Lambda architecture for Twitter sentiment analysis

Ettore Celozzi
E-mail address
ettore.celozzi@stud.unifi.it

Luca Ciabini
E-mail address
luca.ciabini1@stud.unifi.it

Abstract

Twitter is a social networking service on which users post and interact with messages known as "tweets".
Every second, on average, around 6,000 tweets are tweeted on Twitter, which corresponds to over 350,000 tweets sent per minute, 500 million tweets per day and around 200 billion tweets per year.
Sentiment Analysis is the process of determining whether a piece of writing (a tweet in our case) is positive, negative or neutral. It can be used to identify the customer or follower's attitude towards a brand through the use of variables such as context, tone, emotion, etc. Marketers can use sentiment analysis to research public opinion of their company and products, or to analyze customer satisfaction. Organizations can also use this analysis to gather critical feedback about problems in newly released products.
Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods.
In this paper we implement a lambda architecture in order to find the sentiment of a list of keywords with respect to time.

1. Introduction

Twitter is a huge network and due to the massive quantity of data (tweet in our case) produced every second the process of information extraction is really difficult.
Big Data differs from traditional data processing through its use of parallelism: only by bringing multiple computing resources together we can process terabytes of data but often this is not enough.
Suppose you want to process all the tweet of the year (approximately 200 billion) until now. Distributing both data and computation over clusters of tens or hundreds of machines maybe is possible to process all the tweet in few hours, but the results would ignore all the tweets arrived during the computation and this, for some kind of requests, is not acceptable.
So the main idea of the Lambda Architecture is to build Big Data systems as a series of layers:

- Batch layer that is responsible of storing immutable, constantly growing master dataset(raw data), and compute arbitrary functions on the dataset. This type of processing is best done using batch-processing systems: in our case **Hadoop**;
- Fast layer that ensure new data is represented in query functions as quickly as needed for the application requirements. The speed layer only looks at recent data, whereas the batch layer looks at all the data at once. Whe used **Storm** for the implementation of this layer;
- Serving layer that is a specialized distributed database. The serving layer takes both data from the batch layer and the fast layer, keeping it consistent and provides a global view. In our case we used **Cassandra** as distributed db.

Our objective is to simulate the API of twitter using a dataset of over a million tweet and get a chart of the sentiment with respect to time of a given set of keywords using our lambda architecture. All the architecture will run on a small cluster of machines in AWS.

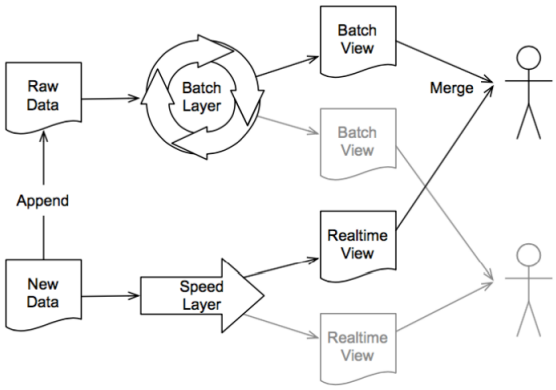

Figure 1. Lambda architecture representation

2. Sentiment analysis

The sentiment analysis performed by the software either in batchlayer or in fastlayer, uses the library NLP developed by the Stanford university. The function $NLP.findSentiment(tweet)$ for a given tweet can return:

- -1 if the sentiment is negative;
- 1 if the sentiment is neutral;
- 1 if the sentiment is positive

3. Batch Layer

This is the first of the three layers of the Lambda Architecture.
The core of this tier is the **masterdataset**, the place where all the tweets are stored. Data is saved as "raw", so that is possible to implement further queries on data with low effort. In our case, a **Tweet class** define the structure of tweets through three attribute:

- **tweet**: the sequence of words of the tweets;
- **timestamp**: hours, minutes and seconds of tweet publication;
- **date**: the date (gg/mm/aaaa) of tweet publication;

Although only tweets are needed to perform sentiment analysis, to maintain the rawness also timestamp and date are saved.
The main aim of the batch layer is to guarantee **fault-tolerance** either from human or machine. The first of this peculiarity is obtained through **immutable** data: once a (raw) tweet is appended, there is no way to modify this. In such a way *no data can be lost*, in fact if bad data is written, earlier good data still exist, so is pretty easy to retrieve good data deleting those wrong.
The second one is achieved thanks to a common strategy: machine and data replication. The first means **machine cluster**, instead, to accomplish data duplication **Hadoop distributed filesystem**(HDFS) is employed. A normal filesystem in fact, is not able to handle the fast grown of masterdataset's dimension (**scalability**) and neither to guarantee writing/reading efficiency. Moreover the HDFS support parallel processing that further increase the speed.

3.1. Pail

As suggested from [1], **Pail** is used to abstract HDFS complexity. This is an open-source library, that yield easy perform operations on HDFS.
The methods used to write to and read from HDFS involves two important tasks:

- **Serialization**: define the way to serialize (encode) objects to binary data;
- **Deserialization**: specify the way to retrieve the object from binary data.

This two operations are performed automatically from pail if the structure of the pail folder is defined like a *Pail Structure*, in our case *Tweet Pail Structure*. Pail objects are templates, so to create the own pail folder, is needful to "cast" the pail object to the own type, that is **Tweet**. To notice that all the pail operations are performed through Map-Reduce so the scaling property of masterdataset is assured.

3.2. JCascalog

Once the tweets are stored in the HDFS is possible to query the masterdataset to perform the sentiment analysis task. This operation is not

straightforward, in fact query directly the masterdataset could request massive time depending on dataset dimensions. The smarter way to perform this job is to precompute some informations before someone ask. So we created **batch views**, that are a "partial" sight of masterdataset's dataset.
To implements this strategy another tool comes to help, **JCascalog**[1]. Also this is an open-source library that allows to build a "cascading" query to interrogate the dataset. To suit this library to our problem we implement three **query class**:

- ***SentimentAnalysis***: if the tweets in input contain one of the keywords, perform sentiment analysis through NLP (see 2). The result is saved in two fields: $keyword$ and $sentiment$;
- ***Count***: this is an aggregate function made available by the library. This implicity group the former output by $keyword$ and $sentiment$, and count the occurrences. This return $keyword$, $sentiment$ and $count$ fields;
- ***QueryResult***: this query class update the Cassandra's **batchtable**, see later.

JCascalog make easy query the dataset, moreover it implicity performs all operations through MapReduce, so it is also fast.

4. Fast Layer

For the fast layer we used **Storm** that is useful in order to process continuous streams of data. The Storm model represents the entire stream-processing pipeline as a graph of computation called a topology. Rather than write separate programs for each node of the topology and connect them manually the Storm model involves a single program thats deployed across a cluster.
A stream is an infinite sequence of tuples and the spout is the source of streams in a topology. While spouts are sources of streams, the bolt abstraction performs actions on streams. A bolt takes any number of streams as input and produces any number of streams as output. Bolts implement most of the logic in a topologythey run

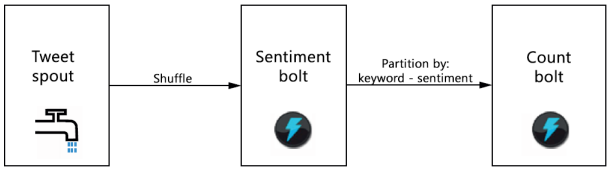

Figure 2. Storm topology

functions, filter data, compute aggregations, do streaming joins, update databases, and so forth.
In our topology there is one spout and two bolts that forms the topology in figure 2.

4.1. Tweet Spout

The tweet spout read from the database of our tweets in order to simulate twitter's API, and perform two actions:

- Write the tweet in a special folder of the HDFS where new tweets are stored until the batch layer finishes his computation;
- Send each tweet to the first bolt with a shuffling operation: the tweets are sent to the workers, that can operate in different machine, of the first bolt randomly.

4.2. Sentiment Bolt

For each tweet received the sentiment bolt first check if one of the keywords selected at the start by the user are contained in the tweet; if so the bolt perform a sentiment analysis (see 2) to this tweet and emit the tuple $(keyword, sentiment)$ that will be processed by the count bolt (the same tuple will always be processed by the same task tanks to the Fields grouping).

4.3. Count Bolt

Finally the count bolt simply make a query to Cassandra adding 1 to the count relative the keyword and sentiment arrived from the previous bolt.
Further details about Cassandra schema and operations will be given in the next section.

5. Serving Layer

For the serving layer we decided to use Cassandra which is a distributed NoSQL database man-

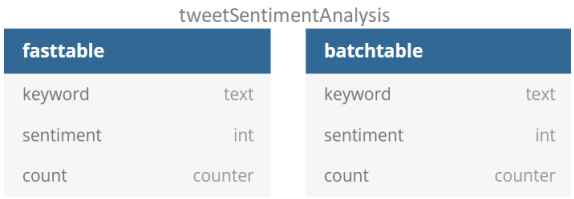

Figure 3. Cassandra keyspace

agement system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.
We used only one keyspace consisting of two column families :

- **batchtable** that contains the informations about the results of the batch layer;
- **fasttable** that contains the information about the results of the fast layer.

Both tables has the same schema consisting of:

- $keyword$: a string;
- $sentiment$: a string;
- $count$: a counters that is a special type that allows to make atomic addition and subtractions to avoid data inconsistency.

The pairs $(keyword, sentiment)$ are the key of the tables.
Our Cassandra schema is presented in figure 3.

6. Presentation Layer

The presentation layer consists of a thread that runs independently in local that at regular intervals carry out these operations:

- query the db (Cassandra) asking for the information contained in the fasttable and in the batchtable;
- merge these two results;
- create a chart where each requested keyword and it's sentiment is represented by a line with respect to time.

In our case we choose 10 seconds as time interval as showed by image 5.

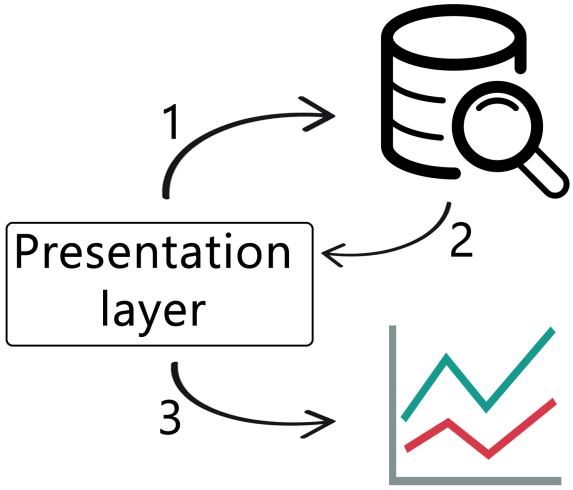

Figure 4. Presentation layer schema

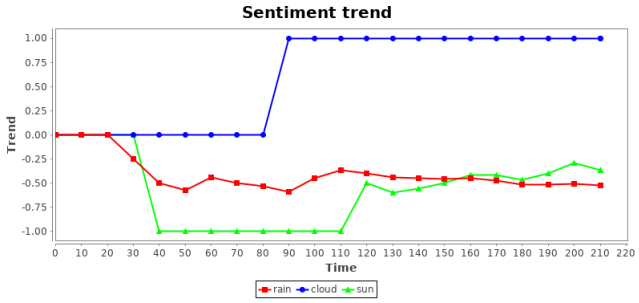

Figure 5. As visible, the chart is updated every 10 seconds with a new point

7. Functioning of Lambda Architecture

Once configured all the layers, is needful to combine all of these and to guarantee the right functioning. The result of the query performed by presentation layer returns a **consistent** outcome. Consistent means that the result has to be the join of batch and fast table but, updating the batch table with a tweet process result, imply that this must be deleted from the fast table. To implement this task we used the pail *snapshot* function and atomic Cassandra *update query*. The first allows to save the tweets written from storm and, when batch computation ends, to erase only the tweets just processed, leaving the others that storm has wrote during the batch computation. The second ensure that there isn't race condition during Cassandra update operation.
This approach is called **incremental**, and even if it is the most reasonable implementation, it could

brings to human errors due to the complexity introduced in the database handling.
At the opposite there is the **recomputation** approach. This method instead of update both the views (Cassandra tables), adjourn the fast table and *recompute*, at each new tweet addition, the batch view. From the performance poit of view, this strategy is more expensive than incremental one, but if an error occur is easier to locate the corrupted part, because an update involves just a part of tweets. To take both the benefits we employ both of them: we alternate incremental and recomputation approach as suggested by [1].

8. Tests

All the tools employed as explained earlier are tested. There are **two** class test:

- **PailTest**: this tests try to write to and to read from HDFS using pail. In particular are inserted "fake" tweets trough pail's method directly and taking tweets from a precomputed "fake" list of tweets. There is also a test that verifies the ingestion function; the result shows that all the tweets of the source pail folder are "ingested" by destination pail folder and then source folder is emptied.

- **LATest**: this tests check the initialization of all the tools in the static *setUp()* method. Then there are three test function that verify respectively, a cycle of tweets passing from fast layer to batch layer, a query to the serving layer with a predefined list of keywords and the recomputing function of the whole batch layer.

All the tests are passed.

9. AWS

To execute our lambda architecture we used Amazon Web Service (AWS) that provides on-demand cloud computing platforms and allows subscribers to have at their disposal a virtual cluster of computers.
For this project we created three cluster: one for Hadoop, one for Storm and one for Cassandra. Our AWS schema is showed in figure 6.

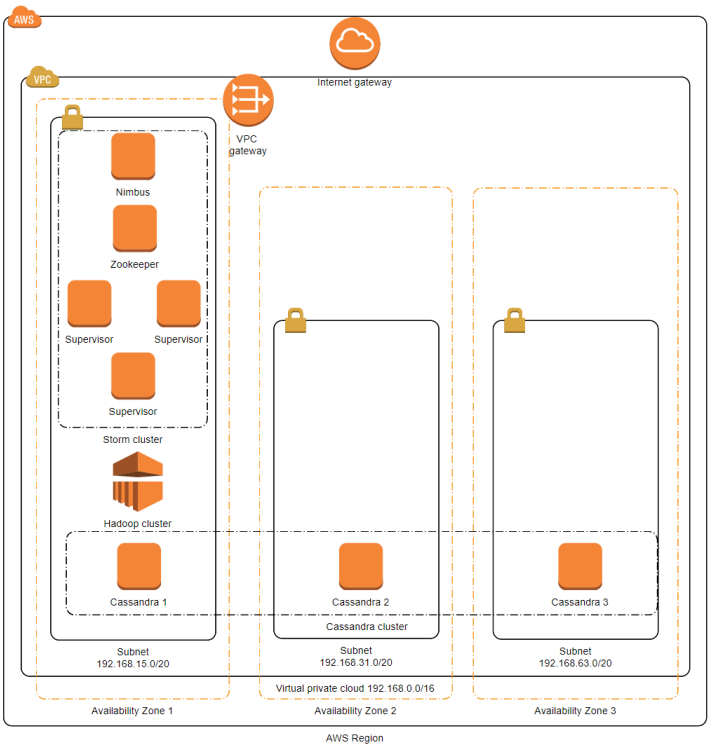

Figure 6. AWS schema

9.1. Hadoop

Setting up Hadoop was the easiest part since AWS provide a service called $Amazon Elastic MapReduce$ that allow developer and companies to create their Hadoop cluster in few seconds. Clearly is necessary to allow inbound and outbound communication in order to let the storm cluster write in the HDFS and to write in Cassandra.

9.2. Storm

For storm we created a cluster of five Ec2:

- one Nimbus machine which is responsible for distributing data among all the worker nodes;

- the Zookeeper machine to coordinate the nodes in the cluster and maintaining shared data with robust synchronization techniques. Nimbus is stateless, so it depends on

ZooKeeper to monitor the working node status;

- three supervisors (a supervisor has multiple worker processes) that follow instructions given by the nimbus.

9.3. Cassandra

Setting up the Cassandra cluster has definitively been the most difficult part. Since it is a database without a single point of failure creating a cluster of machines all in the same AZ (Availability Zone) would have been senseless; so we created a cluster of three machine in three different AZ of the Central European region installed Cassandra in each of them and then configured their $Cassandra.yaml$ files in order to make them communicate.
After that we opened all the necessary communication ports in order to make queries to the distributed db from other locations. cit [1]

10. Results

The results showed by the presentation layer are, clearly, keyword dependent: meaning that they really depends on the keyword that the user choose and on the dataset used.
In figure 7 we can see the results using the keywords: google, microsoft and amazon. Generally the results tend always to be quite negative.

References

[1] N. Marz, *Big Data Principles and best practices of scalable real-time data systems*. Manning, 2015.

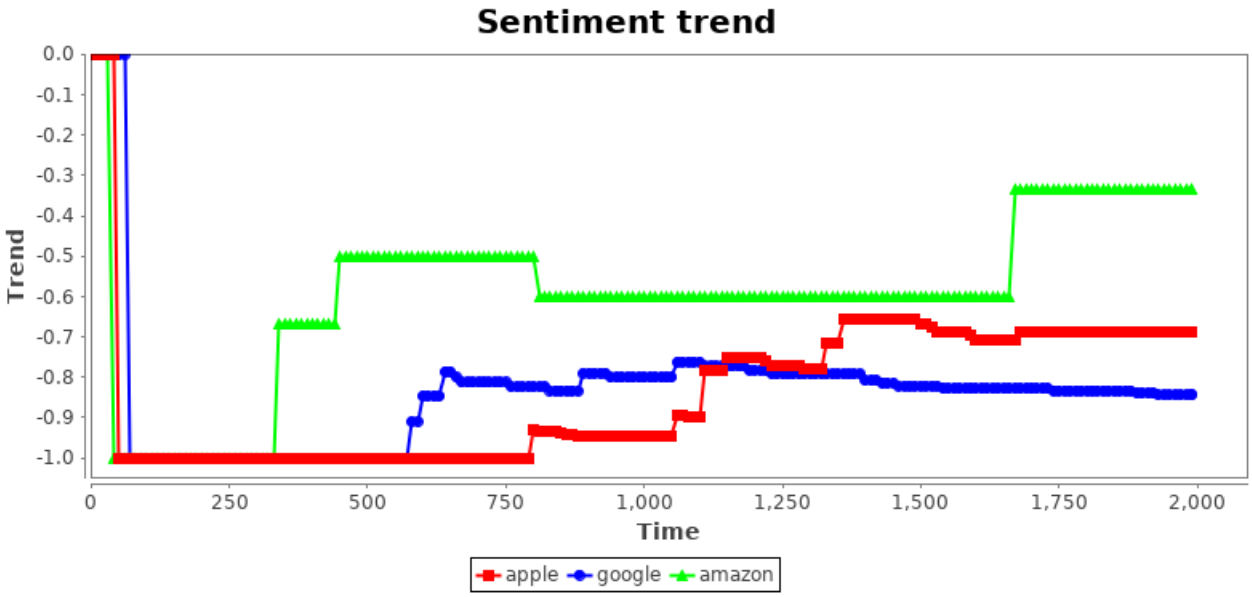

Figure 7. Results