*Hadoop offers a range of options for processing and storing data, such as MapReduce, Spark, and Hive. Outside of the HDFS ecosystem, data can be stored in MySQL and Cassandra. An interesting tool to benchmark database performance is YCSB. VADER, in conjunction with a multi-class neural network and time series analysis, can decipher hidden patterns and behavior in text. All these techniques will be applied to a set of tweets sent from April 6th to June 25th, 2009.*

This project has two parts Big Data and Advanced Data Analytics. In the first section Hadoop, database utilization and database benchmark will be demonstrated and in the second section, sentiment analysis, time series analysis and dashboards will be assessed.

According to Manwal and Gupta ,large organizations such as *Twitter, Facebook*, and *LinkedIn* use Hadoop to handle the vast amounts of data they generate daily. As the starting point of this project is the dataset *ProjectTweets.csv*, it would be beneficial to emulate the data processing methods used in *Twitter* analytics department.

Full implementation of this step can be seen at the annex section 6.1.1, the relevant part is that file now is into Hadoop and from there *MapReduce* jobs can be deployed, data can be read using Hive and data can be streamed for the analytics part using Spark.

Four different *MapReduce* jobs have been implemented with the aim of demonstrating how to perform these tasks. These jobs were also necessary to identify duplicates within ProjectTweets.csv, to clean the dataset before importing it into *Cassandra* and *MySQL* (as the last column contains commas and quotes, which are incompatible with those databases), and to demonstrate that Hive can achieve the same outcomes as a *MapReduce* job. Please note that full *MapReduce* jobs implementation can be found in the annex sections 6.1.2, 6.1.3, 6.1.4 and 6.1.5.

A given task could be to count all mentions and hashtags contained in this file. The mapper processes tweet text to find hashtags and mentions and then emits them as intermediate key-value pairs where the keys are the entities with a prefix, values are all 1 indicating a single occurrence for each entity.

The reducer sums above calculated occurrences to get a count of how often each hashtag and mention appears as it can be seen in Figure 3.

This *MapReduce* job is paired with a Hive query that will produce the same output, counting distinct values for *tweet\_id*. This mapper outputs each *tweet\_id* it encounters as a key-value pair, with the *tweet\_id* as the key and 1 as the value.

The reducer counts unique *tweet\_id* values received from the mapper.

Result to be compared with Hive query, from 1,600,000 rows, 1,598,315 are unique *tweet\_id*.

After several attempts to insert *ProjectTweets.csv* into *Cassandra* and *MySQL*, it was impossible because the text in the last column was full of commas (","). Since a comma is used as a delimiter, every attempt to import disrupted the file structure, which did not match the created table structure. Hence, this *MapReduce* job was necessary. The mapper reads all the lines, stripping any commas and quotes. It preserves only the first five commas to delimit six columns, ensuring the structure matches the table for a smooth load.

The reducer merely passes the cleaned data through and saves it.

Above *MapReduce* output was not ordered by *ids. MapReduce* paradigm does not guarantee an ordered output as seen in Figure 9.

This mapper transforms raw input into a structured key-value format, separating *ids* from the rest of the data.

The reducer sorts the output by the key *ids.*

Having the data ordered is useful for *MySQL*, as this database inserts the data in the given order. In contrast, Cassandra, like *MapReduce*, does not input the data in an ordered manner due to its distributed nature. Both *MapReduce* and *Cassandra* are designed to handle large-scale data across distributed systems, which prioritize scalability and fault tolerance over maintaining data order. This distribution means that data is processed in parallel across multiple nodes, making the preservation of order less practical and often unnecessary for the intended analytical or transactional operations. Figure 12 shows rows in ascending order.

This step is crucial for the project as it is the connection between *Big Data Storage and Processing*, and *Data Advanced Data Analysis.* This consideration has been made with two reasons first to demonstrate *Spark* streaming functionality and second moving the *ProjectTweets.csv* into *Windows*. As *Ubuntu* host has lower specs than *Windows* guest OS, the decision was made deliberately for the ADA where high computational demand was needed for machine learning.

This step is crucial for the project as it connects *Big Data Storage and Processing* with *Advanced Data Analysis.* This consideration was made for two reasons: first, to demonstrate *Spark* streaming functionality, and second, to move the *ProjectTweets.csv* into *Windows*. As the *Ubuntu* host OS has lower specs than the *Windows* guest OS, the decision was made deliberately for the ADA part, where high computational demand is needed for machine learning.

*Facebook* engineers developed this technology in 2010 to simplify the complexity of writing *MapReduc*e jobs by utilizing *SQL* syntax. *Facebook's* analysts were familiar with *SQL,* which is why this querying language was used to extract information from its vast *Hadoop* datasets (Thusoo et al., 2010). Establishing an analogy the complexity of point *2.1.2.2. Distinct tweet\_Id count* can be solved in just one line of code, *SELECT COUNT(DISTINCT tweet\_id) FROM tweets;*

Both *MapReduce* and *Hive* yield the same outcome of 1,598,315 distinct t*weet\_ids.* This is a simple and practical way of demonstrating why *Hive* was developed.

In section 2.2 a comparative performance analysis for *MySQL* will be conducted. Prior to that, the output of the fourth *MapReduce* job (a dataset ordered by ID) was smoothly introduced into MySQL.

One of the strong points of *MySQL* is its syntax, which is easy to interpret and perform.

Also, *Cassandra* will be evaluated in Section 2.2. Before that, the output from *MapReduce* job four was loaded. It is worth mentioning in this section the problems I encountered before concluding that *Cassandra* does not like commas prior to a data load; it was also skipping rows that contained quotes. In the screenshot below, the rows are not ordered by *ids*. This is because *Cassandra*, due to its distributed nature, does not concern itself with order but simply distributes the data across its nodes, Figure 16 shows it.

Similarly to Brian F. Cooper, who published his paper on benchmarking databases (Cooper et al., 2010), this section will compare *Cassandra*, a distributed NoSQL database, *MySQL*, a relational database, and *MongoDB*, a document-oriented NoSQL database. All *YCSB* workloads, plus an additional one, Workload G, have been tested. Workload G (100% Write) serves to contrast with Workload C (100% Read). Note that Workload E has been modified from 95/05 to 65/35 scan/insert, because when running the tests initial outputs were not conclusive hence the change.  
The strategy for testing is based on five iterations (see Figure 17), with changes in the number of rows inserted in each one. Full workload implementation can be seen in the annex sections 6.1.10, 6.1.11, and 6.1.12.

Figure 18 shows that *MySQL* performs better than *Cassandra* in balanced read scenarios, in contrary *Cassandra* performs similarly to *MongoDB* at writing.

Figure 19 shows that even lowering the load to 5% *MySQL* stills performing poorly compared to *Cassandra* and *MongoDB.*

Nothing new in a red scenario *Cassandra* has higher latency, Figure 20 *Overall* shows another way of comparing *YCSB* outputs, runtime and throughput, *MongoDB* has the lowest runtime handling bigger volume of operations.

Figure 21 shows that *MySQL* performs better than *Cassandra* in heavy read scenarios, in contrary *Cassandra* performs similarly to *MongoDB* when inserting.

Initially this workload was 95/5 Scan/Insert, the change to 65/35 was made deliberately aiming to see shift for *Cassandra* or *MySQL, MongoDB* performs exceptionally under all circumstances as seen in Figure 22.

Interesting workload as usually in read scenarios *MySQL* outperformed *Cassandra*, in Figure 23 when combining reading with modify and read, that changes performing *Cassandra* significantly better than *MySQL.*

This additional workload was created to confront workload C. Same as the other scenarios *MySQL* has high latency and runtime when writing, see figure 24.

Every time record count was changed in workload A for database insertion that gave an output metric, these are plotted in figure 25, similarly with writing *MySQL* has high latency and runtime in this scenario.

After the tests the conclusion is clear *MongoDB* is the best performer due to its document-oriented design, allows efficient retrieval and its *BSON* format enables quick scanning and indexing. *MySQL* is strong in reading scenarios. *Cassandra* is designed for high write throughput and excels in writing scenarios.

As part of this conclusion a test for workload distribution has been carried out. Above tests have been run using *zipfian* distribution, there are five more, *uniform, hotspot, sequential, exponential,* and *latest.* It is convenient to see how these perform, same approach five iterations for database *MongoDB* under workload A, to determine which distribution performs best. Choice of database was easy as *MongoDB* has the lowest runtime compared to *MySQL* and *Cassandra*.

First iteration for write and read scenarios, the highest latency is for *uniform* and the lowest latency for *latest.* In thethird, fourth, and last iterations, all distributions seem to perform equally, with *exponential* and *sequential* showing slightly higher latency.

Overall performance-wise, *latest,* is the worst, displaying a steep linear trend. Looking at the last iteration in Figure 28, *sequential* has the best performance. I conclusion *zipfian* distribution was used as a default setting, after evaluation it seems not a bad choice.

The rationale for the choices in data processing and storage, as well as programming language selection, is purely to handle Big Data effectively. The use of *Hadoop* Distributed File System (*HDFS*) for storing *ProjectTweets.csv* is because this system has strong scalability and fault tolerance for large datasets. *MapReduce* jobs process large datasets with a parallel, distributed algorithm on a cluster, that enabled tasks like counting mentions, hashtags and cleaning quotes and commas.

The choice of *Spark* as a streaming tool comes handy as it speeds up data processing tasks. *Hive* is used as an alternative to *MapReduce*, it can be used to summarise data, query, and analyse, as it provides a SQL syntax to query data stored in *Hadoop.*

*MySQL* and *Cassandra* were chosen for their strengths in data storage. *MySQL* is widely used as a relational database and offers *ACID* compliance. *Cassandra,* on the other hand, provides high availability and scalability for unstructured data, being ideal for large datasets.

Programming languages were selected in need of dealing with Big Data. *Python*, used in *MapReduce,* is powerful to prepare and process data, while SQL used in *Hive* is universally known for data manipulation.

As a final reflection, there are many examples across industries using these technologies, that was also a motivation to select them.

Figure 29 illustrates all tasks carried out for the *Big Data* section. There are two well-established sections: all tasks carried out in the HDFS ecosystem and those outside Hadoop, though the data has been stored and processed there. Starting with the first section, a *CSV* file was stored in *Hadoop*, and four *MapReduce* tasks were completed. *Hive* assisted in reading the data, and Spark was used to stream the data into the *Advanced Data Analytics* part. Technologies outside *Hadoop* include *MySQL* and *Cassandra*, but these required a *MapReduce* job to clean and order the data prior to loading. Also note database benchmark was conducted in this section.

Dataset was brought without modifications into *Windows* host machine as high computational capacity was needed in this part.

As previously seen in BDSP section the dataset contained 1685 duplicate records. Also, tweets are not ordered by date, this is important to perform time series, tweets must be ordered chronologically from the 06-04-2009 to 25-06-2009.

This circumstance had special impact when performing time series analysis, however different approaches were conducted to mitigate. In section 3.3. these will be discussed in detail.

Several sentiment analysis tools, such as *TextBlob, RoBERTa, BERT* and *VADER,* were used. The final choice was *VADER,* as it offered lower computational time compared to the rest. *VADER* provides a sentiment range from -1 to 1,with scores greater than 0.05 indicating positive sentiment, scores less than -0.05 indicating negative sentiment, and scores between -0.05 and 0.05 indicating neutral sentiment.

After VADER scores were calculated, sentiments were averaged and grouped by day of the week. This approach is not ideal due to the imbalanced nature of the dataset, with some days having higher frequencies than others.

Analysing the frequency grouped by day of the week, a similar plot to Figure 31 can be seen at Figure 32. Therefore, the average daily sentiment is biased by daily frequencies; days with higher *VADER* scores appear with higher daily tweet frequencies.

To get a uniform metric to find out which days sentiments tend to be higher or lower. Tweets were averaged hourly, and the top sixty scores were selected to determine which weekdays had highest or lowest sentiment score.

As seen in Figure 32, Twitter users react more positively during the weekend. This trend starts on Friday, improves on Saturday, and peaks on Sunday. In contrast, Thursday, Tuesday, Monday, and Wednesday, which did not make it into the top sixty, have less positivity among Twitter users.

Aligning with the above findings, Twitter users are less prone to react negatively on weekend days. This trend changes starting on Monday and reaches its highest point on Thursday, as shown in Figure 33.

*VADER* analysis required cleaning all tweets, which included removing mentions (‘@’) and hashtags (‘#’). This approach is necessary for *VADER* to accurately assign a sentiment value to each row. However, by removing hashtags and mentions, the very nature of Twitter is truncated, and the inner significance of the tweets is lost. There are many studies on this topic. Therefore, after performing the *VADER* analysis, the original tweets were brought back to be paired with scores to determine sentiment fluctuations over time.

As seen in Figures 34 and 35, a sentiment analysis change must include hashtag and mention analysis. Musicians such as *@ddlovato, @mileycyrus, @taylorswift13, @DavidArchie,* and *@andyhurleyday* have a huge impact on driving positive sentiment Additionally, events such as *#mothersday, #eurovision,* and *#MTVmovieawards* drive positivity across Twitter users. Nevertheless, not all sentiments are positive. On June 12, 2009, Iran held presidential elections and shortly after, the results led to violent riots that shocked Twitter, with many users condemning the events.

With such a large sentiment dataset, a classification neural network (NN) was the natural step. VADER scores were labelled as seen in Figure 36, with three categories: positive, negative, and neutral.

A NN with one input feature, three layers, and an output layer using *softmax* to classify into categories 0, 1, or 2. The activation function choice is based on classifying more than two categories.

The model performed extremely well, achieving 99.79% accuracy and 0.001% loss. Validation accuracy and loss were identical, indicating no overfitting. However, an extra step to confirm this result was necessary. Cross-validation over ten folds showed 99.99% accuracy and a 0.01% standard deviation, indicating consistency across different data subsets. These results confirm that the model performs extremely well in classifying sentiment based on *VADER* scores.

To demonstrate this section anecdotally, the approach was to make the NN model perform worse. The next sections show how to achieve this.

As models usually perform well with more data this time the approach was the opposite: downsizing the dataset to find a poorer model. The same NN for multi-class classification was used with only 1% of the dataset. After training, the model's accuracy dropped to 97.17% and loss increased to 39.86, but it still performed well. The model can be tuned to increase performance, but there is little margin for gain in accuracy. The next section explores a different approach.

Five features were added to the model as seen in Figure 39:

An *LGBM* classifier was used, achieving an accuracy of 100% with the same 1% of data as used before. The model still performing extremely well, but why is that?

Model still performing well because of the feature importance of *vader\_score* and the strong correlation that has with sentiment. Next section has the solution to downgrade model performance.

Target variable was shuffled changing the distribution of labels:

Seventy percent of the *vader\_score* values were randomly transformed to zero. Now the dataset is set to train the NN used before, with the only difference being the increase in the number of input features from one to six. Important to note that dataset size is increased to 2% of the original dataset.

After training, accuracy was at 61.76% and loss at 63.88%. Finally the model is ready for hyperparameter tuning.

Full implementation can be seen at *“3.Advanced\_Data\_Analytics.ipynb”,* as seen in Figure 44, all models performed similarly and tunning techniques increased 10% the initial accuracy, a significant improvement.

As seen in section 3.1.2 about how imbalanced this dataset is, plotting tweets over time will result in this graph with a non-continuous trend line due to missing data:

There are solutions to this problem, first interpolate method:

After seeing the interpolation results, which visually are not the best, a different approach was used. Forward filling will use the next available value for the same day, and backward filling, in the absence of the next same-day available value, will go to the next one and then backward fill that missing value. Figure 47 show a more realistic trend line using this method.

The *Dickey-Fuller* test showed that the time series shown above were not stationary, and spectral entropy indicated moderate predictability.

For accurate predictions, intervals of twenty-four hours were chosen. The strategy to fill missing values is defined in these functions:

Figure 49 shows the trend line after using the above functions.

This time series is also non-stationary and has a moderate value for predictability.

For predictions, *ARIMA* and *SARIMA* were tested, but the results were not satisfactory. Predictions were carried out using *ForecasterAutoreg* and *LSTM.*

First round of predictions:

Finding best parameters using *ParameterGrid:*

Slightly improvement after hyperparameter tuning:

A different approach for time series predictions is using a NN, in this case, an *LSTM* model was used.

Results after training showed RMSE improved with increasing predictions length.

Visually, *LSTM* results make more sense than *ForecasterAutoreg*. In this case, due to the non-stationary and imbalanced nature of the time series, an *LSTM* model is the right choice.

All illustrations shown in this project aim to follow Tufte’s principles by just showing the data. The primary goal is to present data clearly, eliminate chartjunk, avoid unnecessary elements that could distract readers' attention, and encourage data comparison.

Four examples of adherence to Tufte’s principles from Figures 56 to 59.

There is an early dashboard version; however, a more elaborated and interactive dashboard was created using *Streamlit*.

Several conclusions can be established after this experiment:

* *Hadoop* alongside *MapReduce, Spark* and *hive* offer an incredible ecosystem to handle large files. Another advantage is its open-source nature which lowers costs in terms of licensing.
* *MySQL* and *Cassandra* are great choices for data storage. However, depending on project specifics, *MySQL* excels in read capacity, while *Cassandra* is better suited for those needing high write capacity.
* *YCSB* confirmed point above and proved that *MongoDB* outperformed *MySQL* and *Cassandra* in all scenarios.
* *VADER*, as a text sentiment analysis tool, determined the sentiment of tweets from an imbalanced dataset, which has proven difficult for making predictions. However, with the help of *ForecasterAutoreg* and an *LSTM* NN, decent forecasts were made.
* From time to time, a robust model containing a highly correlated feature with the output target may need to undergo reverse engineering of its data structure to make it suitable for hyperparameter tuning.

All conclusions stated are based on experimentation, and the results can be replicated at any time.