



OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

INF

FAKULTÄT FÜR
INFORMATIK

Documentation VLBA II – System Architectures

Usage of Apache Spark and Graphx

Subash Prakash
220408
Magdeburg, 29. Juni 2018



Table of contents

Table of contents	1
Introduction	2
1. Apache Spark Eco System	2
1.1 Spark Core API	3
1.1.1 Spark SQL	3
1.2 Spark Streaming	3
1.2.1 MLlib Machine Learning Library	3
1.2.2 GraphX	4
1.3 Installation	4
1.3.1 Deployment Plan:	4
1.3.2 Pre-Installations (Installation of java, pip3):	4
1.3.3 Installation of Spark in both VMs	5
1.3.4 Setup of pyspark and jupyter-notebook, so that the code can be executed from my local machine to VM.....	9
2. Examples.....	9
2.1 Apache Spark Exploratory Data Analysis of Imdb Dataset with pyspark	10
2.2 Apache Spark Streaming Twitter Hashtags – a simple way to build hash tag counter on a topic.....	14
2.2.1 Architecture.....	15
2.2.2 Execution of notebook to get the hashtag and their count of twitter through kafka server	16
2.3 Exploratory data analysis of Graphx:	17
3. References	19



Introduction

Apache Spark is an open-source cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the [Apache Software Foundation](https://www.apache.org/foundation/), which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.

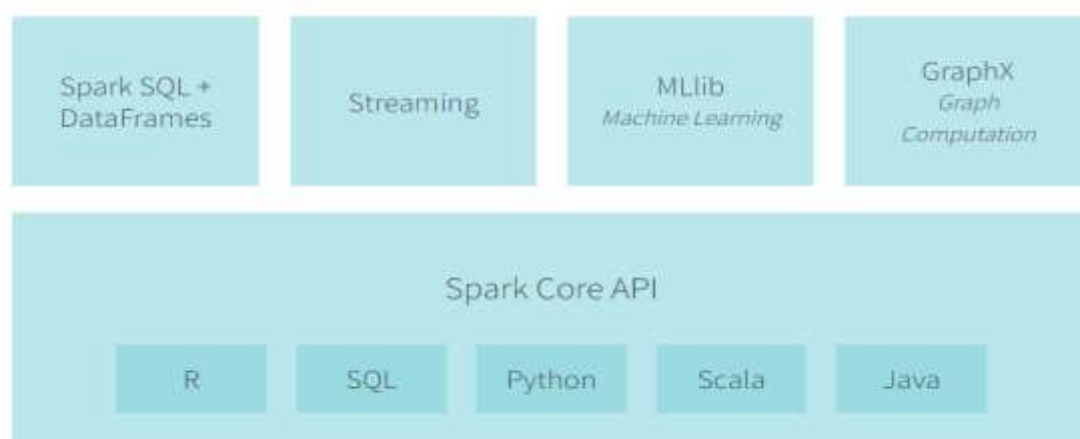
It is a largest data processing open source platform available to handle very large data.

Spark is an all-purpose data processing engine that can be used in a variety of circumstances. Application developers and data scientists can incorporate Spark into their applications to quickly query, analyse, and transform data at scale.

Since the start, Spark was optimised to run in memory, allowing it to process data far more quickly than alternative approaches like Hadoop's MapReduce, which tends to write data to and from computer hard drives between each stage of processing.

Some claim that **Spark running in memory can be up to 100 times faster than Hadoop MapReduce**. Yet this comparison is not entirely fair as raw speed tends to be more important to Spark's typical use cases than it is to batch processing, at which MapReduce-like solutions still excel.

1. Apache Spark Eco System



source: <https://databricks.com/spark/about>



1.1 Spark Core API

Spark Core is the foundation of the overall project. It provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) centered on the RDD abstraction. This interface mirrors a functional/higher-order model of programming: a "driver" program invokes parallel operations such as map, filter or reduce on an RDD by passing a function to Spark, which then schedules the function's execution in parallel on the cluster. These operations, and additional ones such as joins, take RDDs as input and produce new RDDs. RDDs are immutable and their operations are lazy; fault-tolerance is achieved by keeping track of the "lineage" of each RDD (the sequence of operations that produced it) so that it can be reconstructed in the case of data loss. RDDs can contain any type of Python, Java, or Scala objects.

Besides the RDD-oriented functional style of programming, Spark provides two restricted forms of shared variables: broadcast variables reference read-only data that needs to be available on all nodes, while accumulators can be used to program reductions in an imperative style.

A typical example of RDD-centric functional programming is the following Scala program that computes the frequencies of all words occurring in a set of text files and prints the most common ones. Each map, flatMap (a variant of map) and reduceByKey takes an anonymous function that performs a simple operation on a single data item (or a pair of items) and applies its argument to transform an RDD into a new RDD.

1.1.1 Spark SQL

Many data scientists, analysts, and general business intelligence users rely on interactive SQL queries for exploring data. Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

1.2 Spark Streaming

Many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

1.2.1 MLlib Machine Learning Library

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights. Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

1.2.2 GraphX

GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

1.3 Installation

I have made the below setup in the following order of deployment:

1.3.1 Deployment Plan:

Below figure-1 is the deployment plan. Next sections highlight on them:

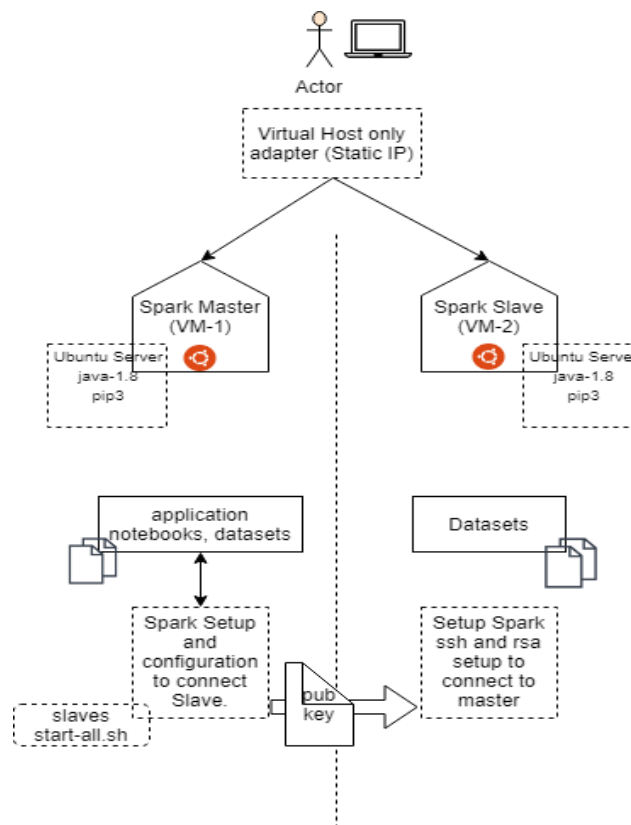


Figure-1

1.3.2 Pre-Installations (Installation of java, pip3):

Installation of java and pip3 has to be done as follows on both the virtual machine installed on Ubuntu server (16.X).

Java Installation:

```
sudo apt-get install default-jdk
```

Pip3 Installation:



At first setup python:

```
sudo apt-get update  
  
sudo apt-get install python  
sudo apt-get install python-pip
```

1.3.3 Installation of Spark in both VMs

Now that the preinstallation are ready, we will have to setup Spark in below fashion

1.3.3.1 Spark Installation:

To install apache spark, follow the below steps:

- Download the latest tar file from here:

(<https://www.apache.org/dyn/closer.lua/spark/spark-2.3.1/spark-2.3.1-bin-hadoop2.7.tgz>)

Version: 2.3.1

- Extract it and move it to /usr/local/spark:
Command to extract: `tar -zxvf spark-2.3.1-bin-hadoop2.7.tgz`
Command to move: `mv spark-2.3.1-bin-hadoop2.7 /usr/local/spark`

Do the above as root user.

1.3.3.2 Making Master and Slave Talk to each other:

Figure-1 of the deployment plan mentioned above gives a high level idea of how the master and slave actually are talking to each other. Below commands enable certain communication via ssh.

➔ If ssh not installed, then install it:

```
sudo apt install -y openssh-server  
  
//Check if is running  
sudo systemctl start ssh
```

➔ Generate rsa keys for both master and slave:

```
ssh-keygen -t rsa
```

//I have kept no password for authentication.

//This will create two files a private file and a public file. Use the public and add it to authorized_key located at ~/.ssh/authorized_keys and add the key.

Repeat this, for both VMs to have a connection.

Screenshot:

```
prakass1@ubuntu-server:~/shared$ ssh-keygen -t rsa  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/prakass1/.ssh/id_rsa):
```



1.3.3.3 Configurations:

- ➔ Editing spark-env.sh at /usr/local/spark/conf/:
- Below are the configuration changes to be made for master (IP address will subject to change):

```
//Changes to spark-env.sh
#Java HOME
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")
SPARK_WORKER_MEMORY=1g
export PYSPARK_PYTHON=python3
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook--ip='192.168.56.103'
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=/usr/local/hadoop/conf
export SPARK_MASTER_HOST=192.168.56.103
export SPARK_WORKER_CORES=1
export SPARK_WORKER_INSTANCES=1
export SPARK_MASTER_PORT=7077
export SPARK_WORKER_MEMORY=4g
export MASTER=spark://${SPARK_MASTER_HOST}:${SPARK_MASTER_PORT}
export SPARK_LOCAL_IP=192.168.56.103
```

Slaves spark-env.sh to be edited at the location /usr/local/spark/conf:

```
#Java HOME
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")
SPARK_WORKER_MEMORY=4g
export PYSPARK_PYTHON=python3
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=/usr/local/hadoop/conf
#SPARK_MASTER_HOST=sparkubuntu
export SPARK_MASTER_HOST=192.168.56.103
export SPARK_WORKER_CORES=1
export SPARK_WORKER_INSTANCES=1
export SPARK_MASTER_PORT=7077
export SPARK_WORKER_MEMORY=4g
export MASTER=spark://${SPARK_MASTER_HOST}:${SPARK_MASTER_PORT}
export SPARK_LOCAL_IP=192.168.56.102
```

- ➔ Editing slaves:
- At the location, /usr/local/spark/conf/ there is a file called slaves.template. Rename that file to slaves:
- Commands:

Below changes should be done:

sudo vi /etc/hosts and add the worker ip address:



Example:

```
root@sparkubuntu:/usr/local/spark/conf# cat /etc/hosts
127.0.0.1        localhost
127.0.1.1        sparkubuntu

# The following lines are desirable for IPv6 capable hosts
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters

#IP
192.168.56.102   sparkworker
```

After above, change the slaves as below example: (Note: the hostname can change according to the setup made)

Command: `vi /usr/local/spark/conf/slaves`

```
# A Spark Worker will be started on each of the machines listed below.
#localhost
sparkworker
root@sparkubuntu:/usr/local/spark/conf#
```

1.3.3.4 Startup and checking Web UI:

To start the master and all workers use this below command:

```
prakass1@sparkubuntu:~/shared$ sudo su -
[sudo] password for prakass1:
root@sparkubuntu:~# start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /usr/local/spark/logs/spark-root-
org.apache.spark.deploy.master.Master-1-sparkubuntu.out

sparkworker: starting org.apache.spark.deploy.worker.Worker, logging to
/usr/local/spark/logs/spark-root-org.apache.spark.deploy.worker.Worker-1-ubuntu-
server.out
```

Below logs should be seen at worker:

```
root@ubuntu-server:~# cd /usr/local/spark/logs/
root@ubuntu-server:/usr/local/spark/logs# tail -50 spark-root-
org.apache.spark.deploy.worker.Worker-1-ubuntu-server.out
```


SPARK WORKER



If the above is seen it means the setup to connect master and slave is successful.

1.3.4 Setup of pyspark and jupyter-notebook, so that the code can be executed from my local machine to VM.

Referring to configurations, pyspark is configured already to talk to jupyter notebook.

1. Install jupyter-notebook:
sudo apt-get install jupyter-notebook
2. Start pyspark as a normal user:
To execute pyspark, either add the pyspark path into the ~/.bashrc or call it from absolute path.

Below screenshot shows bashrc configuration:

```
#####PATH Additions#####
export JAVA_HOME=$(readlink -f /usr/bin/java | sed "s:bin/java::")
export SBT_HOME=/usr/share/sbt-launcher-packaging/bin/sbt-launch.jar
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:$JAVA_HOME/bin
export PATH=$PATH:$SBT_HOME/bin:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Now start pyspark by calling \$pyspark. This should start the jupyter notebook like this:

```
prakassi@sparkubuntu:~$
prakassi@sparkubuntu:~$ pyspark
[1] 00:10:05.422 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter/notebook_cookie_secret
[1] 00:10:06.952 NotebookApp] Serving notebooks from local directory: /home/prakassi
[1] 00:10:06.952 NotebookApp] 0 active kernels
[1] 00:10:06.953 NotebookApp] The Jupyter Notebook is running at:
[1] 00:10:06.954 NotebookApp] http://192.168.56.103:8888/
[1] 00:10:06.954 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 00:10:06.955 NotebookApp] No web browser found: could not locate runnable browser.
```

If you see the image there is a message that **No web browser found**. This is not a problem as the local machine is already in the full network of Virtual machine. Simply can use the url and execute on the browser to have access without any **tunneling**.

2. Examples

The examples are divided into three categories:

1. Apache Spark Exploratory Data Analysis of Imdb Dataset with pyspark
2. Apache Spark Streaming Twitter Hastags and a simple way to build a HashTag count.
3. Apache Graphx exploring the Imdb dataset with Scala

2.1 Apache Spark Exploratory Data Analysis of Imdb Dataset with pyspark

The notebook to refer is Imdb_Analysis_and_Prediction.ipynb.

First step is to run pyspark on master not as root which is already described above.

Below are some important snippets which gives an idea on the working of spark:

The notebook is self-explanatory. At first, importing all the spark libs required.

The first step is to call the SparkConf so that the context is ready

```
conf=SparkConf().setAppName("Imdb data Analysis")
spark = SparkSession.builder.appName('Imdb').getOrCreate()
```


with a SparkSession:

To load the dataset into the memory we make use of textFile() provided by apache spark.

```
titles = sc.textFile('/home/prakass1/imdb_dataset/data_title.tsv').cache()
rating = sc.textFile('/home/prakass1/imdb_dataset/data_rating.tsv').cache()
```

Note: the dataset can be cached so the dataset can be loaded faster.

Screenshot of the worker:



← → ↻ ⓘ 192.168.56.102:8081 ☆ ⓘ ⚙ ⚠ ⚡ ⋮

 2.3.0 **Spark Worker at 192.168.56.102:33255**

ID: worker-20180626235419-192.168.56.102-33255
Master URL: spark://192.168.56.103:7077
Cores: 1 (1 Used)
Memory: 4.0 GB (1024.0 MB Used)
[Back to Master](#)

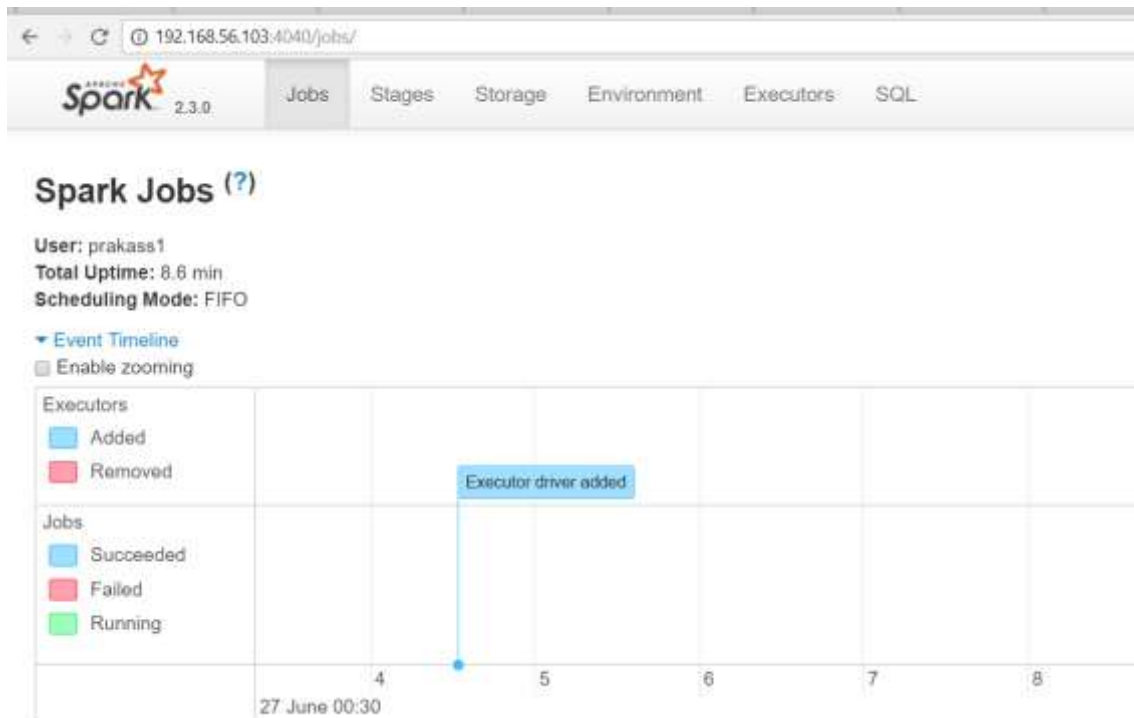
Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	RUNNING	1024.0 MB	ID: app-20180627003004-0000 Name: PySparkShell User: prakass1	stdout stderr



Now you can see that a job is being executed at the worker from the master as part of our cluster setup.

Worker jobs can easily be monitored as below:



Data Preprocessing is handled in the code only to remove the first row which is header.

```
def filter_header(data):  
    header = data.first()  
    data = data.filter(lambda line: line != header)  
    return data
```

There are other methods like parsing the data which is a tab separated file. More information is provided with the notebook.

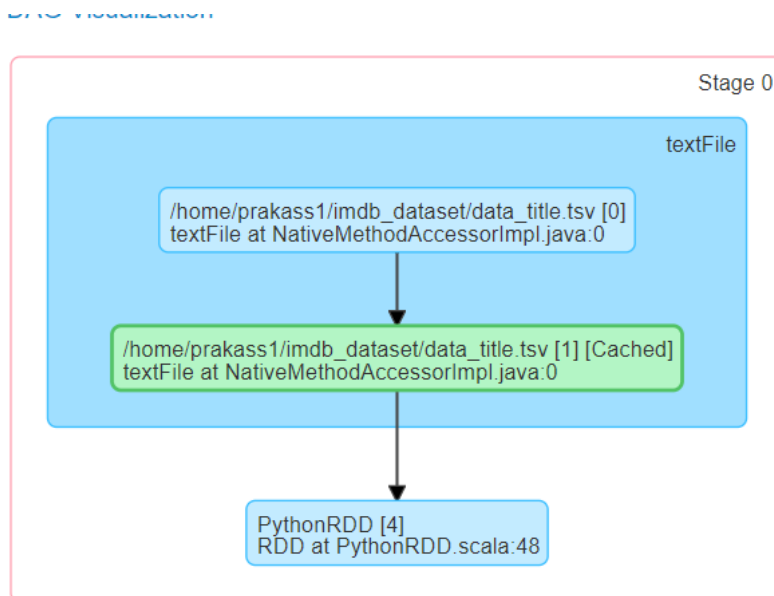
Below is a details for map() in spark:

```
def perform_title_mapping(lines):  
    rows = lines.map(parse_data)  
    return rows
```

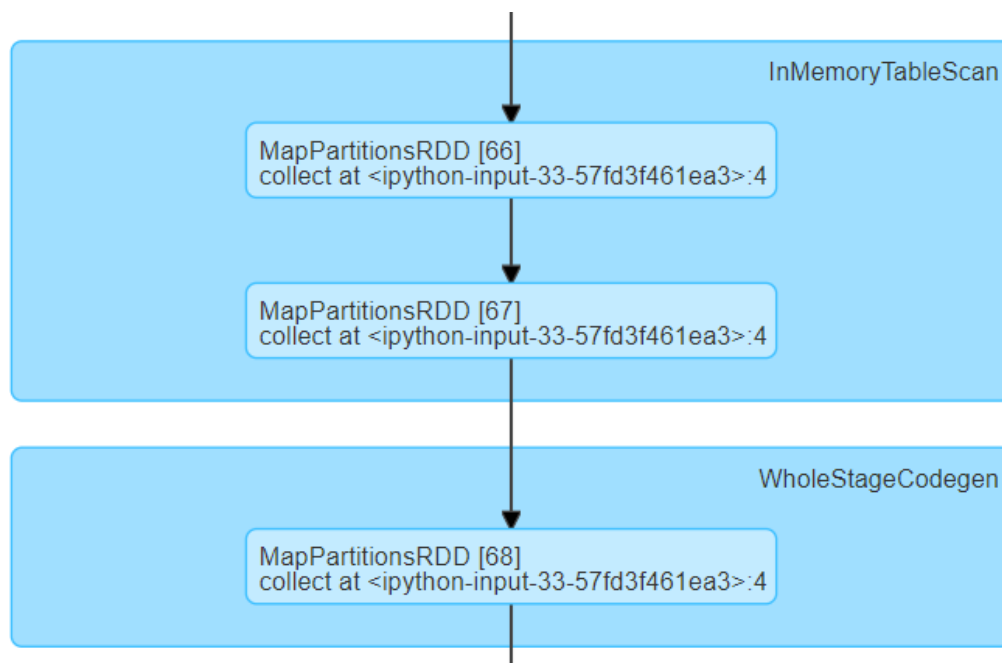


What happens when `map()` is run at the worker?

Apache Spark works using DAG (Direct Acyclic Graph), more understanding on this is given by the below image:



In Memory staging from the DAG visualization is:



The real power to perform in-memory computing of apache spark can be seen from the below image:



Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
1	/home/prakass1/indb_dataset/data_title.tsv	Memory Serialized 1x Replicated	1	8%	15.8 MB	0.0 B
3	/home/prakass1/indb_dataset/data_rating.tsv	Memory Serialized 1x Replicated	1	50%	3.4 MB	0.0 B

Some finding of the dataset are:

Code snippet taking top 10 movies:

```
#Collect all data for top ten movies
titles=[]
for movies in topTenMovies:
    titles.append(cachedTitles.filter((cachedTitles.movieId == movies[0])).collect())
titles
```

Some Result of it are shown in graph using matplotlib:

Note: These are additional installations.

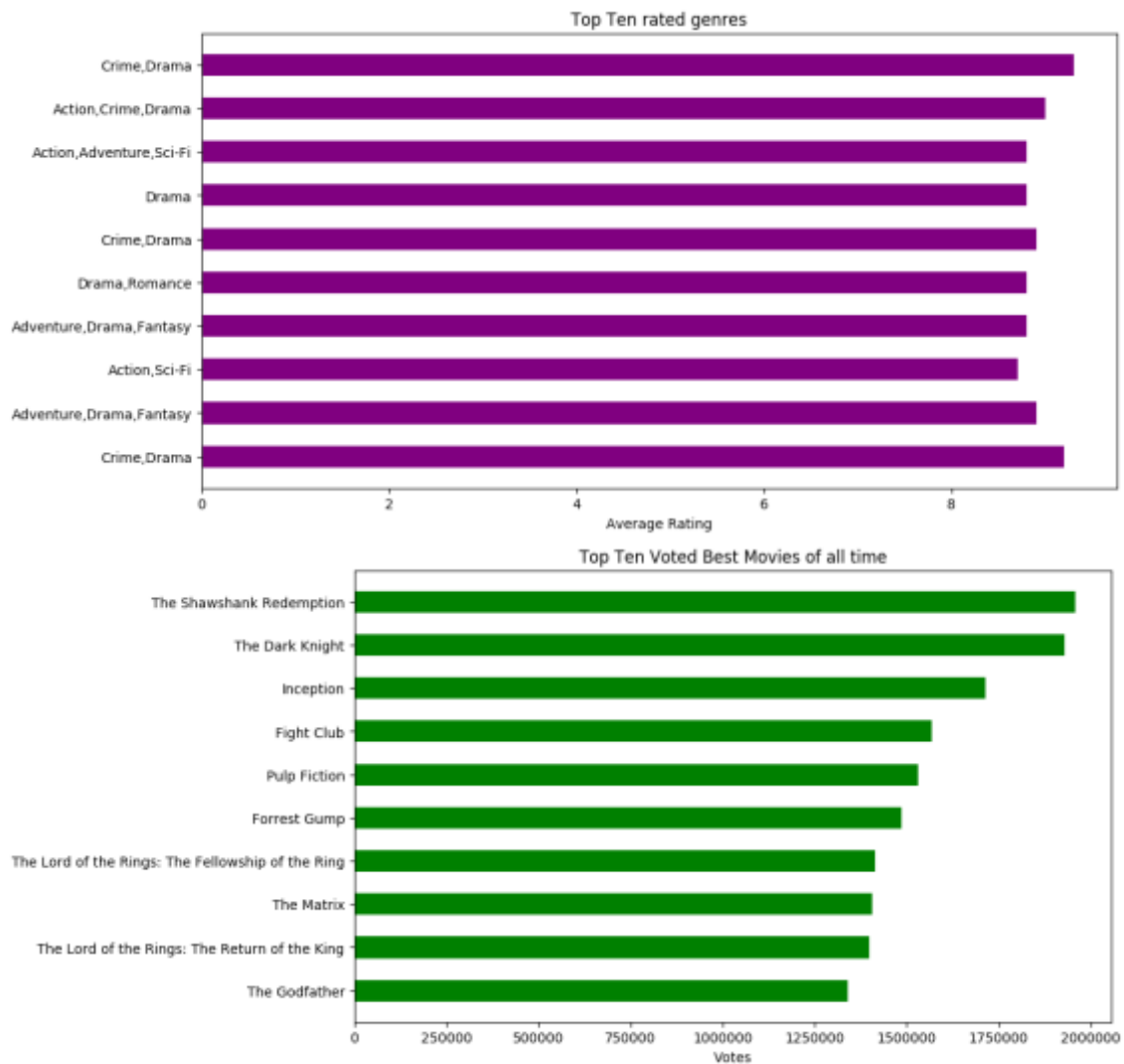
Installation:

pip3 install matplotlib

pip3 install seaborn

pip3 install numpy

pip3 install scipy

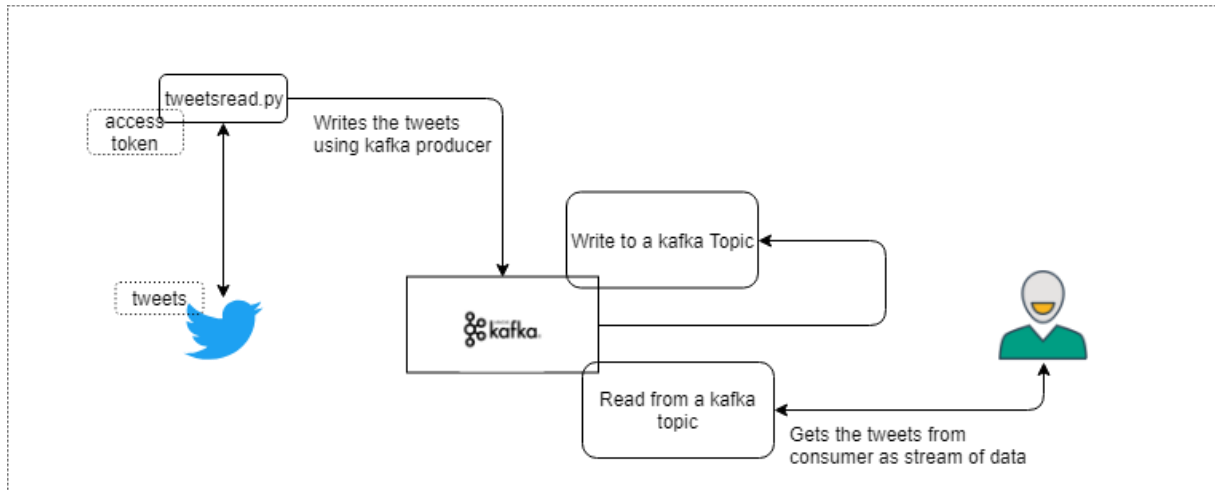


A small use case is presented to see if by seeing run time can we predict whether it is a movie or a short film.

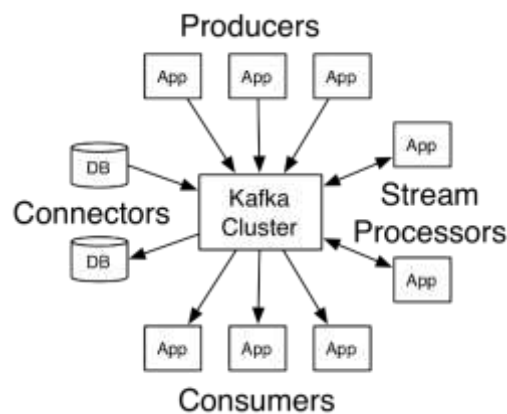
All details for this present in the notebook.

2.2 Apache Spark Streaming Twitter Hashtags – a simple way to build hash tag counter on a topic

2.2.1 Architecture



➔ About Kafka server:



Source: <https://www.kafka.apache.org/intro>

- Apache Kafka is a publish-subscribe messaging system. By saying that, we need to describe a messaging system. A messaging system lets you send messages between processes, applications, and servers.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.
- Kafka is very fault tolerant meaning if one node goes down there is another node which could still send/receive messages from/to the application.
- Kafka can be used for stream processing, making store in storage etc..

Below are important installation steps to be followed:

1. Downloading and Extracting kafka:



- Download the latest version of kafka (https://www.apache.org/dyn/closer.cgi?path=/kafka/1.1.0/kafka_2.11-1.1.0.tgz) .
 - Extract it using tar command: `tar -zxvf kafka_2.11-1.1.0.tgz`
 - Move it: `mv kafka_2.11-1.1.0 /usr/local/kafka`
 - As prerequisites, please install tweepy and kafka as follows:
`pip3 install tweepy`
`pip3 install kafka-python`
 - Then execute step 2.
2. Starting a kafka server and creating a topic: The kafka installation setup is fully automated meaning I have developed a script which will perform the startup of kafka without needing to type the commands manually.
- 3.
- Script Name: `create-kafka-server.sh`
Starting the script:
`./<<script name.sh>> <<topic name>>`
Example:
`./create-start-kafka.sh tweets`
4. Executing tweetsread.py:
A python script is written which gets the tweets based on the access token of the tweet and adds that into the kafka topic.

Script Name: `tweetsread.py`
Execution: `python3 tweetsread.py`

Tweets should come into kafka as below:

```
prakash@ubuntu-server1:~$ kafka-console-consumer.sh --bootstrap-server 192.168.36.102:9092 --topic tweets --from-beginning
RT @VGLBapp: ? Since hangisi daha iyi futbolcu?

@ Cristiano Ronaldo = RT
* Lionel Messi = Beğen https://t.co/3BfcwLEpqr
RT @Scotfield340: En vrai France ou Messi je supporte personne je peux pas, la France j'ai jamais connu un vrai bonheur, 2006 et 2010 m'a tué...
RT @IndyFootball: Most take-ones completed in #WorldCup history:

** Lionel Messi (107)
** Diego Maradona (105)

Argentinian legends, https://t.co/3BfcwLEpqr
RT @CafeDeRick: Cuando veais a Maradona en el estadio pensab que ese fue el seleccionador que le dieron a Messi en un Mundial con 23 años
RT @Guldenhorst1888: God bless you a million times
I just tweeted my mind https://t.co/4YRbifn8K0
RT @ _Karma_: the difference is one was an attempt on goal indeliberately blocked , the other was a failed attempt on clearance, and left a...
We that is here begging man utd to sell Polish Rojo@ https://t.co/3BfcwLEpqr
```

2.2.2 Execution of notebook to get the hashtag and their count of twitter through kafka server

2.2.2.1 Starting and initialize of pyspark:

Since kafka jars are not in spark a little modification to pyspark command is done:



pyspark --packages org.apache.spark:spark-streaming-kafka-0-8-assembly_2.11:2.3.1

Code Snippets:

➔ Initialization and start of stream:

```
#sc = SparkContext()
ssc = StreamingContext(sc, 10)
#sqlContext = SQLContext(sc)
```

➔ Connect to the kafka as streaming: Here "tweets" is the kafka topic.

```
kvs = KafkaUtils.createDirectStream(ssc, ["tweets"], {"metadata.broker.list": "sparkworker:9092"})
kvs.window(20)
```

➔ Use the above to perform map reduce and filtering to get the hashtag counts as :

```
ssc.awaitTermination()
```

```
Time: 2018-06-27 11:37:20
```

```
Tweet(tag='#TeenChoice', count=1)
Tweet(tag='#WorldCup\nMessi', count=1)
Tweet(tag='#ChoiceFandom', count=1)
Tweet(tag='#Directioners', count=1)
Tweet(tag='#No.', count=1)
```

```
Time: 2018-06-27 11:37:30
```

```
Tweet(tag='#messi', count=1)
Tweet(tag='#wk2018', count=1)
Tweet(tag='#StopRacism...', count=1)
Tweet(tag='#Messi31', count=1)
Tweet(tag='#penalty', count=1)
Tweet(tag='#ISL:', count=1)
Tweet(tag='#CuneytCakir', count=1)
```

2.3 Exploratory data analysis of Graphx:

This section is similar to 3.1, but here the library is Graphx to process the data like a graph.

For graphx python support is not there, hence I code is written in scala. Already spark comes with scala hence, there is no need to change anything expect to add kernel for scala in jupyter-notebook.

pip3 install https://dist.apache.org/repos/dist/dev/incubator/toree/0.2.0-incubating-rc3/toree-pip/toree-0.2.0.tar.gz
pip3 install

pip3

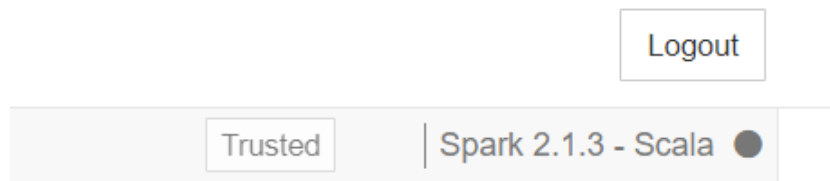
jupyter toree install --spark_home="\$\${SPARK_HOME}" --kernel_name="Spark 2.1.3" --interpreters="Scala"

Note: After successful installation jupyter-notebook should have scala as a kernel.



Execute: prakass1@sparkubuntu:~\$ jupyter-notebook --ip=192.168.56.103

Screenshot:



Code Snippets:

- At first, the classes need to be defined.
Example:

```
//Lets define movie class//  
case class Movie(  
  tconst:String,  
  titleType:String,  
  primaryTitle:String,  
  originalTitle:String,  
  isAdult:String,  
  startYear:String,  
  endYear:String,  
  runtimeMinutes:String,  
  genre:String  
)  
  
defined class Movie
```

- The Data is parsed and header filter is done. Also, graph data structure does not accept String hence the values must be Long. So as part of preprocessing we remove alphabet characters from ids.
- Perform required mapping as below:

```
val titles = moviesRDD.map(movies => (movies.tconst.toLong,movies.primaryTitle)).distinct()
```

- Edge is defined as:

```
val edges = votes.map {  
  case ((tconst, averageRating), noOfVotes) => Edge(tconst.toLong,averageRating.toLong,noOfVotes.toLong) }
```

- Graph is defined as:

```
val graph = Graph(titles, edges)  
  
graph.vertices.take(2)
```

```
graph = org.apache.spark.graphx.impl.GraphImpl@2a9d6697  
[(1115179,Introduction), (354796,Parade de quarte)]
```

- Some outputs from the graph are:
1. Basic Stats like the number of movies to that of ratings:



Number of movies in the dataset

```
val noOfMovies = graph.numVertices  
noOfMovies = 5040244  
5040244
```

Number of votes

```
val values = graph.numEdges  
values = 838182  
838182
```

2. Get the highest voted movie:

Get the Highest Voted Movie

```
val highVotedMovies = graph.edges.filter { case ( Edge(tconst, averageRating, noOfVotes) ) => noOfVotes >= 1500000 }.take(10)  
highVotedMovies = Array(Edge(111161,9,1957480), Edge(468569,9,1929695), Edge(110912,8,1530948), Edge(137523,8,1569481), Edge(175666,8,1714048))  
[Edge(111161,9,1957480), Edge(468569,9,1929695), Edge(110912,8,1530948), Edge(137523,8,1569481), Edge(1375666,8,1714048)]
```

3. References

- [1] www.wikipedia.org
- [2] www.spark.apache.org
- [3] <https://docs.databricks.com/spark>
- [4] <https://docs.oracle.com/en/cloud/paas/database-dbaas-cloud/csdbi/generate-ssh-key-pair.html#GUID-4285B8CF-A228-4B89-9552-FE6446B5A673>
- [5] <http://jupyter.readthedocs.io>
- [6] http://docs.tweepy.org/en/v3.5.0/getting_started.html
- [7] <https://kafka.apache.org/quickstart>
- [8] <https://matplotlib.org/gallery>
- [9] <https://databricks.com/blog>