the CDR for the same terminal in the same hour but one.
* The dataframe is turned into an RDD to apply a User Defined Aggregation Function. For every terminal, a reduce function is defined to get the first and last use of the terminal. The resulting RDD (turned again into a dataframe) contains the different terminals and their usages: subscriber number, terminal identifier, terminal model, first use and last use.

# In[17]

rdd = dg\

        .select(['DS\_CDMSISDN', 'DS\_IMEI', 'DT\_CDHOURSTART', 'TAC\_IMEI', 'TERM\_NAME', 'TERM\_TYPE'])\

        .rdd\

        .map(**lambda** line: ((line[0], line[1], line[3], line[4], line[5]), [line[2], line[2]]))\

        .reduceByKey(**lambda** x, y: consolidate(x, y))\

        .map(**lambda** line: (line[0][0], line[0][1], line[0][2], line[0][3], line[1][0], line[1][1])

# In[18]

dh\_schema = StructType([StructField("DS\_CDMSISDN", StringType(), True),

                        StructField("DS\_IMEI", StringType(), True),

                        StructField("TAC\_IMEI", StringType(), True),

                        StructField("TERM\_NAME", StringType(), True),

                        StructField("USE\_START", IntegerType(), True),

                        StructField("USE\_END", IntegerType(), True),

                       ])

dh = spark.createDataFrame(rdd, dh\_schema)

* Terminals belonging to subscribers with just one terminal in the analysis period are determined and removed (using a left anti join operation).
* A windowed lag function is used to determine the activations.

# In[11]

di=dh.groupBy('DS\_CDMSISDN').count().filter(col('count') == 1)

dj=dh.join(di, dh['DS\_CDMSISDN'] == di['DS\_CDMSISDN'], "leftanti")

# In[12]

windowSpec = Window.partitionBy(col('DS\_CDMSISDN')).orderBy(col('USE\_START').desc())

dl=dj.withColumn('ACTIVATION\_TIME', lag(dj['USE\_START']).over(windowSpec))\

      .filter(col('ACTIVATION\_TIME').isNotNull())

This analysis task introduces several challenges:

* User Defined Aggregation Functions are not supported yet in PySpark. Therefore, dataframes must be turned into RDD, an aggregation function be implemented, and the resulting RDD turned back into a dataframe again.
* SQL-like window functions are needed to get the activation time (the first use of a terminal, but dropping the first used terminal).

The following visualization is obtained (shaded area for the period without available data):

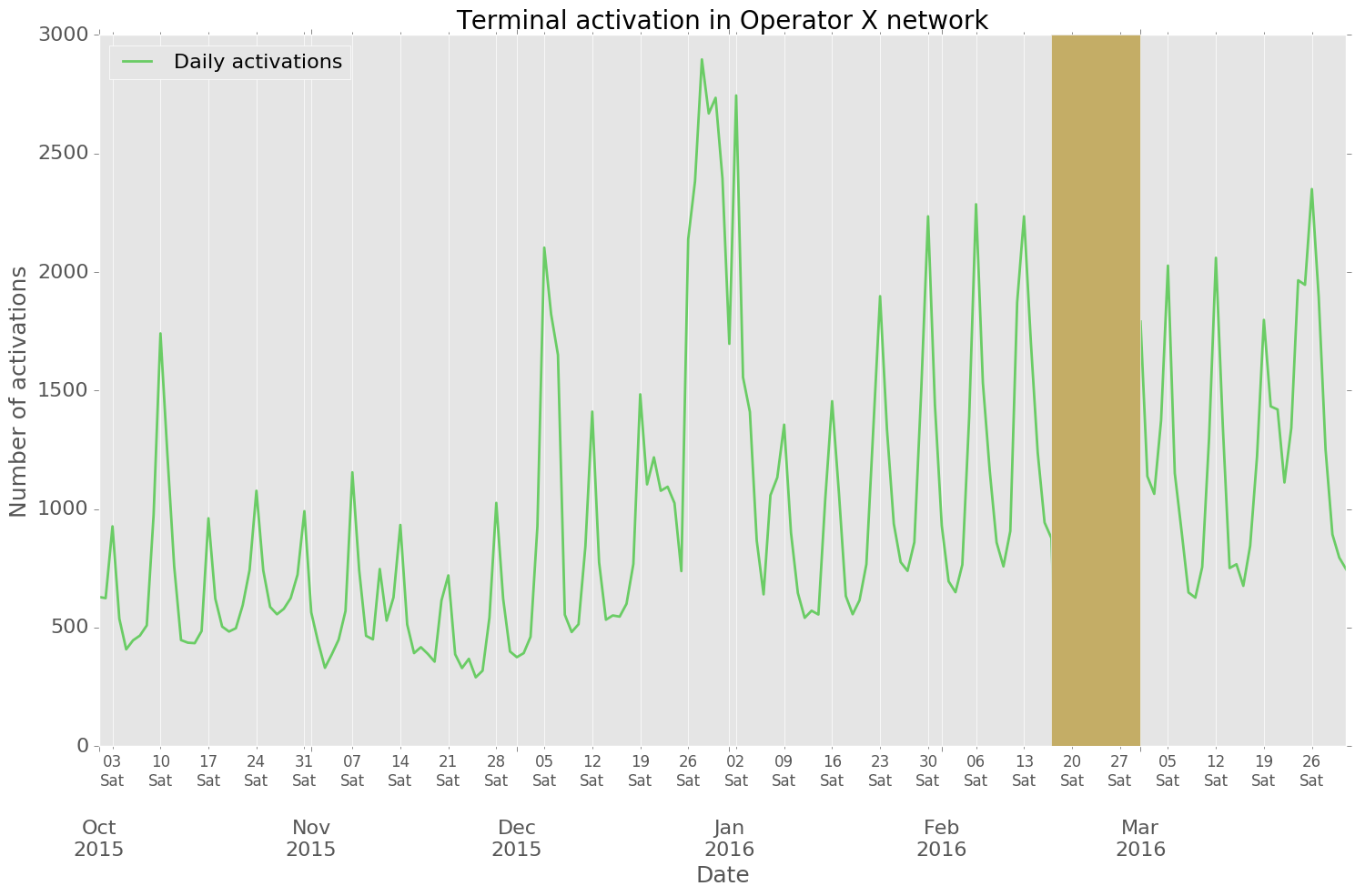


Figure 19: Terminal activation in Operator X

It is possible to monitor tasks being run at the cluster through the Spark Context UI:

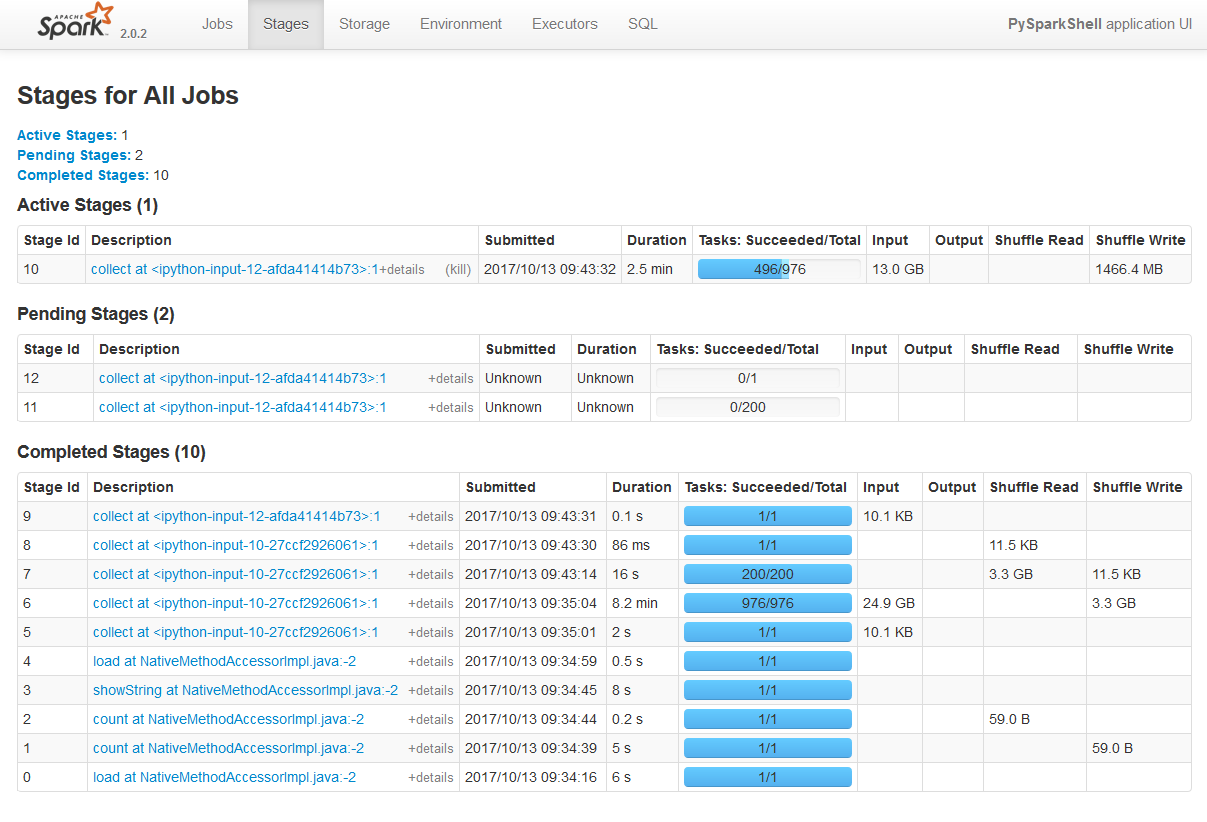


Figure 20: Spark Context UI: Stages for All Jobs

### Subscriber analysis

Exploratory Data Analysis tasks on subscriber information follow the same approach than those related to terminals Tasks are fulfilled with notebooks 03\_subscriber\_queries\_i.ipynb and 04\_subscriber\_queries\_ii.ipynb within the master-thesis-notebooks repository.

Subscriber analysis relies on the MCCMNC dataset and therefore, the first operation to run is the retrieval of the dataset as a Spark dataframe. Next, it is joined with the overall dataset and basic queries are run.

# In[5]:

mccmnc\_file = u'hdfs://cluster-master:9000/data/MCCMNC.parquet'

mccmnc\_df = spark.read.format('parquet').load(mccmnc\_file)

# In[6]:

mccmnc\_df.count()

# Out[6]:

2300

# In[7]:

mccmnc\_df.printSchema()

# Out[7]:

root

 |-- OP\_NAME: string (nullable = true)

 |-- OP\_COUNTRY: string (nullable = true)

 |-- OP\_CODE: string (nullable = true)

# In[10]:

dg\_subscribers = df\_subscribers\

.join(mccmnc\_df, df\_subscribers['ID\_CDOPERADORORIGEN'] == mccmnc\_df["OP\_CODE"])\

  .drop('ID\_CDOPERADORORIGEN')

In order to find the top countries the unique users in the Operator X network come from, the dataframe is grouped by the home country ('OP\_COUNTRY') and each group is counted. The result is shown in the figure below:

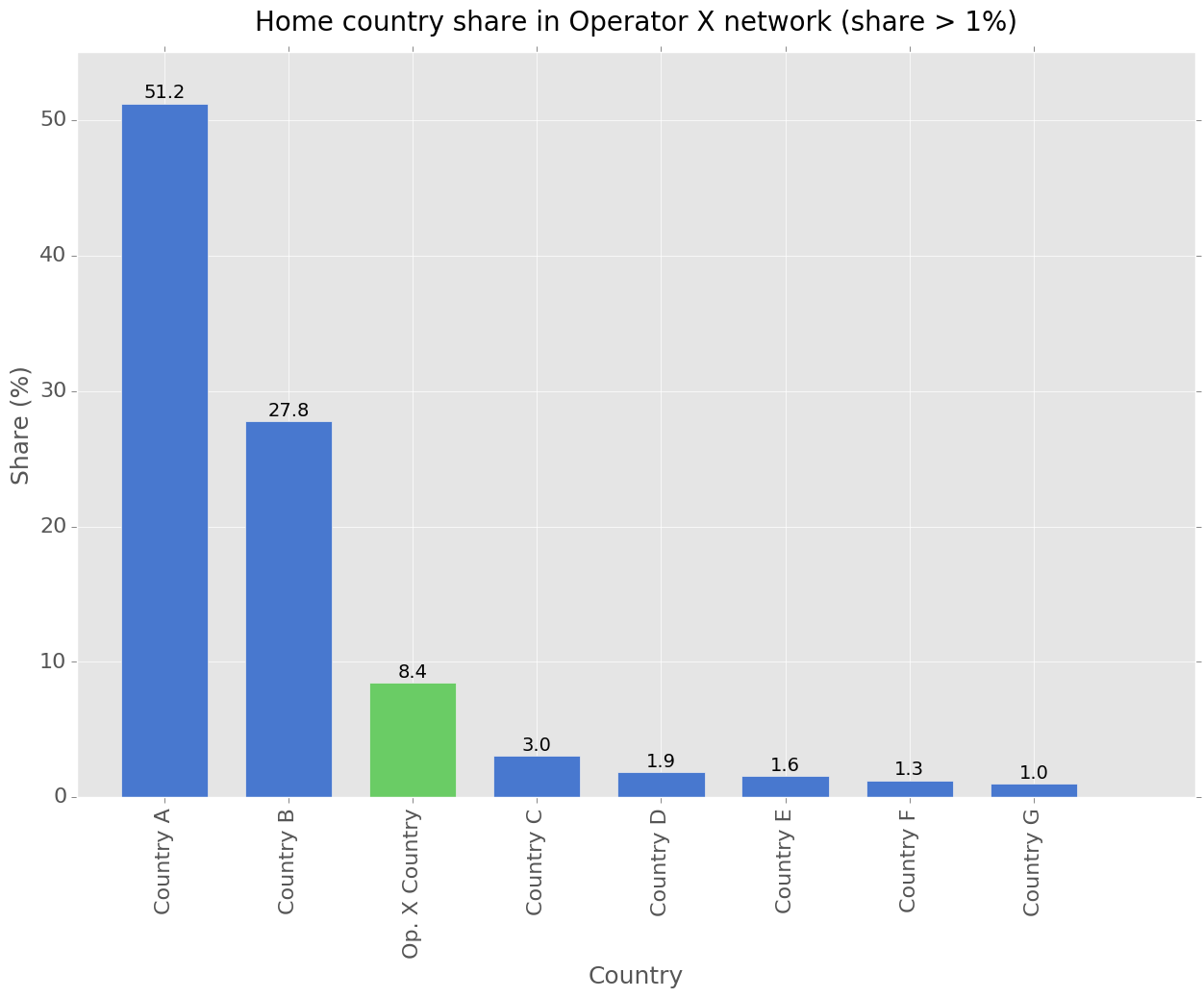


Figure 21: Top countries in the Operator X network

However, the results above appear to be rather counterintuitive. Although country X is small and almost surrounded by countries A and B, it does not seem sensible to have up to six times more subscribers from country A than from the home country, country X. Thus, a different approach was tried and the daily share of subscribers in the Operator X network was determined. That is, whether those unique users were short-term visitors that did not stay for long. The approach taken is simple: the dataframe is grouped not only by home country but also by day. Next, the daily mean value is computed for every home country. The results can be found in the figure below (comparing the global values with the average daily values).

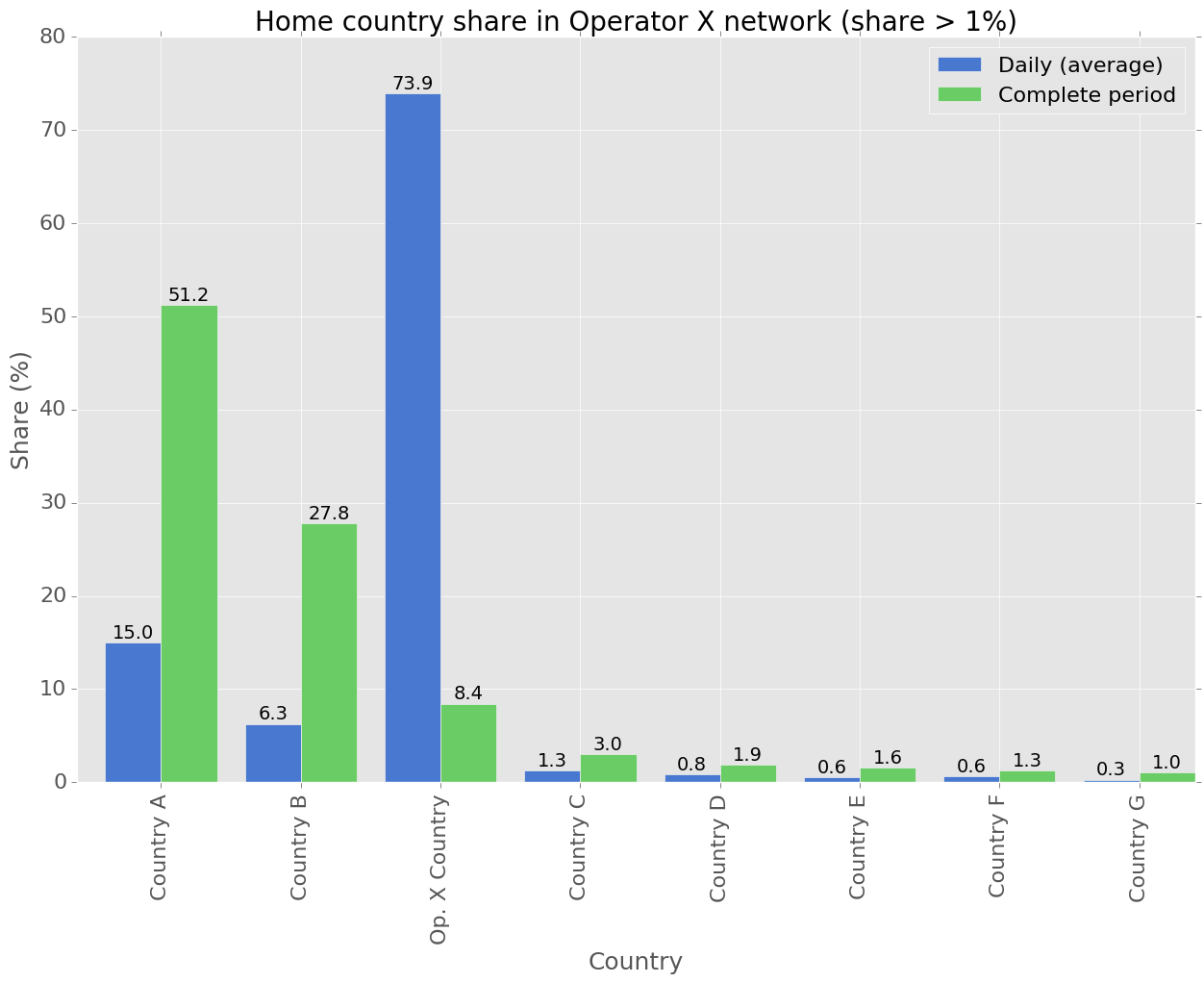


Figure 22: Top daily countries in the Operator X network

Now, the results are much more sensible as, the average daily percentage of subscribers from country X are about 73%. The obvious consequence is the wide presence of short-term visitors from abroad in the Operator X network. Next, an obvious question is to find out the distribution of foreign visitors over time. The results, for the three top countries, is as follows (again, the period without data is shaded):

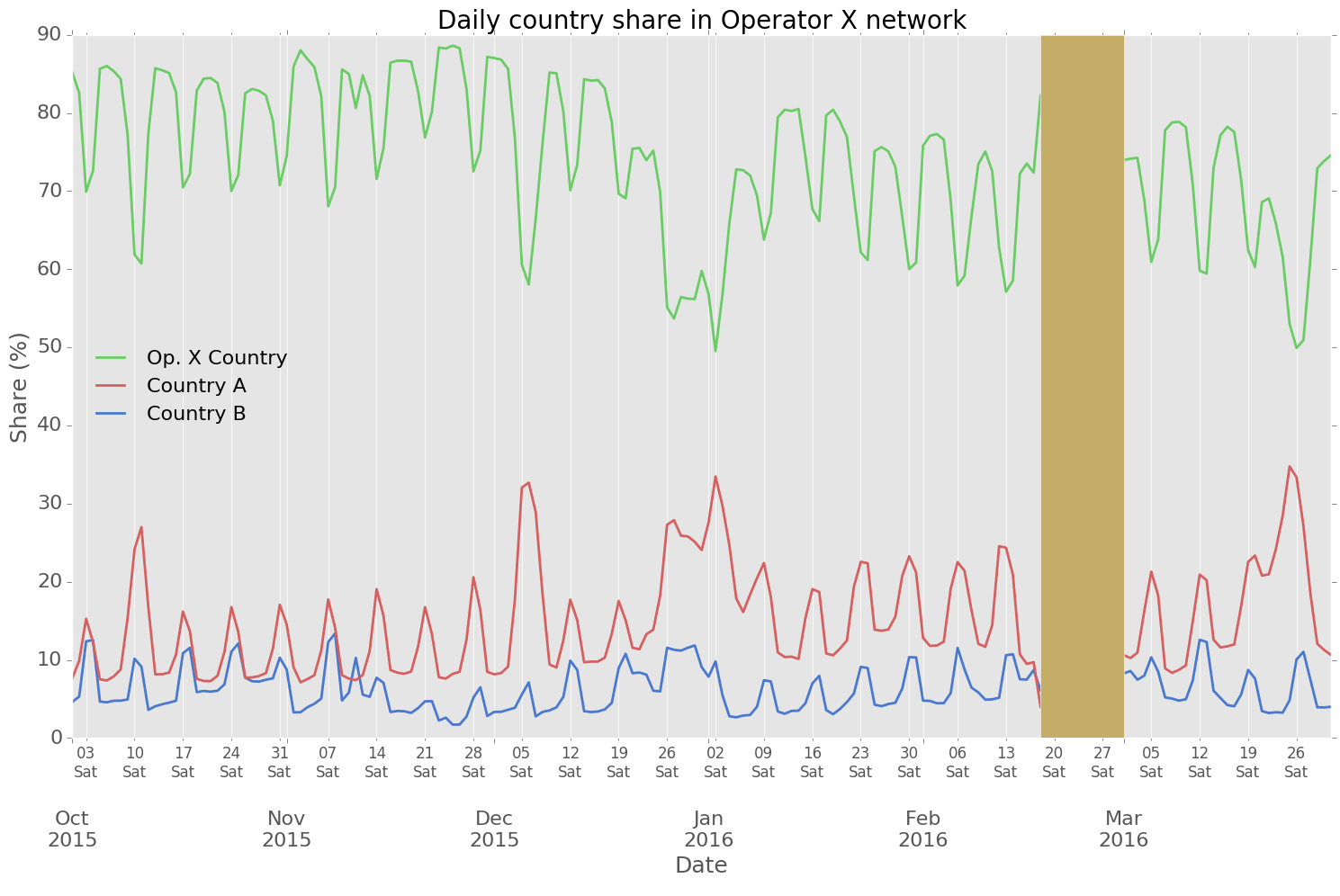


Figure 23: Evolution of country share (top three countries) over time

The figure shows straightforward results (beyond the lack of data in the second half of February). Presence of foreign visitors is higher in weekends and holidays. An interesting finding is that peaks in country A are found on Saturdays, while for country B, they happen on Sundays. Some zoom on the Christmas season can be also applied:

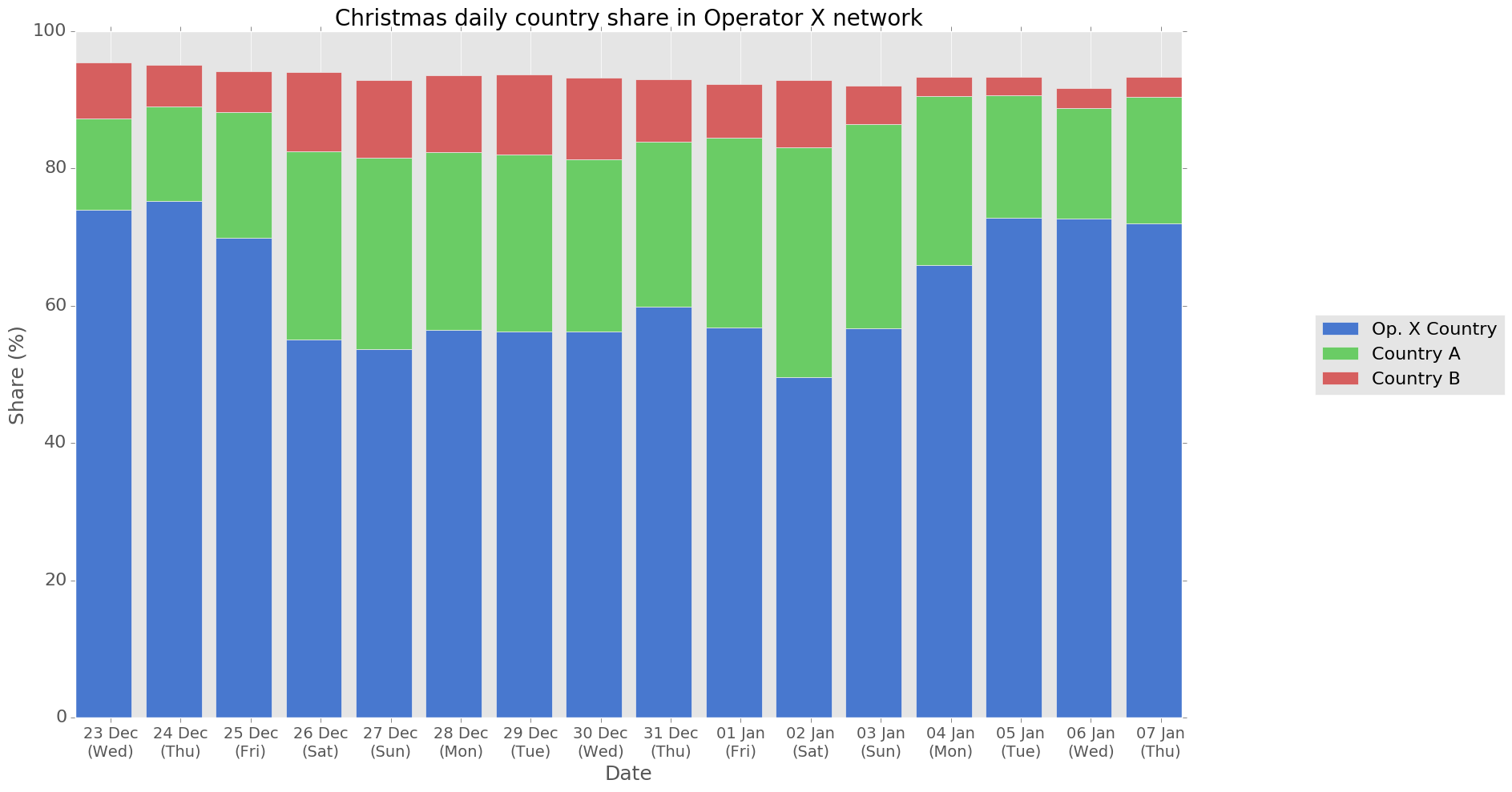


Figure 24: Evolution of country share (top three countries) in the Christmas season

A next question can how long visitors stay in the country, to classify them between tourists (stays no longer than 1 day) and commuters. The procedure is like the one used to determine terminal activations (the dataframe is turned into a RDD; for every subscriber, a reduce function is defined to get the first and last communication of the subscriber; finally, the result is turned again into a dataframe containing the subscriber identifier, home country, first and last recorded presence and days between such values.

# In[25]:

input\_hdfs\_path = u'hdfs://cluster-master:9000/data/DWFET\_CDR\_CELLID\_\*.parquet'

df\_unique\_calls = spark.read.format('parquet').load(input\_hdfs\_path)\

                      .filter(col('ID\_CDOPERADORORIGEN') != home\_mccmnc.value)\

                      .join(mccmnc\_df, col('ID\_CDOPERADORORIGEN') == mccmnc\_df["OP\_CODE"])\

                      .drop('ID\_CDOPERADORORIGEN')

# In[26]:

df\_roamer\_calls = df\_unique\_calls.withColumn("DT\_CDDAYSTART", udf\_get\_day("DT\_CDDATAINICI"))\

                     .withColumn("DT\_CDDAYEND", udf\_get\_day("DT\_CDDATAFI"))\

                     .withColumn("DT\_CD", array('DT\_CDDAYSTART', 'DT\_CDDAYEND'))\

                     .drop\_duplicates(['DS\_CDMSISDN', 'DT\_CD'])\

                     .select(['DS\_CDMSISDN', 'OP\_COUNTRY', 'DT\_CD'])

# In[27]:

**def** consolidate (x, y) :

    total\_set = set(x).union(set(y))

**return** [min(total\_set), max(total\_set)]

# In[28]:

rdd\_calls = df\_roamer\_calls.rdd\

                    .map(**lambda** line: ((line[0], line[1]), line[2]))\

                    .reduceByKey(**lambda** x, y: consolidate(x, y))\

                    .map(**lambda** line: (line[0][0], line[0][1], line[1][0], line[1][1]))

# In[29]:

dh\_schema = StructType([StructField("DS\_CDMSISDN", StringType(), True),

                        StructField("OP\_COUNTRY", StringType(), True),

                        StructField("STAY\_START", IntegerType(), True),

                        StructField("STAY\_END", IntegerType(), True),

                       ])

dh=spark.createDataFrame(rdd\_calls, dh\_schema)\

                   .withColumn("STAY\_LENGTH", col('STAY\_END') - col('STAY\_START'))

The results, for the five top countries is as follows:

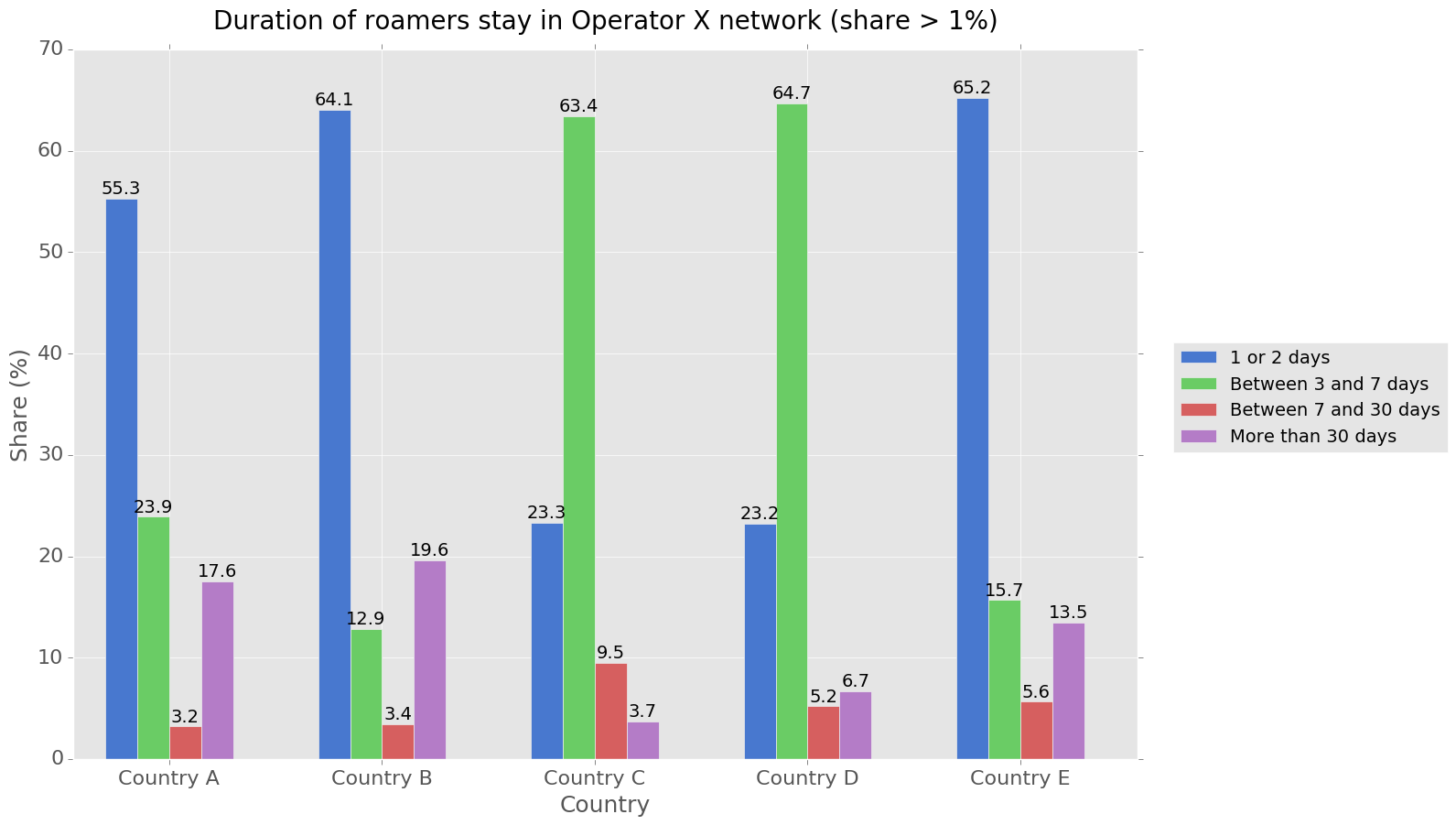


Figure 25: Duration of roamers stay in Operator X network

Finally, a plot describing the evolution of short-term visitors (duration shorter than two days) in the Operator X network over time is created:

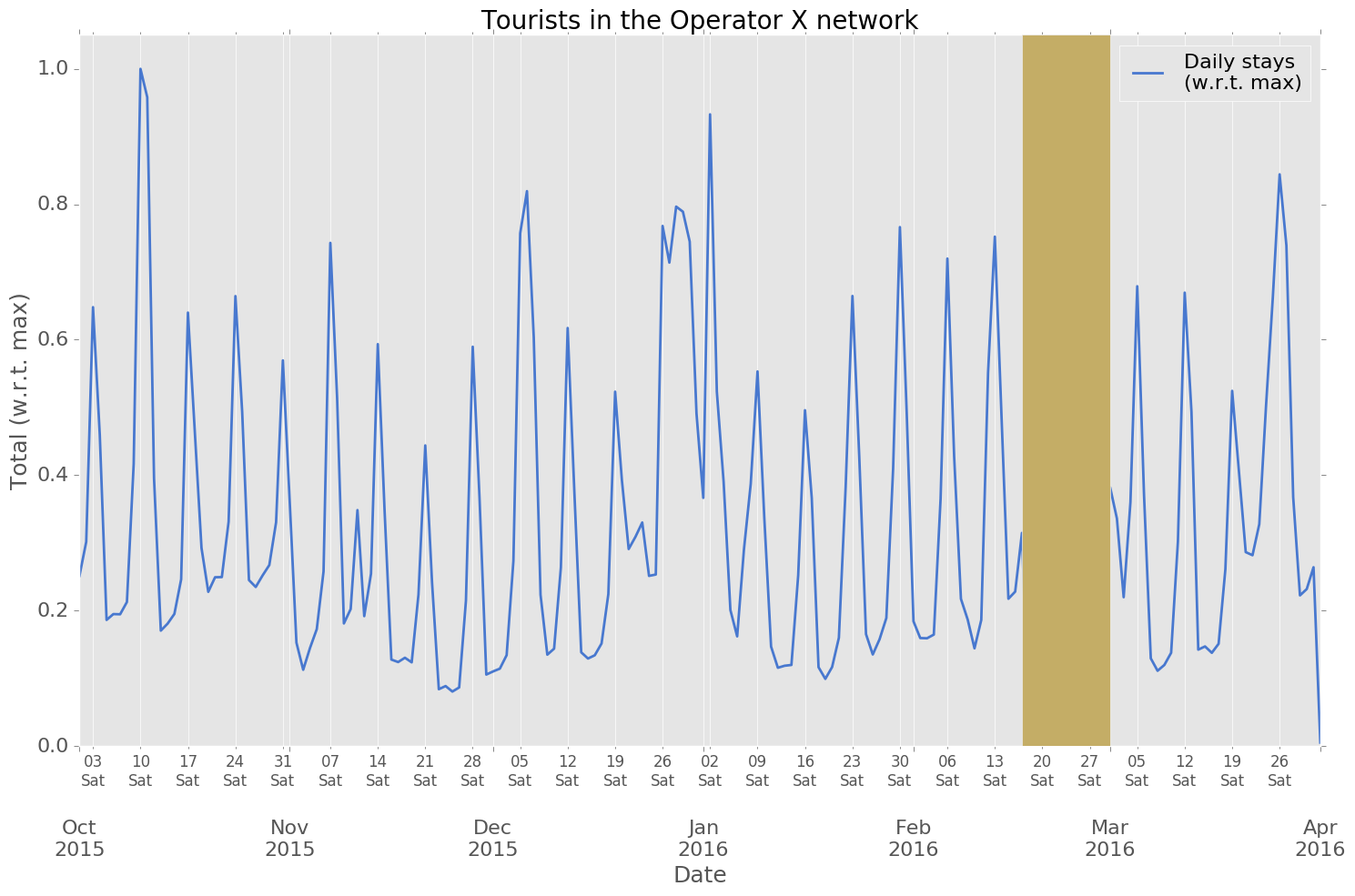


Figure 26: Short-term visitors (less than two days) in the Operator X network

The plot is like the one in Figure 22 and show clear peaks in weekends and holiday periods.

The last analysis that has been carried out describes the home operators of foreign visitors to the Operator X network. The most popular operators in the top six countries are shown:

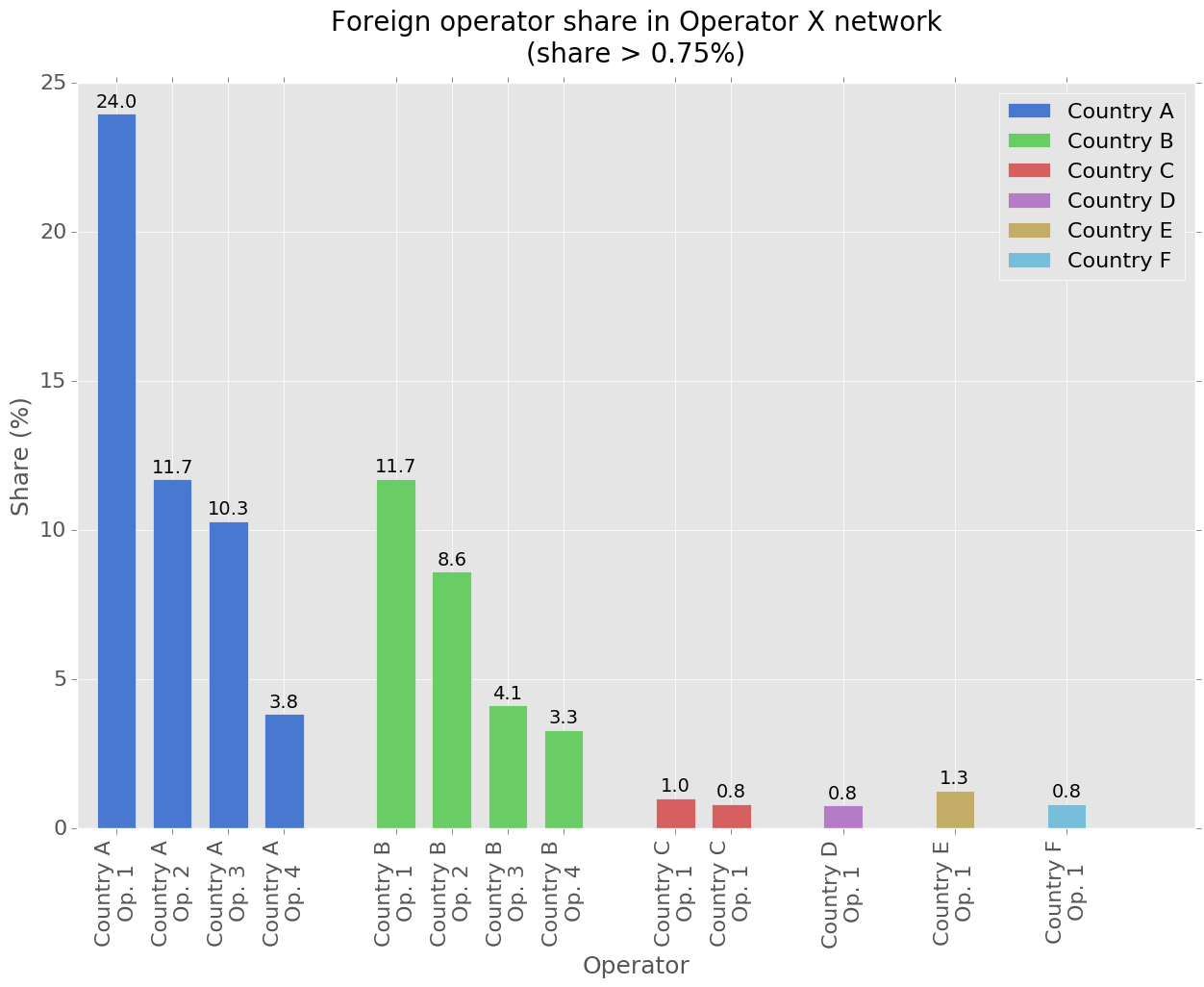


Figure 27: Operator split for top countries among visitors to Operator X network.

## Cell load computation and visualization

This task has two different phases: in the first phase, Spark computes the load of every cell over the analysis period and represents them as time series. Each point in the series contains the number of concurrent data connections each cell handles. In a second phase, Kibana is used to visualize the results. Thus, interworking between the Spark cluster and the Elastic Stack instance must have been configured beforehand according to what has been described in section 3.2.7.

### Spark processing

As mentioned, the result of the Spark processing must be as many time series as cells are active over the considered analysis period. The time series period has been set to five minutes (300 seconds): a smaller period would mean a more precise result but with no actual gain in visualization and higher computation requirements. Tasks are fulfilled with notebook 05\_get\_cell\_load.ipynb within the master-thesis-notebooks repository.

The procedure take as inputs the following fields in every CDR: ID\_CDTIPUSCOM, DT\_CDDATASTART, NUM\_LENGTH, and ID\_CELL\_START. It is described as follows:

* Data connections records are selected (ID\_CDTIPUSCOM==S-CDR). Others are discarded.
* A User Defined Function is used to get the instants, according to the series period (5 minutes) where a given CDR is active.

# In[4]:

poll\_period = 300            # Five minutes (300 seconds)

start\_of\_period = 1443657600 # 2015.10.01 00:00:00 (GMT)

# In[5]:

**def** get\_samples (x, y):

    start\_epoch = x

    session\_length = float(y)

    number\_of\_ticks=int(round(session\_length / float(poll\_period))) + 1

    ticks\_to\_start=int(round(float(start\_epoch - start\_of\_period) / poll\_period))

    ticks\_list = []

**for** i **in** range(number\_of\_ticks):

        tick = start\_of\_period + ((ticks\_to\_start + i) \* poll\_period)

        ticks\_list.append(tick)

**return** ticks\_list

udf\_get\_samples = udf(get\_samples, schema)

* An explode function is used to transform each record in as many records as active instants it contains.
* The result is grouped by cell identifier and instant and counted.

# In[6]:

input\_hdfs\_path = u'hdfs://cluster-master:9000/data/DWFET\_CDR\_CELLID\_\*.parquet'

df = spark.read.format('parquet').load(input\_hdfs\_path)\

                .filter(col('ID\_CDTIPUSCOM') != 'S-CDR')\

                .select(['DS\_CDMSISDN', 'DT\_CDDATASTART', 'NUM\_LENGTH', 'ID\_CELL\_SOURCE'])\

                .withColumn('DT\_CDDATASTART', unix\_timestamp('DT\_CDDATASTART'))\

                .withColumn('DS\_DATACOMM', udf\_get\_samples('DT\_CDDATASTART', 'NUM\_LENGTH'))\

                .select('ID\_CELL\_SOURCE', explode(col("DS\_DATACOMM")).alias("TICK\_ACTIVE"))\

                .groupBy(['ID\_CELL\_SOURCE', 'TICK\_ACTIVE']).count()

* A dataframe with all the possible pairs cell identifier/instant set to zero is created. It is used to create complete time series.
* Both dataframes are united, and the result is grouped by cell identifier and instant. The sum of each group is generated.

# In[39]:

df\_cells=df.union(df\_empty)\

          .groupBy(['ID\_CELL\_SOURCE', 'TICK\_ACTIVE']).agg(sum('count').alias('CELL\_LOAD'))\

          .orderBy(['ID\_CELL\_SOURCE', 'TICK\_ACTIVE'], ascending=False)  

### Elasticsearch management

There are several approaches to save results of Spark computations in an Elasticsearch index:

* Save results as a CSV file and use Logstash to load it to Elasticsearch.
* Save results as a JSON file and use elasticdump to load it to Elasticsearch (a description on how to properly install elasticdump in the Elastic Search instance can be found in <https://github.com/miguel-angel-monjas/master-thesis/blob/master/doc/spark-elasticsearch-setup.md>).
* Save results (as an RDD) directly from Spark. This the option that will be used and described.

Regardless of the approach, a mapping for the index that will host the data is required. In the project, it is necessary to store several time series, each of them describing the load of a specific cell of Operator X for six months. Thus, each point in the series is modelled as an Elasticsearch document (a document is a basic unit of information that can be indexed in Elasticsearch), made of three elements: a string identifying the cell (as this type has become deprecated[[28]](#footnote-28) in Elasticsearch, the keyword type will be used instead), a date representing a timestamp as seconds after the epoch (it is important to note that a right format definition must be added to the date type definition: epoch\_second; otherwise, Elasticsearch will consider the date element as a long number), and an integer representing the amount of concurrent connections at the timestamp in the given cell. Thus, the following mapping is defined in the Elasticsearch cluster (available in the elastic folder within the master-thesis repository at GitHub as <https://github.com/miguel-angel-monjas/master-thesis/blob/master/elastic/cell_load_mapping.json>):

{

  "cell\_info": {

    "mappings": {

      "cell\_load": {

        "properties": {

          "cell": {

            "type": "keyword"

          },

          "date": {

            "type": "date",

            "format": "epoch\_second"

          },

          "load": {

            "type": "integer"

          }

        }

      }

    }

  }

}

An index, cell\_info, and an associated document type, cell\_load, are thus defined.

Index creation can be accomplished in different ways: via the command line (using curl -X), by using Sense[[29]](#footnote-29) or the Dev Tools in Kibana, or by means of elasticdump.

**Index mapping creation via elasticdump**

The JSON document provided above is saved as a file, cell\_info\_mapping.json and the following command is run at the Elastic Stack instance:

elasticdump --type=mapping \  
--input=./cell\_info\_mapping.json \  
--output=http://localhost:9200 \  
--output-index=cell\_info

**Index mapping creation via the web interface**

The following HTTP query must be provided via Sense or the Kibana Developer Tool:

PUT cell\_info  
{  
 "mappings": {  
 "cell\_load": {  
 "properties": {  
 "cell": {  
 "type": "keyword"  
 },  
 "date": {  
 "type": "date",  
 "format": "epoch\_second"  
 },  
 "load": {  
 "type": "integer"  
 }  
 }  
 }  
 }  
}

If the index had to be deleted, a DELETE query could be used: DELETE cell\_info.

Once the index has been created, it is possible to write to it. Three elements must be considered when writing from PySpark to an Elasticsearch index:

* Access to the *elasticsearch-hadoop* binaries
* Proper configuration to route and enable writing operations from PySpark.
* Use of the *elasticsearch-hadoop* API from PySpark.

**Access to the *elasticsearch-hadoop* binaries**

As the *elasticsearch-hadoop* binaries have been copied to the $SPARK\_HOME/jars folder, there is no need to explicitly load them (either via --jars or via --packages).

**Configuration settings**

Several settings must be configured to write to Elasticsearch indices [16] (the full list of configuration settings is available in the official Elastic documentation):[17]

* es.nodes: List of Elasticsearch nodes, defaults to localhost.
* es.port: Elasticsearch port, defaults to 9200.
* es.resource: Where the Elasticsearch data is read and written to. It follows the format <index>/<type>.
* es.nodes.client.only: If the Elasticsearch cluster allows access only through client nodes, then this setting is necessary; defaults to *False*.

Other settings must be also configured:[18]

* es.nodes.discovery: to use the only the nodes in the Elasticsearch cluster given in es.nodes setting for metadata queries. Defaults to *True*. In the project environment, it must be set to *False*.

Thus, the PySpark notebooks running in the Spark cluster must define or load the following configuration settings to enable subsequent writing on the Elasticsearch cluster:

# In[12]:

es\_conf = {'es.resource': 'cell\_info/cell\_load',

          'es.nodes': '<elk-floating-ip-address>',

          'es.nodes.discovery': 'false',

          'es.nodes.data.only': 'false',

          'es.mapping.date.rich': 'false'}

**Write operation from PySpark**

Write operation to an Elasticsearch index is currently only possible from a pair RDD and not from dataframes. Thus, the dataframe must be first turned into an RDD and subsequently into a pair RDD (by means of a map transformation). This pair RDD must have a key (not relevant, any value can be used as key) and a value, which contains the actual content to write to the Elasticsearch index as a document, mapped to the Elasticsearch index mapping.

In the project, the cell load result is a dataframe with the following schema:

# In[40]:

df\_cells.printSchema()

# Out[40]:

root

 |-- ID\_CELL\_INI: string (nullable = true)

 |-- TICK\_ACTIVE: integer (nullable = false)

 |-- CELL\_LOAD: long (nullable = true)

Provided that es\_conf has been already defined in the PySpark notebook, the write operation is executed in the following way (a map transformation is used with the string id as key):

# In[42]:

df\_cells.rdd\

        .map(**lambda** item: ("id", {"cell": item[0], "date": item[1], "load": item[2]}))\

        .saveAsNewAPIHadoopFile (

                  path='-',

                  outputFormatClass="org.elasticsearch.hadoop.mr.EsOutputFormat",

                  keyClass="org.apache.hadoop.io.NullWritable",

                  valueClass="org.elasticsearch.hadoop.mr.LinkedMapWritable",

                  conf=es\_conf

                  )  

### Kibana visualization

Once the dataset is uploaded to Elasticsearch from the Spark application (it takes about two hours), it is possible to start the visualization task. The first step is the creation of an Index Pattern in Kibana (*Management > Index Patterns > Create Index Pattern*) from the index cell\_info\* in Elasticsearch. In the figure below, the date field is highlighted to show that the right type has been assigned. Proper date type assignation is key to create visualizations in Elasticsearch. Otherwise, it will not be possible to devise time series visualizations:

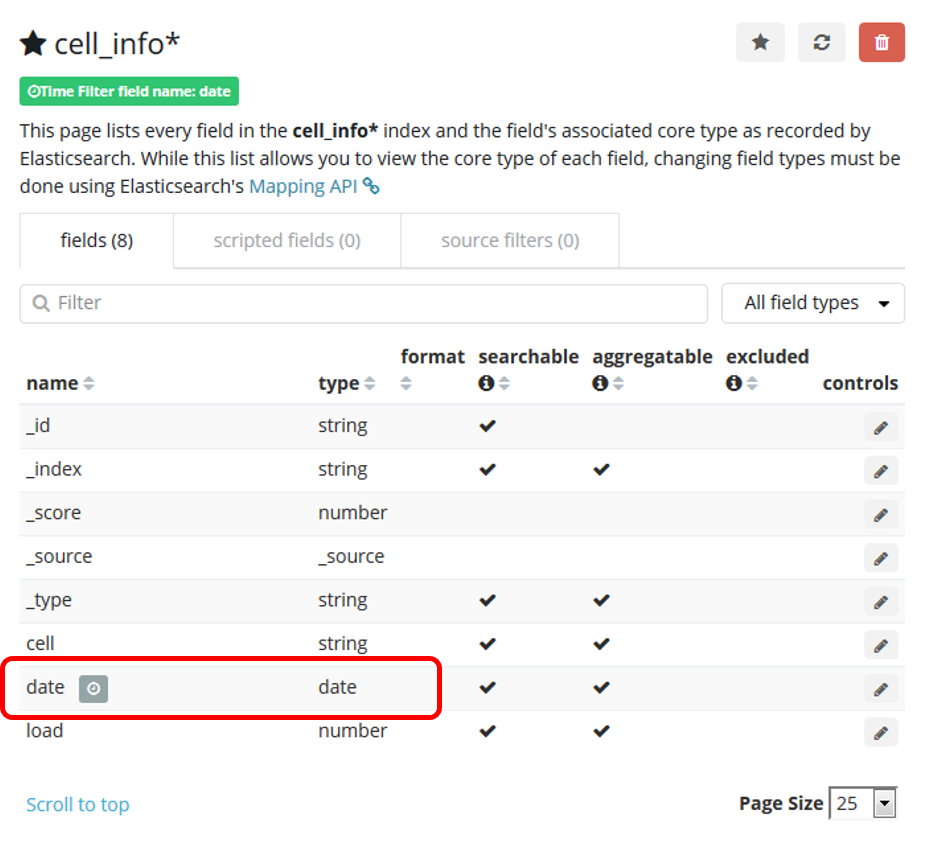


Figure 28: cell\_info\* Index Pattern at Kibana

Once the cell load information is available in Elasticsearch, it is possible to start handling it from Kibana. First, several searches are created, for they to be used to devise visualizations:

* **Not null values search**: Search that contains the documents with a not null load (according to the following Lucene query syntax: -load:0).
* Next, several filters to split the previous search across months are built. Separate searches are stored.

Next, the visualizations are designed:

* **Operator X cells**. It is a Metric visualization (*Visualize > Create new visualization > Metric*) showing the number of unique cells in the analyzed period. The aggregation used in the metrics is ‘Unique count of cell’.
* **Evolution of active cells in Operator X over time (weekly)**. It is a Line chart (*Visualize > Create new visualization > Line*) applied to the Not null values search (that is, non-zero load values). The metrics used is ‘Unique count of cell’ with a Date Histogram as aggregation in the Buckets for the Y axis (weekly).
* **Average cell load in Operator X network (daily)**. It is a Line chart applied to the Not null values search. The metrics used is ‘Average load’ and the buckets, over the X axis, are ‘date per day’.
* **Number of connections in Operator X network (daily)**. It is a Line chart applied to the Not null values search. The metrics used is ‘Sum of load’ and the buckets, over the X axis, are ‘date per day’.
* **Top loaded cells in Operator X network**. It is a Time Series visualization (*Visualize > Create new visualization > Time Series > Visual Builder*). In there, an aggregation of type Max applied to the field load, grouping by terms applied to the field cell creates the visualization. Only the top 20 cells are shown.

Finally, three dashboards are created:

* **Active cells in Operator X network**:

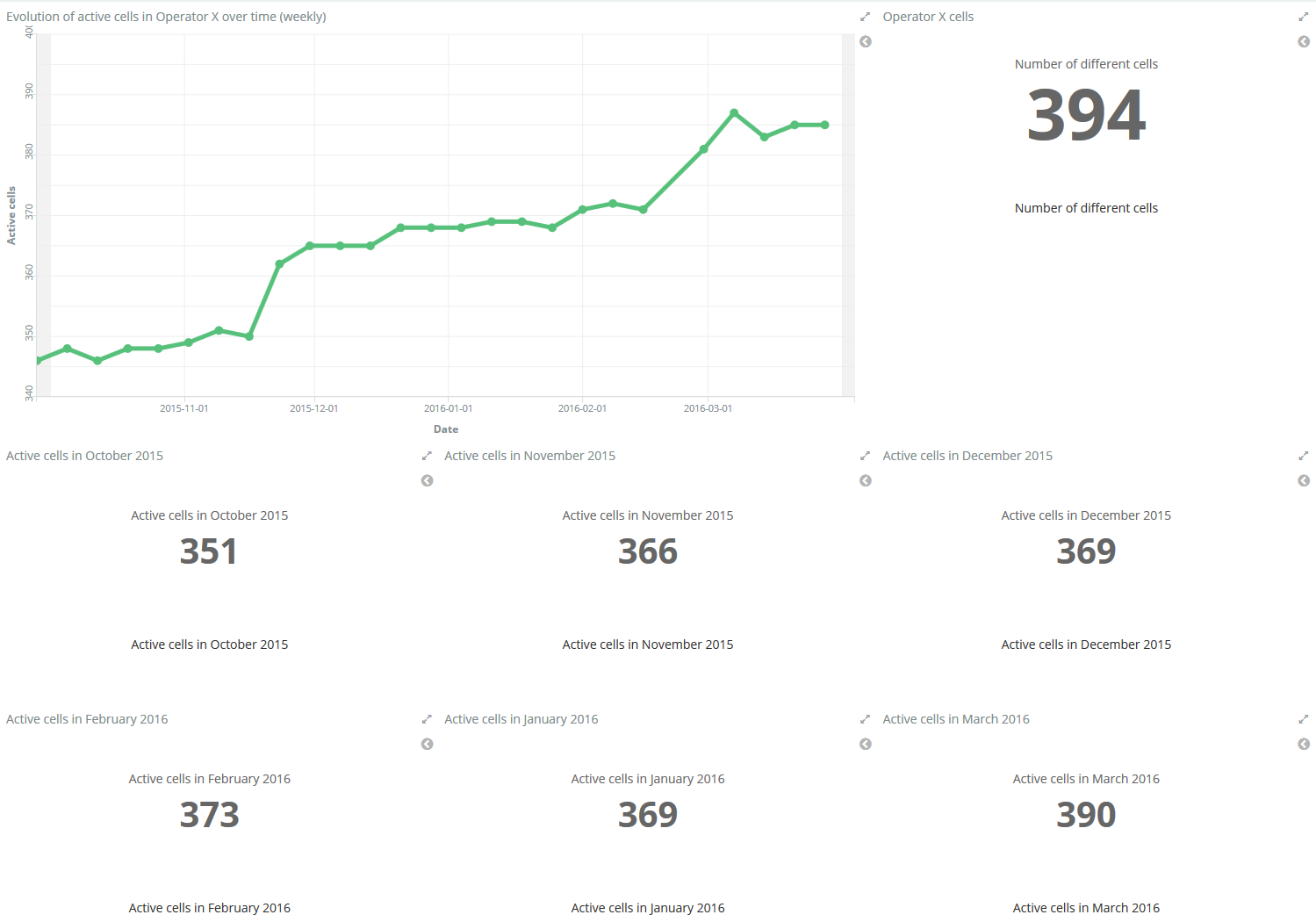


Figure 29: Active cells in Operator X (Kibana)

* **Loaded cells in Operator X network**:

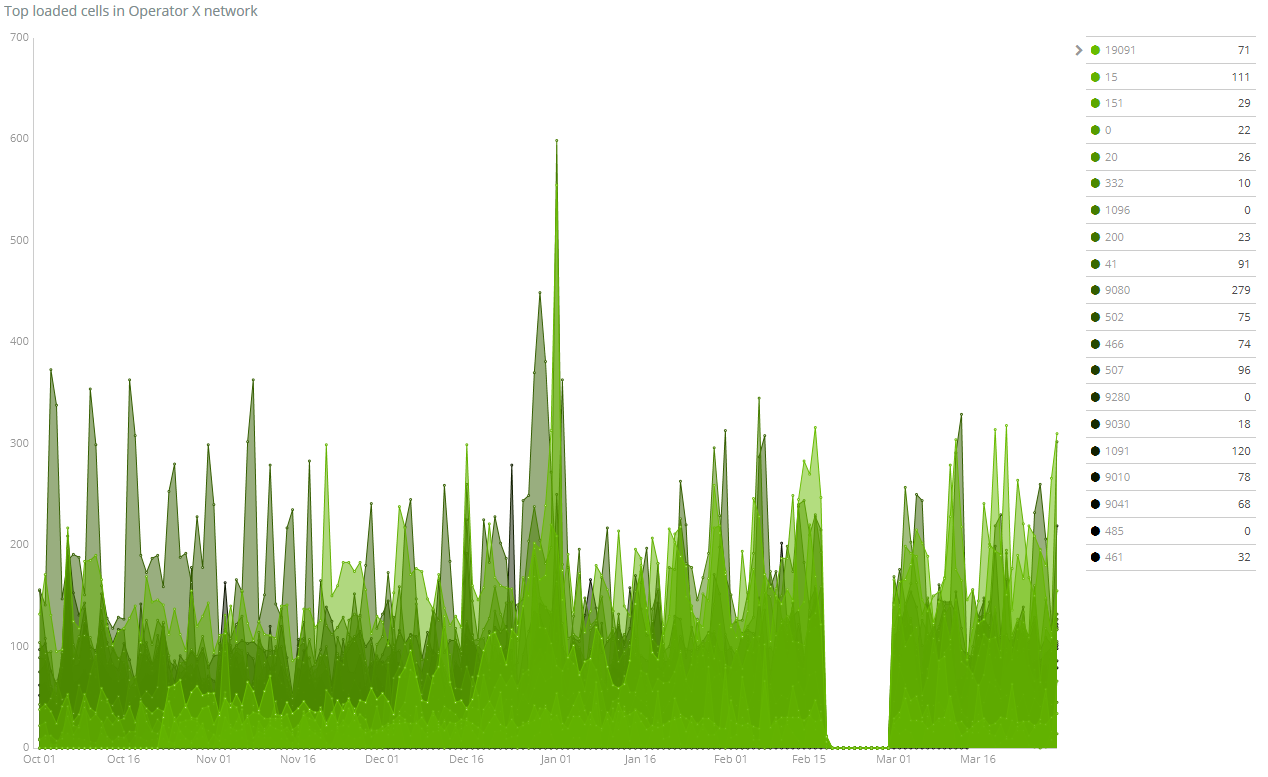


Figure 30: Top loaded cells in Operator X (Kibana)

* **Traffic in Operator X network**:



Figure 31: Traffic in Operator X network (Kibana)

All the objects related to the dashboards (searches, visualizations, and the actual dashboards) have been saved as JSON files and are available at the elastic folder within the master-thesis repository at GitHub).

## Cell graph computation

This task aims to extract the cell graph by analyzing the same user communications close in time. It is assumed that two communications that happens from different cells one immediately after the other come from two contiguous cells. The application run as a Spark notebook: 06\_get\_cell\_graph.ipynb within the master-thesis-notebooks repository. The procedure take as inputs the following fields in every CDR: NUM\_LENGTH, DS\_CDMSIDN, DT\_CDDATASTART, DT\_CDDATAEND, and ID\_CELL\_START. It is described as follows:

* Records describing communications longer than a given threshold (60 seconds) are not considered.
* For each record, and by means of a windowed lag function, the end time and cell of the previous communication for a same user are determined.
* Unsuitable records are discarded (those where the previous communication was carried out from the same cell, i.e. no cell hand over; those too long after the previous communication: threshold has been set to 30 seconds).
* The edges are extracted and stored in the HDFS cluster.

The following code snippet describes the procedure to extract the edges:

# In[5]:

call\_duration = 60 # only calls shorter than 60 seconds are considered

call\_interval = 30 # only calls started less than 30 seconds

                   # after the end of the previous call are considered

# In[6]:

windowSpec = Window.partitionBy(col('DS\_CDNUMORIGEN'))\

        .orderBy(col('DT\_CDDATAINICI'))

df = spark.read.format('parquet').load(input\_hdfs\_path)\

        .filter(col('NUM\_LENGTH') < call\_duration)\

        .select(['DS\_CDMISDN, 'DT\_CDDATASTART', 'DT\_CDDATAEND', 'ID\_CELL\_START'])\

        .withColumn('PREVIOUS\_CELL', lag(col('ID\_CELL\_START')).over(windowSpec))\

        .withColumn('END\_COMM\_TIME', lag(col('DT\_CDDATAEND')).over(windowSpec))\

        .filter(col('PREVIOUS\_CELL').isNotNull())\

        .filter(col('PREVIOUS\_CELL') != col('ID\_CELL\_START'))\

        .withColumn('DT\_CDDATASTART', unix\_timestamp('DT\_CDDATASTART'))\

        .withColumn('DT\_CDDATAEND', unix\_timestamp('END\_COMM\_TIME'))\

        .filter(col('DT\_CDDATASTART')-col('DT\_CDDATAEND') < call\_interval)\

        .withColumn('EDGE', udf\_get\_edge('ID\_CELL\_START', 'PREVIOUS\_CELL'))

The screenshot below show the DAG (Directed Acyclic Graph) of the count() action associated to the transformations in the code snippet above:

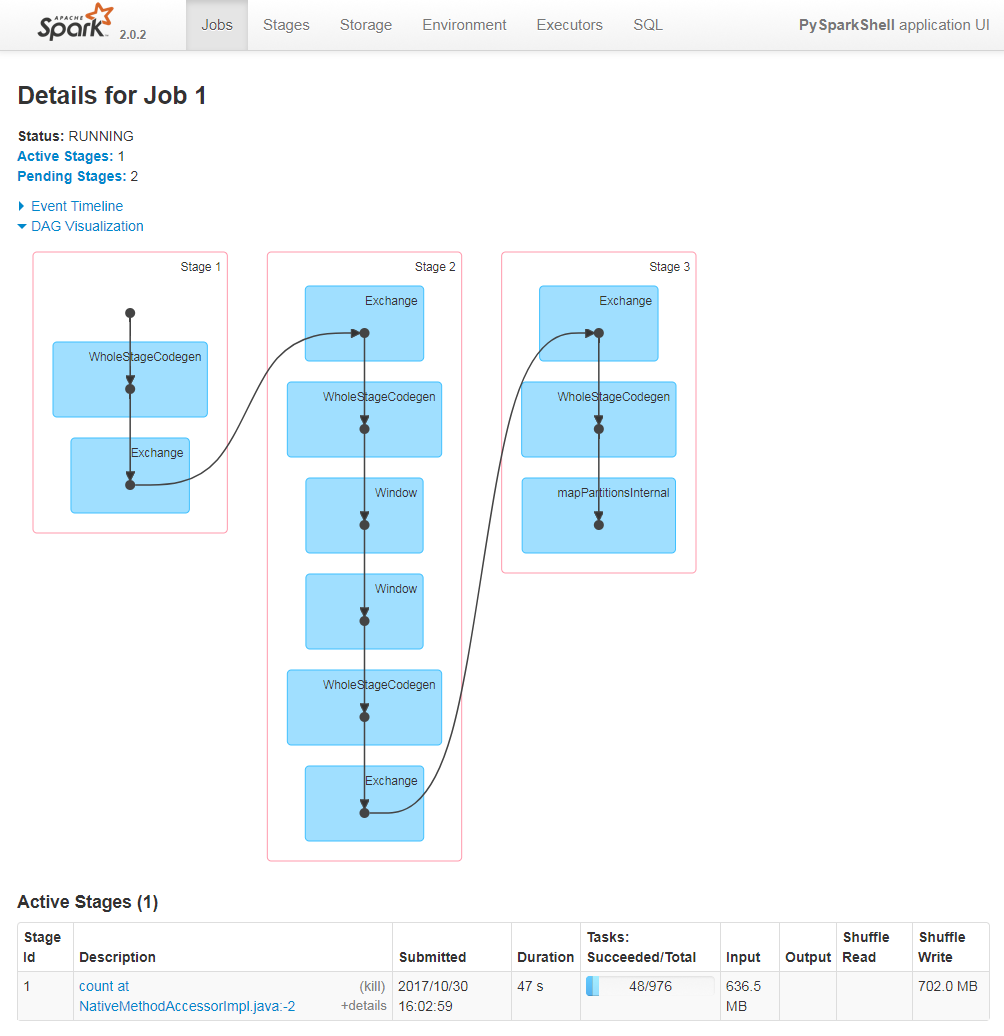


Figure 32: Spark Context UI: DAG for a count() action executed after dataframe transformations

Once extracted, the dataframe is saved as a CSV file in the HDFS cluster, as described in section 4.3.2 (Terminal analysis).

## Congestion management

Congestion management aims to detect when the number of concurrent connections in any cell exceeds a predefined threshold. It is enabled by a stream data source that publishes connection start and stop events. According to the visualizations generated in section 0, a period with high load (last week in 2015) has been selected and the connection information is extracted from the dataset so that it is possible to publish the start and stop information. A Spark Streaming application consumes said events, continuously computes the number of concurrent connections in every cell, and finally sends the information to the Elasticsearch instance, where an online visualization is provided. Inbound and outbound data to and from the Spark cluster is handled by means of the Kafka message broker.

Thus, in a first phase, and provided that the Kafka broker has been set up, a Pyspark application extracts the information from the dataset in the HDFS cluster and stores start and stop details in a CSV file. Next, the CSV file is saved locally and a Python script processes it, creates Kafka messages passing on the details of the start and the end of the communication through a topic in the Kafka broker. Each message contains the cell identifier and the type of event (start, stop). No timestamp is included as the data source sends the messages as it were created. In a second phase, a Spark Streaming application listens to the topic defined in the first phase and continuously computes the load in all cells. It also determines how many cells have their load above the threshold. Finally, in a third phase, the Spark Streaming application sends the outputs of the computation to output topics in the Kafka broker. The Elastic Stack instance has been configured, through Logstash, to receive the information, write it to an Elasticsearch index, and visualize it through Kibana.

### Data source simulation

The Kafka message broker is available at <kafka-ip-address>. The extraction of connection information in the selected time window is done by means of a Python Spark notebook (11\_extract\_connection.ipynb within the master-thesis-notebooks repository). The connection information is saved as a CSV file in the HDFS cluster, which is saved locally afterwards. Finally, a Python script, which plays the role of a Kafka producer, publishes messages to a Kafka topic, cdr, using a CSV-like format: <epoch\_time>;<cell identifier>;<type\_of\_event>, where type\_of\_event can have two values: start and end.

### Spark processing

The Spark Streaming application reads from the Kafka cdr topic and computes the number of active connections for every cell over the batch period. A pre-requisite for reading and writing from Kafka is to have the *spark-streaming-kafka* binaries available. As they have been copied to the $SPARK\_HOME/jars folder in every instance in the cluster (see 3.2.8), there is no need to explicitly load them (either via --jars or via --packages).

The Spark Streaming context is created by using the following configuration:

# In[4]:

config = {

  "output\_topic\_load": "load",

  "output\_topic\_congestion": "congestion",

  "input\_topic": "cdr",

  "congestion\_threshold": 50,

  "kafka\_ip": "10.10.10.66:9092",

  "checkpointKafka": "checkpoint",

  "checkpoint\_dir": u'hdfs://cluster-master:9000/checkpoints',

  "streamBatchSize": 10,

  "windowSize": 30,

  "execTime": 10

}

# In[5]:

ssc = StreamingContext(sc, config["streamBatchSize"])

The Spark Streaming processing generates two outcomes:

* For each batch, the number of concurrent connections being handled by each cell at the end of the batch. The output is pushed, as a JSON message, to a Kafka topic, load (see code snippet below.)
* For each batch, the number and ratio of congested cells (a congested cell is defined as that with a load beyond a given threshold) and the number of cells under the congestion threshold. The output is pushed, as a JSON message, to a Kafka topic, congestion (see code snippet below.)

# In[13]:

# connections per cell in every batch

cdrs = input\_kafka\_stream.map(**lambda** x: x[1])\

                            .flatMap(parse\_event)\

                            .map(**lambda** item: (item["cell"], (((2\*int(item["event"] == True))-1), item["time"])))\

                            .reduceByKey(**lambda** x, y: (x[0]+y[0], max(x[1], y[1])))

# per cell state is updated and published

load = cdrs.updateStateByKey(update\_load)

load.map(create\_load\_json)\

    .foreachRDD(**lambda** rdd: rdd.foreachPartition(**lambda** x: send\_to\_kafka(x, config['output\_topic\_load'])))

# congestion share is computed

top\_alarms = load.filter(**lambda** x: x[1][1] > config['congestion\_threshold'])\

                 .count().map(**lambda** x: ("congested", x))\

                 .union(load.count().map(**lambda** x: ("total", x)))\

                 .union(load.map(**lambda** x: x[1][2]).reduce(**lambda** x,y: max(x,y)).map(**lambda** x: ("time", x)))\

                 .reduce(consolidate\_alarms)

top\_alarms.map(create\_congestion\_json)\

          .foreachRDD(**lambda** rdd: rdd.foreachPartition(**lambda** x: send\_to\_kafka(x, config['output\_topic\_congestion'])))

The messages published to the load topic follow the following JSON schema:[[30]](#footnote-30)

{

  "$schema": "http://json-schema.org/draft-06/schema#",

  "properties": {

    "date": {

      "description": "Date of the load event.",

      "type": "number"

    },

    "cell": {

      "description": "Cell the load value refers to.",

      "type": "string"

    },

    "load": {

      "description": "Number of concurrent connections.",

      "type": "integer"

    }

  },

  "type": "object"

}

Messages published to the congestion topic are even simpler:

{

  "$schema": "http://json-schema.org/draft-06/schema#",

  "properties": {

    "congested": {

      "description": "Number of congested cells.",

      "type": "integer"

    },

    "uncongested": {

      "description": "Number of uncongested cells.",

      "type": "integer"

    },

    "total": {

      "description": "Total number of cells.",

      "type": "integer"

    },

    "ratio": {

      "description": "Ratio of congested cells.",

      "type": "number"

    },

    "date": {

      "description": "Time the congestion share value refers to.",

      "type": "number"

    }

  },

  "type": "object"

}

Tasks are fulfilled with notebook 07\_congestion\_streaming.ipynb within the master-thesis-notebooks repository.

### Elastic Stack management

Unlike the scenario described in section 4.4, the interworking between the Spark Streaming application and Elasticsearch indices is handled through Kafka and Logstash. Logstash is properly configured to consume messages from the Kafka topics where the Spark Streaming application publishes its outcomes, transforms the JSON content passed on by the Kafka messages and write them to the Elasticsearch indices.

The first step to take is the definition of the Elasticsearch indices that will continuously store the cell load and the cell congestion information and the associated mappings (if mappings for the index are not available, Elasticsearch will infer the types). One of the indices, realtime\_load\_info, will be the identical with the index in section 4.4.2. A new index, realtime\_congestion\_info, is defined to host the share of congested cells. Thus, the following indices are defined beforehand in the Elasticsearch cluster:

PUT realtime\_load\_info  
{  
 "mappings": {  
 "cell\_load": {  
 "properties": {  
 "cell": {  
 "type": "keyword"  
 },  
 "date": {  
 "type": "date",  
 "format": "epoch\_second"  
 },  
 "load": {  
 "type": "integer"  
 }  
 }  
 }  
 }  
}

PUT realtime\_congestion\_info  
{  
 "mappings": {  
 "congestion": {  
 "properties": {  
 "date": {  
 "type": "date",  
 "format": "epoch\_second"  
 },  
 "ratio": {  
 "type": "float"  
 },  
 "total": {  
 "type": "integer"  
 },  
 "uncongested": {  
 "type": "integer"  
 },  
 "congested": {  
 "type": "integer"  
 }  
 }  
 }  
 }  
}

Next, appropriate pipeline configuration files to cope with Kafka messages must be defined in Logstash. The Elastic Stack deployment was described in section 3.2.7: it involved the creation of a folder in the host instance where Logstash pipeline configuration files must be placed (logstash/pipeline). It is not possible to define a separate pipeline configuration file for each index, as each of them is written from a different Kafka topic and Logstash only supports one Kafka consumer per running instance (each time a Kafka input section is found, Logstash attempts to start a consumer). Thus, a single file is needed, logstash.conf:

input {

kafka {

decorate\_events => true

bootstrap\_servers => "10.10.10.66:9092"

topics => ["congestion", "load"]

}

}

filter {

json {

source => "message"

}

if [kafka][topic] == "congestion" {

mutate {

add\_field => {"[@metadata][index]" => "realtime\_congestion\_info"}

add\_field => {"[@metadata][document\_type]" => "congestion"}

}

} else {

mutate {

add\_field => {"[@metadata][index]" => "realtime\_load\_info"}

add\_field => {"[@metadata][document\_type]" => "cell\_load"}

}

}

mutate {

remove\_field => ["kafka"]

}

}

output {

elasticsearch {

action => "index"

hosts => "elasticsearch:9200"

index => "%{[@metadata][index]}"

document\_type => "%{[@metadata][document\_type]}"

}

Next, Index Patterns must be defined in Kibana to provide visualizations of the Elasticsearch indices: realtime\_load\_info\* and realtime\_congestion\_info\*.

Finally, some visualizations and dashboards are created:

* **Streaming-Average cell load**. It is a Time Series visualization. In there, an aggregation of type Average applied to the field load in the index realtime\_load\_info.
* **Streaming-Congestion share**. It is a Vertical Bar visualization showing the number of congested and uncongested cells as a stacked bar (from the realtime\_congestion\_info index). The two aggregations used in the metrics (Y-Axis) are ‘Max of Congested’ and ‘Max of Uncongested’.
* **Streaming- Congestion ratio**. It is a Metric visualization showing the congestion ratio. The aggregation used in the metrics is ‘Average of ratio’ (from the realtime\_congestion\_info index). A filter over the last 20 seconds (now-20s to now) is used.
* **Streaming-Congested cells**. It is a Metric visualization showing the number of congested cells. The aggregation used in the metrics is ‘Max of congested’ (from the realtime\_congestion\_info index). A filter over the last 20 seconds (now-20s to now) is used.

A single dashboard is created:

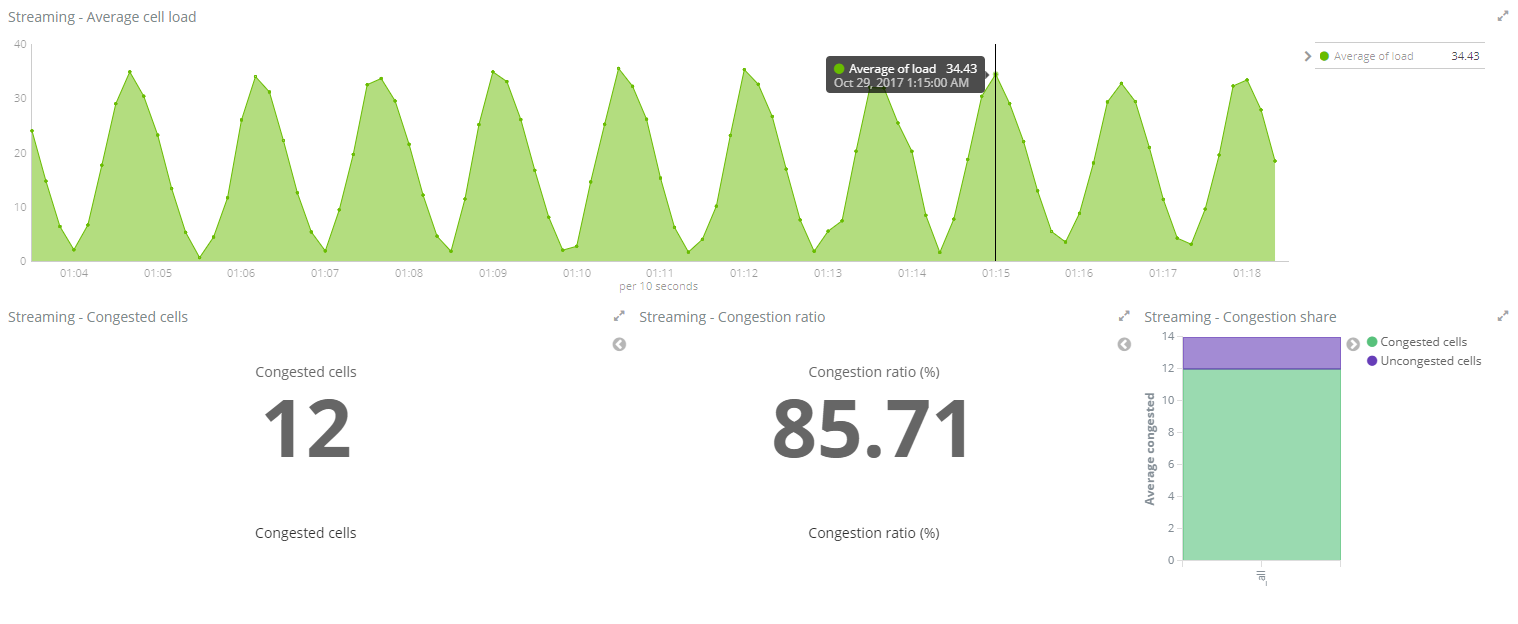


Figure 33: Congestion status (Kibana)

# Open issues and further work

Data availability enable an endless field of applications both in the deployment and the execution phases. Without a sizeable amount of data, analysis is futile. This master’s thesis has carried out a modest work with many elements that can be subject of further experimentation:

* Deployment:
  + Containerization and automatization of the Spark Standalone Cluster deployment. The deployment documentation generated makes it simple to create appropriate Dockerfiles and build Docker images.
  + Orchestration based on Docker Swarm or Kubernetes.
  + Cluster management based on YARN.
* Execution:
  + Visualization based on seaborn, as it provides a higher abstraction level than matplotlib.
  + User of Zeppelin as notebook technology.
  + Integration of graph information extracted in section 4.5 to refine the functionality in section 0 (that is, determining whether a congestion happens not only in each cell but also in contiguous cells). Visualization of graphs.
  + Cell load analysis as time series in order to fill the gap in late February.
  + Clusterization of time series (cell load).

# Conclusions

An aspect that is frequently avoided (in fact, given for granted) is the infrastructure needed to carry out analytics task when large amounts of data are involved. Toy datasets can be processed with local tools deployed in a simple laptop. However, real life is a little bit more complicated, as large datasets must be stored and processed.

The project has been run for two months and a half. The amount of effort required to implement it has been about one person-month. At least 40% of such effort has been invested in section 3. That is, infrastructure deployment has required a non-negligible effort. Worse enough, the execution could not start until it was ready (new projects will not have to take this toll again). This, somehow unexpected, finding supports the need for new approaches such as DevOps,[[31]](#footnote-31) which emphasizes the value of collaboration between development and operations staff throughout all stages of the development lifecycle when creating and operating a service. Development requires a clear and first-hand understanding of what operation means and requires.

When it comes to the execution, some findings can be reported:

* Use of notebooks is a powerful tool not only for testing and debugging, but also for interactive execution of tasks. It must be recommended.
* matplotlib is a comprehensive package for visualization, but with a low level of abstraction. If the project were run again, seaborn would be used.
* The integration between Spark and the Elastic Stack has been smooth (both in the batch and streaming scenarios). The combined use of both technologies is a good option for exploratory data analysis.
* Full product maturity has not been reached yet. Misalignment between releases (Spark vs. Zeppelin, for instance), unclear or uncompleted documentation (stack overflow is a must when running these technologies) … are some examples.

The conclusion cannot be but positive. Even if the challenges have been serious, it is possible to acknowledge that powerful enough tools are already available and can be used to get the expected results.

# References

1. Apache Hadoop 2.7.4. 2017. HDFS Users Guide. [ONLINE] Available at: <https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>.
2. Apache Hadoop 2.7.4. 2017. Hadoop Cluster Setup. [ONLINE] Available at: <https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/ClusterSetup.html>.
3. Chawla, Sumit. 2015. Install a Multi Node Hadoop Cluster on Ubuntu 14.04. [ONLINE] Available at: <https://chawlasumit.wordpress.com/2015/03/09/install-a-multi-node-hadoop-cluster-on-ubuntu-14-04/>.
4. Apache Hadoop 2.7.4. 2017. *File System (FS) Shell overview*. [ONLINE] Available at: <https://hadoop.apache.org/docs/r2.7.4/hadoop-project-dist/hadoop-common/FileSystemShell.html>.
5. Apache Hadoop 2.7.4. 2017. HDFS Support for Multihomed Networks: Ensuring HDFS Daemons Bind All Interfaces. [ONLINE] Available at: <https://hadoop.apache.org/docs/r2.7.4/hadoop-project-dist/hadoop-hdfs/HdfsMultihoming.html#Ensuring_HDFS_Daemons_Bind_All_Interfaces>.
6. Apache Spark 2.0.2. 2016. Spark Standalone Mode. [ONLINE] Available at: <https://spark.apache.org/docs/2.0.2/spark-standalone.html>.
7. stack overflow. 2016. How to get rid of derby.log, metastore\_db from Spark Shell. [ONLINE] Available at: <https://stackoverflow.com/questions/38377188/how-to-get-rid-of-derby-log-metastore-db-from-spark-shell/44048667#44048667>.
8. Jupyter Team. 2015. Running a notebook server: Running a public notebook server. [ONLINE] Available at: <http://jupyter-notebook.readthedocs.io/en/latest/public_server.html#running-a-public-notebook-server>.
9. Apache Zeppelin. 2017. Zeppelin: Quick Start. [ONLINE] Available at: <https://zeppelin.apache.org/docs/0.7.2/install/install.html>.
10. Docker Inc. 2017. Get Docker CE for Ubuntu. [ONLINE] Available at: <https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/>.
11. Docker Inc. 2016. Install Docker Compose. [ONLINE] Available at: <https://docs.docker.com/compose/install/>.
12. Lapenna, Anthony. 2017. Docker ELK stack. [ONLINE] Available at: <https://github.com/deviantony/docker-elk/>.
13. Elasticsearch BV. 2017. Install Elasticsearch with Docker edit. [ONLINE] Available at: <https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html>.
14. Elasticsearch BV. 2017. Elasticsearch for Apache Hadoop [5.6]. [ONLINE] Available at: <https://www.elastic.co/guide/en/elasticsearch/hadoop/5.6/reference.html>.
15. ITU-T. 2016. *Recommendation* E.212: The international identification plan for public networks and subscriptions. [ONLINE] Available at: <https://www.itu.int/rec/T-REC-E.212-201609-I>.
16. Kothuri, Prasanth. 2016. Integrating Hadoop and Elasticsearch – Part 2 – Writing to and Querying Elasticsearch from Apache Spark. [ONLINE] Available at: <https://db-blog.web.cern.ch/blog/prasanth-kothuri/2016-05-integrating-hadoop-and-elasticsearch-%E2%80%93-part-2-%E2%80%93-writing-and-querying>.
17. Elasticsearch BV. 2017. Elasticsearch for Apache Hadoop [5.6]: Configuration. [ONLINE] Available at: <https://www.elastic.co/guide/en/elasticsearch/hadoop/5.6/configuration.html>.
18. Discuss the Elastic Stack. 2016. SparkStreaming to Elasticsearch ERROR NetworkClient: Connection timed out: connect. [ONLINE] Available at: <https://discuss.elastic.co/t/sparkstreaming-to-elasticesrahc-error-networkclient-connection-timed-out-connect/45834/2>.

1. For privacy reasons, Operator X is not identified. [↑](#footnote-ref-1)
2. <https://www.openstack.org/> [↑](#footnote-ref-2)
3. <https://docs.openstack.org/sahara/latest/> [↑](#footnote-ref-3)
4. <http://www.agilemodeling.com/artifacts/userStory.htm> [↑](#footnote-ref-4)
5. Access to the private repository can be required by sending an email to the repository owner. [↑](#footnote-ref-5)
6. <http://hadoop.apache.org/docs/r2.7.4/> [↑](#footnote-ref-6)
7. <https://spark.apache.org/releases/spark-release-2-0-2.html> [↑](#footnote-ref-7)
8. <https://zeppelin.apache.org/docs/0.7.2/> [↑](#footnote-ref-8)
9. <https://www.anaconda.com/distribution/> [↑](#footnote-ref-9)
10. <https://github.com/minrk/findspark> [↑](#footnote-ref-10)
11. <https://seaborn.pydata.org/> [↑](#footnote-ref-11)
12. <https://www.elastic.co/products> [↑](#footnote-ref-12)
13. <https://kafka.apache.org/> [↑](#footnote-ref-13)
14. <https://www.docker.com/community-edition> [↑](#footnote-ref-14)
15. <https://docs.docker.com/compose/> [↑](#footnote-ref-15)
16. <https://www.manning.com/books/spark-in-action> [↑](#footnote-ref-16)
17. <https://jupyter.org/> [↑](#footnote-ref-17)
18. Unlike Jupyter, notebooks are not stored directly under the configured directory, but under subfolders (one folder by each notebook) with a random name. The note is available inside the folder, as a JSON file called note.json. [↑](#footnote-ref-18)
19. <http://central.maven.org/maven2/org/elasticsearch/elasticsearch-spark-20_2.11/5.6.1/elasticsearch-spark-20_2.11-5.6.1.jar> [↑](#footnote-ref-19)
20. <http://download.elastic.co/hadoop/elasticsearch-hadoop-5.6.1.zip> [↑](#footnote-ref-20)
21. <https://mvnrepository.com/artifact/org.elasticsearch/elasticsearch-spark-20_2.11/5.6.1> [↑](#footnote-ref-21)
22. <https://hub.docker.com/r/spotify/kafka/> [↑](#footnote-ref-22)
23. <https://mvnrepository.com/artifact/org.apache.spark/spark-streaming-kafka-0-8_2.11/2.0.2> [↑](#footnote-ref-23)
24. <http://en.wikipedia.org/wiki/Call_detail_record> [↑](#footnote-ref-24)
25. <https://parquet.apache.org/> [↑](#footnote-ref-25)
26. <http://www.urbanophile.com/arenn/hacking/gzrt/gzrt.html> [↑](#footnote-ref-26)
27. For subsequent retrieval of the CSV file, the following HDFS File System shell command can be used:

    hdfs dfs -get /dataframes/subscriber\_df\_01.csv/\*.csv ./subscriber\_df\_01.csv

    (as the actual file is named part\* under the folder subscriber\_df\_01.csv [↑](#footnote-ref-27)
28. <https://www.elastic.co/guide/en/elasticsearch/reference/5.6/string.html> [↑](#footnote-ref-28)
29. <https://chrome.google.com/webstore/detail/sense-beta/lhjgkmllcaadmopgmanpapmpjgmfcfig?hl=en> [↑](#footnote-ref-29)
30. <http://json-schema.org/latest/json-schema-core.html> [↑](#footnote-ref-30)
31. <http://radar.oreilly.com/2012/06/what-is-devops.html> [↑](#footnote-ref-31)