Lambda Architecture for Twitter image processing analysis

Alessio Venturi
University of Florence
alessio.venturi@stud.unifi.it

Abstract

In today's modern society, the importance of social networks and Big Data is growing exponentially. For this reason, the need to find ways to analyze these huge amounts of data becomes fundamental. The Lambda Architecture was developed specifically to resolve this problem in an efficient and reliable way. The aim of this work is to present a Lambda Architecture to perform an image processing analysis on Twitter data in real-time using some of the frameworks of the Apache suite: Hadoop, HBase and Storm.

1 Introduction

The advent of Big Data has brought changes in data processing over recent years. Big Data differs from traditional data processing through its use of parallelism: only by using multiple computing resources it is possible to process terabytes of data. The Lambda Architecture is a particular approach popularized by Nathan Marz[5] that combines the large-scale batch-processing strengths of MapReduce with the real-time responsiveness of stream processing. The main goal is to create scalable, responsive and fault-tolerant solution to Big Data problems. There's no single tool that provides a complete solution. Instead, you have to use a variety of tools and techniques to build a complete Big Data system. The main idea of the Lambda Architecture is to build Big Data systems as a series of layers. Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it.
These layers are :

- **Batch Layer**: It needs to be able to do two things: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset. It uses batch-oriented technologies like MapReduce to precompute batch views from historical data and this is effective but latency is high.
- **Serving Layer**: It is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. A serving layer database supports batch updates and random reads. Most notably, it does not need to support random writes.
- **Speed Layer**: It only looks at recent data, whereas the batch layer looks at all the data at once. It updates the real-time views as it receives new data instead of recomputing the views from scratch like the batch layer does.

So the Lambda Architecture can be summarized by these three equation :

- batch view = function(all data)
- real-time view = function(real-time view, new data)
- query = function(batch view, real-time view)

Instead of resolving queries by just doing a function of the batch view, you can resolve queries by looking at both the batch and realtime views and merging the results together. Real-time views contain only information derived from the data that arrived since the batch views were last generated and are discarded

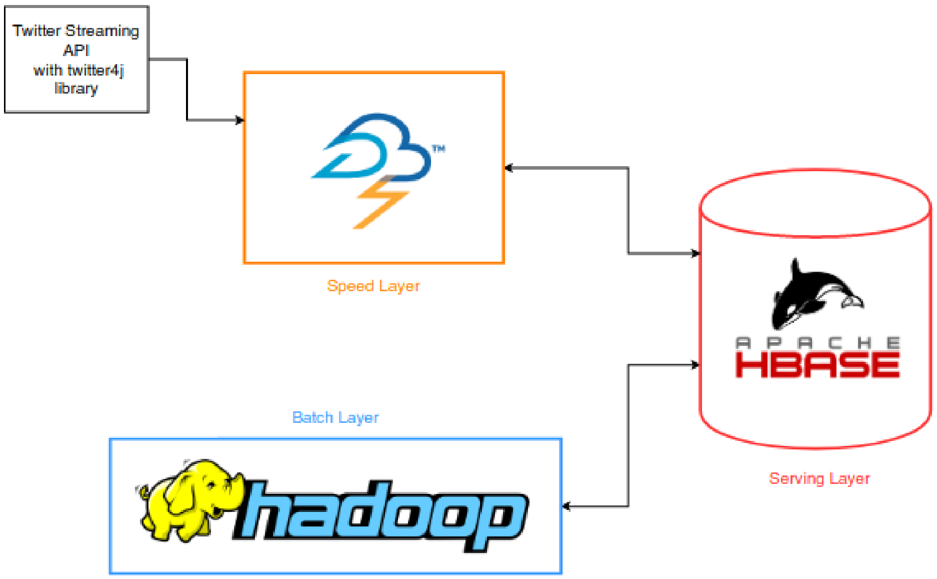


Figure 1: Lambda Architecture Structure

when the data they were built from is processed by the batch layer. The batch and real-time views are combined to create query results.
The analysis of the image processing in this project, is used to find from a given set of image data obtained from Twitter (joined by the same hashtag), the most representative image / s of the set of images analyzed, through the implementation of Kmeans with features vector obtained with Lire Library.

2 Proposed Approch

The main goal of this project is the implementation of a Lambda architecture capable of efficiently providing the analysis statistics of the image processing found on the tweets in real time. This project has done completely with Java and Apache Hadoop[1], Apache Storm [3] and Apache HBase [2] have been executed in a pseudo-distribuited mode on a local cluster 2. The speed layer represents the core of this work and it is launched first. It takes as arguments the keyword on which the image processing will be performed by the whole architecture and creates the tables of the speed layer (in our case are 4) . Then the batch layer is started so that the Lambda Architecture is complete. The keyword is used to find tweets with image containing the hashtag (ie. #apple), then from each of these image, we will extract the feature vector with Lire [4]. The structure of Lambda Architecture proposed is shown in $figure$ 2.

2.1 KMeans

The K-means clustering is a method of vector quantization, originally from signal processing, that aims to partition ”n” observations into ”k” clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. To belong to a center, the squared Euclidean distance from that center must be less than the distance from the other centers. We will use the K-Means algorithm inside the MapReduce of Hadoop.

2.2 Image processing with LIRE

As said above, the keyword is used to find tweet with image/s containing the #keyword using the Twitter4J library. Then, from each image found, we will save it into a folder and we will extract the feature vector with Lire library obtaining a 144-dimension vector. LIRE (Lucene Image Retrieval) is an open source library for content based image retrieval, which means you can use LIRE to implement applications that search for images that look similar. These vector will be used in the MapReduce Kmeans to find ”k” center of ”k” cluster, from which we will obtain the closest image to the center using the squared Euclidean Distance.

2.3 Serving Layer

The serving layer uses Apache HBase. The tables are created by the speed layer at the beginning of the execution. There are 4 tables:

- **Tweet master database**: it represents the master database of the Lambda Architecture. Each row contains the path_img, the keywords and the ID of each tweet

- **Tweet real-time database**: it contains only the information regarding the tweets that arrived since the batch view was last generated. Each row contains the keyword , the feature vector and the path_img to a certain tweet and, as every other thing that is stored in HBase, the timestamp in which the row was inserted in the database. In this case the timestamp is particularly useful because it is used to discard the rows corresponding to tweets already processed by the batch layer.

- **Tweet batch view**: it is the result of the computation of the batch layer. It contains ”k” center that show us the closest path_img and the feature vector of it.

- **Synchronization table** : it is composed by only 2 rows that represent respectively the timestamps of the start and the end of the batch processing. These timestamps are used to synchronize the batch and the speed layer.

2.4 Batch Layer

The batch layer is represented by Apache Hadoop, that runs in an infinite loop and computes a Map Reduce job, where we developed the K-Means algorithm, on tweet master database, and then it writes 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.
Before to run the mapper, we create ”k” center from the first ”k” image inside the master database and record them in a file.

To simply the execution, we created two classes: Point and Center.
Point have as parameters :

- listOfFeatures

- index

- path

Center extends Points and it has two more parameters :

- indexCenter

- numberOfPoints

After that, the Mapper reads the path_img of each tweet from ”tweet master database” and then extract the feature vector using Lire library. Now each tweet will be a point containing its feature vector and the path_img. Then for each ”Point”, the Euclidean quadratic distance is calculated to assign the closest center to the point. The output will be a tuple structured as $<$ Center, Point $>$.
The combiner works on the points associated with the same center. It recreates a Point object where all the characteristic vectors are summed and saves the number of points associated with each center. At the same time, each point is saved in a list to keep track of, indicating which center it is associated with.
The Reducer takes a tuple <Center, Point> in input and makes a further sum of the partial results obtained by the combiner. Inside we have two HashMaps that keep track of old and new centers. In the cleanup phase, the results are concretely updated by dividing the coordinates with respect to the number of points in the cluster. The convergence condition is divided into two constraints::

- The converging centers are greater than or equal to 90% (the distance between the old and the new value does not exceed the threshold)

- The average of the distances between the old and the new values does not exceed the threshold. The average is calculated by taking the square root of the sum of the squared distances, divided by K.

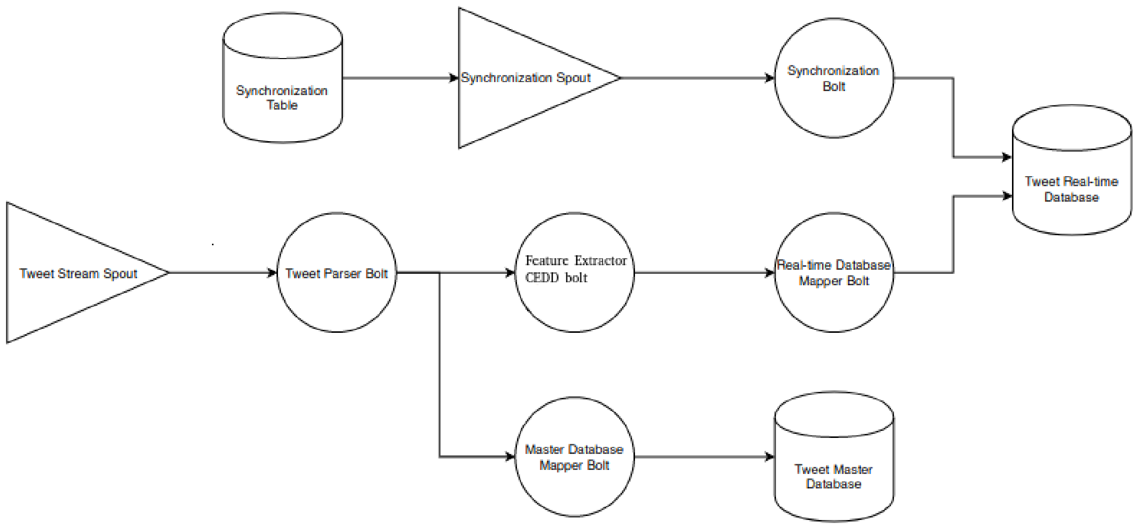


Figure 2: Storm Topology

If either of these is met, then a global variable is incremented.

All this is controlled by a Driver class, which has the task of setting the configuration, managing the job and creating the initial centers. The main task of the Driver is to start the Kmeans loop, where the MapReduce takes place. The classes used and the input and output elements are initialized. After each ”waitForCompletation”, it is checked that the general counter contains the convergence value. If this value is not equal to 1, then we must continue iterating. An additional exit condition has been set even if the iterations exceed ”n”.Q At each cycle the batch view table is updated with the current centers and the images closest to them.

2.5 Speed Layer

The core of the speed layer is Apache Storm. The program takes the keyword as argument and creates all the tables of the serving layer at the beginning of the execution (if they do not already exist). To discard the data already processed by the batch layer from the real-time view it was necessary to implement synchronization spout and synchronization bolt, combined with synchronization table. In this way, when the batch layer terminates its execution, the rows inserted before the start of the batch computation are no longer part of the real-time view. At the same time, the rows inserted during the batch computation are not discarded and still are a part of the real-time view. The Storm Topology is formed as follow :

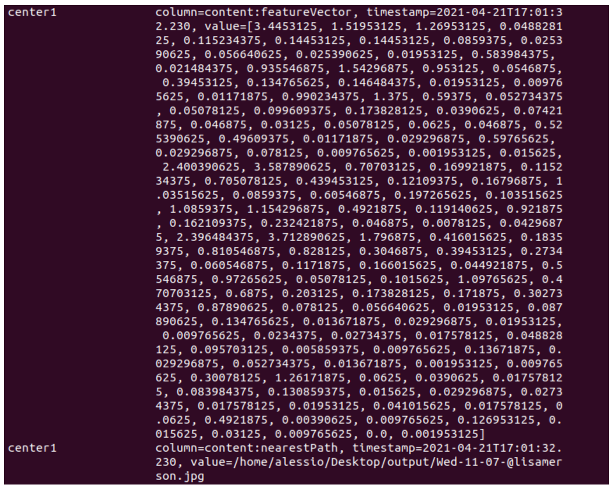
- **Tweet Spout** : it uses the twitter4j library and the Twitter Streaming API to get a real-time stream of tweets. It filters the tweets to retrieve only the ones that contain images and the keyword.
- **Tweet Bolt** : it takes a tweet object in input and parse it to output a tuple containing the ID, the keyword of the tweet and the path_img, where it will be stored.
- **Master DB Mapper Bolt**: it inserts a row in tweet master database for each parsed tweet.
- **Feature Extractor CEDD Bolt** : it uses the Lire library to extract the feature vector from the path_img of the tweet. Then, it outputs a tuple containing the keyword , the path_img and the feature vector.
- **Real-time DB Mapper Bolt** :for each tuple outputted by Feature Extractor CEDD Bolt, it inserts a row in tweet real-time database.
- **Synchronized Spout** : it checks the two rows of the synchronization table and when both the

start and the end timestamps are modified, it outputs a tuple containing the start timestamp.

- **Synchronized Bolt** : it takes the start timestamp outputted by synchronization spout and it deletes the rows of tweet real-time database with a preceding timestamp.

3 Result

To test this project, we tried using it with some words like ”#dog” and ”#apple”. Whenever the images found exceeded hundreds, but due to the (possibly wrong) use of hashtags, photos were found that did not refer to the real word. Despite this, the results obtained are always acceptable. In both tests, the centers searched corresponded to the centers obtained (3 required / 3 obtained). By re-running the program several times on the same images, the centers have not changed, confirming the acceptability of the output. In the figure below 3, you can see one final center obtained.



```
center1                 column=content:featureVector, timestamp=2021-04-21T17:01:3
                        2.230, value=[3.4453125, 1.51953125, 1.26953125, 0.0488281
                        25, 0.115234375, 0.14453125, 0.14453125, 0.0859375, 0.0253
                        90625, 0.056640625, 0.025390625, 0.01953125, 0.583984375,
                        0.021484375, 0.935546875, 1.54296875, 0.953125, 0.0546875,
                         0.39453125, 0.134765625, 0.146484375, 0.01953125, 0.00976
                        5625, 0.01171875, 0.990234375, 1.375, 0.59375, 0.052734375
                        , 0.05078125, 0.099609375, 0.173828125, 0.0390625, 0.07421
                        875, 0.046875, 0.03125, 0.05078125, 0.0625, 0.046875, 0.52
                        5390625, 0.49609375, 0.01171875, 0.029296875, 0.59765625,
                        0.029296875, 0.078125, 0.009765625, 0.001953125, 0.015625,
                         2.400390625, 3.587890625, 0.70703125, 0.169921875, 0.1152
                        34375, 0.705078125, 0.439453125, 0.12109375, 0.16796875, 1
                        .03515625, 0.0859375, 0.60546875, 0.197265625, 0.103515625
                        , 1.0859375, 1.154296875, 0.4921875, 0.119140625, 0.921875
                        , 0.162109375, 0.232421875, 0.046875, 0.0078125, 0.0429687
                        5, 2.396484375, 3.712890625, 1.796875, 0.416015625, 0.1835
                        9375, 0.810546875, 0.828125, 0.3046875, 0.39453125, 0.2734
                        375, 0.060546875, 0.1171875, 0.166015625, 0.044921875, 0.5
                        546875, 0.97265625, 0.05078125, 0.1015625, 1.09765625, 0.4
                        70703125, 0.6875, 0.203125, 0.173828125, 0.171875, 0.30273
                        4375, 0.87890625, 0.078125, 0.056640625, 0.01953125, 0.087
                        890625, 0.134765625, 0.013671875, 0.029296875, 0.01953125,
                         0.009765625, 0.0234375, 0.02734375, 0.017578125, 0.048828
                        125, 0.095703125, 0.005859375, 0.009765625, 0.13671875, 0.
                        029296875, 0.052734375, 0.013671875, 0.001953125, 0.009765
                        625, 0.30078125, 1.26171875, 0.0625, 0.0390625, 0.01757812
                        5, 0.083984375, 0.130859375, 0.015625, 0.029296875, 0.0273
                        4375, 0.017578125, 0.01953125, 0.041015625, 0.017578125, 0
                        .0625, 0.4921875, 0.00390625, 0.009765625, 0.126953125, 0.
                        015625, 0.03125, 0.009765625, 0.0, 0.001953125]
center1                 column=content:nearestPath, timestamp=2021-04-21T17:01:32.
                        230, value=/home/alessio/Desktop/output/Wed-11-07-@lisamer
                        son.jpg
```

Figure 3: Feature Vector and nearest path of one of the three centers

4 Conclusion

In this work, it has been shown an implementation of Lambda Architecture capable of getting the most representative image of real-time tweets using several techniques together. The final result can be seen on ” owner’s Hbase Shell”, where you can see also the tables used in this project. We have encountered several problems connecting the various technologies and versions, in fact Apache Hadoop is not the best technology for KMeans. Furthermore, the same project was tested on a Windows platform, where, however, we did not obtain the desired results.

4.1 Future Development

Future work may implement another technology as Apache Mahout, to see the difference between it and Hadoop. A GUI (**Graphics User Interface** could be implement to see ,without ”hbase shell” , everything about the center (# number of points for each center, which points are connected with a certain center etc.). Moreover, it could try to filter the images relevant to a particular hashtag so that you don’t have images that are completely out of place.

References

[1] Apache hadoop. `https://hadoop.apache.org/`.

[2] Apache hbase. `https://hbase.apache.org/`.

[3] Apache storm. `https://storm.apache.org/`.

[4] Lire : Lucene image retrieval. `https://github.com/dermotte/LIRE/`.

[5] J. W. Nathan Marz. Big data: Principles and best practices of scalable real-time data system. 2015.