[Github link](https://github.com/Carloselrecharlie/BD_assessment.git)

Word count: 1503

Table of Contents

[1. Introduction 1](#_Toc142233537)

[2. Data preprocessing and EDA 2](#_Toc142233538)

[2.1 Exploration of text feature 5](#_Toc142233539)

[3. YCSB 7](#_Toc142233540)

[4. References 11](#_Toc142233542)

# Introduction

The twitter dataset was handled with *Apache Spark* technology mainly. Some of the reasons to choose this open-source distributed computed framework are because of its fast in-memory processing capabilities (compared with for example MapReduce) and it can be used together with *Hadoop’s HDFS* for storage. It includes a distributed computing engine and a set of high-level APIs for building applications. When handling data there are three logical abstractions: *RDDs* (Resilient Distributed Datasets) used to be the only option, they are immutable distributed collections of elements of data and they are useful when a dealing with unstructured data or when, for instance, a developer wants a low-level transformation and actions and control on the dataset. In this case tweets are semi-structured data and the tweets themselves (text) are unstructured data.

*Spark DataFrames* also are immutable distributed collections of data, but they are organized into named columns, like a table in a relational database and are designed to make large data sets processing easier (data manipulation, filtering, aggregations, and transformations). They allow users to impose a structure onto a distributed collection of data, allowing higher-level abstraction (Damji 2016). Considering these facts the choice was to use *DataFrames*. Additionally, they also offer better performance optimizations like Project Tungsten and Catalyst optimizer (NK 2023), which can lead to faster data processing compared to RDDs.

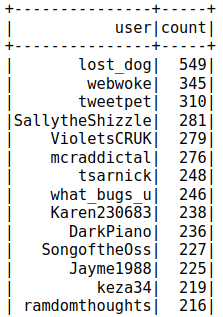
Something significant to point out is that these logical abstractions are not mutually exclusive, actually *Dataframes* are built on top of the *RDDs* and can be easily converted into each other. For the purpose of this assignment, I also used *RDDs* for didactive reasons, unless performance was degraded. This was because *DataFrames* are more intuitive and similar to the usual python for DA. Most operations with *RDDs* were handled with inline functions (*lambda*) which were applied to each element with *map* function.

Also used SQL queries for the same reason, even though there was not an actual connection between spark and this DBMS due to the unresolved conflicts with dependencies.

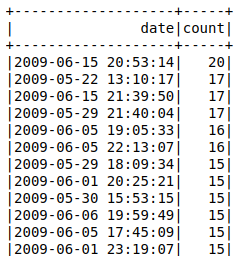
# Data preprocessing and EDA

The dataset was taken to *Hadoop* filesystem and read in to memory as a spark *DataFrame*, solving the issue with PDT timezone by converting the dates with to\_*timestamp* and setting to legacy the *timeParserPolicy*. Found 1685 duplicate ids (duplicate tweets), confirmed duplicates by aggregating all features except index since it is a unique parameter. Once the duplicates were removed there were 1,598,315 rows remaining. And the non-English tweets flagged by *langdetect* were also removed.

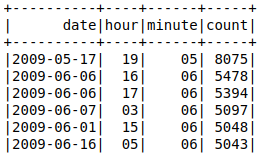
The flag column has one single unique value (NO\_QUERY) which means it does not add anything to the analysis. 659,775 unique users, being the ones with greatest count of tweets as per below:



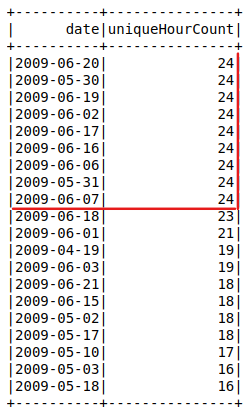
The collection of tweets covers around 2 months and a half in 2009, from 07/04 to 25/06 and the greatest count of tweets per timestamp (HH:MM:SS) is 20. Top counts below:



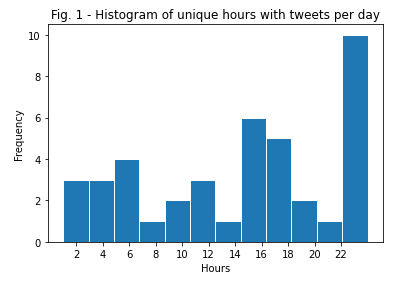
Regarding natural minutes, the greatest count of tweets is above 8 thousand:



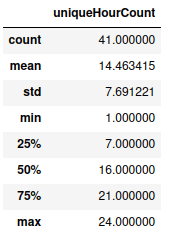
In order to plan the time series, the analysis of the timestamps showed that only 10 days have tweets in each of the 24 hours which make up a day. Considering this, an hourly time series would imply an excessive amount of imputation.



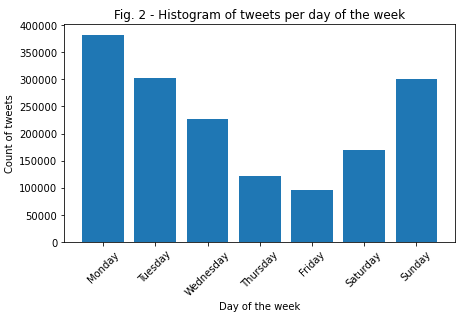
Same as a histogram:



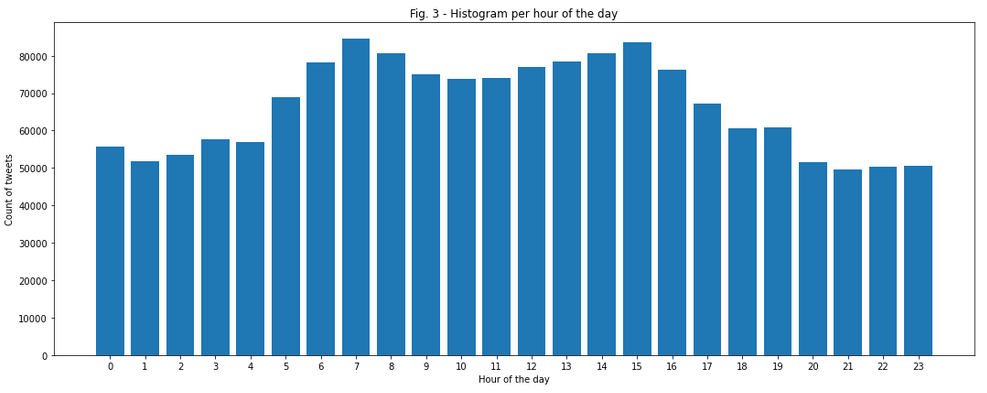
The percentiles of the counts above showed that within the 2 and a half months (with timestamps from 80 days) there are just 41 dates present, being the rest missing. Again this represents a significant amount of imputation. Also noticeable how half of the dates present hold tweets from up to 16 hours.



Sunday, Monday and Tuesday are the days of the week with greater count of tweets:

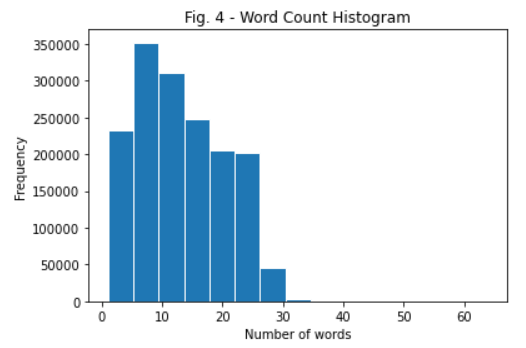


Approximately business hours (UTC due to the time conversion applied) the count of tweets is greater:



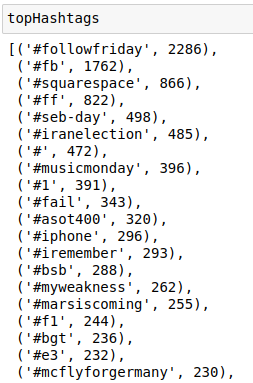
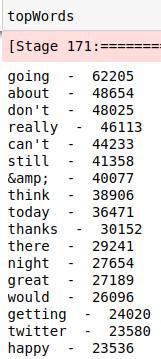
## 2.1 Exploration of text feature

There are 21,052,270 words in the data set with an average of 13.2 words per tweet and a standard deviation of 6.9, facts that are represented in the histogram below:



The average of characters per tweet was 74, just 0.05% of all the words were in upper case, over 8 million of stop words and there were nearly 28 million of special characters (27,809,083).

After the analysis of the most common words and most common hashtags it was concluded that the dataset was not gathered around a topic in particular due to the variety and randomness of hashtags and words. The screenshots below show the ones with greatest counts:

Since there did not seem to be a predominant subject, no sentiment scores were manually assigned to any tweets, which would have been based on, for instance, certain hashtags. For this reason *TextBlob* was the choosen tool to tag the entire set of tweets with an individual sentiment score.

Before giving scores the text was cleaned with funcion which removed:

- user mentions

- hashtags

- URLs

- stopwords (over 8 million)

- some of the most common words which would not be expected to add sentimient, stored in wordsToRemove variable. For example the three most common words were: ‘going’, ‘about’, ‘still’, ‘&amp;’, etc.

- extra white spaces

- single characters (over 27 million)

- special characters, except apostrophes which were between letters n and t. The intention was for the negations to remain (ca**n’t**, should**n’t**, etc) obvious so that *TextBlob* could handle them appropriately.

This function also implemented spelling correction and lemmatization with *WordNetLemmatizer(),* process which reduces words to their base or root form in order to achieve a lower count of unique words. The expected outcome is to improve accuracy of the *NLP* algorithm.

Once the sentiment scores were assgined the performance issues came up when calculating the proportion of positives, negatives and neutral. It seemed due to the function *.count()* and there were multiple attemps to sort this out by converting the *RDD* to a *DF*, getting a list of tuples (date,score), exporting the *RDD* to *HDFS* and importing it into a new *Jupyter Notebook* but none attempts improved performance and the tasks ended up erroring out or making the VM crash after over one day with CPU uilization at 100% constantly.

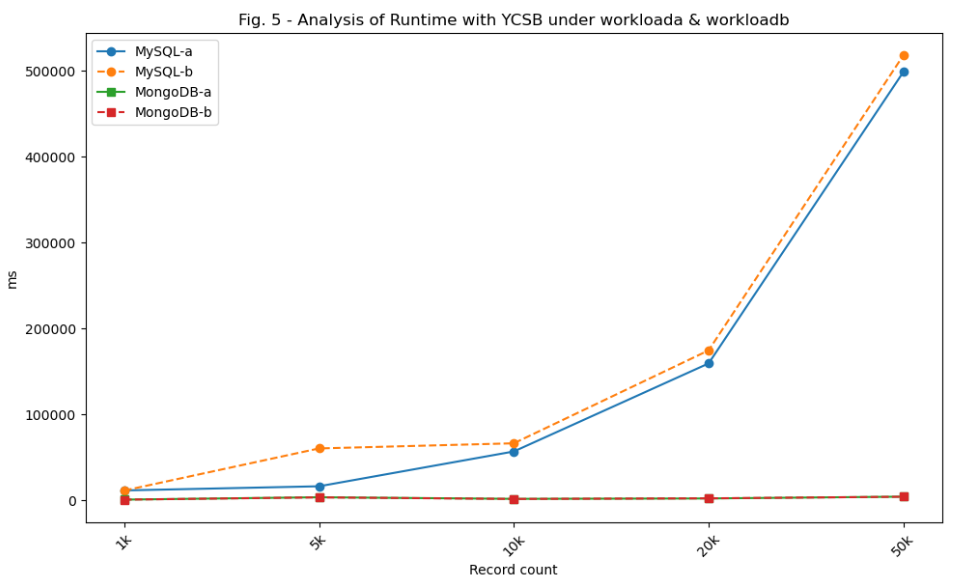
However the process continued with *RDDs* by creating a new one with just date and score for the time series, also hoping that the performance improved but this was not the case. The scores were aggregated by date (natural days) and the corresponding averages were calculated, assigning a score of 2 to the dates which were missing. These scores would have been replaced with quadratic interpolation most likely, but again the conversion to *Spark DataFrame* before transforming it to *pandas df* had a really poor performance.

Once the interpolation was completed I would have checked trend, seasonality and presence of noise. Then autocorrelation and possible significant lags would have determined model and order in the context of ARIMA. Anyhow considering how narrow the time frame of the study dataset is (approximately 2 months and a half), forecasting 1 week going forward may take more raw data for an accurate prediction. And all the more reason for 1 month and 3 month forecasts.

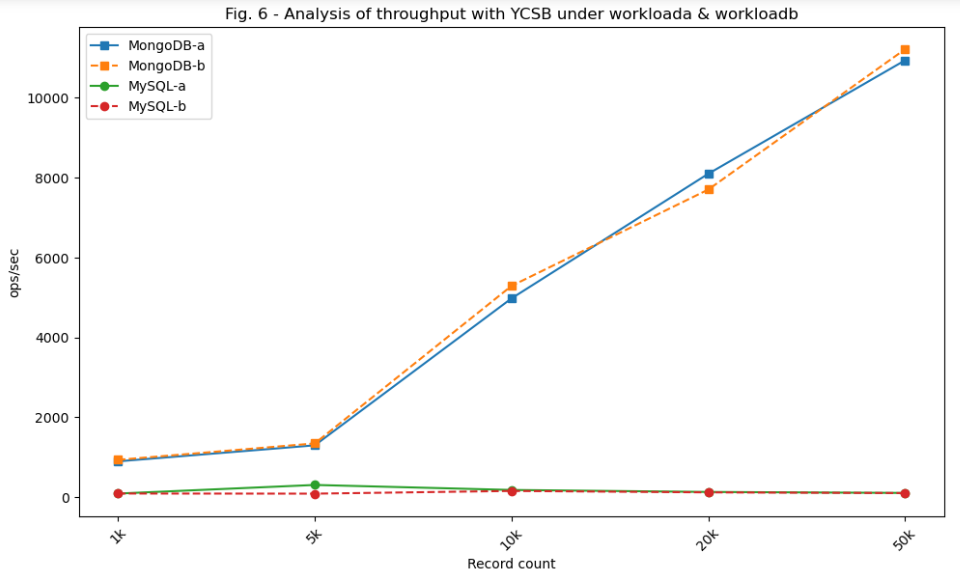
# YCSB

This benchmarking suite was used to compare the performance from two *DBMSs*, *MySQL* (relational) and *MongoDB* (non-relational). They were tested under different workloads (workloada - 50% Read, 50% Update - & workloadb - 95% Read, 5% Update -) within a range of record counts between 1,000 and 50,000. The output of each test was redirected to a csv file, then all csv files were merged and read in as a pandas dataframe. Regarding the charts below, the round markers represent the relational database whereas square markers point to the non-relational one.

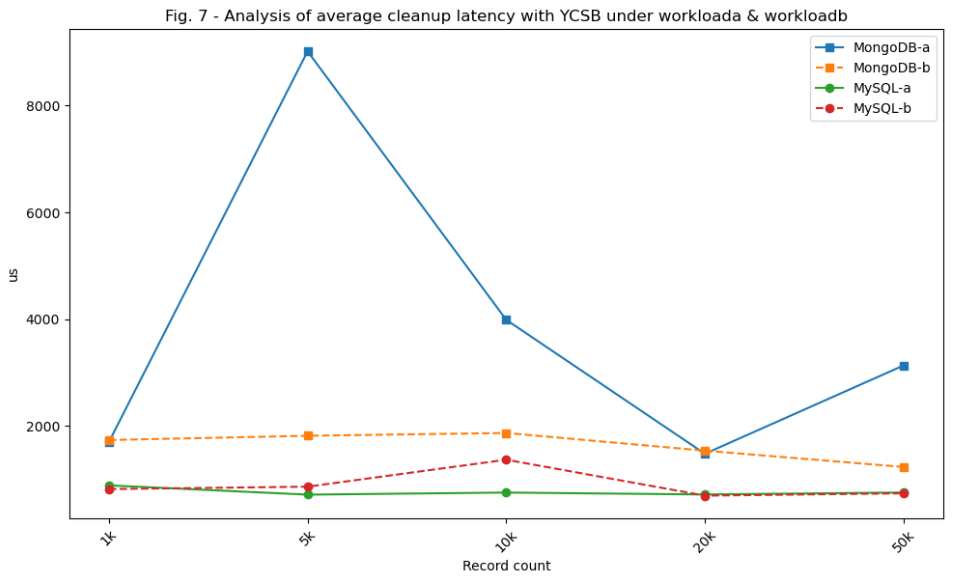
What first stands out is the exponential increment of runtime from *MySQL* as the record count increases, this represents a massive difference between the two *DBMSs* and it is true for both workloads, going above 8 minutes for *MySQL*, whereas *MongoDB* remains below 5 seconds. The consistency of *MongoDB* is represented with a flat line, at least when compared with the other one.



As it could be expected after seeing the above, the throughput of *MongoDB* raises with big steps, especially when there are more than 5,000 records. However *MySQL* remains very consistent at all times regardless of the number of records. The greatest difference at 50,000 records is around 100 vs. 11,000 ops/sec.



However the cleanup latency is worse for mongo, particularly for workloada where it is not consistent at all. There is a very prominent spike at 5K records (~9000 us) but at 10K and 50K it is still high and keeps fluctuating, so it seems like a similar rate of reads and updates affect the performance of cleanups.



Finally the insert latency was lower for *MongoDB* and much more consistent than for *MySQL*. There is a significant difference at 5K for the relational one, as it is below 4ms for workloada but it approaches 12 ms for workloadb. Then this effect that seems to be due to the type of workload becomes much more balanced.

# 

In general terms *MongoDB* performed better than *MySQL* for the two workloads tested and was able to deal with the increase of the record counts in a much more efficient way. It is only behind at the cleanup latency but overall, its runtime is much lower and the throughput greater.

For the dynamic dashboard *ipywidgets* and *matplotlib* were used. It prompts the user to select a performance metric and the corresponding plot is shown accordingly.

# References

Cooper, B.F. *Core workloads*, *GitHub*. Available at: <https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads> (Accessed: 28 July 2023).

Damji, J. (2016) *RDD vs DataFrames and datasets: A tale of three apache spark apis*, *Databricks*. Available at: https://www.databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html (Accessed: 17 July 2023).

*Datetime patterns for formatting and parsing* (no date) *Datetime patterns - Spark 3.4.1 Documentation*. Available at: https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html (Accessed: 13 July 2023).

NK, N. (2023) *Spark Performance Tuning & Best Practices*, *Spark By {Examples}*. Available at: https://sparkbyexamples.com/spark/spark-performance-tuning/?expand\_article=1 (Accessed: 9 July 2023).

Soni, A. (2021) *User defined functions(UDF) in pyspark*, *Medium*. Available at: <https://medium.com/analytics-vidhya/user-defined-functions-udf-in-pyspark-928ab1202d1c> (Accessed: 10 July 2023).