Radish: a distributed relevant image retrieval system

Giovanni Berti
E-mail address
giovanni.berti2@stud.unifi.it

Abstract

The rise in data produced on the internet every day, and the need to quickly and efficiently analyze those data pose the basis of Big Data architectures. Our work explores an implementation of a distributed image retrieval system based on Hadoop and Storm designed in line with the notorious Lambda Architecture.

1. Introduction

In the modern world, image retrieval systems must face the challenges posed by an Internet with more than 4 billion users and exabytes of data created daily. All of this data creates the need of a distributed architecture to properly and efficiently ingest, process and present results of interest in response to a user-provided query.

Our work consists in a distributed relevant image retrieval system called *Radish* based on *Lambda Architecture* as described in [1]. Here is a brief outline of the components involved in the architecture:

- a *batch layer*, processing incoming data in a batch, periodical fashion, computing and persisting accurate results for the task at hand;
- a *speed layer*, processing real-time data incrementally, whose primary concern is low latency over result correctness;
- a *serving layer*, usually a thin abstraction layer that accepts user query, routes them to the other two layers and presents results to the user.

These layers work together in parallel to process incoming data. When new data is input to the system, the speed layer updates its internal view (initially derived from batch processed data). After a batch is complete, the *speed view* is then updated. The system is initialized with a list of keywords and continuously crawls for tweets with text that contains one of the keywords and has at least one image.

2. Batch layer

The batch layer's job is to read image descriptors (called also feature vectors, we will use the two terms interchangeably throughout the paper) from a database and cluster these descriptors in order to extract a *most relevant* image for each feature cluster.

The system is designed to work in a distributed processing environment with enormous amounts of data and lots of concurrent readers and writers, to the point that traditional SQL databases fail to scale effectively under these circumstances. Considering this, a more suitable choice of database is HBase[2], a column-oriented database tailored to work in distributed settings. This is the database implementation used for the system, but is not the only one: Cassandra[3] for example is another popular choice when it comes to distributed databases.

The system implementation uses HBase, with the following schema for the batch layer[1]:

[1] Note that these are not rigid schemas as in relational databases, as column can be added at any time for any row

- *images* table:
  - *image_id*, a UUID for the image
  - *image_path*, a HDFS path for the image
  - *features*, a Base64 encoded `double` array, the image descriptor
  - *keyword*, the search query associated to the image
- *clusters* table:
  - *cluster_id*, an integer identifying the cluster
  - *keyword*, the search query associated to the images in the cluster
  - *centroid*, a `double` array, the cluster centroid
  - *center_image_id*, the UUID of the image whose feature vector is nearest to the centroid

The batch layer's job is accomplished using a map-reduce pattern implemented with the widespread Hadoop[4] framework. The batch layer is initialized with a list of keywords, and begins by creating a MapReduce job for each of them. The map operation reads all the rows in the *image* and filters them by keyword. Each reducer is then assigned a list of feature vectors (all of which have the same associated keyword) and invokes another MapReduce job that perform a k-means clustering on those vectors. After k-means clustering has converged, the reducer computes the feature vector that is nearest to the cluster centroid and outputs the computed data in the *clusters* table.

2.1. Image descriptors

As image descriptors we choose CEDD descriptors[5], whose output is a 144-dimensional vector of integers in the range $[0, 7]$. Even though it is not a very accurate descriptor like those based on neural network models, its ease of computation and consequent fast computation made it a good candidate for the system.

We used the LIRE[6] library to compute feature vectors, which provides a `CEDD` class and relevant methods to extract a `double` array from an image.

2.2. Filesystem organization

According to the distributed nature of the system, it is necessary to have some component or arrangement to synchronize and provide the necessary redundancy and fault tolerance to files that a single non-distributed filesystem can't provide. To this aim HDFS (Hadoop Distributed File System), part of the Hadoop framework, provides a convenient distributed filesystem interface.

Below we outline the directory structure used by *Radish* in HDFS:

- *images*, containing all images crawled by the speed layer
- *clusters*, containing files pertaining to k-means MapReduce jobs

2.3. *k*-means implementation

Because of the scalability requirements of the system, the standard approach of calling a k-means library in the reducer is not applicable (the reducer collects all feature vectors associated with a keyword; if this number wasn't unbelievably massive it wouldn't need a distributed solution). Thus, we implemented the k-means algorithm with the MapReduce framework to be used as a subjob during the Reduce phase.

The k-means variant we implemented, called k-means++, features an improved initialization phase that avoids poor clusterings generated by unfortunate initial choices in the vanilla k-means algorithm. The pseudocode for k-means++ is reported in algorithm 1 on the following page.

After k-means++ initialization, the standard k-means is applied. Before going on to detail the Hadoop implementation we will briefly describe the backing filesystem structure:

- *points data file*, an input file with one row per data point; in our case it will contain feature vectors

Algorithm 1: k-means++ initialization

Input: Dataset X

Output: set C of centroids to bootstrap k-means

$C \leftarrow \{\text{random point in } X\}$
repeat
  foreach $x_i \in X$ **do**
    $d(x_i) \leftarrow \min_{c \in C} \|x_i - c\|$
    $p(x_i) \propto d(x_i)^2$
    $C \leftarrow C \cup \{\text{random } x_j \text{ with probability } p(x_j)\}$
  end
until $|C| = k$

- *centroids file*, a file produced by the initialization phase, where each line is a cluster centroid (note that this file always contains exactly k lines)
- *points input file*, a file produced by the initialization phase and input to each iteration of k-means. Each line is in the format $\langle point,\ assigned\_centroid\_id \rangle$

Points are serialized to strings by joining each component with a space. In the *points input file* the two parts (point and centroid id) are separated with a semicolon (`;`) character. The paths of these files are all configurable, and by default the *centroids file* and *points input file* will be placed in `/clusters/kmeans_centroids_`*`keyword`* and `/clusters/input_`*`keyword`* respectively.

The initialization phase is composed by two different MapReduce jobs. The first job computes the square of the distance to the nearest centroid for each point and writes the sum of the squared distances S in a temporary file. The second MapReduce job computes the square of the distances again and chooses one new centroid according to the pseudocode of algorithm 1 . The map part of this MapReduce jobs outputs both the point and the squared distance. To choose a new centroid, the following strategy has been implemented: the reducer generates a random number r in the range $(0, 1]$ and multiplies this number by S. While iterating on the received points, their squared distances are accumulated until they reach the rS threshold, and the first point to do so is chosen. If the threshold is never reached (for example because of very low distances and floating point rounding errors) the last point is chosen instead. This couple of MapReduce jobs are then iterated until we get k centroids. Note that these two jobs always have an in-memory access to the centroids list (we can do so under the assumption that k is sufficiently small, which is usually the case). After generating k centroids, a preliminary assignment of the input points is made by matching each point to the nearest centroid. By this time both the *centroids file* and the *points input file* are filled.

After the initialization phase is completed, the k-means loop can begin. The mapping phase maps each point to the index of the nearest centroid and outputs a $\langle index, point \rangle$ key-value pair, while the reducing phase computes the new centroid for each cluster associated to a centroid. After the job is completed, both the *centroids file* and the *points input file* are updated. If the new centroid's distance to the old centroid is below a set threshold, a job counter is incremented, marking the convergence of said centroid. The MapReduce job is then repeated until one of two conditions is met: either the count of converged centroids has reached k, meaning that the whole k-means procedure has converged, or a maximum number of iterations has been reached. Because we are also interested in computing the nearest point to each centroid, the reducer is also tasked with this computation.

During a k-means iteration it can happen that a cluster becomes empty, and the algorithm comes to a stall. To deal with this issue, every time the reducer detects an empty cluster the corresponding centroid is marked for elimination. After the job is completed, all marked centroids are subsequently deleted from the centroid file, and a new initialization phase is launched with the remaining centroids already present (borrowing notation from algorithm 1 , C is initialized to the remaining centroids instead of a random point in the

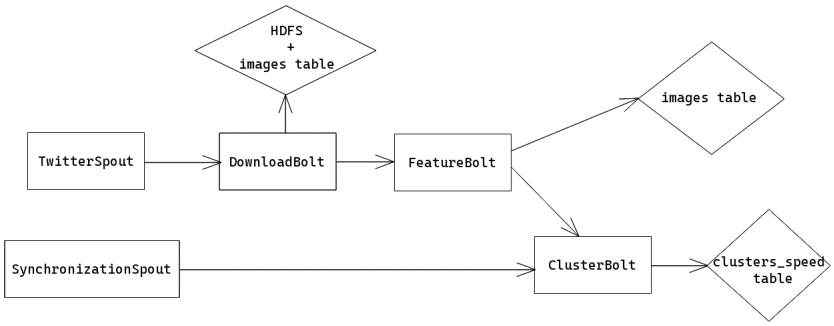

Figure 1. Speed layer topology diagram

dataset).

3. Speed layer

The speed layer's *raison d'être* is to continuosly crawl for new Twitter images, download them and compute their feature vectors, all while providing a coherent view of both the new observed data and the already present data.

To implement the speed layer we employed the Apache Storm[7] stream processing computation framework. In order to define a stream processing pipeline, a Storm *topology* is to be defined. This topology is made of *spouts*, which continuously produce new data, and *bolts*, which process data produced by spouts. Essentially, spouts act as sources while bolts can be both intermediate nodes or sinks. All Storm processing is tuple-oriented, meaning that all objects that travel between bolts and spouts are dynamically-typed tuples of variable length, addressed by field name or integer index.

We defined a Storm *topology* with the following bolts and spouts:

- *TwitterSpout*, which crawls for URLs of images that are associated to tweets
- *DownloadBolt*, picks up URLs, download the respective images and persists them to HDFS. Outputs the corresponding HDFS path along with an image UUID
- *FeatureBolt*, takes UUIDs and paths produced by *DownloadBolt* and outputs a triple $\langle id, keyword, features \rangle$
- *SynchronizationSpout*, a spout that periodically checks whether a batch round is complete and new centroid data is available
- *ClusterBolt*, a bolt that reads tuples from *FeatureBolt* and updates a speed layer view of the data
- *FeatureMapperBolt*, a utility bolt that converts tuples output from *FeatureBolt* in a format suitable to be used with HBase

HBase compatibility is achieved by using the `storm-hbase` library, also provided under the Storm project. Both FeatureMapper and ClusterBolt are connected to a corresponding *HBaseBolt*. FeatureMapper writes to the same *images* table mentioned in the batch layer section, while ClusterBolt persists data in a *clusters_speed* table that has the same schema as the *clusters* table.

FeatureBolt computes CEDD descriptors as described in subsection 2.1 on page 2.

ClusterBolt keeps an in-memory table mapping keywords to a list of $\langle cluster\_centroid, nearest\_feature \rangle$ pairs. This table's data is then persisted to the *cluster_speed* table. At every tuple received from FeatureBolt, ClusterBolt checks whether the received feature vector is nearer to any centroid than the current nearest point, and consequently updates its table. This mechanism makes it possible to quickly respond to incoming data while accounting for the possible change in clusters' distribution of the crawled data.

3.1. Speed and batch layers synchronization

The HBase table *clusters_speed* makes up the speed layer's real-time view that will be then queried by clients. Whenever a batch job is complete and new centroids are computed, the *clusters_speed* table will have to be updated. This is achieved with a simpler variant of the ping-pong scheme[8], which we will affectionately call *sugar candy*[2] scheme. Every time SynchronizationBolt detects that a new batch has been completed, it issues a *sugar candy* message, containing new centroids and their respective nearest vectors, to which ClusterBolt reacts by overwriting its internal structure and re-emitting a tuple for each new centroid. This way, both ClusterBolt and *clusters_speed* are updated.

The usage of this variant is possible thanks to the simple update rule employed by ClusterBolt, where only the nearest feature vectors are not recomputed.

4. Presentation layer

We also developed a frontend to demo the system. It queries the speed layer view and shows the most relevant images for each keyword. It's implemented as a relatively simple Spring Boot[9] application.

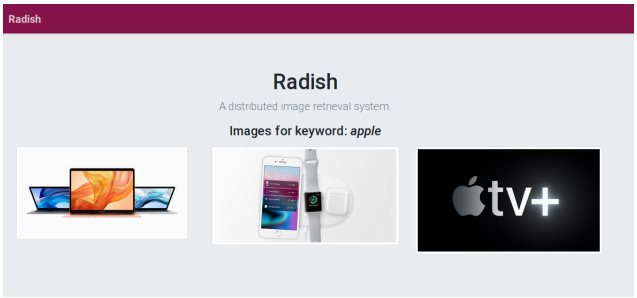

Figure 2. Radish frontend. Clustering set with $k = 3$

5. Conclusions and future developments

We have shown how a retrieval and ranking system can be implemented in a distributed architecture using Hadoop and Storm, to efficiently handle a large quantity of files.

There are some possible improvements to the system: for example replacing the current updating scheme in the speed layer by using some kind of heuristics to approximate new centroids as new images come. This will very probably require to switch from a sugar candy scheme to a full-fledged ping pong scheme.

A more sophisticated updating mechanism will allow to have even better image results.

[2] As opposed to *poison pill*

References

[1] Nathan Marz and James Warren. *Big data: principles and best practices of scalable real-time data systems*. Manning, 2015.

[2] Apache Software Foundation. *HBase*. Version 2.2.4. Apr. 1, 2020. URL: https://hbase.apache.org.

[3] Apache Software Foundation. *Cassandra*. Version 3.11.6. Apr. 1, 2020. URL: https://cassandra.apache.org.

[4] Apache Software Foundation. *Hadoop*. Version 3.2.1. Apr. 1, 2020. URL: https://hadoop.apache.org.

[5] Savvas A. Chatzichristofis and Yiannis S. Boutalis. *CEDD: Color and Edge Directivity Descriptor: A Compact Descriptor for Image Indexing and Retrieval*. Ed. by Antonios Gasteratos, Markus Vincze, and John K. Tsotsos. Berlin, Heidelberg, 2008.

[6] Mathias Lux. *LIRE*. Version 1.0b4. Apr. 1, 2020. URL: http://lire-project.net/.

[7] Apache Software Foundation. *Storm*. Version 2.1.0. Apr. 1, 2020. URL: https://storm.apache.org.

[8] Marco Bertini. *Parallel computing course's slides #20*. 2020.

[9] SpringSource. *Spring Boot*. Version 2.2.6. Apr. 1, 2020. URL: https://spring.io/projects/spring-boot.