Design and Implementation of a Distributed System for Content-Based Image Retrieval

Candidate
Pietro De Rosa

Supervisors
Prof. Claudio Gennaro
Prof. Giuseppe Amato
Prof. Fabrizio Falchi

A.Y. 2015-2016
To my family and
all my friends
Abstract

The aim of this work is to design and implement a distributed system for content-based image retrieval on very large image databases. To realize this system, a standard full-text search engine has been used. In particular, the system has been developed with the open source software Elasticsearch which, in turn, is built on top of Apache Lucene™, a widely used full-text search engine Java library. In order to allow the full-text search engine to perform similarity search, we used Deep Convolutional Neural Network Features extracted from the images of the dataset and encoded as standard text. Given the distributed nature of Elasticsearch, the index can be split and spread among several nodes. This makes it easy to parallelize the search, thus leading to a significant performance enhancement. All the experiments have been conducted on the Yahoo Flickr Creative Commons 100M dataset, publicly available and composed of about 100 million of tagged images. A web-based GUI has been designed to allow the user to perform both textual and visual similarity search on the dataset of images.
# CONTENTS

1 INTRODUCTION  
1.1 Thesis structure ............................................. 2

2 BACKGROUND  
2.1 Deep Learning and Convolutional Neural Networks .......... 4  
2.1.1 Deep Features ............................................ 4  
2.1.2 DCNN ..................................................... 4  
2.2 Information Retrieval systems .............................. 6  
2.2.1 Vector space model ...................................... 7  
2.2.2 Evaluation metrics ...................................... 9  
2.3 Apache Lucene .............................................. 10  
2.3.1 Default model and scoring function ..................... 10  
2.3.2 Index structure ......................................... 11  
2.4 Related Work .............................................. 12

3 ELASTICSEARCH  
3.1 Basic concepts ............................................. 13  
3.2 Sharding and Replication .................................... 16  
3.3 Cluster Health ............................................... 17  
3.4 Communications ............................................ 17  
3.5 Index structure ............................................ 18  
3.5.1 Meta-fields ............................................... 19  
3.5.2 Fields ................................................... 20  
3.6 Operations .................................................. 21  
3.6.1 Indexing .................................................. 21  
3.6.2 Create ................................................... 22  
3.6.3 Get, update, delete ..................................... 22  
3.6.4 Bulk operations ......................................... 22  
3.6.5 Searching ............................................... 23  
3.6.6 Adding nodes ......................................... 23  
3.6.7 Removing nodes ........................................ 24  
3.7 Scoring function ........................................... 25

4 FEATURES ENCODING  
4.1 Overview ................................................... 26  
4.2 Algorithm .................................................... 27  
4.3 Quantization factor ......................................... 27  
4.4 Query reduction ............................................ 28

5 IMPLEMENTATION  
5.1 Overview ................................................... 30  
5.2 Elasticsearch-based CBIR from scratch ...................... 31  
5.2.1 Features extraction and storing ......................... 31  
5.2.2 Interaction with Elasticsearch ........................................... 32  
5.2.3 ElasticImageIndexManager class ...................... 34
5.3 Application on the YFCC100M dataset .................................................. 36
  5.3.1 YFCC100M Lucene index .......................................................... 36
  5.3.2 LuceneIndexReader class ......................................................... 37
  5.3.3 YFCC100M Elasticsearch index ............................................... 38
5.4 Web-based GUI ............................................................................. 39
  5.4.1 Operations ............................................................................ 40
  5.4.2 Screenshots .......................................................................... 45

6 EXPERIMENTS ................................................................................. 47
  6.1 Setup ......................................................................................... 47
  6.2 Results ..................................................................................... 48

7 CONCLUSIONS .............................................................................. 50
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>An high-level view of a CBIR</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>CBIR obtained with a standard full-text search engine</td>
<td>3</td>
</tr>
<tr>
<td>Figure 3</td>
<td>DCNN Features visualization at different layers</td>
<td>5</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Cosine similarity illustrated</td>
<td>8</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Inverted index</td>
<td>9</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Example of ES index logically organized in types</td>
<td>14</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Example of JSON document</td>
<td>15</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Communication in Elasticsearch</td>
<td>18</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Indexing in Elasticsearch</td>
<td>21</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Single VS bulk requests</td>
<td>22</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Search operation in Elasticsearch</td>
<td>23</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Adding a new node to the ES cluster</td>
<td>24</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Removing a node from the ES cluster</td>
<td>25</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Effectiveness (mAP) vs space occupation for increasing values of the quantization factor $Q$ for the Flickr1M dataset</td>
<td>28</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Query reduction mechanism</td>
<td>29</td>
</tr>
<tr>
<td>Figure 16</td>
<td>System overview</td>
<td>30</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Class diagram for the classes performing features extraction</td>
<td>32</td>
</tr>
<tr>
<td>Figure 18</td>
<td>ElasticIndexManager class diagram</td>
<td>34</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Class diagrams for ElasticImageIndexManager, DeepFeatureEncoder and JsonImageBuilder</td>
<td>35</td>
</tr>
<tr>
<td>Figure 20</td>
<td>LuceneIndexReader class diagram</td>
<td>37</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Web-based GUI</td>
<td>40</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Sequence diagram for the &quot;Random search&quot;</td>
<td>41</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Sequence diagram for the &quot;Textual search&quot;</td>
<td>42</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Sequence diagram for the &quot;Visual search&quot; on an existent image</td>
<td>43</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Sequence diagram for the &quot;Visual search&quot; on an upload image</td>
<td>44</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Visual similarity search</td>
<td>45</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Textual search</td>
<td>46</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Results</td>
<td>49</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Comparison between IR and DBs</td>
<td>6</td>
</tr>
<tr>
<td>Table 2</td>
<td>Term-document weight matrix</td>
<td>7</td>
</tr>
<tr>
<td>Table 3</td>
<td>Comparison between ES and Relational DBs</td>
<td>15</td>
</tr>
<tr>
<td>Table 4</td>
<td>YFCC100M Lucene index structure</td>
<td>37</td>
</tr>
<tr>
<td>Table 5</td>
<td>YFCC100M Elasticsearch index structure</td>
<td>39</td>
</tr>
<tr>
<td>Table 6</td>
<td>/encode REST endpoint</td>
<td>43</td>
</tr>
<tr>
<td>Table 7</td>
<td>Details about the shards</td>
<td>47</td>
</tr>
<tr>
<td>Table 8</td>
<td>Results</td>
<td>48</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td></td>
</tr>
<tr>
<td>BM</td>
<td>Boolean Model</td>
<td></td>
</tr>
<tr>
<td>CBIR</td>
<td>Content-Based Image Retrieval</td>
<td></td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
<td></td>
</tr>
<tr>
<td>CRUD</td>
<td>Create Read Update Delete</td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>DataBase</td>
<td></td>
</tr>
<tr>
<td>DCNN</td>
<td>Deep Convolutional Neural Network</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td>ElasticSearch</td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
<td></td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td></td>
</tr>
<tr>
<td>REST</td>
<td>REpresentational State Transfer</td>
<td></td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifiers</td>
<td></td>
</tr>
<tr>
<td>VSM</td>
<td>Vector Space Model</td>
<td></td>
</tr>
<tr>
<td>YFCC100M</td>
<td>Yahoo Flickr Creative Commons 100M</td>
<td></td>
</tr>
</tbody>
</table>
INTRODUCTION

Advances in data storage and image acquisition technologies have led to the creation of large image datasets. In this scenario, it is necessary to develop appropriate information systems to efficiently manage these collections. The commonest approaches use the so-called Content-Based Image Retrieval systems (CBIR). Basically, these systems try to retrieve images similar to a user-defined specification or pattern (e.g., shape sketch, image example). Their goal is to support image retrieval based on content properties (e.g., shape, color, texture), usually encoded into vectors of real numbers called features. One of the main advantages of the CBIR approach is the possibility of an automatic retrieval process, instead of the traditional keyword-based approach, which usually requires very laborious and time-consuming previous annotation of database images. This technology has been used in several applications such as fingerprint identification, biodiversity information systems, digital libraries, crime prevention, medicine, historical research, etc. Figure 1 shows an high-level view of a CBIR.

![CBIR Diagram](image)

Figure 1: An high-level view of a CBIR

The basic steps to build a CBIR are the followings:

1. extract the features from the image dataset
2. store them, possibly into an index to speed-up search
3. define a similarity function to get the most relevant results against a query.

The last two steps come for free if a full-text search engine is used, since the latter has its own data structure - i.e. an inverted index - and defines its own scoring function. So, a CBIR can be achieved by means of standard text-retrieval techniques. The trick is to transform the information associated with the feature vector, which represents the content of the image, into a text suitable to be used and processed by a standard full-text search engine. To this end, several approaches have been proposed during
the last years depending on the kind of features. In this work, a quite recent method proposed by Amato et al. in [1] has been used; this encoding is designed to specifically index Deep Convolutional Neural Network (DCNN) Features, so that it is possible to support efficient content-based research on large datasets.

Figure 2 shows the typical structure of a CBIR obtained with a standard full-text search engine.

Regardless how the system is realized, there are at least two properties that are desirable from a good CBIR, and more in general from an Information Retrieval system (IR); they are listed below:

- high relevance of the results against the query
- low search time.

The relevance of the results retrieved depends on several parameters which, will be discussed later. The search time is a very important factor too and it should be minimized as much as possible. Indeed, a system that takes a very long time to get results is totally useless from the user’s point of view, even if it is very accurate and precise.

The main goal of this thesis is to design a system able to execute efficiently content-based queries on very large datasets. In order to test our system in a realistic context, the Yahoo Flickr Creative Commons 100M (YFCC100M) dataset has been taken as reference one, since experiments have already been done on it in [1]. YFCC100M consists, as the name suggests, of about 100 million of images uploaded to Flickr between 2004 and 2014 under Creative Commons commercial or noncommercial license. More information about the dataset can be found in [2].

To provide a boost in performance, the idea is to split the index into several parts and to distribute them among different nodes, so that search operations can be parallelized reducing as consequence the query execution time. To implement the aforesaid system, the textual search engine software Elasticsearch - ES from now on - has been used. ES is open-source, distributed and it is built on top of Apache Lucene™, an open-source full-text search engine Java library, which allows you to implement search functionality in your own Java application.

A web-based GUI has been developed to allow the user to perform both textual and visual similarity search on the index. It connects through the REST interface to the ES’s cluster, whom receives and serves users’ requests.

Finally, experiments have been conducted to observe the improvement of the searching time.

The structure of the whole document is briefly reported in the remaining part of this chapter.

1.1 Thesis Structure

In Chapter 2, some background knowledge are presented. In particular, the first part talks about features and, specifically, DCNN Features, i.e. those extracted by means of Deep Learning. Then, basic concepts on Information Retrieval are reported. Finally, the last part briefly introduces the Java library Apache Lucene™.
In Chapter 3, the noteworthy parts of Elasticsearch are described in an exhaustive way.

In Chapter 4, we explain in detail the encoding used to build a CBIR from a standard full-text search engine.

In Chapter 5, we explain how the system is realized. More in detail, we describe the implementation of both the system and the web GUI, and how they interact each other. Then, the main implementative choices are discussed and justified.

In Chapter 6, the experiments that have been performed are reported and the results are shown. Specifically, a comparison in search time has been done by letting the number of nodes grow.

In Chapter 7, we conclude and provide suggestions for possible enhancements and future work.

---

**Figure 2:** CBIR obtained with a standard full-text search engine
BACKGROUND

In this chapter, we present some notions of Deep Learning and, in particular, Convolutional Neural Networks. Next, basic concepts on Information Retrieval are presented, focusing the attention especially on the open-source search engine library *Apache Lucene*™.

2.1 DEEP LEARNING AND CONVOLUTIONAL NEURAL NETWORKS

2.1.1 Deep Features

In the previous chapter we said that, in order to perform visual similarity searches based on the content of the images, the so-called features can be used. They are defined in [3] as follows:

“Visual Feature is a mathematical description of the image visual content that can be used to compare different images, judge their similarity, and identify common content.”

Historically, there are two main type:

1. *global features*, which describe an image as a whole
2. *local features*, which represent image patches.

Global features have the ability to generalize an entire object with a single vector. The second ones, on the other hand, are computed at multiple points in the image and are more robust to occlusion and clutter.

Recently, a new class of image descriptor, built upon Deep Convolutional Neural Networks, have been used as effective alternative to descriptors built using local features such as SIFT, SURF, ORB, BRIEF, etc.

DCNN Features - or *deep features* for short - are extracted starting from CNN and they are the representation of one of the internal states of the neural network, when the latter is excited with a given input.

The first layers of DCNN are typically useful in recognizing low-level characteristics of images such as edges and blobs, while higher levels have demonstrated to be more suitable for semantic similarity search. Figure 3 shows this behaviour.

One major obstacle to the use of DCNN features for indexing large datasets of images is its internal representation, which is high dimensional, leading to the curse of dimensionality [4]. For instance, in the well-known AlexNet architecture [5] the output of the sixth layer (fc6) has 4,096 dimensions, while the fifth layer (pool5) has 9,216 dimensions.

2.1.2 DCNN

Deep Features are extracted from DCNNs whose architecture is briefly described in this section.
A Convolutional Neural Network (CNN) [6] consists of several layers, that can be of three types:

1. **Convolutional:**
   - Convolution is a mathematical operation which describes a rule of how to mix two functions or pieces of information: (1) the feature map (or input data) and (2) the convolution kernel mix together to form (3) a transformed feature map. Convolution is often interpreted as a filter, where the kernel filters the feature map for information of a certain kind (for example one kernel might filter for edges and discard other information). Thus, convolutional layer filters inputs for useful information. This layer has parameters that are learned so that filters are adjusted automatically to extract the most useful information for the given task.

2. **Max-pooling:**
   - After each convolutional layer, there may be a pooling layer. The pooling layer takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block. There are several ways to do this pooling, such as taking the average or the maximum. Pooling layers subsample their input.

3. **Fully-Connected:**
   - Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. A fully connected layer takes all the neurons in the previous layer and connects them to every single neuron it has. There can be no convolutional layers after a fully connected layer.

The typical architecture of a CNN consists of one or more convolutional layers - possibly with a subsampling step - and followed by one or more fully connected layers. DCNN also includes several hidden layers, some of which perform non-linear transformation. The actual architecture depends on the specific neural network.
2.2 INFORMATION RETRIEVAL SYSTEMS

Information Retrieval has been defined in [7] as

"the activity of finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)".

So, an IR system is an entity whose goal is to find the documents most relevant to a given query.

IR introduces three key concepts to deal with:

- collection of documents
- query
- notion of relevance.

By documents means the smallest unit of data which can be indexed or searched for. Usually, they are unstructured, i.e. raw text without any predefined or fixed structure. For example, they may be individual memos, chapters of a book, HTML pages etc.

Within a document collection, each document has an unique serial number, known as the document identifier (docID). The system may simply assign progressive integers to each new document when it is first encountered. The query is expressed using natural language keywords as input, without the need for any formal language. Finally, the results set is sorted by rank, which gives a measure of the relevance of each document.

To better understand IR systems, it is useful to make a comparison with Databases (DB). Table 1 reports the major differences.

The simplest form of document retrieval is for a computer to do a sort of linear scan through documents. Given the speed of modern computers, this can be a very effective process for simple searching of modest collections. However, this approach is not suitable for very large collections anymore. The way to avoid linearly scanning the texts for each query is to index the documents in advance, so that, then, they can be quickly searched for.

Many IR models are available in literature. The vector space model, described in the following section, is probably the most used one by the search engines. Indeed, it is supported both by Lucene and, in turn, by Elasticsearch.

<table>
<thead>
<tr>
<th>System</th>
<th>Format of data</th>
<th>Queries</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>Structured data: clear semantics based on a formal model</td>
<td>Formal language, e.g. SQL</td>
<td>Exact match records</td>
</tr>
<tr>
<td>IR</td>
<td>Unstructured or semi-structured, raw text or free text plus markup respectively</td>
<td>Natural language</td>
<td>Ranked documents</td>
</tr>
</tbody>
</table>

Table 1: Comparison between IR and DBs
2.2.1 Vector space model

The vector space model (VSM) belongs to the class of the ranked retrieval models in which, rather than a set of documents satisfying a query expression, the systems returns an ordering over the (top) documents in the collection for a query. The latter - as said before - is typed freeform by users into the search interface, using the natural language.

A way of assigning a score to query/document pair is needed. To this end, the score of a document, with respect to a given query, should satisfy the following conditions:

1. it should be proportional to the number of occurrences of the query’s terms in the document
2. since rare terms are more informative than frequent ones, documents containing them should receive a boost in scoring.

The first point suggests to use the term frequency $t_f t, d$, that is the number of times that term $t$ occurs in document $d$, to compute the score.

On the other hand, the tf weight of a term should be reduced by a factor that grows with its collection frequency, i.e. the total number of occurrences of a term in the collection. So, denoting the total number of documents in a collection by $N$, the inverse document frequency (idf) of a term $t$ is defined as follows:

$$\text{idf}_t = \log \frac{N}{d_f t},$$

where $d_f t$ is the document frequency, that is the number of documents that contain the term $t$. Thus, the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

The tf.idf weighting scheme assigns to term $t$ a weight in a document $d$ given by

$$tf.idf_{t,d} = t_f t, d \times \text{idf}_t.$$

To relate documents with their respective tf.idf weight, the so-called term-document weight matrix is used. The latter is a matrix that puts in relation terms and documents as shown in Table 2. Matrix element $(t, d)$ is equal to the tf.idf weight if the document in column $d$ contains the word in row $t$, and is $0$ otherwise.

| Term  | doc1 | doc2 | doc3 | doc4 | doc5 | ...
|-------|------|------|------|------|------|------|
| term1 | 2.20 | 1.87 | 0    | 0    | 0.30 | ...
| term2 | 0.70 | 2.20 | 0.48 | 0    | 0    | ...
| term3 | 2.37 | 2.36 | 0.48 | 0.30 | 0    | ...
| term4 | 0    | 0.3  | 0    | 0    | 0    | ...
| term5 | 1.76 | 0    | 0.95 | 0.78 | 0.95 | ...
| term6 | 0.48 | 0    | 0    | 0    | 0    | ...
| term7 | 0.48 | 0.30 | 0.30 | 0.30 | 0.78 | ...

Table 2: Term-document weight matrix

At this point, looking at the matrix’s columns, each document may be viewed as a vector with one component corresponding to each term in the dictionary, whose value is equal to its tf.idf weight. For dictionary terms that do not occur in a document, this weight is zero. This is exactly what is done in the vector space model.
In this model, the standard way of quantifying the similarity between two documents $d_1$ and $d_2$ is to compute the cosine similarity of their vector representation $V(d_1)$ and $V(d_2)$

$$sim(d_1, d_2) = \frac{V(d_1) \cdot V(d_2)}{|V(d_1)| \cdot |V(d_2)|},$$

where the numerator represent the inner product\(^1\) of the vectors $V(d_1)$ and $V(d_2)$, while the denominator is the product of their Euclidean lengths\(^2\). This measure is the cosine of the angle $\theta$ between the two vectors, shown in Figure 4.

![Figure 4: Cosine similarity illustrated](image)

To implement such a model, the relationship between a document and its tf.idf weight needs to be saved. However, storing the whole weight matrix may be very space-consuming even for modest collections. Given that the latter is extremely sparse, i.e. it has few non-zero entries, a much better representation is to record only the things that do occur, that is, the non-zero positions. This idea is central to the first major concept in IR, the inverted index.

An inverted index consists of a list of all the unique words that appear in any document, named dictionary of terms or vocabulary, and for each word, a list of the documents in which the term occurs in. Each item in the list - which records that a term appeared in a document and the relative tf statistic - is conventionally called posting. The list is then called a postings list or inverted list, and all the postings lists taken together are referred to as the postings. Usually, the dictionary is kept in memory, with pointers to each postings list, which is stored on disk. The structure of an inverted index in all its components is reported in Figure 5.

To gain the speed benefits of indexing at retrieval time, you have to build the index in advance. The main steps in this are:

1. collect the documents to be indexed
2. tokenize the text, turning each document into a list of tokens
3. perform linguistic preprocessing, producing a list of normalized tokens
4. index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

---

\(^1\) The dot product $x \cdot y$ of two vectors is defined as $\sum_{i=1}^{M} x_i \cdot y_i$.

\(^2\) The Euclidean length of vector $x$ is defined to be $\sqrt{\sum_{i=1}^{M} x_i^2}$. 
The input to indexing is a list of normalized tokens for each document, which can be equally thought of as a list of pairs of term and docID. Next, this list is sorted in alphabetical order. Multiple occurrences of the same term from the same document are then merged. Instances of the same term are then grouped, and the result is split into a dictionary and postings. Since a term generally occurs in a number of documents, this data organization already reduces the storage requirements of the index. The dictionary also records some statistics, such as the document frequency, i.e. the number of documents which a specific term occurs.

### Evaluation metrics

To measure ad hoc information retrieval effectiveness in the standard way, there is the need for a test collection consisting of three things:

1. a document collection
2. a test suite of information needs, expressible as queries
3. a set of relevance judgments, standardly a binary assessment of either relevant or nonrelevant for each query-document pair.

The standard approach to information retrieval system evaluation revolves around the notion of relevant and nonrelevant documents. With respect to a user information need, a document in the test collection is given a binary classification as either relevant or nonrelevant. This decision is referred to as the gold standard or ground truth judgment of relevance.

The most frequent and basic measures for information retrieval effectiveness are reported below.

**Precision ($P$)**

It is the fraction of retrieved documents that are relevant:

\[
P = \frac{\text{#(relevant items retrieved)}}{\text{#(retrieved items)}} = P(\text{relevant} | \text{retrieved})
\]
Recall (R)

It is the fraction of relevant documents that are retrieved:

$$R = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = P(\text{retrieved} | \text{relevant})$$

Mean Average Precision (mAP)

For a single information need, Average Precision is the average of the precision value obtained for the set of top $k$ documents existing after each relevant document is retrieved, and this value is then averaged over all the information needs. That is, if the set of relevant documents for an information need $q_j \in Q$ is $d_1, ... d_{m_j}$ and $R_{jk}$ is the set of ranked retrieval results from the top result until you get to document $d_k$, then

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{Q} \frac{1}{m_j} \sum_{k=1}^{m_j} \text{Precision}(R_{jk})$$

2.3 Apache Lucene

Lucene is a high-performance Java search library that lets you easily add search functionality to any application. In recent years Lucene has become exceptionally popular and is now the most widely used information retrieval library: it powers the search features behind many websites and desktop applications [8].

2.3.1 Default model and scoring function

Lucene supports different Information Retrieval models. By the default, it combines the Boolean model (BM) - which simply assesses if a document is relevant or not to a given query - with the Vector Space Model: documents "approved" by BM are then scored by VSM.

As explained in 2.2.1, in VSM documents and queries are represented as weighted vectors in a multi-dimensional space, where each distinct index term is a dimension, and weights are tf.idf values. However, Lucene refines the VSM’s standard function - i.e. cosine similarity - for both search quality and usability. The latter is reported below:

$$score(q, d) = \text{coord}(q, d) \cdot \text{queryNorm}(q) \cdot \sum_{t \in d} t_{f_1, d} \cdot idf_t^2 \cdot \text{boost}(t) \cdot \text{norm}(d),$$

where

1. $q$ is the given query
2. $d$ is the document retrieved by Lucene
3. $t_{f_1, d} = \sqrt{t_{f_1, d}}$ is the term frequency, defined as the number of times term $t$ appears in the currently scored document $d$. Documents that have more occurrences of a given term receive a higher score
4. \( idf_t = 1 + \log \frac{N}{1 + df_t} \) is the inverse document frequency, where \( N \) is the total number of documents and \( df_t \) is the document frequency, or in other words the number of document in which the term \( t \) appears. This means rare terms give higher contribution to the total score. \( idf_t \) appears for \( t \) in both the query \( q \) and the document \( d \), hence it is squared in the equation.

5. \( coord(q, d) \) is a score factor based on how many of the query terms are found in the specified document, and it may affect the ranking. The coordination factor multiplies the score by the number of matching terms in the document, and divides it by the total number of terms in the query. Typically, a document that contains more of the query’s terms will receive a higher score than another document with fewer query terms.

6. \( queryNorm(q) = \frac{1}{\sum_{t} idf_t^2} \) is the query normalization factor used to make scores between queries comparable, without affecting document ranking since all ranked documents are multiplied by the same factor.

7. \( boost(t) \) is a boost to each query term specified at search time by users. Hence the contribution of a query term to the score of a document is multiplied by the boost of that query term.

8. \( norm(d) \) is the document length normalization which is computed and store at indexing time.

2.3.2 Index structure

The fundamental concepts in Lucene are index, document, field and term. An index contains a sequence of documents, which in turn are sequences of fields, that again are named sequences of terms which are, finally, text strings. It is important remember that the same string in two different fields is considered a different term. Thus terms are represented as a pair of strings, the first naming the field, and the second naming text within the field.

The index stores statistics about terms in order to make term-based search more efficient and follows exactly the structure of the inverted index seen in 2.2.1.

In Lucene, fields may be stored, in which case their text is stored in the index literally, in a non-inverted manner. Fields that are inverted are called indexed. A field may be both stored and indexed. The text of a field may be tokenized into terms to be indexed, or the text of a field may be used literally as a term to be indexed. Most fields are tokenized, but sometimes it is useful for certain identifier fields to be indexed literally. Also, for each field in each document, the term vector may be stored. A term vector consists of pairs of term text and corresponding term frequency.

Lucene indexes may be composed of multiple sub-indexes, or segments. Each segment is a fully independent index, which could be searched separately. Indexes evolve by either creating new segments for newly added documents either merging existing segments. Searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments.
Recently, a new class of image descriptors, built upon Deep Convolutional Neural Networks (DCNNs), have been used as effective alternative to descriptors built using local features such as SIFT, SURF, ORB, BRIEF, etc. Starting from 2012 [5], DCNNs have attracted enormous interest within the Computer Vision community because of the state-of-the-art results achieved in the image classification challenge ImageNet Large Scale Visual Recognition Challenge (ILSVRC). In Computer Vision, DCNN have been used to perform several tasks, including not only image classification, but also image retrieval [10, 11] and object detection [12], to cite some. In particular, it has been proved that the multiple levels of representation which are learned by DCNN on specific task (typically supervised) can be used to transfer learning across tasks. This means that the activation of neurons of a specific layers, preferably the last ones, can be used as features for describing the visual content [13].

Liu et al. [14] proposed a framework that adapts Bag-of-Word model and inverted table to DCNN feature indexing, which is similar to LuQ [1]. However, for large-scale datasets, Liu et al. have to build a large-scale visual dictionary that employs product quantization method to learn a large-scale visual dictionary from a training set of global DCNN features. In any case, using this approach the authors reported a search time that is one order higher than LuQ for the same dataset.
Elasticsearch is a highly scalable full-text search and analytics engine. It is open-source, distributed and built on top of Apache Lucene™ whose functions are extended to make storing, indexing, searching faster and easier. Indeed, on top of what Lucene provides, it adds its own, higher-level functionality.

Even though ES is written in Java, there’s more than a Java API that lets you work with it. It also exposes a REST API, which any application can access, no matter the programming language it was written in. Typically, a REST request has its payload in JSON (JavaScript Object Notation) format and the response is a JSON document too.

Finally, ES is organized in clusters of nodes. It is always possible to add more servers to balance the load, increase capacity and improve the fault-tolerance; similarly, servers can easily be removed from the cluster to reduce costs if the load is lower.

In this chapter, the main features and functionality of Elasticsearch are presented. A complete discussion of this topic is beyond the scope of this thesis. Therefore, more information about it can be found in the official documentation [15], in [16] and [17].

3.1 BASIC CONCEPTS

In order to better understand how ES works, there are few core concepts that need to be described and explained. They are reported below:

- Cluster
- Node
- Index
- Type
- Document

Cluster

A cluster is a collection of one or more nodes - acting as servers - that together holds the whole volume of data and provides federated indexing and search capabilities across all nodes.

Each cluster is identified by a unique name which by default is "elasticsearch". This name is important because a node can only be part of a cluster if the node is set up to join a cluster by its name. To prevent the nodes from joining the wrong cluster, it must be sure not to reuse the same cluster name in different environments. The name of the cluster to join to can be set from configuration file or command line on node’s startup.

Each cluster has a master node which is responsible for keeping information such as the nodes in the cluster, where the parts of the index are located and so on. The master node is appointed by running an election algorithm. Each time the master is unavailable, a new one is elected.
**Node**

A *node* is a single server that is part of a specific cluster, stores data, and participates in the cluster's indexing and search capabilities.

Just like a cluster, a node is identified by a name which by default is a random Universally Unique IDentifier (UUID) that is assigned to the node at startup. You can define any node name you want if you do not want the default one. This name is important for administration purposes because it allows to identify which servers in the network correspond to which nodes in the Elasticsearch cluster.

A node can be configured to join a specific cluster by the cluster name. Unless otherwise specified, each node is set up to join the cluster named "elasticsearch". In a single cluster, you can have as many nodes as you want. Furthermore, if there are no other Elasticsearch nodes currently running on your network, starting a single node will by default form a new single-node cluster named "elasticsearch".

**Index**

An *index* is a collection of documents. It is identified by a name - that must be all lowercase - and this name is used to refer to the index when performing indexing, search, update and delete operations against the documents in it.

In a single cluster, you can define as many indexes as you want, each one with its own settings.

**Type**

Within an index, you can define one or more *types* or mapping types. A type is a logical partition of the data within an index whose semantics is completely up to the developer. In general, a type is defined for documents that have a set of common fields. Typically, documents with different structures - or schemas - should be put in different types.

Creating a new type implies to define its fields and the corresponding data-type, i.e. the range of values they can assume (string, double, date, etc.). Figure 6 shows a possible organization of an index for an e-commerce site. It is logically divided in three types namely "customers-data", "products-catalog" and "order-data".

Obviously, multiple types may coexist within the same index.

---

1 An UUID is a 128-bit number used to identify information in computer systems

2 This separation is an abstraction that makes easier to work with different structures by dividing them into types. However, there’s no physical separation of documents that have different types. All documents within the same Elasticsearch index, regardless of type, end up in the same set of files
Document

A document is the smallest unit of data that can be indexed or searched for. It must be assigned to a type inside an index and, within that type, it must have an unique ID. As said before, a document must be expressed in JSON, an ubiquitous internet data-interchange text format. An example of document expressed in JSON format is reported in Figure 7.

Given its nature, a document has the following properties:

1. self-contained: it contains both the fields and their values
2. hierarchical: a field can be a set of fields with their respective values
3. schema-free: it does not depends on predefined schema.

Within an index and type, you can store as many documents as you want.

```json
{
  "title": "The Godfather",
  "director": "Francis Ford Coppola",
  "year": 1972,
  "mainCharacters": [
    {
      "characterName": "Don Vito Corleone",
      "actorName": "Marlon Brando"
    },
    {
      "characterName": "Micheal Corleone",
      "actorName": "Al Pacino"
    },
    {
      "characterName": "Tom Hagen",
      "actorName": "Robert Duvall"
    },
    {
      "characterName": "Kay Adams",
      "actorName": "Diane Keaton"
    }
  ]
}
```

Figure 7: Example of JSON document

To fix ideas, it is useful to draw some parallels with traditional relational databases. Table 3 reports the above mentioned comparison. An Elasticsearch cluster can contain multiple indices (databases), which in turn contain multiple types (tables). These types hold multiple documents (rows), and each document has multiple fields (columns).

<table>
<thead>
<tr>
<th>Elasticsearch</th>
<th>Relational Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>=&gt; Database</td>
</tr>
<tr>
<td>Types</td>
<td>=&gt; Tables</td>
</tr>
<tr>
<td>Documents</td>
<td>=&gt; Rows</td>
</tr>
<tr>
<td>Fields</td>
<td>=&gt; Columns</td>
</tr>
</tbody>
</table>

Table 3: Comparison between ES and Relational DBs

---

3 JSON is a format for expressing data structures. A JSON object typically contains keys and values, where values can be strings, numbers, true/false, null, another object or an array. More information are available at http://json.org
3.2 SHARDING AND REPLICATION

In this section the mechanisms which allows ES to scale are presented.

Sharding

A single index storing a large amount of documents may not fit on the disk of a single node or may be too slow to serve search requests from a single node alone. So, ES provides the ability to subdivide an index into multiple pieces, called shards or primary shards and to spread them among different nodes. Also, each shard is in itself a fully-functional, independent Lucene index and represents the smallest unit that ES moves from node to node. Obviously, it can be hosted on any node in the cluster.

It’s up to the developer decide how many shards create and it must be specified, once and for all, at index creation time; by default, an index is split into five chunks. This number should be chosen with attention because if it is too small you might lose any benefits of adding nodes to the cluster, e.g. if you have the whole index on a single shard adding a new node is totally useless. The other way around, creating an index with a large number of primary shards may seem like a good idea at first, but again, since a shard is a complete Lucene index, the latter requires a number of file descriptors for each segment of the index, as well as a memory overhead. So, depending on the application, a good trade-off has to be found.

Summarizing, sharding allows to:

1. horizontally split/scale your content volume
2. distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput.

Replication

In a network/cloud environment where failures can be expected anytime, it is very useful and highly recommended to have a failover mechanism in case a shard/node somehow goes offline or disappears for whatever reason. To this end, ES allows you to make one or more copies of your index’s shards into what are called replica shards, or replicas for short. Clearly, a replica shard is never allocated on the same node as the original/primary shard that it was copied from.

Similarly to primary shards, it’s up to the cluster’s administrator or developer decide how many replicas create; by default, ES sets the number of replicas to one. So, replication allows to:

1. provide high availability and fault-tolerance in case a shard/node fails
2. scale out your search volume/throughput since searches can be executed on all replicas in parallel.

In conclusion, sharding and replication are automatically handled by ES and they’re fully transparent to the user.

The number of shards and replicas can be defined per index at the time the index is created. After that, you may change the number of replicas dynamically but you cannot change the number of shards after-the-fact.
3.3 **Cluster Health**

Many statistics can be monitored in an Elasticsearch cluster, but the single most important one is cluster health, which provides an overall indication of how the cluster is functioning. The *status* can be:

- **green**: it means everything is good, the cluster is fully functional
- **yellow**: it means all data are available but some replicas are not yet allocated
- **red**: it means some data are not available for whatever reason; even if the cluster is red, it is still partially functional (i.e. it will continue to serve search requests from the working shards).

3.4 **Communications**

How you talk to Elasticsearch depends on whether you are using Java. Figure 8 illustrates how the communication takes place.

*Java API*

If you are using Java, Elasticsearch comes with two built-in clients that you can use in your code:

1. **Node client**
   The node client joins a local cluster as a non-data node. In other words, it doesn’t hold any data itself, but it knows what data lives on which node in the cluster, and can forward requests directly to the correct node.

2. **Transport client**
   The lighter-weight transport client can be used to send requests to a remote cluster. It doesn’t join the cluster itself, but simply forwards requests to a node in the cluster.

Both Java clients talk to the cluster over port **9300**, using the native Elasticsearch transport protocol. The nodes in the cluster also communicate with each other over the same port. If this port is not open, your nodes will not be able to form a cluster. Moreover, the Java client must be from the same version of Elasticsearch as the nodes; otherwise, they may not be able to understand each other.

*RESTful API*

All other languages can communicate with Elasticsearch over port **9200** using a RESTful API, accessible with a simple browser. ES provides official clients for several languages such as JavaScript, PHP, Perl, Python etc.

A request to Elasticsearch consists of the same parts as any HTTP request:

```plaintext
<METHOD> <PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY STRING> -d <BODY>,
```

where, specifically, `<METHOD>` is an HTTP method (GET, POST, PUT, HEAD, or DELETE), `<PROTOCOL>` is either *http* or *https*, `<HOST>` is the hostname of any node in the cluster, `<PORT>` is 9200 by default, `<QUERY STRING>` specifies optional parameters and `<BODY>` is a JSON-encoded request body.
3.5 INDEX STRUCTURE

In Section 3.1 the basic concepts about ES have been introduced. In particular, we have seen that an index can be split, physically in one or more shards - located eventually on different nodes - and, logically in multiple types. Everytime a new type is created within an index, the relative mapping is defined as well. Mapping is the process of defining how a document, and the fields it contains, are stored and indexed. For instance, use mappings to specify:

- which string fields should be treated as full text fields
- which fields contain numbers, dates, or geolocations
- whether a field should be indexed, stored or both
- ...

Each type has two kind of fields:

1. **Meta-fields**: they are used to customize how a document’s metadata associated is treated

2. **Fields or properties**: a list of user-defined fields pertinent to that type.

Moreover, each field has a data-type - e.g. text, keyword, date, long, double, boolean and so on - which specifies the range of values the field can assume. Since meta-fields are automatically added by ES, their data-type is known a priori and do not require to be specified. On the contrary the data-type of user-defined fields should be defined by the user itself. However, they do not need to be defined before being used. Thanks to dynamic mapping, new types and new field names will be added automatically, just by indexing a document. This autodetection has its downside because Elasticsearch might not guess right. Also, you know more about your data than ES can guess, so while dynamic mapping can be useful to get started, at some point you will want to specify your own explicit mappings.
3.5.1 Meta-fields

Meta-fields are those fields which stores metadata, i.e. the information associated with each document. The behaviour of some of these meta-fields can be customized when a mapping type is created. They are:

1. identity meta-fields: `_index`, `_type`, `_id`, `_uid`
2. document source meta-fields: `_source`
3. indexing meta-fields: `_all`, `_field_names`
4. routing meta-fields: `_parent`, `_routing`
5. other: `_meta`

Identity meta-fields

They specify information about the origin of the document. They are required by ES and can’t be disabled. They are:

- **_index:**
  it specifies the index which the document belongs to. For example, it is used when performing queries across multiple indexes and you want to add query clauses that are associated with documents of only certain indexes.

- **_type:**
  it defines the document’s type. It is useful to easily filter documents by type when you search in a specific one.

- **_id:**
  it’s the document’s ID. It’s not indexed and not stored. If you search in it, `_uid` is used instead. When you get results, contents are extracted from `_uid` as well.

- **_uid:**
  it is a composite field consisting of the concatenation of type and id and equal to the string “type#id”. It is used to identify a document within the same index. ES uses `_uid` internally for identification because, as said before, all documents land in the same Lucene indices. The type and ID separation is just an abstraction.

Document source meta-fields

Document source meta-fields specify information about the content of the document. It is:

- **_source:**
  it contains the original JSON document body that was passed at index time. The `_source` field itself is not indexed - and thus is not searchable - but it is stored so that it can be returned when executing fetch requests, like get or search. Since it is a very space-consuming field, it can be disabled (not stored neither indexed) but then some features are not supported anymore, e.g. update, reindex from and index to another, repair index corruption automatically among others. If disk space is a concern, rather increase the compression level instead of disabling this field.
Indexing meta-fields

Indexing meta-fields are those that index the other fields. Both are present by default, but they can be disabled if not needed:

- **_all:**
  it is a special catch-all field which concatenates the values of all of the other fields into one big string, using space as a delimiter, which is then analyzed and indexed, but not stored. This means that it can be searched, but not retrieved. It is particularly useful when users are looking for something without knowing where to search. Infact, it is the default field where to search if not otherwise specified.

- **_field_names:**
  it indexes the names of every field in a document that contains any value other than null. It is required to find documents that either have or don’t have any non-null value for a particular field.

Routing meta-fields

Routing meta-fields are useful to create relations between documents or to specify a custom route to follow to retrieve a document. They are not present by default, but can be added if needed:

- **_parent:**
  it is used to create a parent-child relationship between two mapping types.

- **_routing:**
  it is used to specify a custom routing value which routes a document to a particular shard. By default, the target shard - responsible for indexing a particular document - is determined on the basis of the document’s ID. However, custom routing patterns can be implemented by specifying a custom routing value per document.

Other meta-fields

- **_meta:**
  it is used to store custom meta data associated with a type. These are not used at all by ES, but can be used to store application-specific metadata, e.g. class, version etc. Clearly, they are not present by default since they have to be defined by the user.

3.5.2 Fields

These are the user-defined fields, pertinent to a given type. Their data-type should be explicitly defined, otherwise it is guessed more or less well by ES thanks to the so-called dynamic mapping.

By default these are indexed but not stored, since the whole JSON document is stored in _source. However the latter behaviour can be customized.
3.6 Operations

Among the many things that can be done with the API there are the following ones:

- check your cluster, node, and index health, status, and statistics
- administer your cluster, node, and index data and metadata
- perform CRUD (Create, Read, Update, and Delete) and search operations against your indexes
- execute advanced search operations such as sorting, filtering, aggregations, and many others.

In this section, the most important operations that can be performed and how they are executed are described. For more information please refer to the bibliography.

3.6.1 Indexing

When you index a document, it is done by following these steps:

1. the indexing application sends its request to one node in the cluster
2. the document is first sent to one of the primary shards; the latter is determined on the base of an hash of the document’s ID, specifically using the following formula

\[
shard\_num = \text{hash}(ID) \% \text{num\_primary\_shards}
\]

3. once the target shard is determined, the current node forwards the document to the node holding that shard; this is transparent to the user
4. subsequently, the indexing operation is replayed by all the replicas of that shard

Figure 9 shows an indexing operation of a document on a index divided in two shards, with one set of replicas and distributed among two nodes.

Finally, it is important remember that ES is a near real time search platform. What this means is there is a slight latency, normally one second, from the time you index a document until the time it becomes searchable.

Figure 9: Indexing in Elasticsearch
3.6.2  

Create

An Elasticsearch index and type can be created by just indexing a new document. It is ES itself that automatically takes the burden of creating a new index and defining a new mapping type, so that an user can start indexing without any prior configuration. In this case, ES’s default settings are applied to the new index and type. However, it is better to manually create a new index and type. Indeed, the latter approach allows to specify different settings than the default ones, e.g. the number of primary shards, the data-type for the mapping type’s fields and so on. Manually creating a new index and type is quite simple. One has just to define a JSON document that specifies the settings for the index and for mapping type’s fields.

3.6.3  

Get, update, delete

Other possible operations are get, update, delete.

A get allows the user to retrieve a specific document from the collection by just giving the document’s ID.

An update operation is used, as the name suggests, to update the content of a document, i.e. its JSON.

Finally, a delete permits to delete a document from the index, again, by giving its ID.

All these operations are performed similarly to the indexing one. Indeed, they are first executed on the target primary shard, and then the same operation is replayed by all the other replicas of that shard.

3.6.4  

Bulk operations

Let’s suppose that a large collection of documents need to be indexed. Indexing documents one at a time implies performance penalties because the application has to wait for a reply before it can move on, and ES has to process all data from the request for every indexed document. So, to speed-up the indexing process, ES offers a set of functions - the so-called bulk API - which can be used to index multiple documents at once, by sending a single request for all the operations to be performed. Within a single bulk, you can have any number of index or create operations and also any number of update or delete operations.

![Figure 10: Single VS bulk requests](image)
3.6.5 Searching

Things are a little bit different when a search request has to be performed. In fact, when an user looks for something:

1. the client sends its search requests to one node in the cluster
2. the node that receives the request forwards it to a set of shards containing all the index. These shards, which can be either primary or replica, are selected by ES using a round-robin policy
3. the request is executed independently on each single shard producing partial result
4. ES then gathers partial results from those shards and aggregates them into a single reply
5. finally, the reply is sent back to the client application.

An example of search request’s execution is reported in Figure 11. By default, primary and replica shards get hit by searches in round-robin, assuming all nodes in the cluster are equally fast, i.e. identical hardware and software configurations. Of course, it might be not the case. So, it is possible to organize data and configure shards to prevent the slower nodes from becoming a bottleneck.

![Figure 11: Search operation in Elasticsearch](image)

3.6.6 Adding nodes

Adding a new nodes to a cluster is as simple as configuring correctly some network settings and running new Elasticsearch processes. When another node is added, the latter automatically joins the corresponding cluster. The nodes can use two different ways to discover one another: multicast or unicast. ES can use both at once but by default is configured to use only multicast because unicast requires a list of known nodes to connect to. Moreover, when new nodes are added to a cluster, ES will automatically try to balance the number of shards evenly across all nodes, because each node added in this way shares the burden by taking a portion of the data. Figure 12 shows how the shards are re-distributed among all cluster’s nodes when a new one is added.
Removing nodes

When a node either accidentally leaves the ES cluster or it is voluntarily stopped, primary shards stored in that node - if there are - go lost. Because indexing first goes to the primary shards, Elasticsearch tries hard to make sure there are always primaries assigned for an index. So, the first thing that is done is to automatically turn the corresponding replica shards, located on different nodes, into primary shards. After ES turns the replicas for the missing primary shards into primaries, the cluster is in a yellow state which means some replica shards aren’t allocated to a node yet. Next, it is required to create more replica shards to maintain the high-availability setup for the given index. Since all the primaries are now available, the data from those ones will be replicated into replicas on the other nodes. Once the replica shards have been re-created to account for the node loss, the cluster will be back in the green state with all primary and replica shards assigned to a node.

Of course, this behaviour is achievable only if there is at least a set of replicas. If it is not the case, some shards won’t be available anymore and the cluster will be in a red state.

Figure 13 shows an example of what happens when a node is removed from the cluster.
Since Elasticsearch is based on Lucene, its default scoring function is the same as the Lucene one, seen in 2.3.1, which is reported here for convenience:

\[
\text{score} (q, d) = \text{coord} (q, d) \cdot \text{queryNorm} (q) \cdot \sum_{t \in d} tf_i d \cdot idf_i^2 \cdot \text{boost} (t) \cdot \text{norm} (d),
\]

However, with ES it is also possible to use different scoring functions. There are a set of predefined functions, or it is possible to define your custom scoring functions giving a boost to some documents than others.
FEATURES ENCODING

As introduced in Chapter 1, a CBIR can be achieved by exploiting a standard full-text search engine. This can be obtained by encoding the information associated with the feature vector, which represents the content of the image, into a text form that can be analyzed and indexed.

To this end, we have used the approach proposed by Amato et al. [1] and in the following, we provide a brief introduction of this method.

4.1 Overview

The proposed method is designed to specifically index DCNN Features to support efficient content-based on large datasets and it is suitable to be used by any full-text engine supporting vector space model.

In principle, to perform similarity search on DCNN feature vectors, one should compare the vector extracted from the query with all the vectors extracted from the images of the dataset, and take the nearest images according to L2 distance. Unfortunately, if the database is large, this brute-force approach can be time-consuming.

However, given the sparsity of the DCNN features, which contain mostly zeros (about 75%), it is possible to use an inverted index, as explained in Chapter 2. In fact, it has been noticed that since typically DCNN feature vectors exhibits better results if they are L2-normalized to the unit length [18][19], if two vectors $x$ and $y$ have length equal to one, the following formula between the L2 distance $d_2(x, y)$ and the inner product $x \cdot y$ is valid:

$$d_2(x, y)^2 = 2 \cdot (1 - x \cdot y).$$

The advantage of computing the similarity between vectors in terms of inner product is that it is possible to efficiently exploit the sparsity of the vectors by accumulating the product of non-zeroes entries in the vector $x$ and their corresponding non-zeros entries in the vector $y$. Moreover, most of the search engines, computes the similarity between documents using the cosine similarity, which is the inner product of the two vectors divided by their lengths product, as seen before. Therefore, in this case, cosine similarity and inner product are the same.

So, the idea is to fill the inverted index with the DCNN features vectors. For space-saving reasons, however, text search engines do not store float numbers in the posting entries of the inverted index representing documents, rather they store the term frequencies, which are represented as integers. This method, associates each component of the feature vector with a unique term and generates a text string in which the relative keyword is proportional to the float value of the corresponding deep feature entry.

The algorithm is presented in detail in the following section.
Let \( w = (w_1, w_2, ..., w_m) \) be the L2-normalized DCNN vector of \( m \) dimensions. Each of its component \( w_i \) is associated with an unique alphanumeric term \( \tau_i \), e.g. the prefix ‘f’ followed by the numeric value corresponding to the index \( i \). Then, the text encoding \( \text{doc}(w) \) corresponding to the vector \( w \) is given by the following formula:

\[
\text{doc}(w) = \bigcup_{i=1}^{m} \lfloor Q \cdot w_i \rfloor \bigcup_{j=1}^{m} \tau_i,
\]

where \( \lfloor \cdot \rfloor \) denotes the floor function and \( Q > 1 \) is the quantization factor. An implementation in pseudo-code is reported below.

```plaintext
encode(w, Q) {
    // initialization
    encoding = "";
    m = w.length;
    for(i=0; i<m; i++)
        for(j=0; j<floor(w[i] * Q); j++)
            encoding += "f" + (i + 1) + ",";
    return encoding;
}
```

4.3 QUANTIZATION FACTOR

The aforesaid algorithm introduces a quantization error due to the representation of float components in integers. However, if \( Q \) is properly set, this error has little impact on the retrieval effectiveness.

Both the accuracy of the approximation and the index’s size depend on the quantization factor \( Q \). Clearly, the greater is \( Q \) the better is the accuracy. The other way around, the smaller is \( Q \), the smaller the inverted index will be, because the floor function will set to zero more entries in the posting lists. So, a good trade-off between the effectiveness of the retrieval system and its space occupation has to be found.

From the experiments conducted in [1], the authors have concluded that an optimal choice of the quantization factor is \( Q = 30 \), which leads to a good trade-off between accuracy and index size.

In particular, they have evaluated the optimal value of \( Q \) over the Flickr1M dataset\(^2\). Figure 14 shows the mAP as function of \( Q \) and the corresponding space occupation.

\(^1\) The union operator \( \bigcup \) denotes the space-separated concatenation of keywords.

\(^2\) http://press.liacs.nl/mirflickr/
of the inverted index. The choice of the quantization factor $Q = 30$ leads to a mAP of 0.62 and a space occupation of 2.31 GB. They stress that the mAP using the brute-force approach (on the exact DCNN feature vectors) is about 0.60. This means that the quantization error leads to a slight improvement of the precision for $Q \geq 30$.

Another important aspect is that this effectiveness was obtained forcing Lucene to use the standard inner product on $tf$ weight without $idf$ and any other document normalization. A further improvement can be obtained using the similarity function of Lucene, which provided a mAP of about 0.64.

![Figure 14: Effectiveness (mAP) vs space occupation for increasing values of the quantization factor $Q$ for the Flickr1M dataset](image)

4.4 Query Reduction

Another aspect that we have to take into careful consideration is the query’s size. For instance, choosing $Q=30$, i.e. the optimal value, and taking the DCNN fc6 layer, which is a 4096-dimensional vector, leads to documents containing on average 275 unique terms (which means that about the 93% of the entries are zero). So, when performing a similarity search, the search engine have to treat query of this size. These unusual long queries, however, can affect the response time if the inverted index contains million of documents.

A way to overcome this issue is to reduce the size of the query by exploiting the knowledge of the $tf_{id} \ast idf_1$ statistic, which comes for free in standard full-text retrieval engines. The idea is to retain the elements of the query that exhibit greater values of $tf_{id} \ast idf_1$ and eliminate the others. For example, for a query of about 275 unique terms on average, taking the first ten terms that exhibits the highest $tf.idf$ results in a query time reduction of about 96%. This procedure comes, however, with a price: it decreases the precision of results. To attenuate this problem, for a top-k query, it is a good idea to reorder the results using the cosine similarity between the original query
(i.e., the one without reduction) and the first $C_r \times k$ candidates documents retrieved, where $C_r$ is an amplification factor called reordering factor. For instance, if we have to return $k = 100$ images and we set $C_r = 10$, we take and reorder the first $10 \cdot 100 = 1000$ candidate documents retrieved by the reduced query.

In order to calculate the cosine similarity of the original query and the $C_r \times k$ candidates, the quantized features need to be reconstructed by accessing to the posting list of the document returned by the search engine. This approach does not affect significantly the efficiency of the query but can offer great improvements in terms of effectiveness. Figure 15 shows the query reduction mechanism.

![Query reduction mechanism](image_url)

Figure 15: Query reduction mechanism
IMPLEMENTATION

In this chapter, we describe the implementation of a Content Based Image Retrieval (CBIR) System built on top of our Elasticsearch index.

Then, we show its employment on the YFCC100M dataset, which is the one taken as reference to conduct experiments.

Finally, we explain how the designed web-based GUI has been realized and how it interacts with the Elasticsearch cluster.

5.1 OVERVIEW

As mentioned in Chapter 1, the main goal of this work is to build an Elasticsearch-based CBIR system, suitable to index and search through very large image databases. In fact, thanks to the sharding mechanism provided by ES, and explained in Section 3.2, it is possible to split an index into different shards and spread them among the nodes of the cluster, so that a given query is distributed and each chunk of the index is searched separately. As consequence, the searching time is reduced, with respect to the case in which the whole index is located on the same physical node. In this way, we are taking advantage of the parallel processing and storing power of different physical servers.

To this end, we have implemented a Java application. Starting from a given local set of images, it is able to extract deep features from them, encodes them into a textual form and, finally, index them by connecting to an Elasticsearch server. Once the index has been built, it allows us to perform visual similarity and textual search against the given index. More details on this are available in Section 5.2.

Even though queries can be executed using the Java application, a web interface is more serviceable. Thus, in order to allow the users to perform both visual similarity and textual search, a web-based GUI has been implemented. The GUI connects directly to the Elasticsearch cluster by means of the REST interface and sends the requests of the users addressing the search REST endpoint. Figure 16 shows this architecture.

![Figure 16: System overview](image-url)
5.2 Elasticsearch-based CBIR from Scratch

A said before, in order to build an ES-based CBIR from scratch, we have developed a Java application, which provides the following functionalities:

- extract the deep features from a local image dataset
- store the features into a binary file, since the extraction process is very costly
- connect to an Elasticsearch cluster
- manage the cluster and its indexes
- execute CRUD (Create Read Update Delete) operations against the indexes
- perform both visual similarity and textual search.

In the following parts of this section, we describe how these operations take place and the main implemented Java classes.

5.2.1 Features extraction and storing

Deep features are extracted by means of the open-source library JavaCV\(^1\), which is a "non-official" Java wrapper of the OpenCV\(^2\) library. Written in C++, OpenCV is a library of programming functions mainly aimed at real-time computer vision. Even if it is cross-platform, unfortunately, the official Java wrapper for OpenCV does not include the C++ APIs for DCNN, so we had to switch necessarily to JavaCV.

The APIs provided by JavaCV allows one to load serialized models of different dnn-frameworks, and then it is possible to extract the features of the specified layer from a given image. In particular, the neural network Hybrid-DCNN for the deep learning framework Caffe\(^3\) [20] has been used. Its model and weights are publicly available in the Caffe Model Zoo\(^4\).

So, the DNNExtractor Java class has been defined. Basically, it imports the Caffe files defining the neural network and provides the extract() method. The latter takes as input a File pointing to an image in the file system and, as the name suggests, returns a float[] which represents the deep features extracted from it. The class Parameters contains only constant strings and it is used to specify some configurations for the DNNExtractor such as the name of the layers of the network etc.

Once the features have been extracted, all the information about the image are saved in the ImgDescriptor class, which stores the ID of the image, its URI, optionally a textual description and its L\(_2\)-normalized deep features.

Since the extraction process is very expensive, the latter can be skipped the next time by serializing and storing the ImgDescriptors on the mass memory. This is done by the FeaturesStorage class which provides two methods, store() and load(). The first

---

1 https://github.com/bytedeco/javacv
2 http://opencv.org
3 http://caffe.berkeleyvision.org
4 http://github.com/BVLC/caffe/wiki/Model-Zoo
takes as input a list of `ImgDescriptors` and stores them in the output file; viceversa, the other one loads the image descriptors from the disk.

Finally, the `SeqImageStorage` class allows to automatically extract and store the deep features from an entire folder. Of course, it is based on the other classes described before. Figure 17 shows the class diagram for the Java classes described before.

![Class Diagram](image)

Figure 17: Class diagram for the classes performing features extraction

### 5.2.2 Interaction with Elasticsearch

In order to communicate with Elasticsearch, we have implemented the class `ElasticIndexManager`. Basically, this class relies on an instance of the ES `TransportClient` class. As said in 3.4, a `TransportClient` is a light ES node that doesn’t join the cluster itself, but simply forwards requests to a node in the cluster. 

`ElasticIndexManager` defines multiple methods which allow to perform the following operation:

- **connect to or disconnect from a set of nodes belonging to the same cluster:**
  this is achieved by simply calling the `connectTo()` method or the `disconnectFrom()` one, respectively. Both require to specify the address and the port of the node you want to connect to or disconnect from.  
  Finally, if you want definitively the client to leave the cluster, you should call the `close()` first.

- **create or delete a new index or type:**
  the `createIndex()` method creates a new index in the cluster with the specified name and settings (number of primary and replica shards, ... ). The `deleteIndex()` removes an index by its name.
  To create a new type within an index you can use the `putMapping()` which takes
as input the name of the index, the name of the type you want to create and the mapping type expressed using the JSON format.

- **CRUD operations within a given index and type:**
  - `index()` indexes a new JSON document within the given index and type.
  - `get()` retrieves an already indexed document on the base of its ID.
  - `update()` and `upsert()` update the content of a document. The difference between them is that the second one creates and indexes a new JSON document if the latter is not already present in the index.
  - `delete()` deletes an existing document, given its ID.

- **Bulk operations:**
  as explained in 3.6.4, executing operations one at a time implies performance penalties because the application has to wait for a reply before it can move on, and ES has to process all data from each single request separately. So, to speed-up this process, ES offers a set of functions - the so-called *bulk API* - which can be used to execute multiple tasks at once, by sending a single request for all the operations to be performed.
  Bulk requests can be sent to index, delete and update documents by using respectively the `bulkIndex()`, `bulkDelete()` and `bulkUpdate()` methods which take as input a list of JSON documents or IDs.
  `ElasticIndexManager` offers another set of functions to send bulk request, namely `indexDocBulkProcessor()`, `deleteDocBulkProcessor()` etc. These uses the `BulkProcessor` class which offers a simple interface to flush bulk operations automatically based on the number or size of requests, or after a given period.

- **Search for documents:**
  the `queryStringSearch()` allows us to perform a query string search on the specified index and type. The inputs to this method are the name of the index and of the type, the field you want to search in, the query text string and, finally, the result’s size. As far as the scoring function is concerned, this method uses the Elasticsearch default cosine similarity function, that is, in turn, the default one of Lucene, explained in 2.3.1.

- **Other possible methods are listed below:**
  - `getTermVector()` and `getTermVectors()` respectively retrieve the term vector from a given field of a specific document and all the term vectors from a list of documents. Obviously, the field of the document from which you want to get the tv has to have the *term vector* property enabled, otherwise it is not possible to retrieve it.
  - `getDocumentCount()` gets the total number of indexed document in an index and/or in a type within an index.
  - `forceRefresh()` refreshes the index and makes immediately searchable the just indexed documents. In fact, as said in 3.6.1, ES is a *near real time* platform, which means that there is a slight latency, normally one second, from the time you index a document until the time it becomes searchable.
  - `setReplicas()` allows to change dynamically the number of replica shards for a specific index.

Figure 18 shows the class diagram for the aforesaid Java class.
The ElasticImageIndexManager class brings together all the functionalities seen so far, to automate the features extraction and indexing process. It is the actual class that builds the Elasticsearch-based CBIR. Basically, it is derived from the ElasticIndexManager superclass and extends its methods to make things easier.

To index a local image dataset, you can follow these steps:

1. create a new instance of the ElasticImageIndexManager class, specifying the clustername and the quantizationFactor to be used

2. extract and load the deep features into ImgDescriptor objects using the extractAndLoad() method or just load them from binary file by calling loadImgDescriptorsFromFile()

3. connect to the cluster, create a new index and call the indexImgDataset() which indexes all the images in the dataset by sending bulk requests to the ES server.
Of course, the images are first encoded as text strings by means of the static method `encode()` belonging to the `DeepFeatureEncoder` class. Given as input the `float[]` representing the deep features extracted and the `quantizationFactor`, it returns the textual encoding according to the algorithm described in Section 4.2.

The JSON document to be indexed - and representing the image - is created with the utility class `JsonImageBuilder`, in particular thanks to its `build()` static method. Each document in the Elasticsearch index made in this way has three `fields`, in addition to the `meta-fields` (_index, _type, _uid, _source, ...): `deep`, `txt` and `uri`. The first indexes the deep features, the second the textual description of the image and the last one its URI.

It is important to remember that if you don’t give any explicit `mapping-type` for the index, ES will use the `dynamic mapping` to guess the data-type and it will apply the default settings for all the `fields` and `meta-fields`. A custom `mapping-type` can be explicitly given to ES by means of the `putMapping()` method.

Once the index has been built, we can perform both visual similarity and textual queries. `visualSearch()` takes the ID of an already indexed image as input and looks for visual similar images. For test reasons, we have implemented also the query reduction mechanism described in Section 4.4: a reduced query can be executed by calling `visualSearchQR()`, while the version with the reordering of the results is coded by the `visualSearchQRReordered()` method. Figure 19 shows the class diagrams.

![Class diagrams for ElasticImageIndexManager, DeepFeatureEncoder and JsonImageBuilder](image-url)
5.3 APPLICATION ON THE YFCC100M DATASET

In order to test the Elasticsearch-based CBIR, we have decided to use the YFCC100M dataset. As mentioned in Chapter 1, it consists of approximately 99.2 million photos and 0.8 million videos, all uploaded to Flickr between 2004 and 2014 and published under a Creative Commons commercial or non commercial license. Metadata for the YFCC100M dataset are publicly available through Yahoo! Webscope. YFCC100M images can be obtained directly from Flickr! using the information reported in the metadata or through the Multimedia Commons Initiative website using the hash of the original image also reported in the metadata.

In principle, to build an ES-based CBIR for the YFCC100M dataset using the Java application described in the previous section, we should download the whole dataset, extract the deep features and index them together to their metadata. However, we have skipped this operation because a Lucene index of the YFFC100M was already available. In this way, we simply had to read data from the Lucene index and re-index them in ES.

5.3.1 YFCC100M Lucene index

This section explains first how the YFCC100M Deep Features have been extracted, then how the relative Lucene index has been built and, finally, which are its fields and its properties.

Deep features have been extracted with the same trained model of the HybridNet mentioned in 5.2.1, publicly available for the popular Caffe framework. The authors have chosen the HybridNet for several reasons: first, its architecture is the same of the famous AlexNet; second, experiments conducted on various datasets demonstrate the good transferability of the learning. In particular, data are 4096 dimensional L₂ Normalized vectors corresponding to the activation of the neurons of the HybridNet fc6 layer after the ReLU. This activation function, which is part of the HybridNet Convolutional Neural Network, simply sets to zero all the elements of the vectors that are negative. The distance to be used to compare them is the Euclidean (aka L₂ distance). The ranking of the results for a query using a combination of the L₂ normalization and Euclidean distance is the same to the ones using the Cosine similarity of the original feature vectors.

Starting from the deep features so obtained, a Lucene index - i.e. an index located on a single shard - has been built using a quantization factor Q=30. Its fields with the corresponding properties are reported in Table 4. The meaning of indexed, stored and term vector is that explained in 2.3.2. Specifically, as regards the indexed property, the fields ID and URI index only the ID of the document in the posting list, while DEEP and TXT index the term frequencies too. In both cases positions and offsets are omitted.

---

5 https://webscope.sandbox.yahoo.com/catalog.php?datatype=i&did=67
6 https://multimediacommons.wordpress.com
7 http://lucene.apache.org/core/4_6_0/core/org/apache/lucene/index/FieldInfo.IndexOptions.html
To import data from a Lucene index, the `LuceneIndexReader` class has been defined and implemented. For each indexed document, it can read the value of all its *stored* fields or reconstruct the content of a field starting from its *term vector*. Obviously, fields that are neither stored neither have the term vector can not be imported.

Once a new instance has been created by giving as input 1) the directory containing the Lucene index, 2) the name of the fields stored and 3) that of the fields whose value can be obtained from the term vector, we can use the following methods:

- `readDocument()` retrieves either the value of the stored fields either the term vectors from the document with id `docId`, and returns them in a `Map<String, Object>`.
- `getNumIndexedDocuments()` returns the total number of documents *indexed*.
- `getNumAllDocuments()` returns the total number of *all* the documents in the index. The difference with respect to `getNumIndexedDocuments()` is that this function takes into account also the *deleted* documents. Indeed, it is important to underline that, in Lucene, for efficiency reasons, *deleted* documents are not really removed from the index, but they are simply marked as such. Therefore, the number obtained with this method is always greater or equal to the one returned by the previous one.
- Finally, the static method `getTextFromTermVector()` reconstructs the text of a field, given the term vector as input. Since it does not take into account the positions of the terms but only their frequencies, it is very likely that the text is different from the original one but the term vectors are the same.

Figure 20 shows its class diagram.
5.3.3 YFCC100M Elasticsearch index

We have imported the YFCC100M Lucene index in Elasticsearch using the classes described before. In particular:

1. we have created an index - called **cbir** - with **16 shards** and **no replicas**.
   We have decided to split the index in 16 chunks because it represents a good trade-off between obtainable level of parallelization and overhead introduced. In fact, the index can be spread on up to 16 different servers and the overhead is reasonable.
   The other way around, the reason why we have decided to have no replicas during the indexing process is straightforward. Indeed, as mentioned in 3.6.1, the indexing operations would be replicated by all the replica shards, leading to a performance degradation

2. we have created explicitly the mapping type **yfcc100m** within the **cbir** index.
   We have kept the default **meta-fields**, except for the **_all** which has been disabled.
   The **_source** has been maintained as well, because we do not want to lose the chance of automatically re-index using a different number of primary shards.
   As regards the **fields**, we have used exactly the same properties as the ones of the Lucene index. The only difference is that all the fields are not **stored**, since their content is already present in the **_source**. The JSON document representing the **mapping type** is reported below

```json
{
  "cbir" : {
    "mappings" : {
      "yfcc100m" : {
        "_all" : {
          "enabled" : false
        },
        "properties" : {
          "deep" : {
            "type" : "string",
            "term_vector" : "yes",
            "index_options" : "freqs"
          },
          "txt" : {
            "type" : "string",
            "term_vector" : "yes",
            "index_options" : "freqs"
          },
          "uri" : {
            "type" : "string",
            "index_options" : "docs"
          }
        }
      }
    }
  }
}
```
3. finally, we have imported the YFCC100M Lucene index by sending indexing bulk requests every 100,000 read documents.

The index so built has the structure reported in Table 5.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Meta-field</th>
<th>Indexed</th>
<th>Stored</th>
<th>Term Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>_field_name</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>_source</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>_type</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>_uid</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>_version</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>deep</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>txt</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>uri</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5: YFCC100M Elasticsearch index structure

5.4 WEB-BASED GUI

This section presents the web-based GUI developed to allow the users to perform both visual similarity and textual search.

It has been designed as an Elasticsearch site plugin. Plugins\(^8\) are a powerful way to extend or enhance the functionality that Elasticsearch provides out of the box. In particular, they are split into two categories: site plugins and code plugins.

A site plugin is one that provides no additional functionality; it simply provides a web page served by ES.

A code plugin is any plugin that includes JVM code that ES executes; this can include plugins that add features to Elasticsearch or even plugins that replace its internal parts such as the shard distributor and discovery mechanisms.

To implement the GUI, we have used:

- jQuery\(^9\)
  it is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. In particular, it has been used to navigate the document, select DOM elements and create animations.

- Elasticsearch.js
  it is the official JavaScript library provided by ES to interact with the cluster of nodes, performing operations on the index. It sends HTTP requests over the port 9200, addressing specific REST endpoints. More in detail, it uses the .ajax() functionality of jQuery.

- Bootstrap\(^10\)
  it is a free and open-source front-end web framework for designing websites

---

8 https://www.elastic.co/guide/en/elasticsearch/plugins/2.4/intro.html
9 https://jquery.com
10 http://getbootstrap.com
and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It has been used to simplify the development and the management of the graphical components.

Figure 21: Web-based GUI

5.4.1 Operations

The main operations that the users can perform using the GUI and how they are executed are presented below. They are:

- random search, which shows a small random set of images
- textual search, which searches the input keywords on the textual description of the images
- visual search, which can be performed using as input a displayed image, an ID or a picture loaded from your local file system

Random search

A random search retrieves 20 documents at random from the index and displays the corresponding images. Starting from these images, the users can look for similar ones just by clicking on one of it.

A random search is performed whenever either the HTML page is loaded for the first time, either when the user clicks on the “Random search” button in the navigation bar. Figure 21 shows the GUI after that the page has been loaded.

To perform such operation the /cbir/yfcc100m/_search REST resource is addressed and the corresponding HTTP request is the following one:
From the JSON document representing the response, we extract the uri field and we use it to display the images. The sequence diagram is reported in Figure 22.

**Random search**

![Sequence diagram for the "Random search"

**Textual search**

An user can perform a textual search by clicking on the "Random search" button in the navigation bar. Next, an input form is displayed where the user can type the text of its query. The keywords are searched in the txt field and the size of the result is 120. The corresponding HTTP request and the sequence diagram are shown below.
Textual search

Visual search

As introduced before, a visual search can be performed either on an already indexed image or on an image uploaded by the user.

In the first case, we can search an already indexed image by clicking on the image itself or by typing its ID. However, these two methods are executed exactly in the same way: first, we retrieve the uri field and the term vector from the deep field; then, we execute a search on the deep field, in a similar way to the case of the textual one. As usual, the HTTP requests and the sequence diagram (Figure 24) are reported below.

```
GET /cbir/yfcc100m/imageID?fields=uri
GET /cbir/yfcc100m/imageID/_termvectors?fields=uri
POST /cbir/yfcc100m/_search?search_type=dfs_query_then_fetch&size=120&fields=
    {  
        "query": {  
            "query_string": {  
                "fields": ["deep"],  
                "query" : TEXT_TO_SEARCH  
            }  
        }  
    }
```

On the other side, to allow the users to look for similar images starting from an uploaded one, we have developed an Elasticsearch code plugin. Basically, it adds a new REST endpoint to ES which, given the Base64 encoding of the uploaded image, extracts the features from it and returns its textual encoding.

To implement such plugin we have defined two main classes: the ImageEncoderRestAction class and the ImageEncoderPlugin.

The first one extends the superclass BaseRestHandler, defines the new resource /_encode and the actions to be performed when the REST endpoint is addressed. Given the Base64 encoding of the image as parameter in the body of the request, it produces the textual encoding using the methods exposed by the DNNExtractor and
Visual search on an existent image

DeepFeatureEncoder classes, described in 5.2.1 and 5.2.3. In particular, this is achieved by overridding the handleRequest() method of the superclass BaseRestHandler.

ImageEncoderPlugin is derived from the Plugin superclass. It simply wraps the other classes and defines the module to be loaded during the bootstrap of Elasticsearch.

The textual encoding is returned in a JSON document with a single field, namely txt. The details about the /_encode endpoint and its parameters are presented in Table 6.

So, when an user uploads an image, the latter is first transformed in a textual form using the Base64 encoding, then it is sent as input to the /_encode endpoint. Once the response, containing the textual encoding of the image, comes back to the client, this is used to perform a new search request to ES. This behavior is shown in the sequence diagram of Figure 25.

![Sequence diagram for the "Visual search" on an existent image](image)

**Table 6: /_encode REST endpoint**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Mandatory</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>img</strong></td>
<td>the Base64 encoding of the image</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td><strong>qFactor</strong></td>
<td>the quantization factor to be used to produce the textual encoding</td>
<td>No</td>
<td>30</td>
</tr>
<tr>
<td><strong>layer</strong></td>
<td>the layer of the DCNN from which you want to extract the features</td>
<td>No</td>
<td>fc6</td>
</tr>
</tbody>
</table>

Figure 24: Sequence diagram for the "Visual search" on an existent image
Visual search on an uploaded image

Cluster information

We can check anytime the state of the cluster and the total number of indexed documents. These information are displayed in the left corner of the navigation bar and are obtained by sending the following HTTP requests:

```
GET /_cluster/health/cbir

POST /cbir/yfcc100m/_count
```

They are executed either when the page is loaded for the first time, either when the user clicks on the relative "refresh" button.
5.4.2 Screenshots

We report below some example of query execution.

Visual similarity search

Figure 26: Visual similarity search
Textual search

(a) 1st example: 'statue of liberty nyc'

(b) 2nd example: 'pisa italy'

Figure 27: Textual search
EXPERIMENTS

In this chapter, we describe the experiments that we have conducted to evaluate our Content Based Image Retrieval System built on top of Elasticsearch.

6.1 setup

Our tests rely on the YFCC100M dataset and, in particular, on the ES-index described in Chapter 5, which consists of 16 shards. Due to the balancing mechanism of ES, each shard indexes about 6.2M images and has approximately a size of 14GB. Thus, the shards can be considered equally each other. The details are provided in Table 7.

Since for this dataset a groundtruth is not available, we only report the performance of the query in terms of average search time. Specifically, we have taken as reference a real context in which we want to scale, by adding a new node to the cluster, when the data volume grows.

Thus, our goal is to prove its scalability, i.e., our CBIR System is able to keep more or less constant the query execution time when the data volume grows and a new node is added.

To this end, we have distributed evenly - i.e two shards per node - our index among 8 different servers with comparable hardware and software. This last point is very important, given the way ES executes the query. In fact, if a node performs poorly than the others, it becomes the bottleneck of the whole cluster slowing down the query execution process. In particular, all the machines are equipped with 12-16 GB RAM and 7200 RPM HD.

<table>
<thead>
<tr>
<th>Shard</th>
<th># Documents</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6,191,849</td>
<td>14.3</td>
</tr>
<tr>
<td>1</td>
<td>6,199,214</td>
<td>14.4</td>
</tr>
<tr>
<td>2</td>
<td>6,195,526</td>
<td>14.3</td>
</tr>
<tr>
<td>3</td>
<td>6,196,288</td>
<td>14.3</td>
</tr>
<tr>
<td>4</td>
<td>6,201,858</td>
<td>14.4</td>
</tr>
<tr>
<td>5</td>
<td>6,196,296</td>
<td>14.3</td>
</tr>
<tr>
<td>6</td>
<td>6,197,192</td>
<td>14.3</td>
</tr>
<tr>
<td>7</td>
<td>6,200,368</td>
<td>14.4</td>
</tr>
<tr>
<td>8</td>
<td>6,197,934</td>
<td>14.4</td>
</tr>
<tr>
<td>9</td>
<td>6,200,047</td>
<td>14.4</td>
</tr>
<tr>
<td>10</td>
<td>6,196,954</td>
<td>14.3</td>
</tr>
<tr>
<td>11</td>
<td>6,194,848</td>
<td>14.3</td>
</tr>
<tr>
<td>12</td>
<td>6,202,070</td>
<td>14.4</td>
</tr>
<tr>
<td>13</td>
<td>6,196,648</td>
<td>14.3</td>
</tr>
<tr>
<td>14</td>
<td>6,196,936</td>
<td>14.4</td>
</tr>
<tr>
<td>15</td>
<td>6,194,659</td>
<td>14.3</td>
</tr>
</tbody>
</table>

Table 7: Details about the shards
6.2 Results

To evaluate the query execution time, we have adopted an incremental approach which consists of the following steps:

1. pick 100 images at random from the available shards
2. compute the average search time on the available shards, giving as input the images selected at the previous point
3. add a new node to the cluster
4. return to step 1.

First, we have measured the mean search time using as input full-length queries. As mentioned in Section 4.4, these are unusual long queries, since they contain on average 275 unique terms, and may lead to a degradation of the response time if the inverted index contains million of documents as in our case. Thus, we have repeated the same procedure exploiting the query reduction mechanism. Specifically, we have used queries of length $L_q$ - i.e. query reduced to the first $L_q$ terms with the greatest $tf.idf$ value - with $L_q = 10, 20, 30, 40, 50$.

The results are presented in Table 8. Using as input a full-length query we have obtained a mean search time of about 25 seconds which is a good result considering that on a single node the query time is of the order of several minutes. However, we are able to reduce drastically this time by using the reduced version.

Moreover, we have plotted these results which are shown in Figure 28 using a logarithmic scale for the y-axis. The graph shows a constant trend for all the curves, which is exactly what we expected.

<table>
<thead>
<tr>
<th># nodes</th>
<th>Full (s)</th>
<th>$L_q=50$ (s)</th>
<th>$L_q=40$ (s)</th>
<th>$L_q=30$ (s)</th>
<th>$L_q=20$ (s)</th>
<th>$L_q=10$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27.14206</td>
<td>3.67346</td>
<td>3.13765</td>
<td>2.68853</td>
<td>2.67118</td>
<td>0.55386</td>
</tr>
<tr>
<td>2</td>
<td>24.49986</td>
<td>3.13516</td>
<td>2.50177</td>
<td>1.85854</td>
<td>1.32713</td>
<td>0.48002</td>
</tr>
<tr>
<td>3</td>
<td>23.97848</td>
<td>3.00497</td>
<td>2.33163</td>
<td>1.71104</td>
<td>1.162</td>
<td>0.56007</td>
</tr>
<tr>
<td>4</td>
<td>27.11426</td>
<td>2.89218</td>
<td>2.2722</td>
<td>1.69005</td>
<td>1.19286</td>
<td>0.72626</td>
</tr>
<tr>
<td>5</td>
<td>27.08174</td>
<td>2.92933</td>
<td>2.32352</td>
<td>1.71497</td>
<td>1.15817</td>
<td>0.61908</td>
</tr>
<tr>
<td>6</td>
<td>25.82184</td>
<td>2.95874</td>
<td>2.41766</td>
<td>1.98111</td>
<td>1.79357</td>
<td>0.47475</td>
</tr>
<tr>
<td>7</td>
<td>25.93324</td>
<td>2.89701</td>
<td>2.2938</td>
<td>1.70026</td>
<td>1.51673</td>
<td>0.62392</td>
</tr>
<tr>
<td>8</td>
<td>25.81558</td>
<td>2.91148</td>
<td>2.27257</td>
<td>1.64934</td>
<td>1.0394</td>
<td>0.48513</td>
</tr>
<tr>
<td>AVG</td>
<td>25.923325</td>
<td>3.05029125</td>
<td>2.44385</td>
<td>1.87423</td>
<td>1.476905</td>
<td>0.56538625</td>
</tr>
</tbody>
</table>

Table 8: Results
Figure 28: Results
CONCLUSIONS

In this work, we have presented a distributed system for content-based image retrieval on very large image databases. It relies on the distributed search engine software Elasticsearch, which allows us to split an index into several parts and to spread them among different servers, so that search operations can be parallelized reducing as consequence the query execution time.

In order to enable the full-text search engine to perform visual similarity search, we have used Deep Convolutional Neural Network Features extracted from the images of the dataset and encoded as standard text.

We have tested our System on the YFCC100M dataset only in terms of average search time, since for this dataset a groundtruth is not available. From this experiment, we have proved its scalability, i.e., our CBIR System is able to keep more or less constant the query execution time when the data volume grows and a new node is added. Moreover, we can further reduce the response time by exploiting the query reduction mechanism at the cost of a lower quality of approximation.

Finally, we have implemented a web-GUI so that the users can perform both visual similarity and textual search. The goodness of the textual search relies on the metadata of the images. For instance, the YFCC100M dataset contains several images without any textual description. As future work, we will investigate new automatic tagging mechanisms based on the metadata of visual similar images.
BIBLIOGRAPHY


