Lambda Architecture for image retrieval and ranking

Federico Palai
Università degli Studi di Firenze
federico.palai1@stud.unifi.it

Federico Tammaro
Università degli Studi di Firenze
federico.tammaro@stud.unifi.it

Abstract

Implementation of a Lambda Architecture to retrieve images from Google and Bing by providing a keyword. The downloaded images are processed using CEDD features extraction and the system returns in real time the three most relevant images for the given keyword.

1. Introduction

Our project consists in the creation of a *Lambda Architecture* to analyze images retrieved from some of the most popular and used search engines given a query, in order to select the most relevant images based on that particular query; to rank images based on their similarity, the CEDD descriptor [1] has been used to extract features from images.
A *Lambda Architecture* is a data processing architecture designed to handle huge amounts of data using both batch and speed methods. This means that not only the analysis is performed in parallel, but also by using two different systems:

- A *batch layer* that will process the data in batches (collections of items) in a more accurate and therefore more expensive way in terms of time and resources, although giving the best results. This layer runs best when there is a significant number of items to process.
- A *speed layer* that has a significantly lower processing time but gives less accurate results, starting to analyze items right away.

The Lambda Architecture used in our project was designed to receive a real-time stream of images from an external source and to pass them to the batch and the speed layer. The latter aims to give results immediately, showing something to the final user in a short period of time. After a certain number of images are downloaded, a more accurate analysis is executed by the batch layer: its results will override the previous ones (as better described later) and the user will not have to wait the execution of the batch layer for some preliminary results.

This mechanism guarantees that users will always be shown some results, with an increasing accuracy over time as more images are processed by the batch layer.

All the steps necessary to implements the architecture are described below and they will be discussed in detail in the following sections:

- Batch Layer
    1. Features are extracted from images, using the CEDD descriptor and are written on a file, with each line containing the image path and its features (144-dimensional point as defined by the CEDD descriptor)
    2. Centroids are generated using K-Means++ on the points (images) set
    3. The Mapper, reading a line of the file at a time, will map each point to its nearest centroid, sending as output to

the Reducer the pair `<query, image features>`

4. The Reducer calculates the new coordinates of each centroid (using K-Means) and, if the stopping criteria are not met, the cycle repeats from step 3
5. A final MapReduce job is started to map each point (image) to the final centroids found in the previous job and the results are written on the database

- Speed layer

1. A *spout* scans a folder to search for new images and sends an image to the first bolt as soon as a yet-to-be-analyzed image is found
2. The first *bolt* extracts the features of every image and sends them to the second *bolt*
3. The second *bolt* calculates a new approximated centroid (without using K-Means) and writes the most relevant images on the database

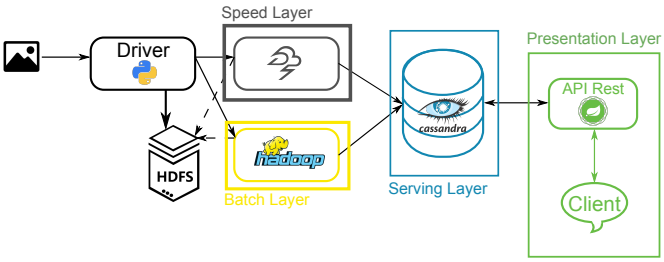

Figure 1. Lambda Architecture schema

1.1. CEDD Descriptor

The CEDD descriptor uses an histogram of 144 elements to describe color and texture features of an image and is a good descriptor to store in a database a large number of images; each element varies in the range [0.0, 7.0]. In our project, this descriptor has been used because of its simplicity and calculation speed, even though results are not very accurate since it is highly based on color values rather than shape and edge features.

In both batch and speed layer, the LIRE[2] Java library has been used, which provides a `CEDD` class with a `extract(BufferedImage image)` method to easily extract an array of `double` values representing the image histogram.

2. K-Means

The K-Means algorithm is used to solve a clustering problem, in this case a query can have different meanings: *apple*, for example, can return images of smartphones, laptops, Apple logos and fruits. So, after downloading hundreds or thousands of images, they can be split into clusters of similar images, and we need to find the most relevant images (i.e. the image that better represents the "average" image of that cluster) for each type. For our experiments we tried splitting the images into 3 clusters, but, depending on each query, there have been cases where the number of clusters have been reduced to 2 or even 1 (more on this in the next section).
This algorithm has been implemented in the batch layer using the MapReduce framework provided by Hadoop by splitting the association of a point and a centroid (the clustering part, done by the Mapper) and the calculation of the new centroids (done by the Reducer).

2.1. K-Means++

A random initialization of the centroids is easy, but it can lead to a poor performance because of the curse of dimensionality: the CEDD descriptor uses 144-dimensional points with sparse arrays, so a random initialization may create centroids very distant from all the points of the set.

For a better centroids initialization, we used the K-Means++ algorithm rather than creating random centroids with a range of [0-7] for each dimension (range of values in the CEDD descriptor). Below are described the steps to achieve a better initialization of centroids:

1. Choose a random point from the points set as centroid and remove it from the set

2. Until the given number of centroids has been chosen:

   (a) Calculate the distance between each point and its nearest centroid

   (b) Choose a new centroid from the remaining points in the set, with a weighted probability distribution based on the square values of the distances calculated before

By using this algorithm, rather than using random generated centroids, we should have a good distribution of centroids among the points in the considered set, reducing the probability of the removal of a centroid with no points associated in the Map process of the MapReduce framework used in the batch layer.
In our experiments, using a random initialization led to a reduction in the number of centroids: this because the CEDD descriptor creates a very sparse set of histograms (most of the features are set to 0.0), so K-Means often associated all the points to one or two centroid instead of the initial three centroids; instead, using the K-Means++ initialization, if there are at least 3 "groups" of different images, the number of centroids is very likely to remain at three.

3. HDFS

The Hadoop Distributed File System is the location where all the images are stored: as indicated by the name, this storage can distributed between multiple machines. This leads to a fault-tolerant architecture but in our case also allows to store relatively big files, splitting them among all the available nodes.
The batch layer also creates the input and output folders of the MapReduce, along with some temporary folders to detect loops in the algorithm, to save the centroids between each iteration and to store the path to images yet to analyze.
So, the HDFS structure looks like this:

- *images* is the folder containing a subfolder for each query, with each subfolder containing the images to be analyzed. Folder and images are created by the external driver.
- *input* is the folder where the driver periodically uploads the *paths* files, each in a subfolder named as the query, for the batch layer to locate and analyze images. This folder is created by the external driver.
- *dataset* is the folder where the extracted features are stored in a file, in a subfolder for each query, to be given as input for the MapReduce algorithm.
- *output* is the output folder for the MapReduce algorithm. It must be created, otherwise the algorithm will not start, but in our case is not used since results are directly stored in the database.
- *centers* is the folder where the MapReduce algorithm stores the centroids coordinates between every MapReduce loop and contains a sequential file for each query.

4. Batch Layer - Hadoop

The batch layer is used to perform a precise but time-consuming calculation of the most relevant images for a query, by using the K-Means algorithm implemented using the Hadoop MapReduce framework. This layer is started only when there is a significant number of images on the HDFS, since it is convenient to use it on a large number of images rather than a small set; also, a low number of images can lead to an inexact clusterization due to an insufficient number of points. The command used to start the batch layer is `java -jar batch.jar [query] [k-means threshold]`; in our case we used a fixed threshold of 0.005.
First, the batch layer reads the *paths* file and extracts the features of all the images listed in the file, then saves them in the database to avoid unnecessary recalculations in the future executions and creates a *dataset* file to use as input in the MapReduce framework. Also, previously analized images for the same query are added to the file by reading them from the database.

4.1. MapReduce

The MapReduce framework reads lines from a file as input and has a generic tuple as output; generally, the output is written on a file, but in this case the results are directly stored in the database, since they are used only to be shown by the presentation layer to the user.

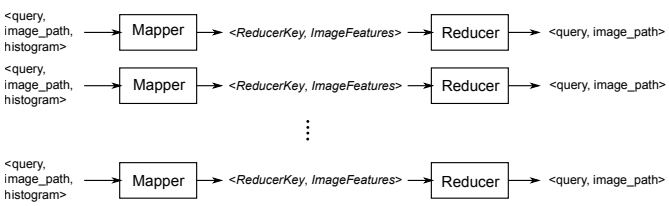

Figure 2. Map-Reduce scheme

First of all, the **Mapper** reads the centroids' coordinates from the *centers* file previously created and/or updated. After this initial setup, it reads a line at a time from the *dataset* file defined as input and splits it into the tuple `<query, image_path, histogram>`: this can be considered the real input of the map function. Then, the Mapper proceeds to pair each point to its nearest centroid and creates a tuple `<<query, centroid>, <image_path, histogram>>` to give as input to the Reducer function.

- `<query, centroid>`, being the key used in the Reducer function, is represented by the custom class *ReducerKey*
- `<image_path, histogram>`, being the value used in the Reducer function, is represented by the custom class *ImageFeature*

Then, the **Reducer** updates the coordinates of each centroid based on its associated points and then checks if the following conditions are true:

1. All the centroids have converged (their previous positions and the new positions are less distant than the defined threshold)
2. The sum of the distances between their previous positions and the new positions is smaller than the defined threshold
3. The sum of the distances between their previous positions and the new positions is the same as the previous iteration, this signals a loop where the centroids are simply being swapped between themselves and this check breaks the loop

If one of these conditions is met, the MapReduce framework ends its cycle.
Then, the framework runs again only one time to map the points to the newly calculated (and final) centroids: before terminating once and for all, the Reducer calculates for each centroid its nearest point (image) and saves on the database the most relevant images for the query.

5. Speed Layer - Storm

The speed layer is used to perform an approximated but real-time calculation of the most relevant images for a query: each image is processed in real-time as soon as it is uploaded on the HDFS, using an algorithm that simulates a K-Means centroid calculation, but yields fast results with a low accuracy. These results are shown to the user while the batch layer has not calculated any result yet or these results are too old (a significant number of new images have been crawled since then). Unlike the batch layer, the speed layer starts immediately and waits for any new image.
The **storm topology** used in our lambda architecture, represented in Figure 3, has one spout and two consecutive bolts, each with its own purpose.

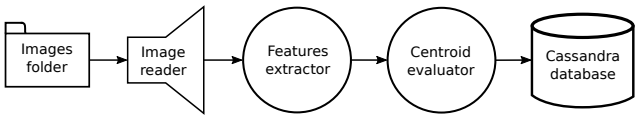

Figure 3. Storm topology used in the speed layer

5.1. Spout - Image Reader

The spout reads the *images* folder on the HDFS and searches for images that has not seen before, since it saves the path of every image sent to the first level bolt. Whenever it finds an unseen image, it emits a tuple `<query, image_path>` to the first level bolt and resumes the scanning of

the folder.

5.2. First Level Bolt - Feature Extractor

The first level bolt receives the tuples from the spout and, retrieving the image path directly from them, extracts the features from the image using the CEDD descriptor; then, it emits a tuple `<query, image_path, histogram>` for the second level bolt. So, its only job is to extract features from images.

5.3. Second Level Bolt - Centroid Evaluator

The second level bolt receives the tuples from the first level bolt and updates the centroid using an approximated method that vaguely resembles K-Means, but it is faster to allow a real-time processing of images. Then, it calculates the 3 most relevant images for the query and saves the results in the database (the mechanism which determines whether to use the batch or speed results is described in Section 7).

5.3.1 Simplified centroid evaluation

A low-accuracy but fast algorithm has been used to avoid a time-consuming calculation such as K-Means:

1. Initialize the centroid coordinates with the coordinates of the first image (point) that will be analyzed
2. For every other image (point) received, the new centroid coordinates are obtained by calculating the average, for each dimension, between the centroid coordinates and the point coordinates

This will give a first estimate of which images can be the most relevant for the given query, but the lambda architecture will update it as soon as the batch layer finishes.

6. Serving Layer - Cassandra

Since a Lambda architecture can be distributed in multiple nodes of a cluster of machines, a traditional database structure could not be used, since its persistence on only one node is a single point of failure of the entire architecture. For a more fault-tolerant approach and to easily scale inside the architecture, Apache Cassandra has been used as the database in which batch and speed results are stored, along the extracted features of every processed image.

The tables used in the database are pretty simple and straightforward.

query	image_path	num_images	is_batch	timestamp

Table 1. Results table

query	image_path	features

Table 2. Features table

To access the database both from the Lambda architecture but also from the Rest APIs exposed for the presentation layer, a package called *cassandra-connector* has been developed, which heavily depends on the Datastax Java Driver[3] for Cassandra: it contains a class with static methods to connect, create tables (if not already present) and insert and retrieve results or features.

7. Combining Speed and Batch Layers

To combine results from both layers, a technique similar to a ping-pong scheme [4] has been used in the *driver.py* script.

1. The speed layer starts as soon as the script is launched, but the batch layer is delayed until the script has uploaded N images (passed as argument to the driver)
2. The batch layer is executed and its results are stored in the database, overriding speed layer results
3. The speed layer continues its execution, but won't update the results in the database for num_images (contained in the database record and initially set to N) images after it sees a batch result in the database

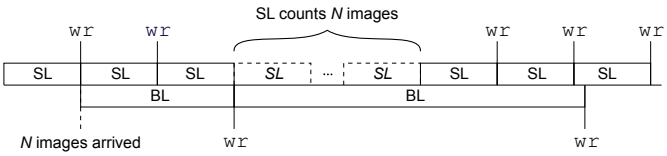

Figure 4. Note that when the BL writes on the database, the SL will stop its writing for a number of times equal to the number of images processed until then

Below a time-line example to better understand how speed and batch layer will alternate and who will write when on the database.

8. Crawler and Driver

The *crawler.py* file is a small Python 3 script used to automatically download images from Google and Bing using the *icrawler* Python package[5]. By launching the script with the query and the number of images (per service) to download as arguments, the script creates a directory and populates it with the images.

The second Python script, *driver.py*, has a more important role in the architecture: its arguments are the query of interest and the number of images to upload before starting the batch layer. First of all, it creates the *images* and *input* folders on the HDFS, if not already present, then proceeds to start the speed layer of the architecture, which will loop independently until all the images are uploaded.
At this point it starts uploading all the images one by one and, when requested, uploads the paths file read by the Mapper in the batch layer. After every image is uploaded, it launches the last execution of the batch layer, waits until it finishes and then kills all the processes and quits.

So, every interaction between the speed layer and the batch layer is regulated by the driver script, which is invoked when a user submits a query in the presentation layer of the architecture.

9. Presentation Layer - API & Web-Page

Some simple REST APIs have been created to run on a Spring Boot server for the frontend to query and to show the user the results in real-time. This interface is where the user will submit queries, resulting in the starting of both architecture layers; below the search box, the user will also see results in real-time as the lambda architecture updates its views in the database.

9.1. Spring Boot server

A **RestController** has been created to use the *cassandra-connector* package previously described and the client can submit the following GET requests to the controller:

- */getMostRelevant/*{*query*} to start the *driver.py* script for the submitted query
- */getResults/*{*query*} is used periodically by the client to retrieve the results for the submitted query, returning only a list of the most revelant images and their paths on the HDFS
- */getImage/*{*path to the image*} is used by the client after retrieving the list of most relevant images to get the images from the HDFS and show them to the user

9.2. Front-end client

A simple user interface has been developed to let users interact with the Lambda Architecture. In particular, users can input the query in a form and, after clicking on the search button, they will be immediately able to see the three most relevant images: initially, those will be given by the speed layer and will not be necessarily very accurate. Later on, images will be replaced with better results given by the batch layer. Note that the results from the batch layer, as already discussed, could change in number; so, after a while, there could be two images or only one image instead of three. The speed layer will replace the batch layer results after a precise number of image given by a chosen heuristic (v. Section 7) and this alternation will end only when there are no more images to be processed.

Here are some screenshots showing the presentation layer:

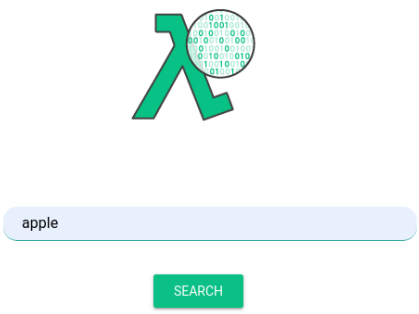

Figure 5. The main page of the presentation layer with the query

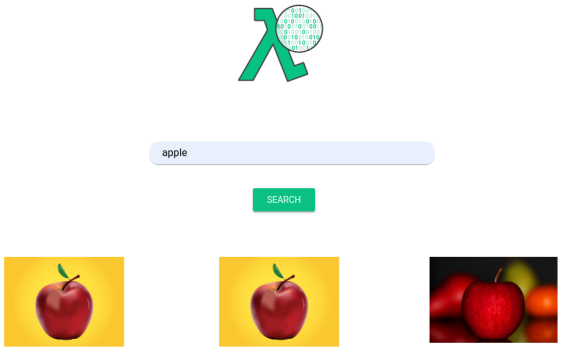

Figure 6. First speed layer results, in this case only images of a type are shown

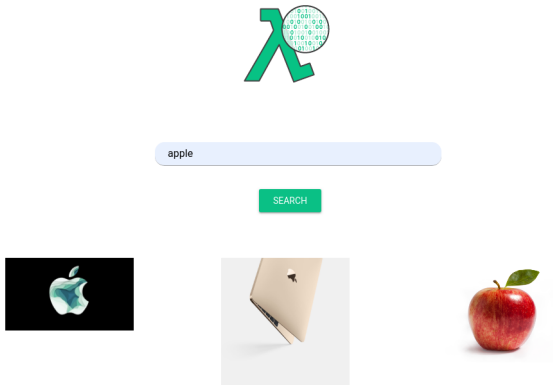

Figure 7. Now different types of images are shown even if it is still a low accuracy result, since the query was ambiguous

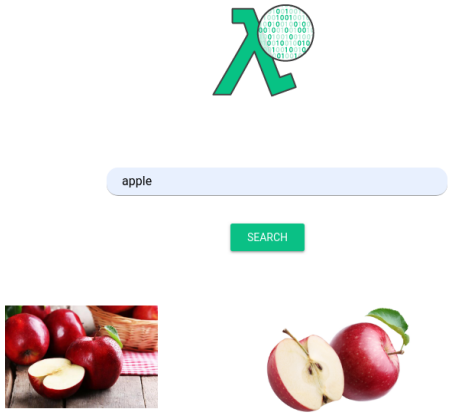

Figure 8. First batch results (on N images). More accurate than the first speed layer results, but only two relevant images

10. Future Developments and conclusions

We have seen how the analysis and ranking of a stream of images can be achieved in a distributed architecture using Hadoop and Storm, providing low-accuracy but fast results and then more precise results after some time. There are some improvements left as future developments to achieve even better performances.
First of all, the architecture can be deployed on an Amazon Web Service cluster to take advantage of its distributed structure, provided by both Hadoop and Storm; this will lead to better performances and a distribution of the workload.
Then, the heuristic used for the ping-pong scheme can be improved upon further studies, decreasing the time between each results update and giving the user a more real-time view of the results.

References

[1] Savvas A. Chatzichristofis and Yiannis S. Boutalis. Cedd: Color and edge directivity descriptor. a compact descriptor for image indexing and retrieval. *ICVS 2008: Computer Vision Systems*, 2008.

[2] Lire: Lucene image retrieval, 2019. http://www.lire-project.net/.

[3] Datastax java driver, 2019. https://github.com/datastax/java-driver.

[4] Marco Bertini. Parallel computing course slides #20, 2019.

[5] icrawler python package, 2018. https://github.com/hellock/icrawler.