Deriving Stock Market Insights Through a Yahoo Finance ETL Pipeline

Ryerson University

DS8003 - Management of Big Data and Tools

Dr. Kucukates and Dr. Padmanabhan

Adam Azoulay, Hee Kyoung Nam, Qinyun Yu

December 10th, 2021

Problem Definition

The usefulness of stock market data cannot be understated. It allows for analysis of investment distribution, future market trend prediction, and correlation discovery. Stock market data is available from a variety of sources, however it tends to either be expensive, old, or short term data. We aim to solve these problems by developing our own source for data and analysis that is up-to-date and reliable, as well as cost effective.

Yahoo Finance [1] provides a limited free API to access this data that we can make use of to build our data warehouse. There are many useful endpoints we can extract data from and we attempt to make use of as many as possible in this project (since we would like to collect too much data rather than too little, in case we would like to run different analyses in the future).

Data Description

Attributes Description

Since we want to collect as much data as possible we extract data from many endpoints of the Yahoo Finance API:

- Chart data to track price movements (/v8/finance/chart)
- Options information (/v7/finance/options)
- Recent news (/v1/finance/search)
- Sustainability report (scraped from yahoo sustainability page)
- Recommendation information (/v6/finance/recommendationsbysymbol)

To each extraction we apply a date column to track when the data was extracted (in the case of news), when the price was observed (chart), or the expiration date (options). Below is a table with the attributes from the chart endpoint as an example. The remaining data source attributes can be found in the *hive_commands* file in the project Git repo [2]. Note that since the data is recorded over an interval, we must remain consistent for each extraction. Typically we choose this to be an hour long due to rate limitations from the API, but in the future this could be reduced.

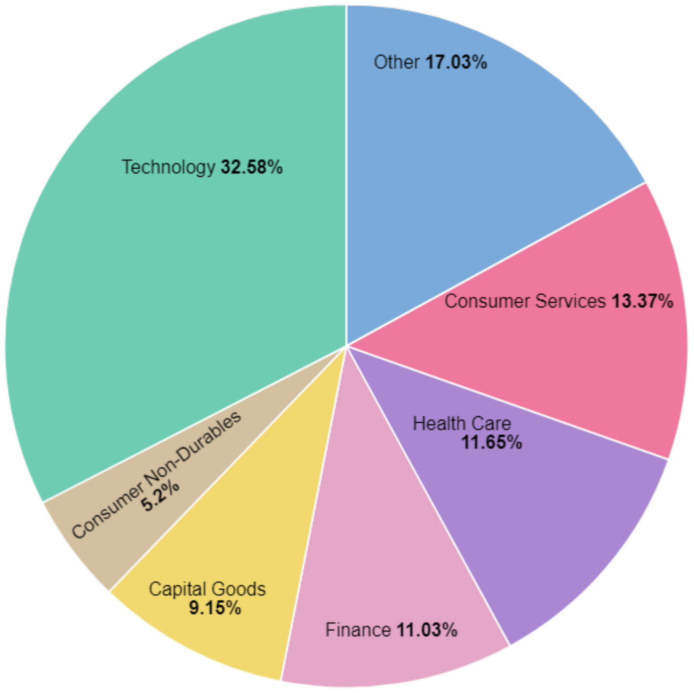
Since we also wish to perform analysis based on market sector and company size (market capitalization), we need to also find some mapping information to use in our analysis on-disk. To solve this problem we make use of the NASDAQ Stock Screener [7] which provides us a csv with mapping information.

Column (Hive Type)	Description
trading_date (STRING)	The date the data was recorded
trading_unix_timestamp (BIGINT)	The exact timestamp of the data
ticker (STRING)	The name of the security
open (FLOAT)	The opening price of the interval
high (FLOAT)	The highest price of the interval
low (FLOAT)	The lowest price of the interval
close (FLOAT)	The final price of the interval
volume (BIGINT)	The total trades during the interval
dividends (FLOAT)	Total dividends paid during the interval
stock_splits (INT)	Total stock splits during the interval

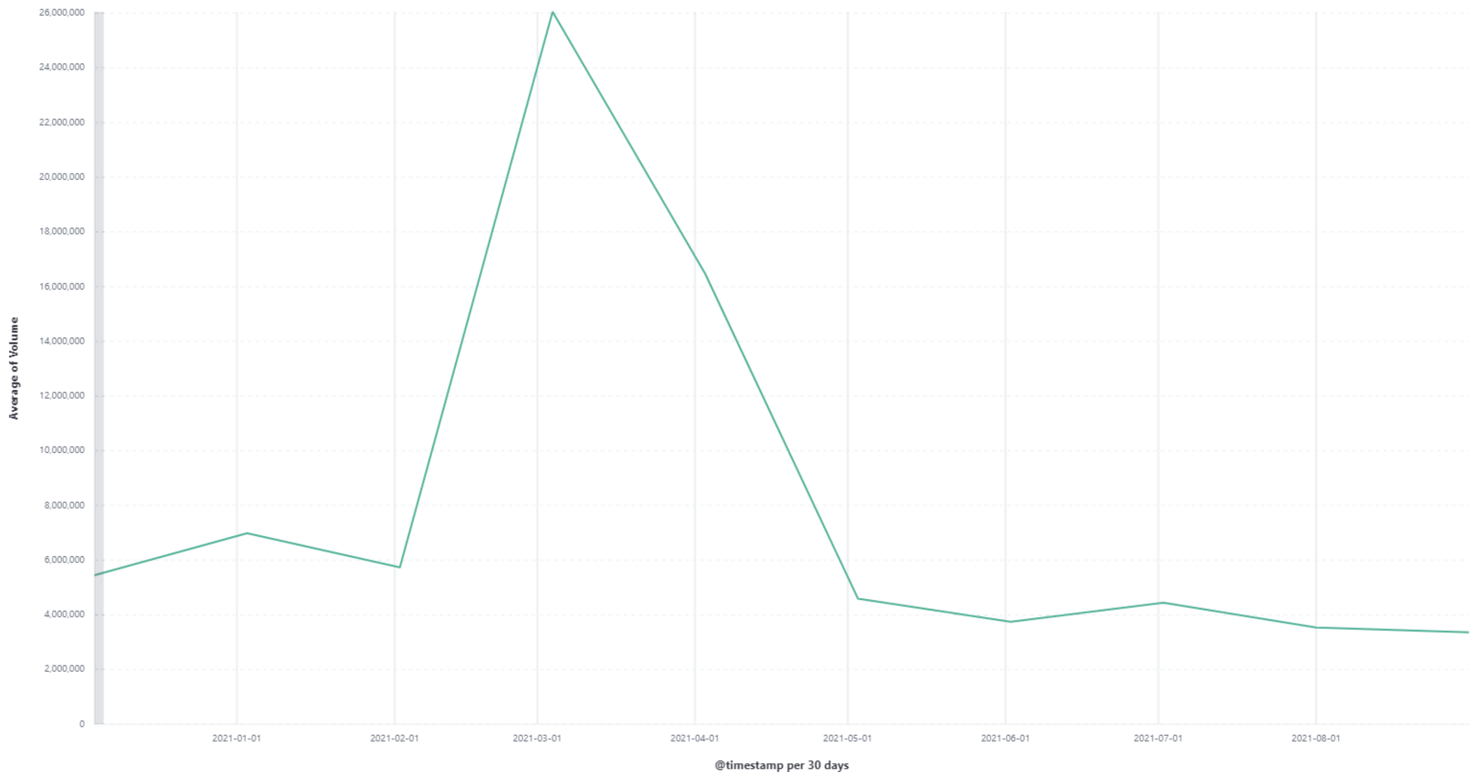
Statistics of the Data

Due to the nature of our project being different from our peers, and how our project provides a pipeline for extracting, transforming, and loading data rather than loading and analyzing an open dataset, the data analysis could vary based on the time of extraction. Some sample analysis highlighting the potential of our pipeline will be provided in lieu of an open dataset.

One example statistic that can be derived from our NASDAQ dataset is the following, which shows the sectors that make up the NASDAQ. This was calculated by summing the total market capitalization of each company grouped by sectors, and dividing by the total market capitalization, which was all done automatically through Kibana. We can see that technology dominates the market, with a whopping 32.58%.



Another example statistic that can be easily derived through our pipeline is analyzing the volume of any stocks that are defined in our Airflow DAG. In our example dataset, we specified that we wanted a years' worth of minute data of TSLA, AAPL, MSFT, FB, GOOG, AMD, and GME. Having loaded the data into Kibana, we can see that by taking the average of summed weekly volumes, we can compare trading volumes of different weeks, as seen below. We see that in the months of February to April, there were very large increases in volume being traded, nearly quintuple of the usual months, which lines up with the retail trading frenzy months [8].



Work Distribution

The work for this project was distributed as follows

Adam Azoulay	Infrastructure, ETL (Airflow ⟶ Hadoop ⟶ Hive)
Hee Kyoung Nam	Data analysis using Spark and Spark SQL
Peter Yu	Data visualization with Kibana

Solution Description

To solve the problems listed in the problem definition, we decided to make use of a variety of tools to create an ETL and analysis stack. We selected various tools that fit the needs of the problem, while keeping in mind the pros and cons of each tool. Below we will discuss each tool used, why it was selected, and how it was used.

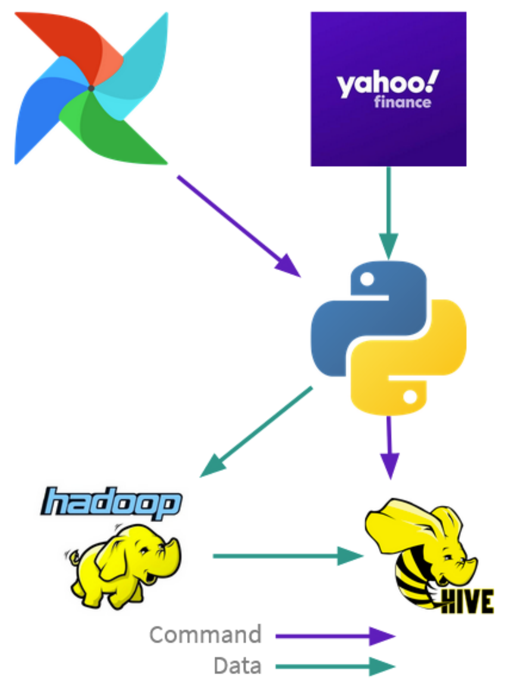


Fig. ETL Stack

Airflow

To begin with, we recognize that data from the stock market is always changing. This means that to receive up to date information we will need to regularly update the data we are storing, at some specified cadence. This can be done in a variety of ways, for example manually running extract scripts when we wish to perform analysis, or letting a third-party service maintain up to date extract for us. Neither of these options is very attractive and can even cause issues with our ETL, so we elected to make use of a scheduled task.

The most ubiquitous tool for scheduling jobs in a unix environment is by far the cron utility. We decided to make use of an open-source project called Airflow [3] which can run jobs in a cron-like manner by scheduling them, provide us a nice GUI to view our jobs, as well as enforce job dependencies and notify us about job status, making it more convenient than cron.

In the figure below we can see an example of the DAG structure for extracting data from the Yahoo finance API. This could be expanded even more using XCOMs to pass the ticker lists through the DAG allowing more complex DAG structure, however for the purposes of this demonstration we have elected to keep the structure more simple.

Lastly we present some code used in the DAG itself. We can see that in the helper function *pull_top_5_finance_data_yesterday_and_write_to_hive* we get some tickers of interest, and pass them to other helper functions which extract, transform, and load the data from various endpoints into Hive for storage and analysis. Note here that we make use of the *PythonVirtualenvOperator* class provided by Airflow. This allows us to keep dependencies localized to the task itself rather than polluting the global namespace. It is not very relevant in the early stages but as we add more functionality and dependencies to our code this will become very useful. This code is run on a schedule of our choosing. In this case we decide to run it daily due to some API call limitations, but if in the future we can access the API more frequently we may decide to increase the rates and use separate DAGs for different needs.

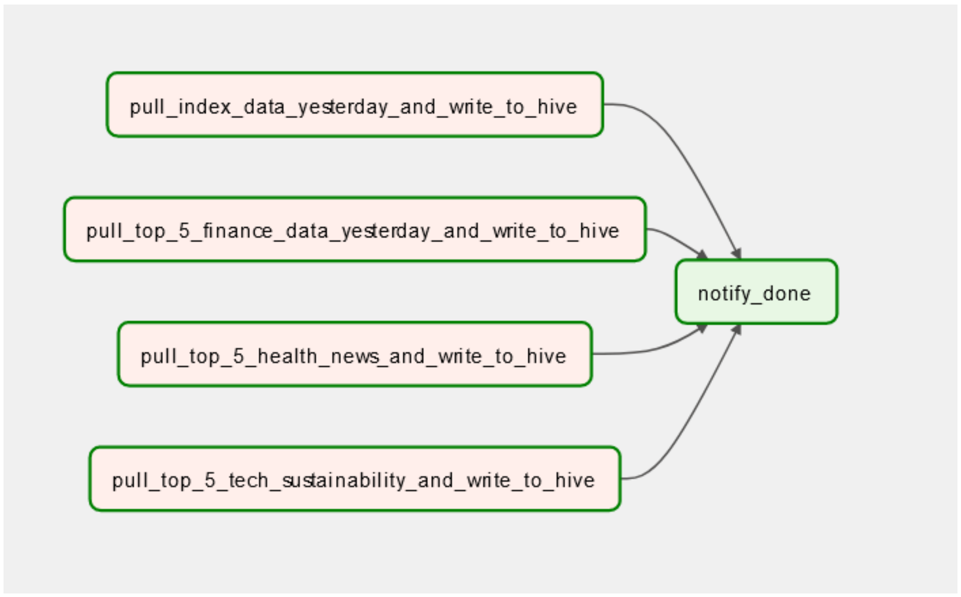
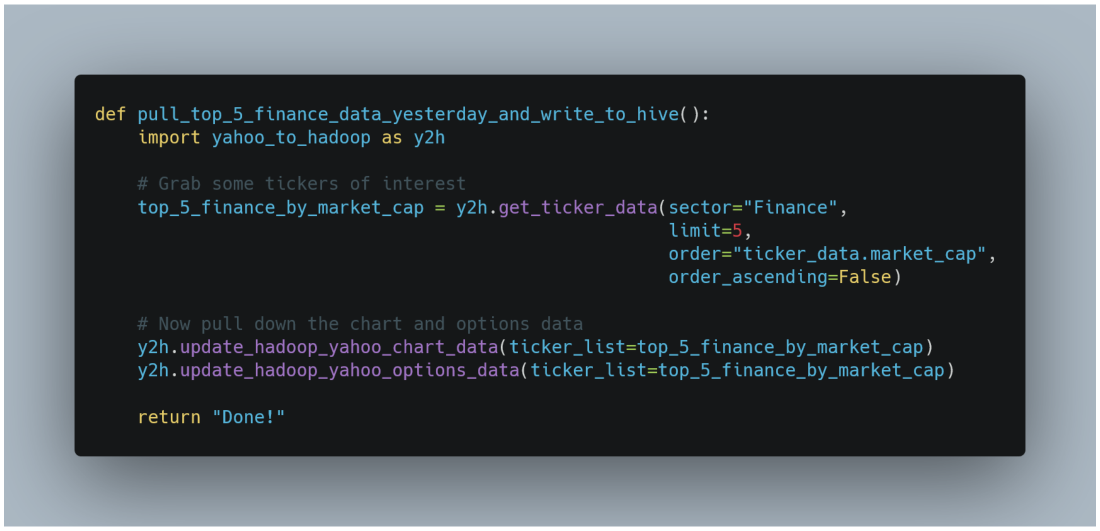
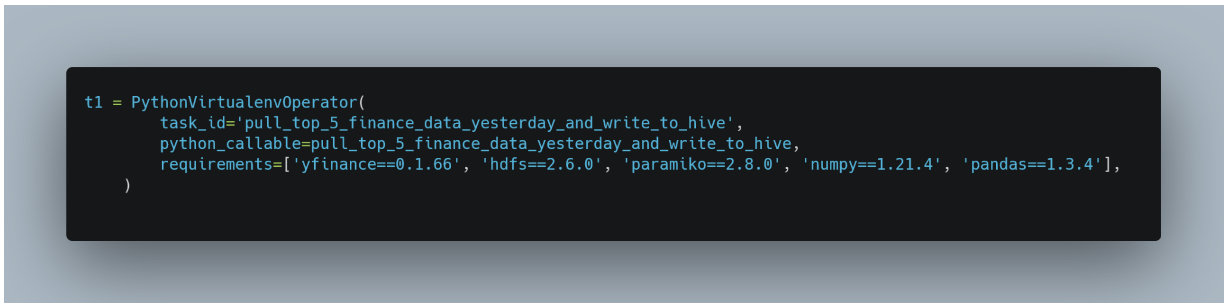

pull_index_data_yesterday_and_write_to_hive
pull_top_5_finance_data_yesterday_and_write_to_hive
pull_top_5_health_news_and_write_to_hive
pull_top_5_tech_sustainability_and_write_to_hive
notify_done

Fig. Example DAG



```python
def pull_top_5_finance_data_yesterday_and_write_to_hive():
    import yahoo_to_hadoop as y2h

    # Grab some tickers of interest
    top_5_finance_by_market_cap = y2h.get_ticker_data(sector="Finance",
                                                      limit=5,
                                                      order="ticker_data.market_cap",
                                                      order_ascending=False)

    # Now pull down the chart and options data
    y2h.update_hadoop_yahoo_chart_data(ticker_list=top_5_finance_by_market_cap)
    y2h.update_hadoop_yahoo_options_data(ticker_list=top_5_finance_by_market_cap)

    return "Done!"
```

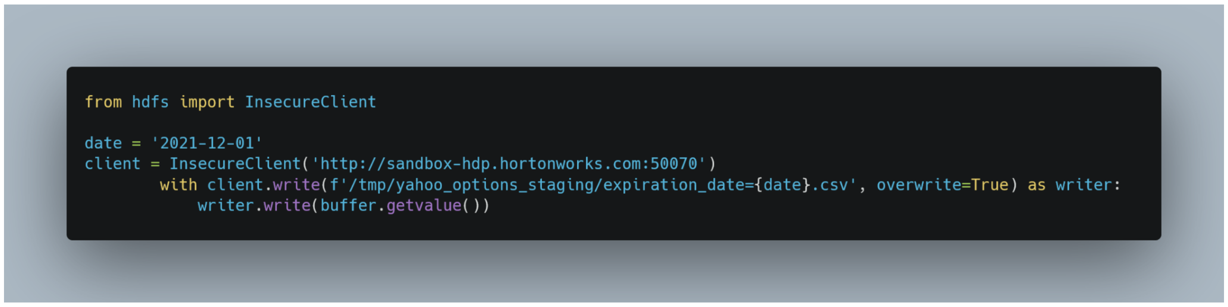


```python
t1 = PythonVirtualenvOperator(
        task_id='pull_top_5_finance_data_yesterday_and_write_to_hive',
        python_callable=pull_top_5_finance_data_yesterday_and_write_to_hive,
        requirements=['yfinance==0.1.66', 'hdfs==2.6.0', 'paramiko==2.8.0', 'numpy==1.21.4', 'pandas==1.3.4'],
    )
```

Fig. Example Airflow code

Hadoop

The next tool in the ETL after Airflow is a place to store our files. Since we would like to have fast access to the files, be fault tolerant in case of failure, be cost efficient, and have support for the tooling if needed, we selected Hadoop [4] for storage. Hadoop has a large user base, is not a cloud service (expensive), and allows us to have the features we want in a data storage tool. It is also very stable which is important to us since we want to reliably access our data at any time, as losing access to analysis can cost us money in a financial space.



```python
from hdfs import InsecureClient

date = '2021-12-01'
client = InsecureClient('http://sandbox-hdp.hortonworks.com:50070')
        with client.write(f'/tmp/yahoo_options_staging/expiration_date={date}.csv', overwrite=True) as writer:
            writer.write(buffer.getvalue())
```

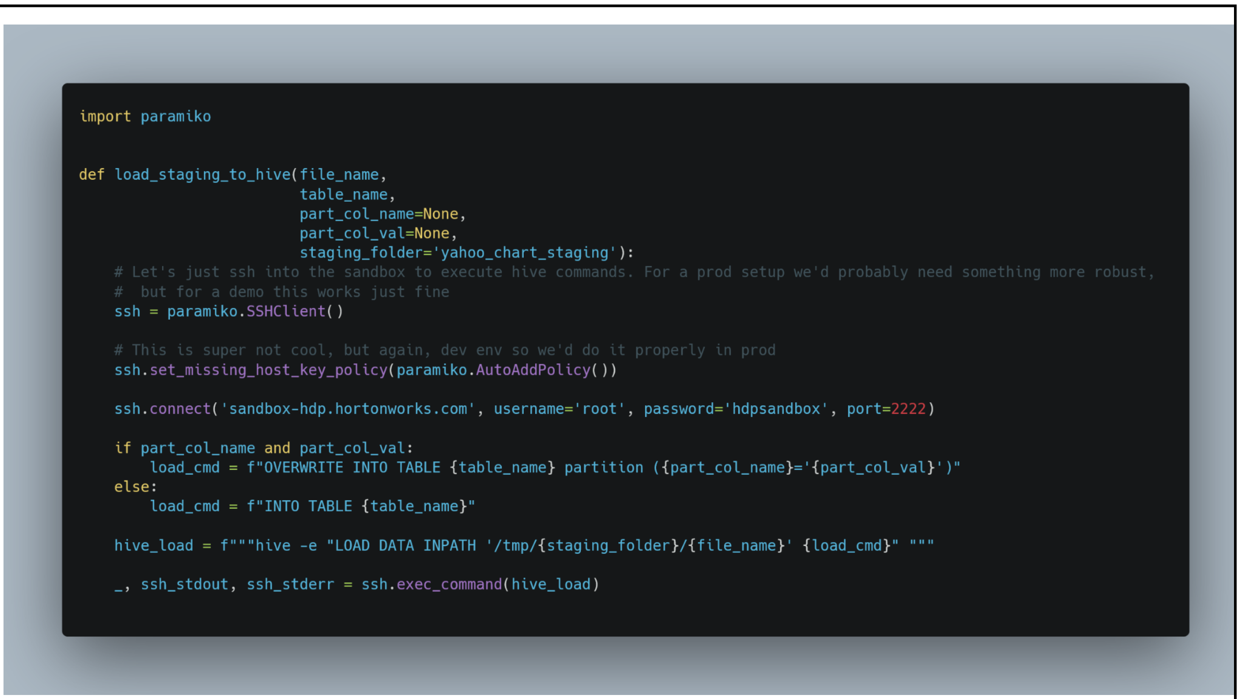
Fig. Hadoop writing

We can see in the figure above that writing to hadoop is a simple process. We can make use of the Python library *hdfs* and point it to our server. Note that here we do need to define the place we will write our data to and how we should name it. We write to a staging area so we know which files need processing, and we name the file clearly with the partitioning.

Hive

Next we need to organize the data in a data warehouse. For us, it makes sense to make use of Hive [5]for this purpose. Like Airflow and Hadoop, Hice has a long history of development, a large community, and is scalable to allow us to store as much data as we like. Some nice advantages of Hive over traditional RDBMS like being able to handle any file format we want to define, allowing us to view store/edit files directly on dist, and giving us the ability to use semi-structured data if we choose to.

In our use case we have a relatively simple data flow. We create a table in Hive (manually defined, but this could potentially be automated to some degree), and then we load data from the staging area into the table. We also partition the data to prevent data duplication and allow faster querying.



```python
import paramiko


def load_staging_to_hive(file_name,
                         table_name,
                         part_col_name=None,
                         part_col_val=None,
                         staging_folder='yahoo_chart_staging'):
    # Let's just ssh into the sandbox to execute hive commands. For a prod setup we'd probably need something more robust,
    #  but for a demo this works just fine
    ssh = paramiko.SSHClient()

    # This is super not cool, but again, dev env so we'd do it properly in prod
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    ssh.connect('sandbox-hdp.hortonworks.com', username='root', password='hdpsandbox', port=2222)

    if part_col_name and part_col_val:
        load_cmd = f"OVERWRITE INTO TABLE {table_name} partition ({part_col_name}='{part_col_val}')"
    else:
        load_cmd = f"INTO TABLE {table_name}"

    hive_load = f"""hive -e "LOAD DATA INPATH '/tmp/{staging_folder}/{file_name}' {load_cmd}" """

    _, ssh_stdout, ssh_stderr = ssh.exec_command(hive_load)
```

Fig. Hive load

We can see in the figure above an example of how the data is loaded into a hive table. We have parameterized the load commands so that we can use it for any table, partitioned or not, by passing in the data location and partition names. A few things to note here: we use SSH to tunnel into the terminal and run the load commands. This is done due to issues with the hive connector libraries in Python. For production, we would want to get those packages working correctly. Also we would like to load the credentials from Airflow rather than plaintext so we don't expose our keys.

Now that we have data loaded into Hive we can query it using HiveQL, query it on disk using Spark or SparkSQL, or connect to it using any other tool that has a Hadoop or Hive connector, which many programs do. This allows us to easily and quickly get insights from the data as soon as it lands in our data warehouse.

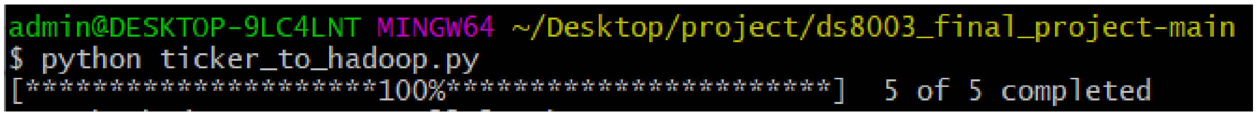
Spark and Spark SQL

Spark was used to calculate correlation between five big tech companies: Apple, Microsoft, Google, Amazon and Meta. The dataset used in Spark is the last one-year records of closing price of each company which are imported from the yahoo finance open source library. Spark MLlib provides machine learning algorithms including vector assembling and correlation features. The vector assembler feature was used to merge and transform each company's one year closing price containing column into a vector column. Correlation feature was used to compute the vectorized column into Pearson correlation matrix.

Spark SQL was used to examine the top analysts' ratings on a specific stock. In this project, Tesla's analyst ratings over the past three month were imported using the recommendation endpoint in the yahoo finance open source library. To filter and trim unnecessary columns in the raw data, SQL was used and it was also used to count the analysts' ratings on Tesla to observe rating distribution.

We used Spark to find out the correlation between different stocks for various reasons. In this project, codes were coded in Python and Spark supports analytics frameworks in Python as well. It also features various options for data visualization including matplotlib. And, we needed a tool to perform data analytics using complex SQL querying which is why Spark was a perfect fit.

Spark Code Snippets



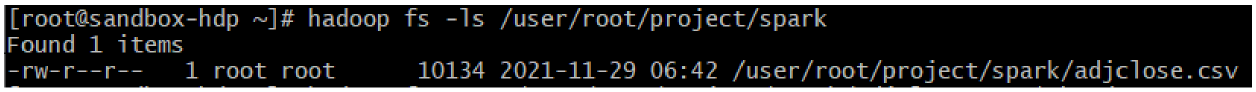
```
admin@DESKTOP-9LC4LNT MINGW64 ~/Desktop/project/ds8003_final_project-main
$ python ticker_to_hadoop.py
[*********************100%***********************]  5 of 5 completed
```

➢ Ticker_to_hadoop.py: import financial data from finance and store data into hadoop

```
from io import BytesIO
import pandas as pd
import yfinance as yf
from hdfs import InsecureClient

data = yf.download('AAPL MSFT GOOG AMZN FB', period="1y", group_by='ticker', actions = False)
data = data.iloc[:,(data.columns.get_level_values[1]=='Adj Close')]
data.reset_index(drop=True, inplace=True)
data.columns = [col[0] for col in data.columns.values]
```

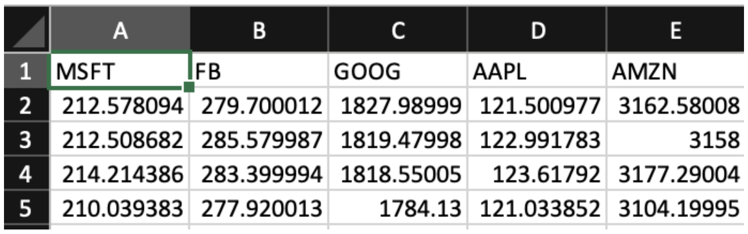
```
buffer = BytesIO()
data.to_csv(buffer, sep=',', na_rep='NaN', index = False)

client = InsecureClient('http://localhost:50070')
with client.write(f'/user/root/project/spark/adjclose.csv', overwrite=True) as writer:
   writer.write(buffer.getvalue())
```

➢ Adjclose.csv saved into hadoop system via pipeline



```
[root@sandbox-hdp ~]# hadoop fs -ls /user/root/project/spark
Found 1 items
-rw-r--r--   1 root root      10134 2021-11-29 06:42 /user/root/project/spark/adjclose.csv
```

➢ Using hadoop fs -get command, download 'adjclose.csv' from hadoop and save it to local.



	A	B	C	D	E
1	MSFT	FB	GOOG	AAPL	AMZN
2	212.578094	279.700012	1827.98999	121.500977	3162.58008
3	212.508682	285.579987	1819.47998	122.991783	3158
4	214.214386	283.399994	1818.55005	123.61792	3177.29004
5	210.039383	277.920013	1784.13	121.033852	3104.19995

```python
from pyspark.ml.stat import Correlation
df = spark.read.csv("adjclose.csv", inferSchema=True, header=True)
df.show()
```

```
+------------------+------------------+------------------+------------------+------------------+
|              MSFT|                FB|              GOOG|              AAPL|              AMZN|
+------------------+------------------+------------------+------------------+------------------+
|212.57809448242188|279.70001220703125| 1827.989990234375|    121.5009765625|   3162.580078125|
|212.50868225097656| 285.5799865722656|  1819.47998046875|122.99178314208984|            3158.0|
|214.21438598632812| 283.3999938964844| 1818.550048828125|  123.617919921875|   3177.2900390625|
| 210.0393829345703| 277.9200134277344|1784.1300048828125|121.03385162353516|3104.199951171875|
|208.77003479003906| 277.1199951171875|1775.3299560546875|122.48490905761719|3101.489990234375|
+------------------+------------------+------------------+------------------+------------------+
only showing top 5 rows
```

```python
from pyspark.ml.feature import VectorAssembler
vector_col="features"
assembler = VectorAssembler(inputCols= df.columns, outputCol= vector_col)
df_vector = assembler.transform(df).select(vector_col)
df_vector.show()
```

```
+--------------------+
|            features|
+--------------------+
|[212.578094482421...|
|[212.508682250976...|
|[214.214385986328...|
|[210.039382934570...|
|[208.770034790039...|
+--------------------+
only showing top 5 rows
```

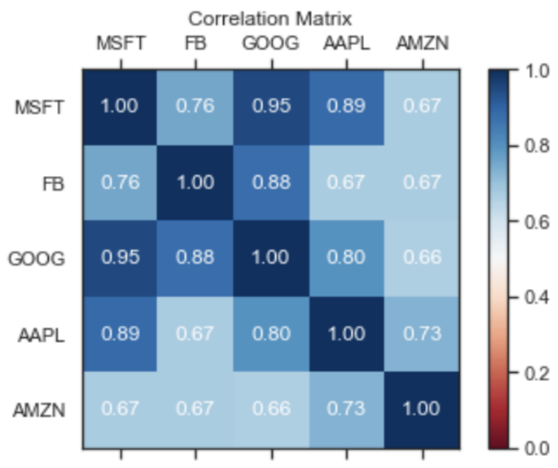
```python
matrix = Correlation.corr(df_vector, vector_col).collect()[0][0]
corrmatrix = matrix.toArray().tolist()
print(corrmatrix)
```

```
[[1.0, 0.7571414900031245, 0.9511371193982352, 0.8876890906264118, 0.6741713713097726], [0.7571414900031245, 1.0, 0.8
806664289762074, 0.6739036594730472, 0.6741427902556327], [0.9511371193982352, 0.8806664289762074, 1.0, 0.79762257198
11988, 0.6649770604427251], [0.8876890906264118, 0.6739036594730472, 0.7976225719811988, 1.0, 0.7308416820173252],
[0.6741713713097726, 0.6741427902556327, 0.6649770604427251, 0.7308416820173252, 1.0]]
```

```python
df_corr = spark.createDataFrame(corrmatrix, df.columns)
df_corr.printSchema()
df_corr.show()
```

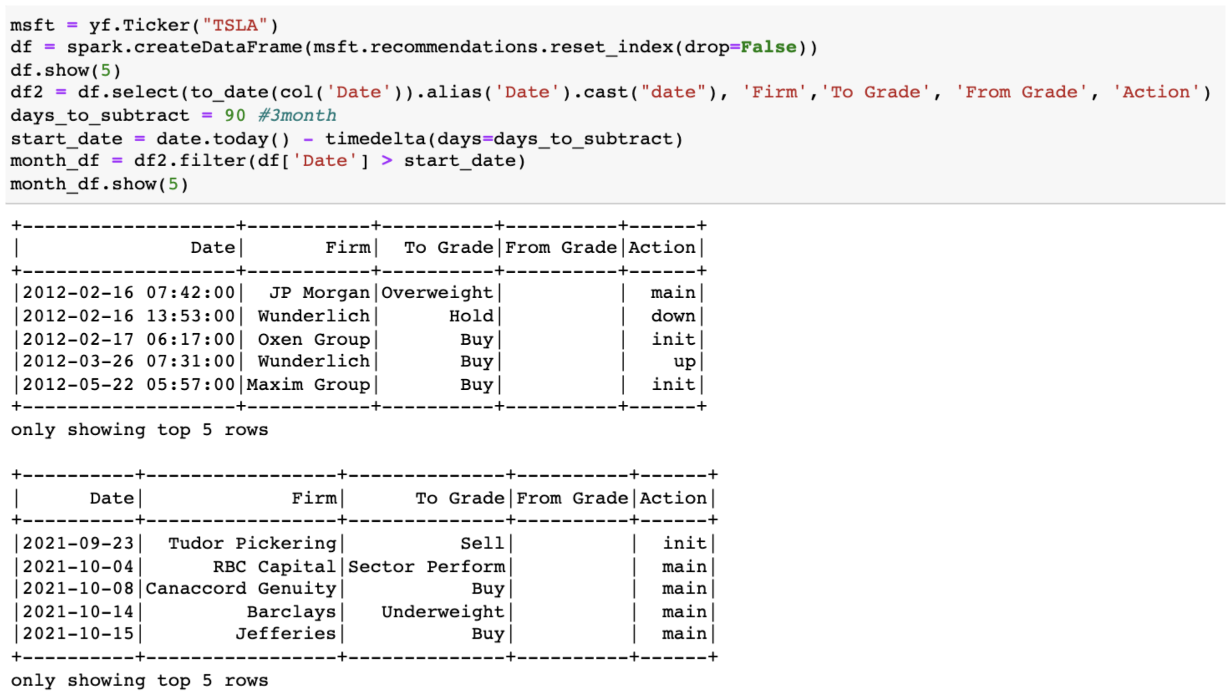
```
+------------------+------------------+------------------+------------------+------------------+
|              MSFT|                FB|              GOOG|              AAPL|              AMZN|
+------------------+------------------+------------------+------------------+------------------+
|               1.0|0.7571414900031245|0.9511371193982352|0.8876890906264118|0.6741713713097726|
|0.7571414900031245|               1.0|0.8806664289762074|0.6739036594730472|0.6741427902556327|
|0.9511371193982352|0.8806664289762074|               1.0|0.7976225719811988|0.6649770604427251|
|0.8876890906264118|0.6739036594730472|0.7976225719811988|               1.0|0.7308416820173252|
|0.6741713713097726|0.6741427902556327|0.6649770604427251|0.7308416820173252|               1.0|
+------------------+------------------+------------------+------------------+------------------+
```

```python
import matplotlib.pyplot as plt

def plot_corr_matrix(correlations,attr,fig_no):
    fig=plt.figure(fig_no)
    ax=fig.add_subplot(111)
    ax.set_title("Correlation Matrix")
    ax.set_xticklabels(['']+attr)
    ax.set_yticklabels(['']+attr)
    cax=ax.matshow(correlations,vmax=1,vmin=0,cmap=plt.cm.RdBu)
    for (i, j), z in np.ndenumerate(correlations):
        ax.text(j, i, '{:0.2f}'.format(z), ha='center', va='center', color='white')

    fig.colorbar(cax)
    plt.show()

plot_corr_matrix(corrmatrix, df.columns, 234)
```



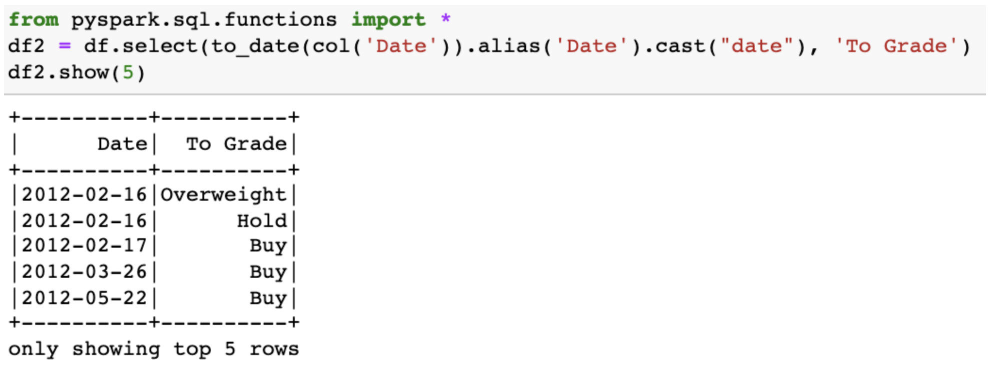
Spark SQL Code Snippets

➢ Using recommendation endpoint, import last three month analyst rating data on ‘Tesla’



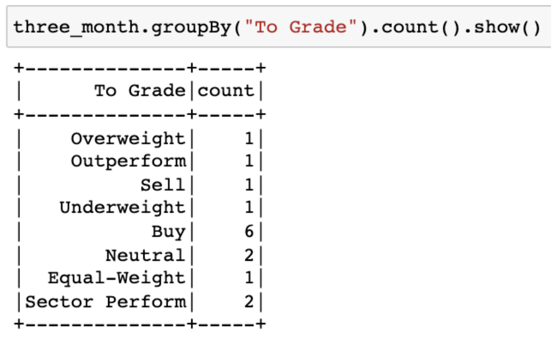
```
msft = yf.Ticker("TSLA")
df = spark.createDataFrame(msft.recommendations.reset_index(drop=False))
df.show(5)
df2 = df.select(to_date(col('Date')).alias('Date').cast("date"), 'Firm','To Grade', 'From Grade', 'Action')
days_to_subtract = 90 #3month
start_date = date.today() - timedelta(days=days_to_subtract)
month_df = df2.filter(df['Date'] > start_date)
month_df.show(5)
```

```
+-------------------+-----------+----------+----------+------+
|               Date|       Firm|  To Grade|From Grade|Action|
+-------------------+-----------+----------+----------+------+
|2012-02-16 07:42:00|  JP Morgan|Overweight|          |  main|
|2012-02-16 13:53:00| Wunderlich|      Hold|          |  down|
|2012-02-17 06:17:00| Oxen Group|       Buy|          |  init|
|2012-03-26 07:31:00| Wunderlich|       Buy|          |    up|
|2012-05-22 05:57:00|Maxim Group|       Buy|          |  init|
+-------------------+-----------+----------+----------+------+
only showing top 5 rows

+----------+-----------------+--------------+----------+------+
|      Date|             Firm|      To Grade|From Grade|Action|
+----------+-----------------+--------------+----------+------+
|2021-09-23|  Tudor Pickering|          Sell|          |  init|
|2021-10-04|      RBC Capital|Sector Perform|          |  main|
|2021-10-08|Canaccord Genuity|           Buy|          |  main|
|2021-10-14|         Barclays|   Underweight|          |  main|
|2021-10-15|        Jefferies|           Buy|          |  main|
+----------+-----------------+--------------+----------+------+
only showing top 5 rows
```



```
from pyspark.sql.functions import *
df2 = df.select(to_date(col('Date')).alias('Date').cast("date"), 'To Grade')
df2.show(5)
```

```
+----------+----------+
|      Date|  To Grade|
+----------+----------+
|2012-02-16|Overweight|
|2012-02-16|      Hold|
|2012-02-17|       Buy|
|2012-03-26|       Buy|
|2012-05-22|       Buy|
+----------+----------+
only showing top 5 rows
```



```
three_month.groupBy("To Grade").count().show()
```

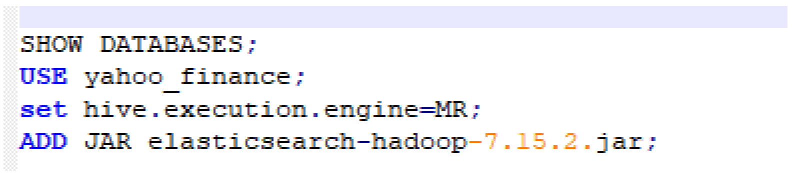
```
+--------------+-----+
|      To Grade|count|
+--------------+-----+
|    Overweight|    1|
|    Outperform|    1|
|          Sell|    1|
|   Underweight|    1|
|           Buy|    6|
|       Neutral|    2|
|  Equal-Weight|    1|
|Sector Perform|    2|
+--------------+-----+
```

➢ Raw rating data was filtered using spark sql and output shows a distribution of analysts’ ratings on ‘Tesla’

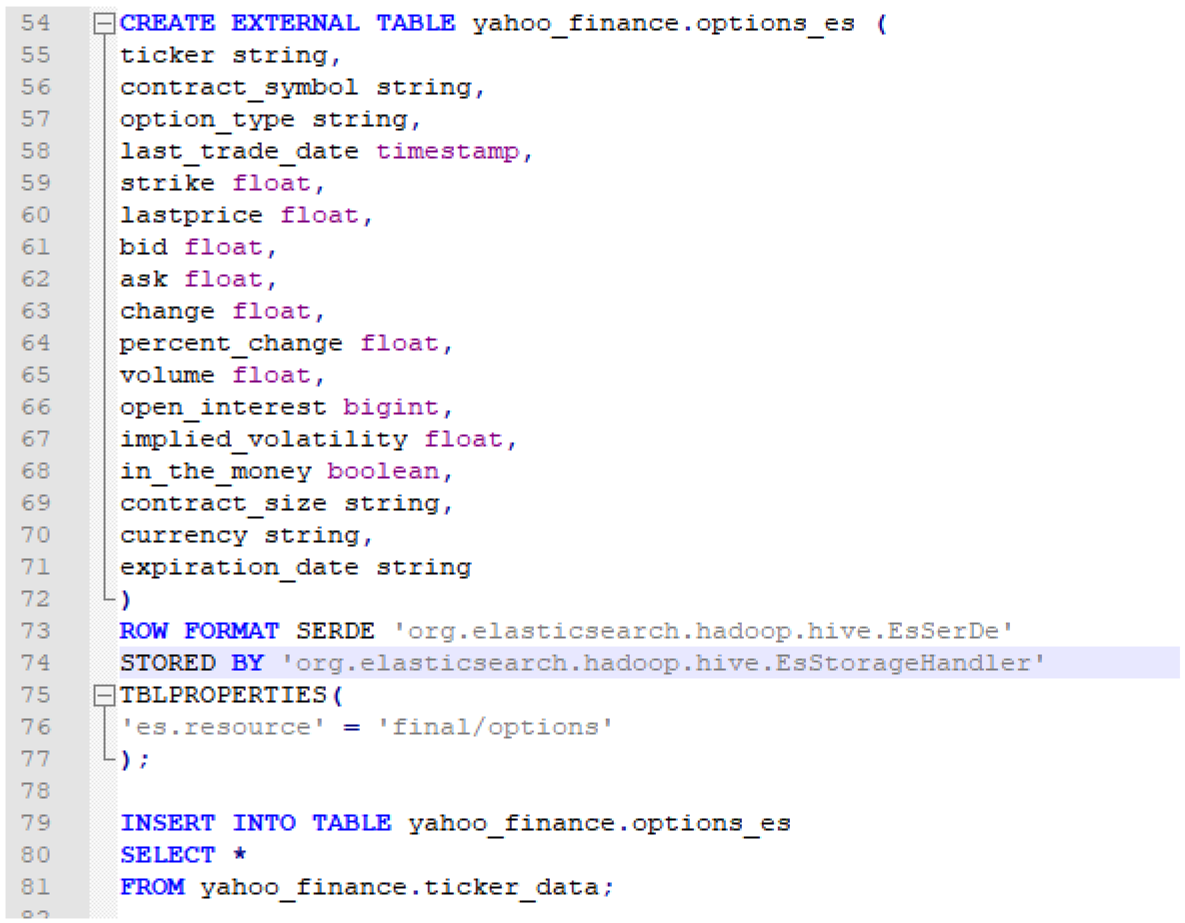
Elasticsearch and Kibana

The stock market contains millions if not billions of daily transactions and stock market ticker updates. Analyzing such data might not be considered feasible for traditional data analysis and visualization methods due to limitations in software, hardware, or having paid license agreements. Elasticsearch and its companion, Kibana, would serve as a perfect solution for our plight, as Elasticsearch's clever implementation of inverted indices and BKD trees would allow our solution to be scalable, and let us analyze and visualize data in near real time, while also being free.

Having loaded our data that was extracted from Yahoo Finance's API into Cloudera's Hadoop file system, our intention was to task Airflow with loading our batch data into Hive tables. Loading this data would process our data into structured tables, which would then be loaded into external Elasticsearch tables on Hive, which would then be stored on Elasticsearch. We could then analyze such data using the front-end tool Kibana, which is perfectly integrated with Elasticsearch. This has been done before and there is Elasticsearch-Hadoop documentation [6] however, perhaps due to how Cloudera's sandbox environment was set up, or due to the particular nature of Linux distributions being finicky, or our lack of experience, this was not achieved. We were to install Elasticsearch and Kibana on Ambari (which failed) and then add elasticsearch-hadoop.jar onto the Hadoop file system, and execute the following Hive commands, which would store our tables into Elasticsearch and allow us to visualize on Kibana.



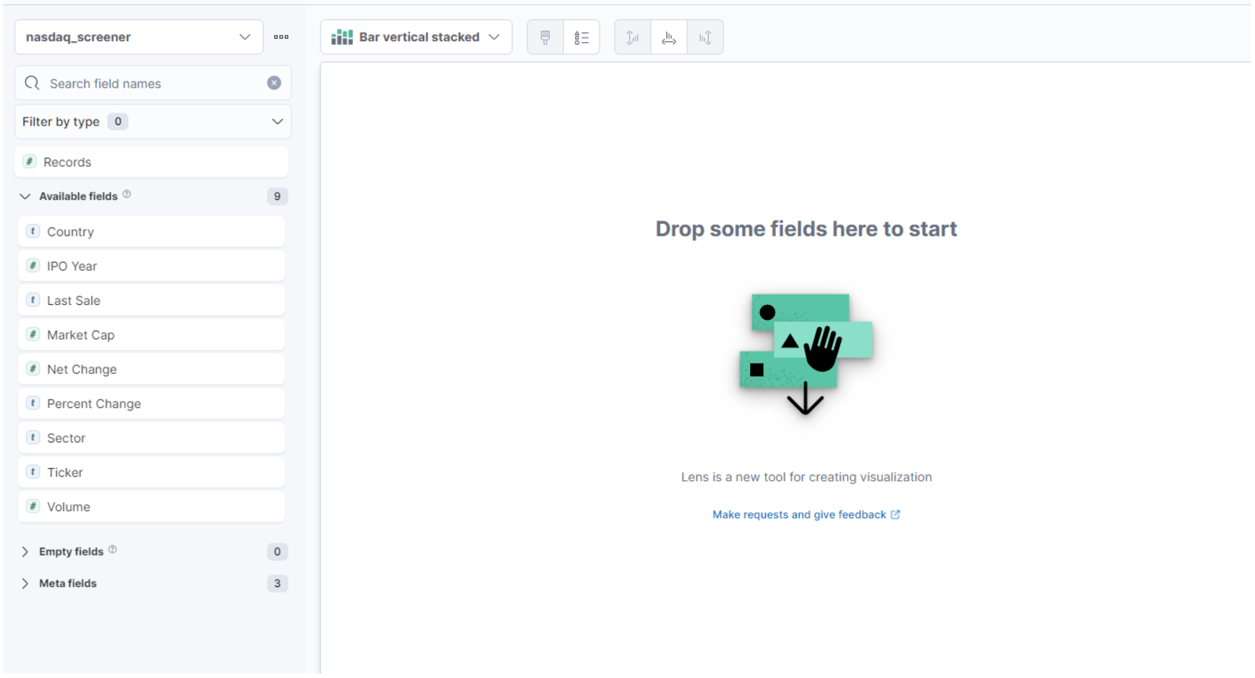
```
SHOW DATABASES;
USE yahoo_finance;
set hive.execution.engine=MR;
ADD JAR elasticsearch-hadoop-7.15.2.jar;
```



```sql
CREATE EXTERNAL TABLE yahoo_finance.options_es (
ticker string,
contract_symbol string,
option_type string,
last_trade_date timestamp,
strike float,
lastprice float,
bid float,
ask float,
change float,
percent_change float,
volume float,
open_interest bigint,
implied_volatility float,
in_the_money boolean,
contract_size string,
currency string,
expiration_date string
)
ROW FORMAT SERDE 'org.elasticsearch.hadoop.hive.EsSerDe'
STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler'
TBLPROPERTIES(
'es.resource' = 'final/options'
);

INSERT INTO TABLE yahoo_finance.options_es
SELECT *
FROM yahoo_finance.ticker_data;
```

Having spent a while deciding on what to do, we decided to try installing Elasticsearch, and Kibana on Windows 10. Luckily this was a much more straightforward process, and after downloading Elasticsearch and Kibana, it only took two clicks (clicking on elasticsearch.bat and kibana.bat) for our local server to run.

Due to the nature of Kibana being very user-friendly and intuitive, there was no code involved in the data analysis and instead, we were presented with a very pleasing user interface which allowed us to create visualizations as seen below. These interfaces allowed us to drag and drop various fields while filtering and specifying the type of visualization we wanted.


nasdaq_screener

Search field names

Filter by type 0

Records

Available fields 9

Country

IPO Year

Last Sale

Market Cap

Net Change

Percent Change

Sector

Ticker

Volume

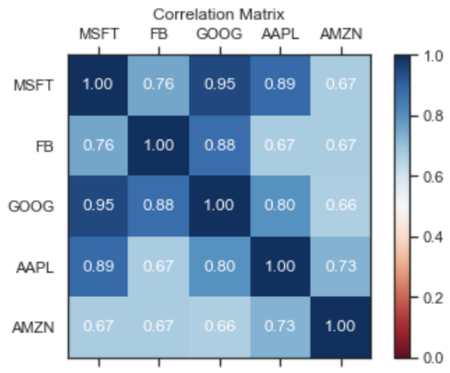
Empty fields 0

Meta fields 3

Bar vertical stacked
Drop some fields here to start
Lens is a new tool for creating visualization
Make requests and give feedback

Insights

Spark

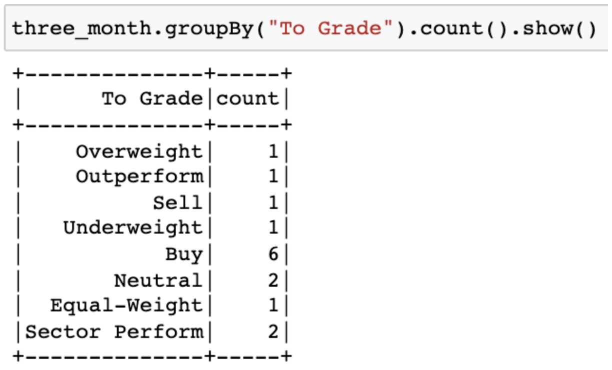


	MSFT	FB	GOOG	AAPL	AMZN
MSFT	1.00	0.76	0.95	0.89	0.67
FB	0.76	1.00	0.88	0.67	0.67
GOOG	0.95	0.88	1.00	0.80	0.66
AAPL	0.89	0.67	0.80	1.00	0.73
AMZN	0.67	0.67	0.66	0.73	1.00

From the correlation matrix of five big tech companies: Microsoft, Meta, Google, Apple, and Amazon, We found that the {Microsoft, Google} pair has the highest positive correlation, having the correlation coefficient of 0.95. This implies that it is highly likely that if one in the pair goes up, the other goes up. There was also a high correlation observed between the {Google, Meta} pair and the {Apple, Microsoft} pair. For low risk investment, choosing assets with relatively low correlation to each other is preferred.

Spark SQL

The table on the right is the past three month top analysts' rating distribution of 'Tesla'. It shows that six analyst groups/firms analyzed that Tesla is a good stock to buy. This type of analysis is crucial when buying a stock for investment and using spark sql we can easily see the experts' thoughts on specific stocks in a table.



```
three_month.groupBy("To Grade").count().show()
```

```
+--------------+-----+
|      To Grade|count|
+--------------+-----+
|    Overweight|    1|
|    Outperform|    1|
|          Sell|    1|
|   Underweight|    1|
|           Buy|    6|
|       Neutral|    2|
|  Equal-Weight|    1|
|Sector Perform|    2|
+--------------+-----+
```

Kibana

Due to the sizes of the figures, they have been moved to the final section titled Figures. In Figure 1, by summing market capitalization and grouping the circles by sector, we see in a different, more encapsulating visualization that technology, consumer services, and finance dominate the NASDAQ scene, and that this could help investors with deciding whether to look into investing in a NASDAQ exchange traded fund or looking elsewhere for more specific sector exposure.

In Figure 2, by summing all the initial public offerings (IPOs) of the NASDAQ stocks, and grouping by sector, we can see that in the recent decade, the number of finance, technology, and healthcare stock IPOs had increased significantly when compared to other sectors. This may have implications for the type of stocks investors prefer, or that there has been great success in those specific sectors.

Expanding on Figure 2, we can see in Figure 3 that we can also query and visualize the headquarter locations of the IPOs. We see that the United States is the main location for the stocks, while China remains as second in nearly all the NASDAQ IPO years. Not much can be said about this, except that perhaps in terms of company formation and innovation, the United States and China go hand-in-hand.

Future Work

While our Yahoo Finance ETL pipeline was ready to use, there are a number of underlying issues that make its usage rather unreliable. One, its reliance on Yahoo Finance, which is a free resource that could have delays in price updates, as well as limitations on API calls. For any professional stock market analysis, we would need to gather data from the source for true accurate real time data. Another issue would be the fact that ultimately, our data storage and visualization tools, Elasticsearch and Kibana respectively, are not being used by Airflow directly.

For a seamless user experience as well as truly calling this project a ETL pipeline that allowed analysis and visualization, we would eventually need to figure out how to add Elasticsearch and Kibana to our pipeline, rather than manually importing the scraped data into Windows-based Elasticsearch.

Thirdly, our project remains a proof of concept, as there is not much demand for batch data. We would need to instead stream our data, so that one could truly analyze the stock market in real time. In conclusion, while our project meets the goal of using Yahoo Finance endpoints to extract, transform, and analyze data, for its intended purpose of being a stock market tracker and analyzer, one needs to add Elasticsearch and Kibana into the Airflow pipeline and change the infrastructure so that it streams real time stock market data so that meaningful data can be obtained.

References

[1] (n.d.). Yahoo Finance API. Retrieved December 8, 2021, from https://www.yahoofinanceapi.com/

[2] (n.d.). GitHub. https://github.com/adamazoulay/ds8003_final_project

[3] (n.d.). Apache Airflow. Retrieved December 8, 2021, from https://airflow.apache.org/

[4] (n.d.). Apache Hadoop. Retrieved December 8, 2021, from https://hadoop.apache.org/

[5] (n.d.). Apache Hive TM. Retrieved December 8, 2021, from https://hive.apache.org/

[6] *Elasticsearch for Hadoop*. (n.d.). Elastic. Retrieved December 8, 2021, from https://www.elastic.co/what-is/elasticsearch-hadoop

[7] Mackintosh, P. (n.d.). *Stock Screener*. Nasdaq. Retrieved December 8, 2021, from https://www.nasdaq.com/market-activity/stocks/screener

[8] Mccrank, J. (2021, April 25). *Analysis: Retail trading appetite robust even as stock-buying frenzy cools*. Reuters. Retrieved December 8, 2021, from https://www.reuters.com/business/retail-trading-appetite-robust-even-stock-buying-frenzy-cools-2021-04-26/

Extra Figures

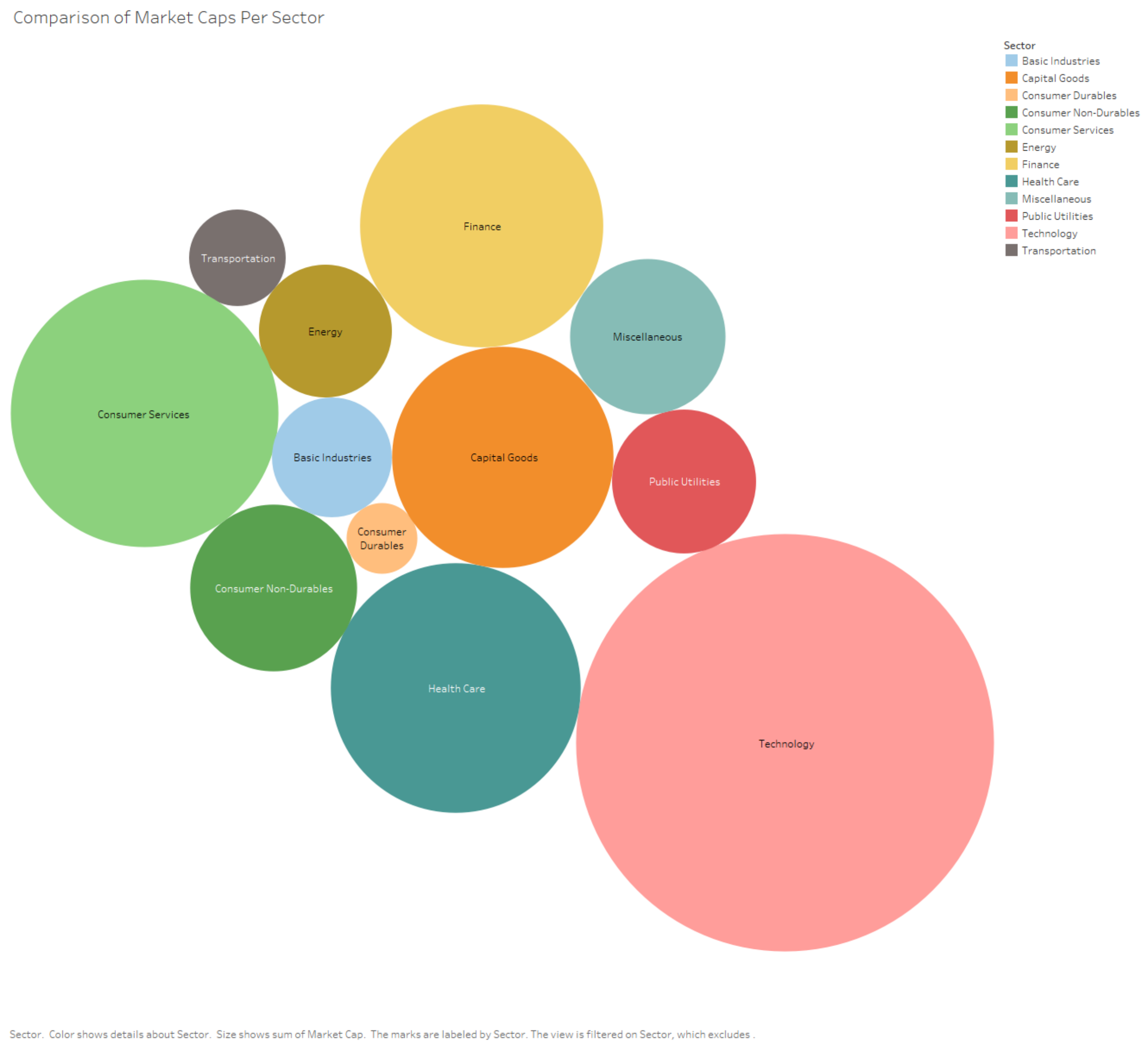


Figure 1. Comparison of Market Capitalization per Sector

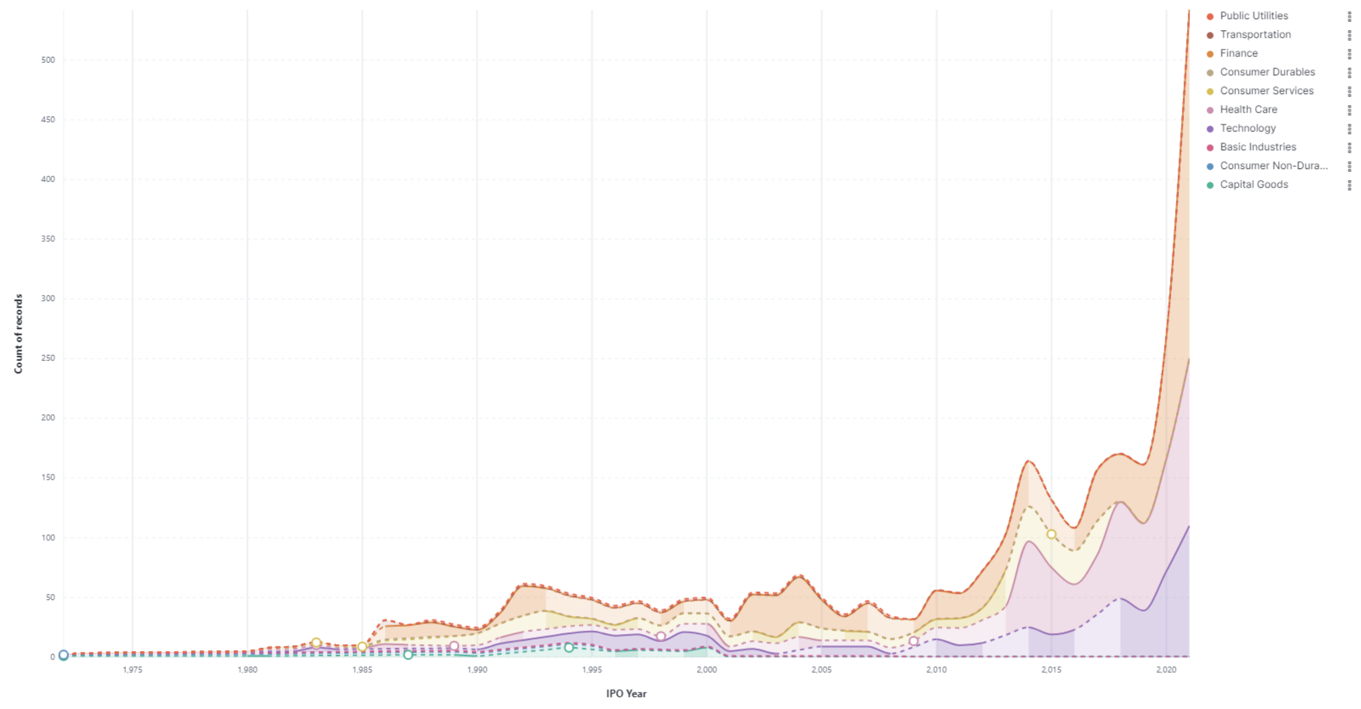


Figure 2. Number of IPOs per Year in the NASDAQ per Sector

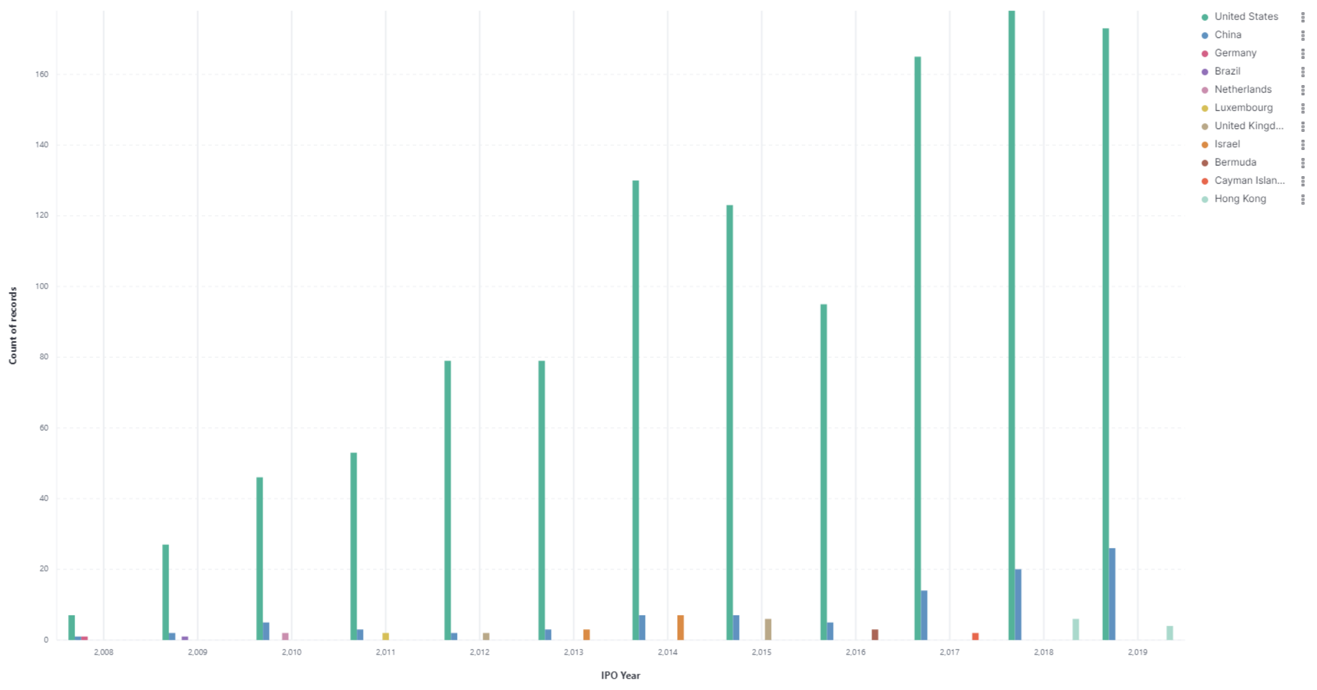


Figure 3. Number of IPOs per Country