*This report focuses on processing and analysing a Twitter dataset, to process the data technologies like Hadoop, MapReduce, MySQL, Cassandra are demonstrated also an exhaustive database performance analysis is performed using Yahoo Cloud Serving Benchmark. To analyse the dataset a neural network using time series is applied to detect and predict any sentiment change throughout the given period….*

This project is divided in two sections Big Data and Advanced Data Analytics.

For Big Data a csv file is processed using Hadoop, from there *MapReduce* is applied to perform four tasks. Reading data from Hadoop is achieved using Spark for streaming and Hive for badging. As a non *HDFS* databases *MySQL* and Cassandra are also explored to finalise this section a thorough analysis for MySQL, Cassandra and MongoDB is carried out using Yahoo Cloud Serving Benchmark (YCSB).

The second section Advanced Data Analytics….

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.