Twitter Sentiment Analysis using Lambda Architecture

Lorenzo Gianassi
lorenzo.gianassi@tud.unifi.it

Abstract

This paper focuses on the study and implementation of a Lambda Architecture to perform Sentiment Analysis on Twitter data in real-time. The objective of this project is to demonstrate the ability of this structure to manage the large amount of data used by a task such as Sentiment Analysis.

1. Introduction

Social networks such as Twitter have gained great attention nowadays. The heavy presence of people on social networks causes a big amount of data ready to be analyzed. However, there are a lot of issues related to the reviewing of large chunks of information. Sentiment analysis (also known as opinion mining or emotion AI) is the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods. This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. So the goal of this project is to solve the sentiment analysis problem on a set of tweets based on the Lambda Architecture structure to process large amounts of data.

1.1. Lambda Architecture structure

Lambda architecture describes a system consisting of three layers: **Batch Layer** to perform processing, **Speed Layer** to real-time processing, and a **Serving Layer** for responding to queries.

1. **Batch Layer**: The batch layer precomputes results using a distributed processing system that can handle very large quantities of data. It uses batch-oriented technologies like MapReduce to precompute batch views from historical data and this is effective but latency is high. Output is typically stored in a read-only database, with updates completely replacing existing precomputed views.

2. **Speed Layer**: The speed layer processes data streams in real time. This layer sacrifices throughput as it aims to minimize latency by providing real-time views into the most recent data. Essentially, the speed layer is responsible for filling the ”gap” caused by the batch layer’s lag in providing views based on the most recent data.

3. **Serving layer**: Output from the batch and speed layers are stored in the serving layer, which responds to adhoc queries by returning precomputed views or building views from the processed data.

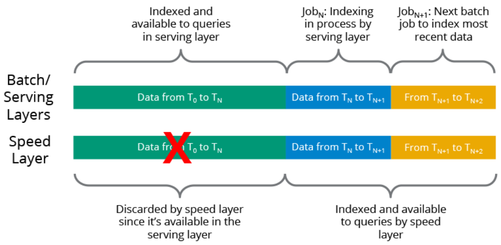

Indexed and available to queries in serving layer
Job_N: Indexing in process by serving layer
Job_{N+1}: Next batch job to index most recent data
Batch/ Serving Layers
Data from T_0 to T_N
Data from T_N to T_{N+1}
From T_{N+1} to T_{N+2}
Speed Layer
Data fr T_0 to T_N
Data from T_N to T_{N+1}
From T_{N+1} to T_{N+2}
Discarded by speed layer since it’s available in the serving layer
Indexed and available to queries by speed layer

Figure 1: Data is indexed simultaneously by both the Serving Layer and the Speed Layer.

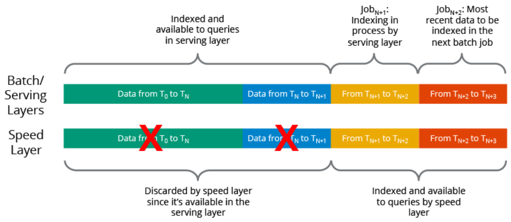

Indexed and available to queries in serving layer
Job_{N+1}: Indexing in process by serving layer
Job_{N+2}: Most recent data to be indexed in the next batch job
Batch/ Serving Layers
Data from T_0 to T_N
Data from T_N to T_{N+1}
From T_{N+1} to T_{N+2}
From T_{N+2} to T_{N+3}
Speed Layer
Data fr T_0 to T_N
Data fror T_N to T_{N+1}
From T_{N+1} to T_{N+2}
From T_{N+2} to T_{N+3}
Discarded by speed layer since it’s available in the serving layer
Indexed and available to queries by speed layer

Figure 2: When the Serving Layer completes a job, it moves to the next batch and the speed layer discards its copy of the data that the Serving Layer just indexed.

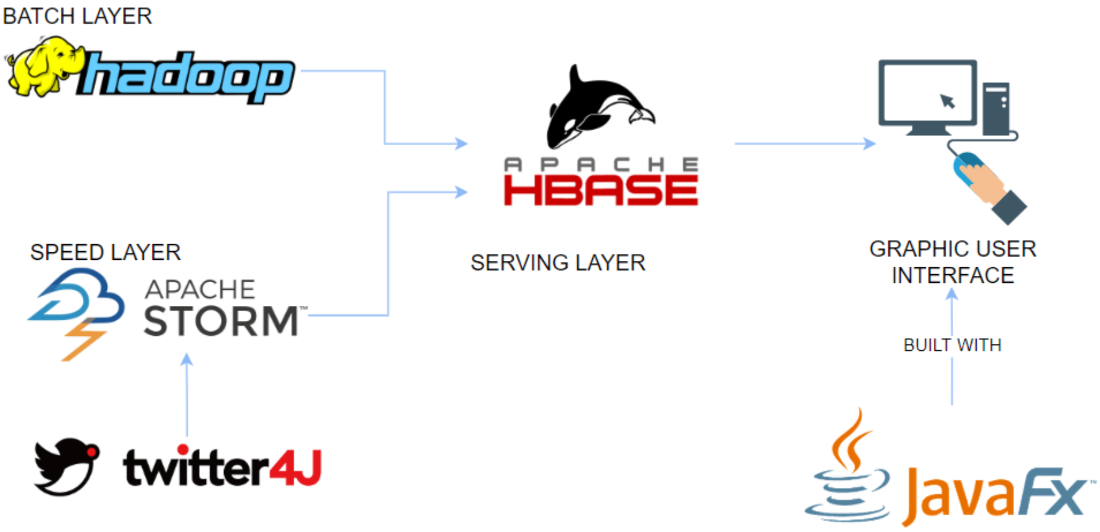


Figure 3: Structure of the proposed Lambda Architecture.

The batch/serving layers continue to index incoming data in batches. Since the batch indexing takes time, the speed layer complements the batch/serving layers by indexing all the new, unindexed data in near real-time. This gives you a large and consistent view of data in the batch/serving layers that can be recreated at any time, along with a smaller index that contains the most recent data. Once a batch indexing job completes, the newly batch-indexed data is available for querying, so the speed layer's copy of the same data/indexes is no longer needed and is therefore deleted from the speed layer. The serving layer then begins indexing the latest data in the system that had not yet been indexed by this layer, which has already been indexed by the speed layer (so it is available for querying at the speed layer). This ongoing hand-off between the speed layer and the batch/serving layers ensures that all data is ready for querying and that the latency for data availability is low. A graphic representation of the data updating process can be seen in the Figures 1 and 2.

2. Implementation

Let's describe the implementations of the components that make up the Lambda Architecture: Serving Layer, Batch Layer, Speed Layer. In particular, the implementation was performed using **Apache Hadoop** for the Batch Layer, **Apache Storm** for the Speed Layer and finally **Apache Hbase** for the Serving Layer. We will run Hadoop in *Pseudo-Ditributed* mode on a single cluster. The pseudo-distributed mode is also known as a single-node cluster where both *NameNode* and *DataNode* will reside on the same machine, that means all the Hadoop daemons will be running on a single node. The Speed Layer represents the main part of this project, in fact it is the first component to be started. In addition to create the topology, it takes as input the keywords on which the sentiment analysis will be performed and creates the tables that will be used by the Serving Layer. To simulate a sentiment analysis process on a tweets stream, it was necessary to work with the Twitter API using Twitter4j. It was necessary to save in a text file the developer access keys to the stream, this file is provided empty in the repo of this project and anyone who wants to access the stream can enter their Developer Credentials.

2.1. Serving Layer

The Serving Layer was built on Apache Hbase [1].The tables used by this module are those created in the speed layer topology, which we will analyze in the next subsection. This layer is composed by the following tables:

- **Master Database**: it represents the master database of the Lambda Architecture. Each row contains the text,

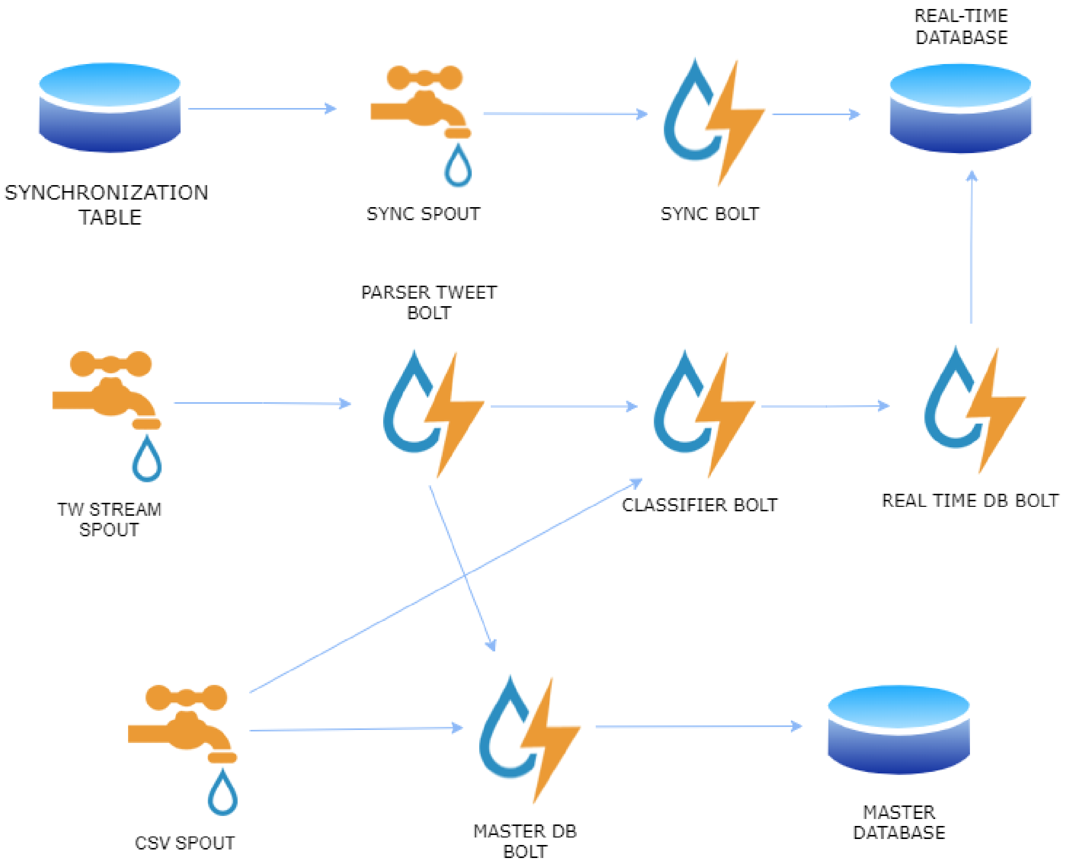


Figure 4: Topology of the proposed Speed Layer.

the keywords and the ID of each tweet.

- **Real-time Database**: it is the database for the real-time part where each row contains the keyword and the related sentiment for each tweet and the timestamps used to perform synchronization.
- **Synchronization**: it contains only two lines that respectively indicate the start and end timestamps of a batch processing. These are used precisely in the synchronization between the batch and the speed layer, particularly useful since they are used to discard the rows corresponding to tweets already processed by the Batch Layer.
- **Batch view**: it is the result of the computation of the batch layer. It contains a row for each keyword and the number of positive and negative sentiments associated to it.

2.2. Classifier Model

Sentiment Analysis requires the text to be classified by assigning a value to various keywords that will correspond to a sentiment. It will therefore be necessary to create and train a classification model. The realization is carried out by making use of the LingPipe library [2]. This library provides the main methods for the training and use of the classifier. For the training phase has been used the dataset [3], while the other dataset [4] is used for testing the accuracy of the just trained model. At the start of the computation is chosen a set of keywords, upon which is done the sentiment analysis. If in a certain tweet is present at least one of this keywords, then the tweet must be classified in order to provide the proper sentiment. The classifier is trained in such a way that is capable to recognize two levels of sentiment:

- *Positive* sentiment, with label 1.
- *Negative* sentiment, with label 0.

Once trained, the model is saved in a file that will be used later in the Speed Layer by the ClassifierBolt for the classification phase and in the Batch Layer. Furthermore a classification test was carried out obtaining a value of 67%. This result obtained is not excellent but the test carried out did not represent the focus of the project.

2.3. Speed Layer

The core of the project is represented by the **Speed Layer**, made with **Apache Storm** [5]. At the beginning of the execution the program takes the keywords as arguments and creates all the tables of the serving layer (if they do not already exist). The idea behind Storm is to use a series of Spouts and Bolts to allow distributed processing of streaming data. A Storm application is designed as a **topology** in the shape of a directed acyclic graph with spouts and bolts acting as the graph vertices. The topology acts as a data transformation pipeline.
Let's analyze the components of the Topology shown in Figure 4:

- **TwStreamSpout**: it uses the twitter4j library and the Twitter Streaming API to get a real-time stream of tweets. It filters the tweets to retrieve only the ones that contain at least one of the keywords and that are written in English.

- **CSVSpout**: : this spout in this work it is used to increment the number of tweets computed by the Lambda Architecture. For each tweet of the dataset [4] it outputs a tuple with the same structure used by TwStreamSpout.

- **ParserTweetBolt**: : it takes a tweet object in input coming from *TwStreamSpout* and parse it to output a tuple containing the ID, the text and the keywords of the tweet.

- **ClassifierBolt**: it classifies the text of the tweet contanied inside the tuple generated by *ParserTweetBolt* using the already trained sentiment classifier model. Then, for each keyword in the text it outputs a tuple containing the keyword and the deducted sentiment.

- **SyncSpout**: takes care of checking if the batch processing start and end timestamps have both remained unchanged since the last check. If not, it outputs a tuple containing the start timestamp to the related bolt.

- **SyncBolt**: as soon as a timestamp is received from the aforementioned spout, it deletes the rows with timestamps prior to the one received from the real-time tweet table, as long as they exist.

- **MasterDbBolt**: it inserts a row in the master database of the architecture for each tuple that arrives, so that the batch will also have real time tweets.

- **RealTimeDbBolt**: for each tuple generated by *ClassifierBolt* it inserts a row containing the values in the appropriate real-time database.

2.4. Batch Layer

The batch layer is composed by three modules: Driver, Mapper and Reducer of the Hadoop system respectively. This layer is represented by Apache Hadoop [6], its goal is to run in an infinite loop and compute a MapReduce job on tweet master database, and then write its results from scratch in batch view. It also writes the timestamps of the beginning and of the end of the execution in synchronization table. **MapReduce** is a framework composed of a map procedure, which executes some filtering and sorting, and a reduce method, which performs a counting operation. MapReduce is so important because it allows to orchestrates the processing of a massive amount of data, even distributed on different servers. Also it does so running the various tasks in parallel and managing the communications between the different servers, but still guaranteeing redundancy and fault tolerance. Therefore the main idea of MapReduce is to ensure scalability and fault tolerance. So the *Batch Layer* is composed by three elements:

- **BatchDriver**: it executes a MapReduce job on the master database, writing the results obtained in the relative batch view table. It also takes note of the start and end processing timestamps in the appropriate table, so that, when the batch layer finishes its execution, it is possible to discard the data processed by it from the real-time view through the synchronization mechanism and the rows inserted before the start of the batch run will be discarded from the real-time table.

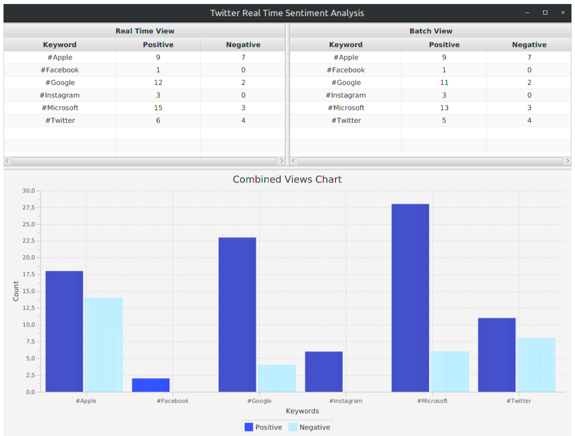
- **BatchMapper**: at each iteration it reads the text and the keywords of each tweet from tweet master database. Then it classifies the text by recalling the appropriate method of the model trained previously and produces, for each keyword of the tweet, a tuple structured as [*Keyword*, *Sentiment*].

- **BatchReducer**: at the end of the Mapper execution, the Reducer receives a pair of values for each of the considered classifications, where, in particular, the second term represents the list of sentiment related to the key. Therefore, a positive/negative sentiment is counted and its value increased accordingly in the batch view.

2.5. Graphic User Interface

A GUI (Graphic User Interface) has been implemented to obtain an intuitive display of the results. This Graphic User Interface was developed with Java FX.
It is composed by two tables, representing respectively the content of the real-time view and the batch view, and a bar chart, which combines the data of the two views and to display the results of the query made. The data of the batch view is retrieved directly from the batch view table, as each

of its rows represents the number of positive and negative sentiments associated to a specific keyword. For real-time view is necessary to aggregate the rows of the real-time database with the same keyword and sentiment and count them.



Twitter Real Time Sentiment Analysis

Real Time View

Keyword	Positive	Negative
#Apple	9	7
#Facebook	1	0
#Google	12	2
#Instagram	3	0
#Microsoft	15	3
#Twitter	6	4

Batch View

Keyword	Positive	Negative
#Apple	9	7
#Facebook	1	0
#Google	11	2
#Instagram	3	0
#Microsoft	13	3
#Twitter	5	4

Figure 5: Display of the Graphical User Interface.

- **GUIController**: : is the GUI software, which retrieves the information from the batch and real-time views and refreshes the data every second, in order to keep the GUI always updated.
- **GUIlauncher**: it is the class that contains the code to start the interface and it details its style.

A representation of the GUI can be seen below in the *Figure 5*.

3. Conclusions

In conclusion, we can say that we have obtained the desired result described at the beginning of the paper. We were able to complete the Sentiment Analysis task that had to process a large amount of data thanks to the use of the Lambda Architecture which was responsible for managing and processing the expected amount of data in parallel. So the Lambda Architecture is capable of handling *Sentiment Analysis* statistics of real-time Tweets. Furthermore thanks to the Lambda Architecture it has been possible to achieve improvements in various aspects such as: latency, data consistency, scalability, fault tolerance, and human fault tolerance. In addition, a classification value was also provided with regard to Sentiment Analysis although it is main focus of the project.

3.1. Future Developments

A possible future development could be to introduce new classes of sentiment corresponding to intermediate values. By introducing these new values it would be possible to obtain a more complete and exhaustive Sentiment Analysis.
A Pseudo-Distributed approach on a single node was chosen for this project, so the problem of Sentiment Analysis could be addressed using a Fully-Distributed approach.

References

[1] Apache Software Foundation, “Hbase,” 2.3.4, https://hbase.apache.org.

[2] Alias-i, “Lingpipe 4.1.0,” 2008, http://alias-i.com/lingpipe.

[3] A. Go, R. Bhayani, and L. Huang, “Twitter sentiment classification using distant supervision,” *CS224N project report, Stanford*, vol. 1, no. 12, p. 2009, 2009, https://www.kaggle.com/kazanova/sentiment140.

[4] N. J. Sanders, “Twitter sentiment corpus,” 2011, https://github.com/guyz/twitter-sentiment-dataset.

[5] Apache Software Foundation, “Storm,” 2.1.0, https://storm.apache.org.

[6] Apache Software, “Hadoop,” 3.2.1, https://hadoop.apache.org.