# *Market prediction with Big Data Tools and Technologies*

Team 11: Robin Liu, Frank Li, Daniel Rodriguez

## Introduction:

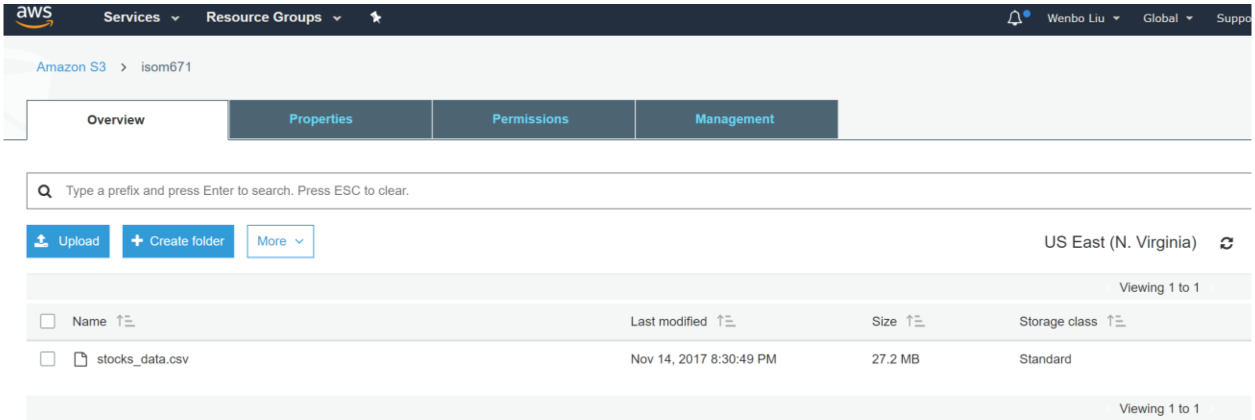
The Big Data Ecosystem, or Big Data Zoo, contains a plethora of tools to collect, store, process, and analyze large amounts of data in a relatively short amount of time. For our Big Data project we wanted to explore a subset of this ecosystem and apply it to real world data in a way that would allow us to utilize a combination of big data tools.

After much deliberation, we ultimately decided on using stock market data as our example data set. Due to the small number of columns and large amount of tuples, this data set was ideal for dipping our tows into new big data tools. Our initial goal was to simply process the dataset using a tool like Hive and compare it to SQL, but we grew more ambitious. Instead, we decided to use R to pull the data from Yahoo's API, use Amazon's S3 service to store the dataset, use Hive to process the data into a table, and load that table onto Spark with Zeppelin to prime it for any analysis one might want to do.

## Methods:

1:Pull data Through R package

2:Log on to AWS and upload the file to a S3 bucket




Using putty SSH to connect to the master node and execute

aws s3 cp s3://isom671/stocks_data.csv ./

Now that we have our data in the terminal, let's create a hql script to create a database called 'stock_info' and a table called 'stock'. At this point, a lot of skills and experiences we learned in lectures and labs become very handy:

```
[hadoop@ip-172-31-16-161 ~]$ ll
total 27872
-rw-rw-r-- 1 hadoop hadoop      290 Nov 14 03:44 create_table.hql
-rw-rw-r-- 1 hadoop hadoop 28535496 Nov 14 03:21 stocks_data.csv
```

```
[hadoop@ip-172-31-16-161 ~]$ cat create_table.hql
DROP DATABASE IF EXISTS stock_info CASCADE;
CREATE DATABASE stock_info;
USE stock_info;
DROP TABLE IF EXISTS stock;
CREATE EXTERNAL TABLE stock (
open_date STRING,
open DOUBLE,
high DOUBLE,
low DOUBLE,
close DOUBLE,
volume INT,
name STRING)
ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','
```

Using hive -f to run a hiveql script

```
[hadoop@ip-172-31-16-161 ~]$ hive -f create_table.hql

Logging initialized using configuration in file:/etc/hi
properties Async: false
OK
Time taken: 1.414 seconds
OK
Time taken: 0.041 seconds
OK
Time taken: 0.027 seconds
OK
Time taken: 0.109 seconds
OK
Time taken: 0.535 seconds
```

And let's have a look at our newly created databases

```
hive> show databases;
OK
default
stock_info
Time taken: 0.014 seconds, Fetched: 2 row(s)
hive> USE stock_info;
OK
Time taken: 0.015 seconds
hive> describe stock;
OK
open_date               string
open                    double
high                    double
low                     double
close                   double
volume                  int
name                    string
Time taken: 0.043 seconds, Fetched: 7 row(s)
```

After we created this database, let's load this database. First of all use

```
hadoop fs -mkdir final_project
```

to create a folder to hold our stock data. Then:

```
[hadoop@ip-172-31-16-161 ~]$ hadoop fs -put stocks_data.csv final_project
[hadoop@ip-172-31-16-161 ~]$ hadoop fs -ls final_project
Found 1 items
-rw-r--r--   1 hadoop hadoop   28535496 2017-11-15 01:42 final_project/stocks_da
ta.csv
```

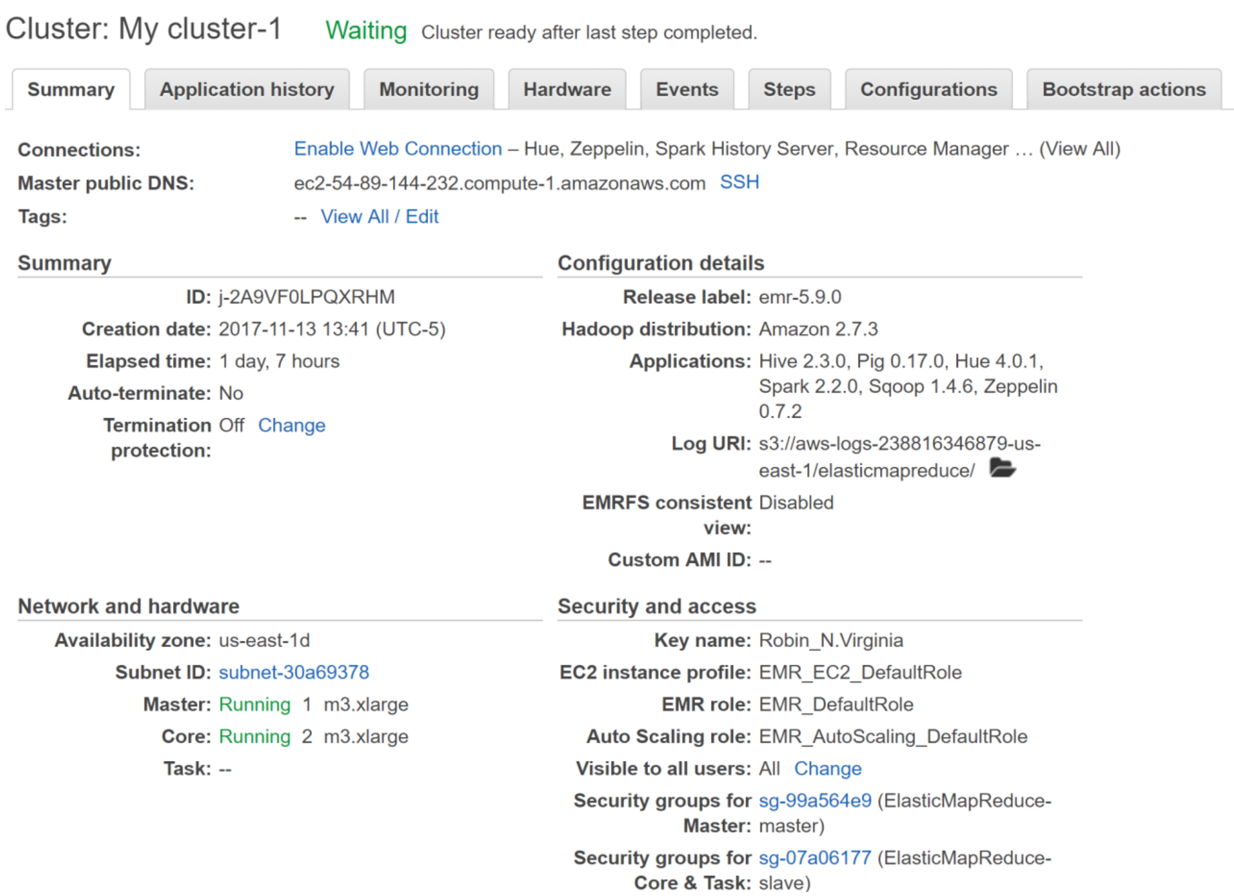
Call hive again, this time tell it to load our database.

```
hive> USE stock_info;
OK
Time taken: 0.01 seconds
hive> LOAD DATA INPATH 'final_project/stocks_data.csv' INTO TABLE stock;
Loading data to table stock_info.stock
OK
Time taken: 0.732 seconds
```

Try to see what's in our hive system:

```
hive> SELECT * FROM stock_info.stock LIMIT 5;
OK
8/13/2012       92.29   92.59   91.74   92.4    2075391 MMM
8/14/2012       92.36   92.5    92.01   92.3    1843476 MMM
8/15/2012       92.0    92.74   91.94   92.54   1983395 MMM
8/16/2012       92.75   93.87   92.21   93.74   3395145 MMM
8/17/2012       93.93   94.3    93.59   94.24   3069513 MMM
Time taken: 1.59 seconds, Fetched: 5 row(s)
```

Seems like we've successfully loaded our database. Let's move on to setting up zeppelin and spark.




AS we see, Zeppelin has been added when we created the cluster. We just need to access it's interface. There are two ways to do this: establishing an SSH tunnel with the master node using either local port forwarding or dynamic port forwarding. If we choose to do the latter option, we would also need to Configure a proxy management Configure a proxy management too tool. Therefore, let's use a local port forwarding method.

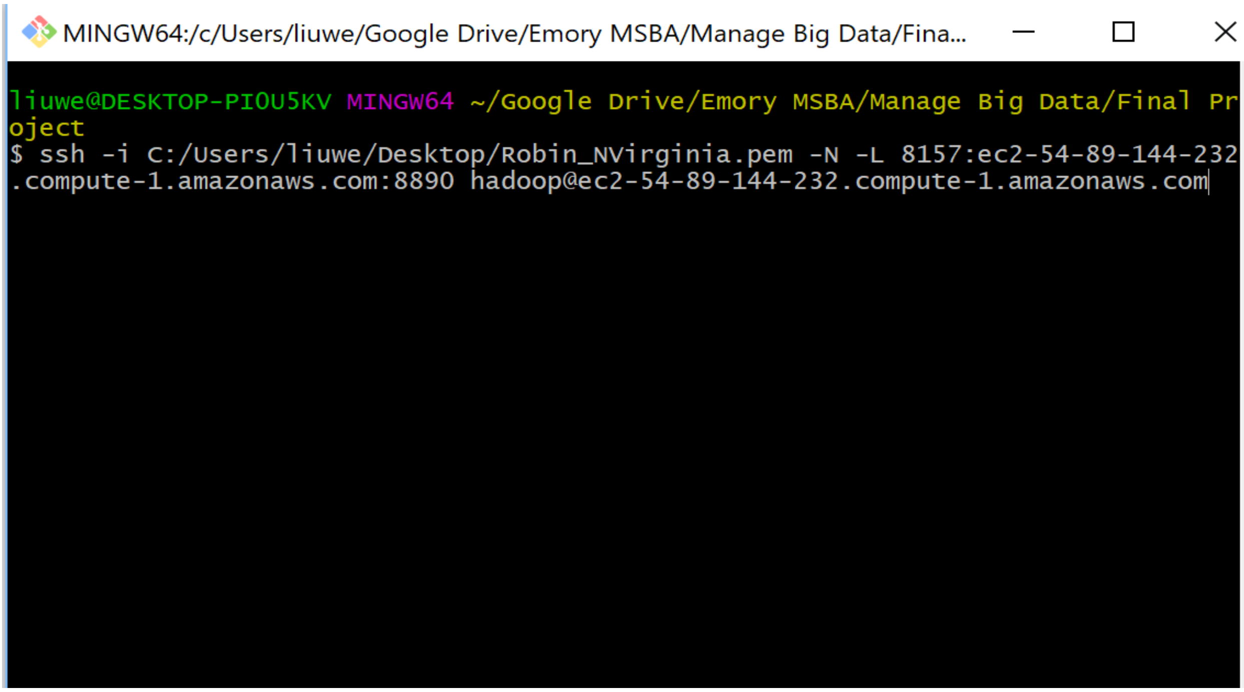
**In bash, write down there commands:**

ssh -i C:/Users/liuwe/Desktop/Robin_NVirginia.pem -N -L 8157:ec2-54-89-144-232.compute-1.amazonaws.com:8890 hadoop@ec2-54-89-144-232.compute-1.amazonaws.com

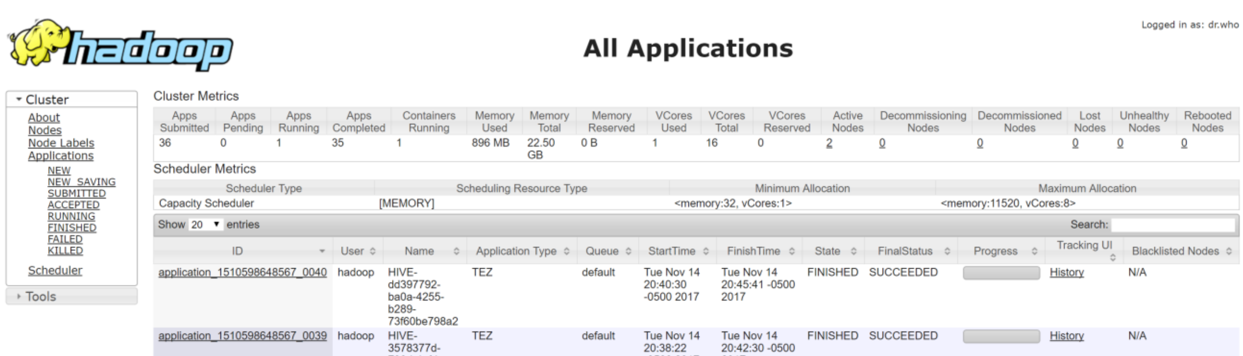
where:

- -i tells it to read a file as public key
- -N tells it not to do not execute anything
- -L Forwards local port to remote address

what it does is that it establishes a connection between a local port(in our case local port 8157 ) and a port on our master node(in our case 8890, on which Zeppelin interface can be accessed.)



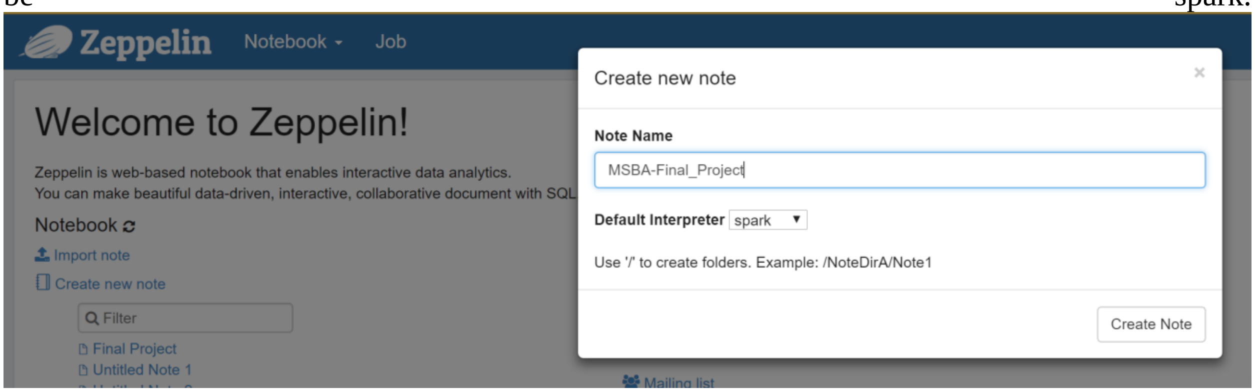

We choose to access port 8890 because that's where our Zeppelin interface resides; there are different ports for other applications if you included them when you set up your cluster. For example, changing '8157' to '8826'(which we randomly selected and is not used by any other application) and '8890' to '8088' will take you to YARN interface.



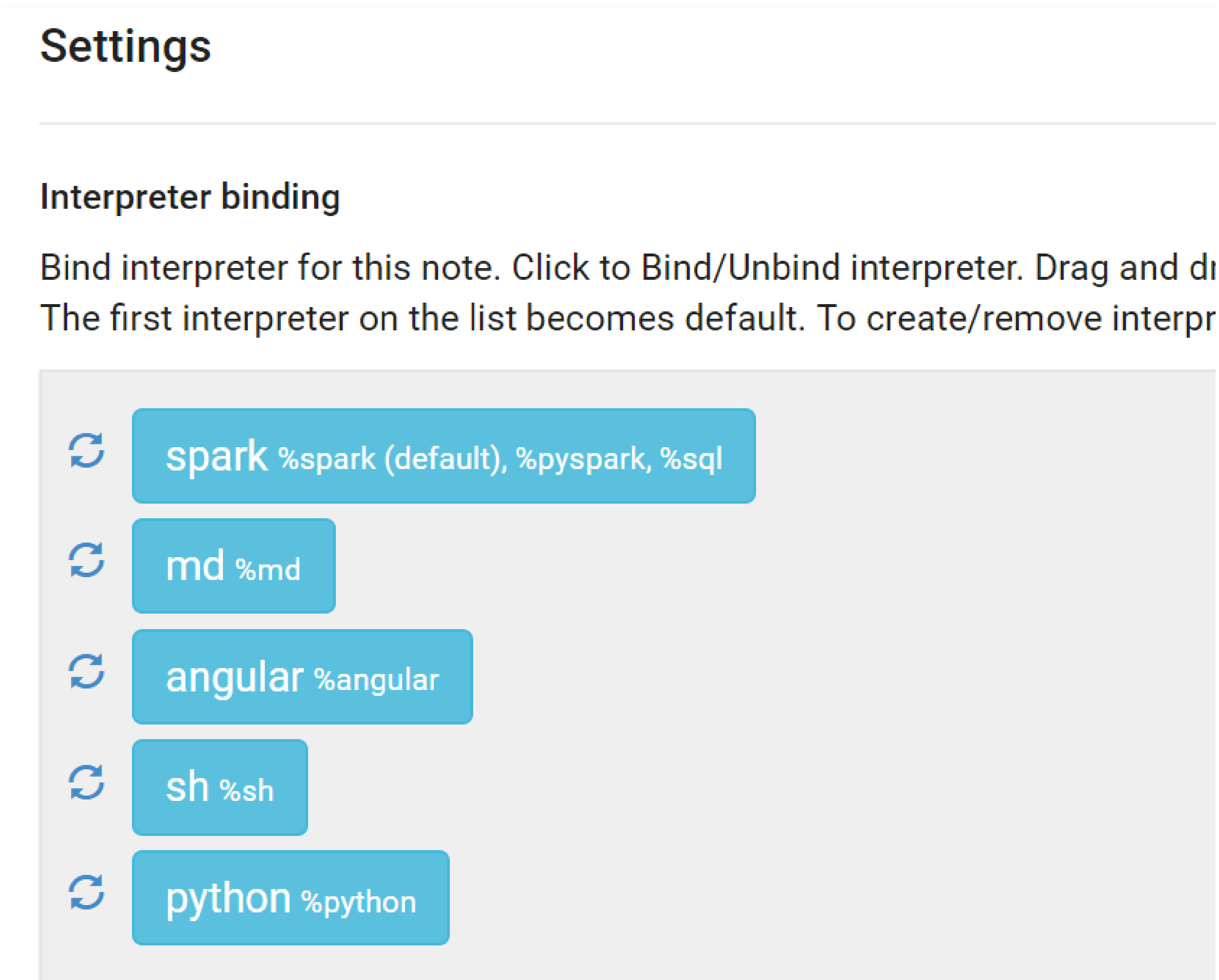
A complete list of application and their ports can be found on aws website.

| Name of interface | URI |
|---|---|
| YARN ResourceManager | http://*master-public-dns-name*:8088/ |
| YARN NodeManager | http://*slave-public-dns-name*:8042/ |
| Hadoop HDFS NameNode | http://*master-public-dns-name*:50070/ |
| Hadoop HDFS DataNode | http://*slave-public-dns-name*:50075/ |
| Spark HistoryServer | http://*master-public-dns-name*:18080/ |
| Zeppelin | http://*master-public-dns-name*:8890/ |
| Hue | http://*master-public-dns-name*:8888/ |
| Ganglia | http://*master-public-dns-name*/ganglia/ |
| HBase UI | http://*master-public-dns-name*:16010/ |

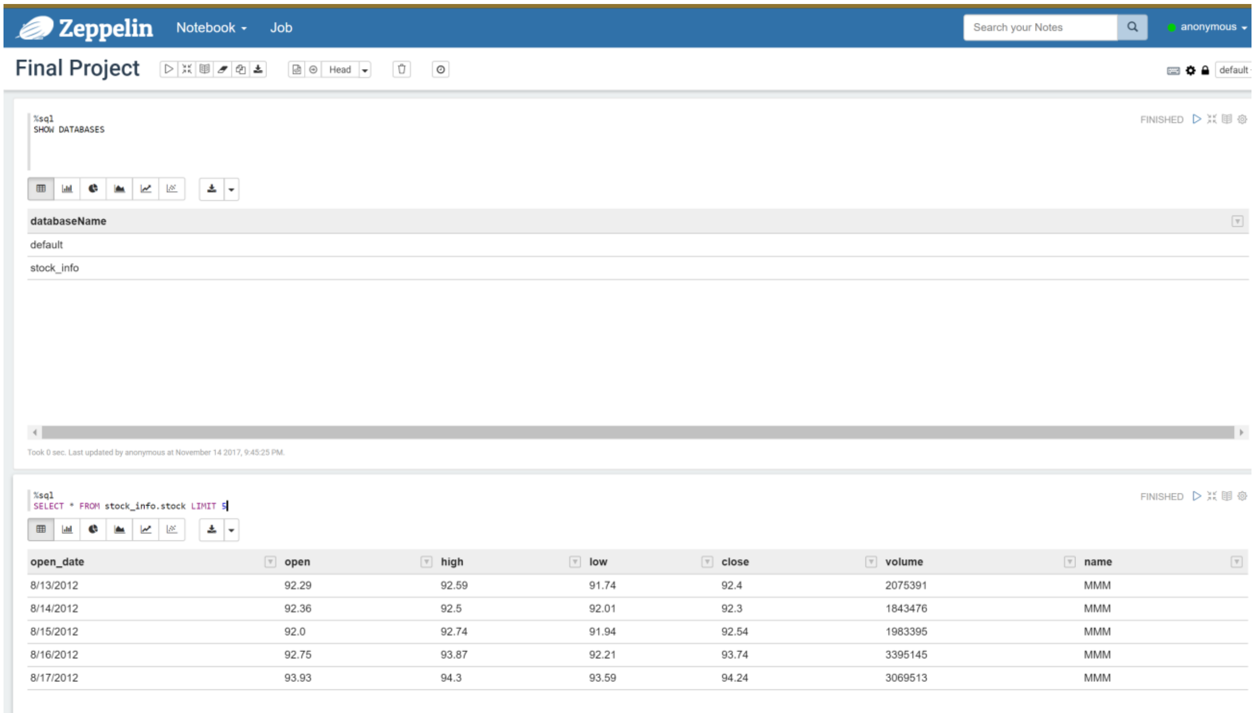
Now that we have the access to Zeppelin, let's create a notebook and choose default interpreter to be spark.




If we want to know what interpreters are being provided by zeppelin:




As we can see, spark provides a sql shell as well as a python interface, pyspark, which provides python api. One thing EMR does very nicely is that it integrated Hive with Spark. Therefore, when you query in Spark's sql shell, it creates a sqlContext object as well as a HiveContext object which queries Hive for you. This is very important because right now when you want to run your Spark application and do analysis on files, you can translate your text file into Hive (or when you have a database, you can use tools such as sqoop to transform data into hive) and directly run query through Spark SQL as you go. This adds great flexibility into project development, and it reduces cost: all the data can be stored in S3, which provides cheap storage option. Whenever you need to run analysis on the data, you can simply acquire the portion you need and put it into Hive.




| databaseName |
|---|
| default |
| stock_info |



| open_date | open | high | low | close | volume | name |
|---|---|---|---|---|---|---|
| 8/13/2012 | 92.29 | 92.59 | 91.74 | 92.4 | 2075391 | MMM |
| 8/14/2012 | 92.36 | 92.5 | 92.01 | 92.3 | 1843476 | MMM |
| 8/15/2012 | 92.0 | 92.74 | 91.94 | 92.54 | 1983395 | MMM |
| 8/16/2012 | 92.75 | 93.87 | 92.21 | 93.74 | 3395145 | MMM |
| 8/17/2012 | 93.93 | 94.3 | 93.59 | 94.24 | 3069513 | MMM |

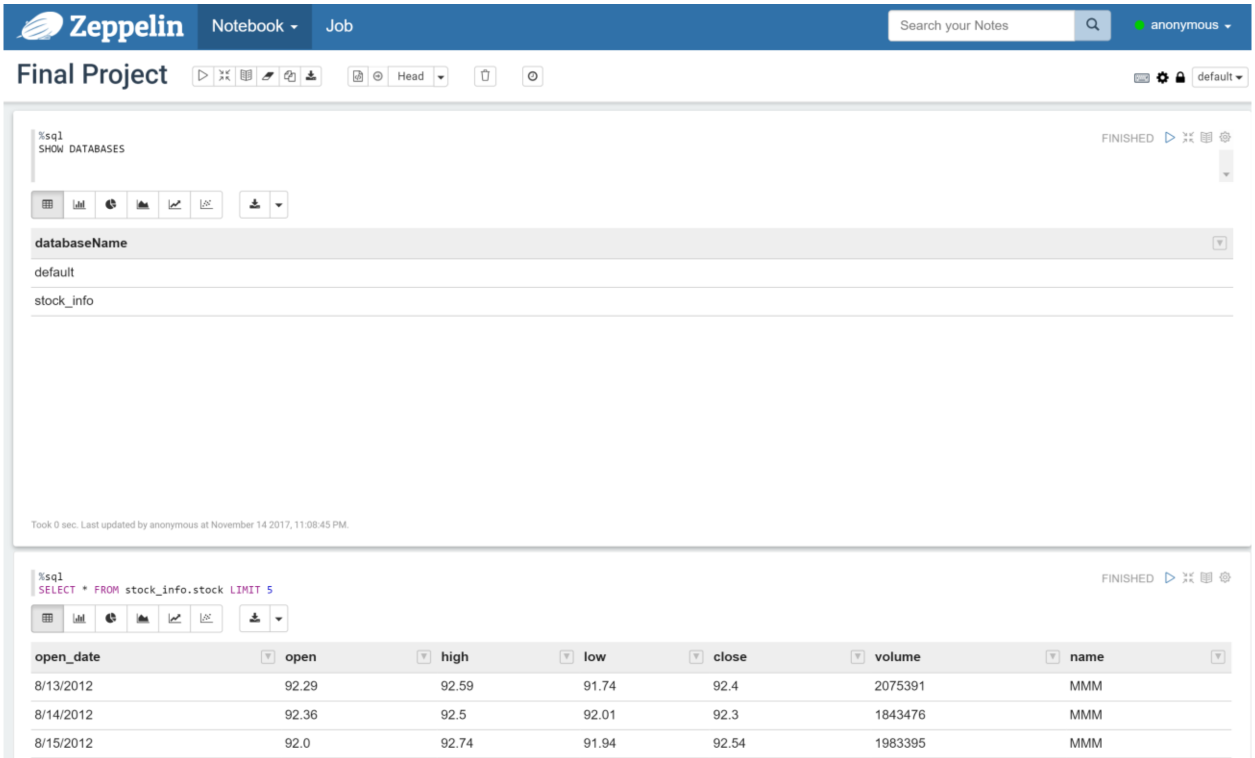
As we see, when we run query in Zeppelin through Spark SQL, our data stored in Hive showed up.

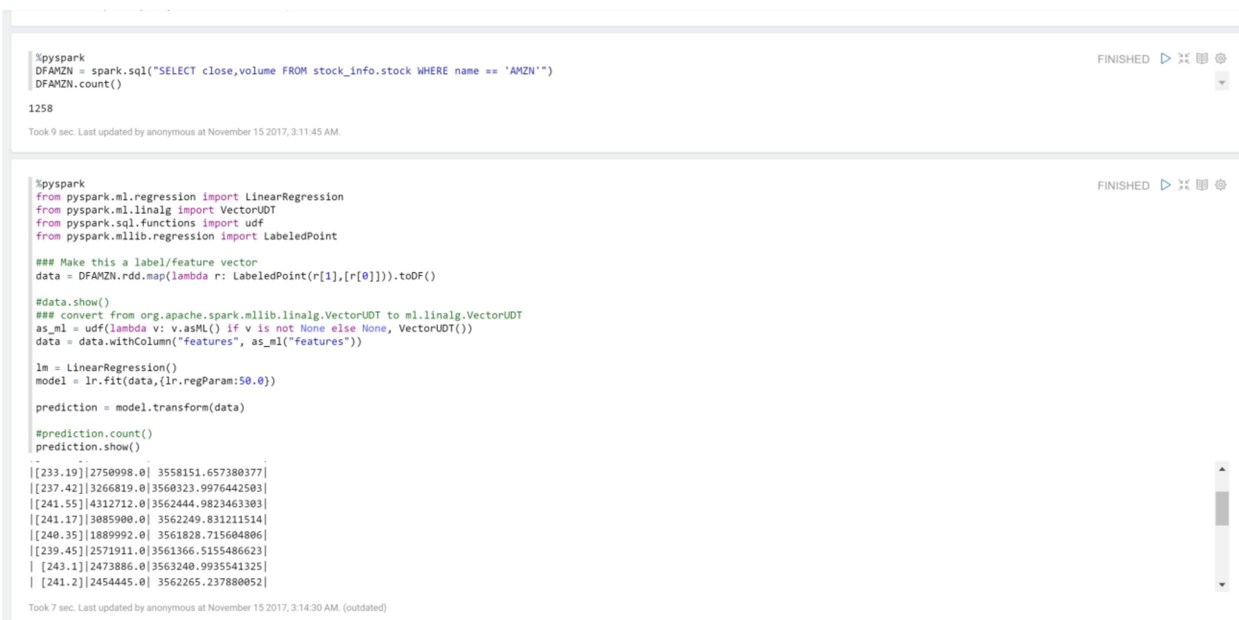
And from this Spark History Server we can see that every time we run Zeppelin, it creates a new SparkContext object with a different AppID.


| App ID | App Name | Started | Completed | Duration | Spark User | Last Updated | Event Log |
|---|---|---|---|---|---|---|---|
| application_1510598648567_0043 | Zeppelin | 2017-11-15 04:07:17 | 2017-11-15 04:07:44 | 27 s | zeppelin | 2017-11-15 04:07:44 | Download |
| application_1510598648567_0042 | Zeppelin | 2017-11-15 04:04:34 | 2017-11-15 04:06:25 | 1.8 min | zeppelin | 2017-11-15 04:06:25 | Download |
| application_1510598648567_0041 | Zeppelin | 2017-11-15 04:03:55 | 2017-11-15 04:04:18 | 24 s | zeppelin | 2017-11-15 04:04:18 | Download |
| application_1510598648567_0019 | Zeppelin | 2017-11-13 22:57:54 | 2017-11-15 03:52:56 | 28.9 h | zeppelin | 2017-11-15 03:52:57 | Download |



And our analysis:




Nowadays, Financial market depends on the ability to process huge volume of data and unearth information from it. Despite we wanted to implement investment strategies mentioned in our proposal, we really wanted to play around with the machine learning library that pyspark provides.

Therefore, we ran a little demo: using volume to predict closing price. Of course, what we did today is quite simple, and this might not be the best predictive model in the financial industry. However, the big data technology set up for more challenging problems could be similar, and we can definitely apply what we learned from this project to run more complex algorithms and analysis in the future.

We also did our original goal: to test our investment strategy. The code below will be on our presentation tomorrow and meant to be flawed: it will have had ran for several hours. The reason it will be very slow is that it is querying Hive many times, and querying in Hive is very time-consuming since it's based on HDFS. The correct way to do the exact same task would query Hive only once and do the analysis in-memory, something spark is excel at, and it would only take seconds. Eventually, out strategy beat the average S&P market return by 2%, though this doesn't take into account any extra costs such as commission fees.

```
%pyspark
change_date = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))
    new_date,open,name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))='1')")
change_date.show()
```

FINISHED

```
+--------+------+----+
|new_date|  open|name|
+--------+------+----+
|   24154|  92.9| MMM|
|   24155| 87.94| MMM|
|   24158|101.44| MMM|
|   24159|103.39| MMM|
|   24160| 106.0| MMM|
|   24161|104.78| MMM|
|   24163|108.37| MMM|
|   24164|118.38| MMM|
|   24166|119.69| MMM|
|   24167|126.76| MMM|
|   24172|135.88| MMM|
|   24173| 139.5| MMM|
|   24175|143.41| MMM|
|   24176|140.15| MMM|
```

Took 0 sec. Last updated by anonymous at November 15 2017, 6:05:23 AM.

RUNNING 0%

```
%pyspark
best_overall = 0.0
worst_overall = 0.0
for i in range(1,len(value)) :
    best = np.zeros(10)
    worst = np.ones(10)
    for c in set(cp):
        current = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))) AS
            new_date,open,name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))='1') where new_date= "+str(value[i])+ " AND name = '"+c+"'")
        past = spark.sql("SELECT * FROM (SELECT 12*INT(YEAR(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd')))+INT(MONTH(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))) AS
            new_date,open,name FROM stock_info.stock WHERE DAY(from_unixtime(unix_timestamp(open_date,'MM/dd/yyyy'),'yyyy-MM-dd'))='1') where new_date= "+str(value[i-1])+ " AND name = '"+c+"'")
        current_open = sc.parallelize(current.select("open").collect()).map(lambda d: d[0]).collect()
        past_open = sc.parallelize(past.select("open").collect()).map(lambda d: d[0]).collect()
        if(len(current_open)==0 or len(past_open)==0):
            pass
        else :
            current_open = float(current_open[0])
            past_open = float(past_open[0])
            ratio = (current_open-past_open)/past_open
            best_min_index = np.argmin(best)
            worst_max_index = np.argmax(worst)
            if(best[best_min_index]<ratio) : best[best_min_index]=ratio
            elif(worst[worst_max_index]>ratio) : worst[worst_max_index]=ratio
    ave_best = np.mean(best)
    ave_worst = np.mean(worst)
    best_overall = (best_overall+ave_best)/2
    worst_overall=(worst_overall+ave_worst)/2
```

Speaking of Future:

APACHE Spark™ Streaming

AWS Data Pipeline

With these two technologies, we can automate most of our work: Using spark stream real-time market data through API and run analysis, and dump everything to S3 bucket once we finish with our analysis.

Last but not least, THANK YOU so much for an amazing class, we enjoyed your class, we learned a lot of exciting things, and we are looking forward to seeing you next semester's machine learning II!

# THE END