UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONA TECH

Real-Time Near Replica Detection Over Massive Streams Of Shared Photos

by

Daniel Mora de Checa

Directed by

Rubèn Tous Liesa

A thesis submitted in partial fulfillment for the degree of Enginyer Superior en Informàtica

in the
Facultat d'Informàtica de Barcelona (FIB)
Department of Computer Architecture

June 2015

Acknowledgements

I would like to thank my parents, my sister, Marc, Marta and my university colleagues and friends for giving me support throughout these months. I am also grateful the Barcelona Super Computing Center for providing computing resources.

I want to specially thank my tutor Rubèn, for encouraging and guiding me and for sharing his expertise, time and experience.

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONA TECH

Abstract

Facultat d'Informàtica de Barcelona (FIB)
Department of Computer Architecture

by Daniel Mora de Checa

Being able to identify whether an image is a modified version of another is a common problem in the web and useful in spam detection, redundant information detection or copyright infringement detection. In this work we present the distributed, scalable and real-time implementation of an image near replica detection.

Local features have been used to detect and describe interest points by building invariant feature vectors around them. Filtering has been used to discard weak image features and afterwards they have been sketched into binary representations to embed them into a Hamming space and to reduce their dimensionality. To estimate the similarity between two images we compute the similarity between their features by discarding those whose Hamming distance is below a threshold. An approximation of the Hamming distance has been used to reduce the search space to just a small amount of candidates.

Results show that we can detect multiple common image modifications achieving precisions below 0.01 within few seconds on both memory and disk using resources efficiently.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Being able to identify if an image is a replica of another one is a common problem in the web. We can see how modification of images are usually spread around social media or how duplicate results are retrieved in image search engines.

Precise near replica detection can be achieved with high computational, memory costs and time. However, considering the big growth of the data and the need to find real-time solutions, we cannot afford these burdens. Though it has not been one of the main focus of work in computer vision and information retrieval, it has been tackled from many different sides finding promising solutions but there is no public implementation fitting these needs. In this work we want to design and build a framework capable of identifying replicas in a feasible amount of time time and being able to scale easily using low memory and disk storage requirements.

1.2 What is a replica?

Before getting into details we must define what a replica is. We consider a replica as an image modification that still allows the user to identify the original picture. However, this abstract description may be too complex and somehow impossible to translate into a machine and it is usually difficult to draw the line between similarity and replication. Figure 1.1 shows some example of image replicas.

To be able to tune and evaluate our replica detection system we defined a set of image modifications (e.g. cropping, compression, channel modification) that, at most, we want to be able to identify and always trying to generalize to any type of image transformations (except from viewport changes, that we have not considered as replicas). We applied these transformation on random images to conform the *UPCReplica* data set.

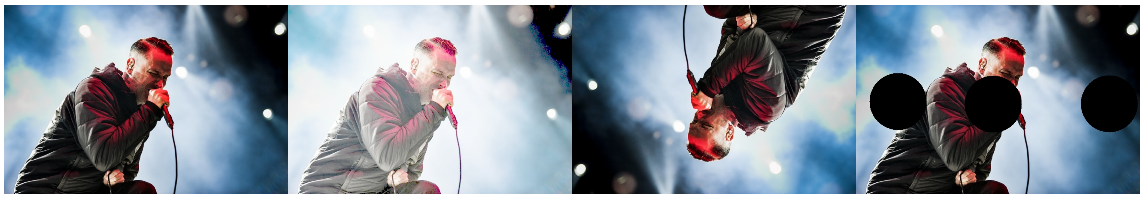


FIGURE 1.1: Three examples of image replicas from the original one (left)

1.3 Why do we need to detect replicas?

Identifying replicas can be useful in many different applications. Here are some examples:

- Distribution of illicit content and copyright infringement. Illicit content (e.g. offensive, racist, explicit violence) from sites could be easily identified if images are queried against a data base of common infringements. Copyright violations could also be tracked the same way.
- Image spam detection: spam usually is presented as undesired text messages received inside emails. But there is also another type of spam where the unsolicited content is embedded into images, allowing the senders to break through text-based spam detectors. To avoid being detected by image-based detectors, modifications of the original spam image are used. Though image spam is rarely used nowadays, near replica detection may help to reduce its effectiveness.
- Reduction of the redundancy of the web: as stated in [4] nearly 40% of the web is already replicated. Memory and computing resources are wasted on information that already exists and identifying which data is replicated could lead to save resources and help for a better management of them. An example of this application can be found in search engines. Figure 1.2 shows the Google image search result of the query 'Persistence of memory' from Salvador Dali. As we can see, there are lots of results containing exactly the same image. Many decisions can be taken from the assumption that a group of images are replicas of each other (e.g. it would be friendly for the user to group all the search results that are related or the site owner could consider not to store/retrieve all of them).

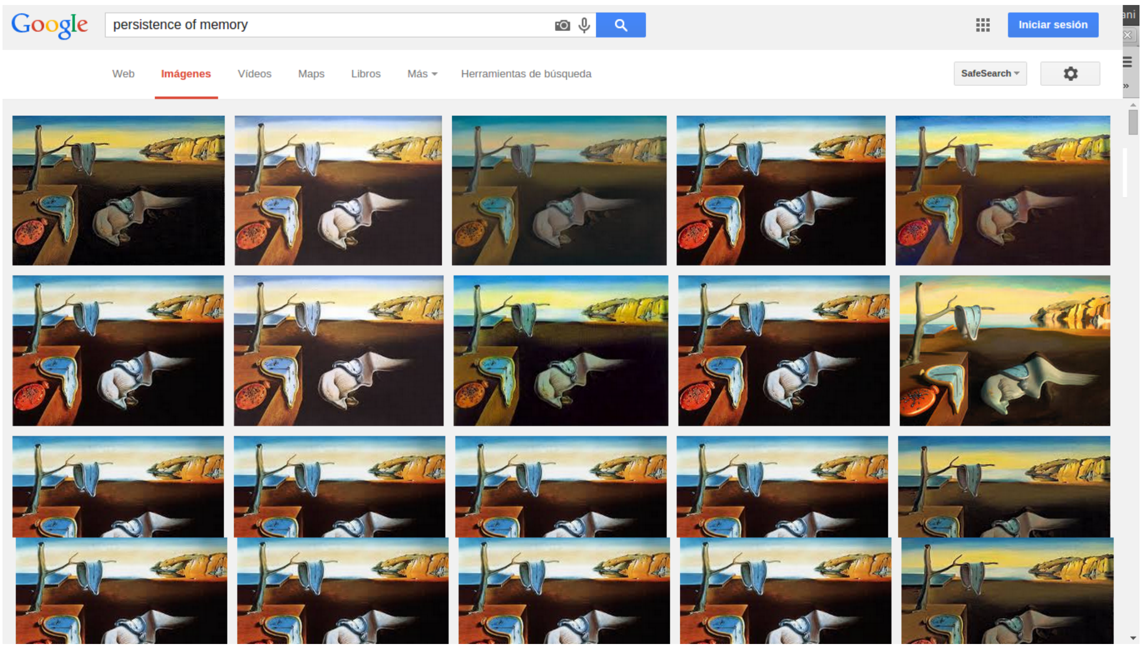


FIGURE 1.2: Google image top results from 'Persistence of memory'

1.4 Technologies used

For the implementation of our system we have both developed and tested on a single machine with a Xubuntu 12.0.4 operative system, 4GB of RAM memory and a Pentium(R) Dual Core T4300 2.10 Ghz. We have used the Eclipse Indigo 3.7.2 Ide with Maven integration for the Java (Java 7) development and OpenCV 2.3.9 for the image processing functions.

Apache Spark 1.2.0 for Hadoop 2.2 has been chosen the distributed computational framework and HBase has been set as the data base management system. Code has been under versioning using Git and all plots, data analysis and log parsing have been drawn with RStudio.

More details about these technologies and why have chosen them can be found in Chapter 4.

1.5 Requirements

In this section we describe the functional and non-functional requirements identified during the scheduling process.

Code	Title	Description
NFR1	Real-time	The system must respond a replica detection query in a real-time manner
NFR2	Good performance	Performance of the system must be close to the state-of-the-art
NFR3	Open source	Tools used and outcomes of the project must be open source
NFR4	Project Outcomes	Generate a dataset and/or a publication from our work
NFR5	Media integration	Integration of our system with well-known social media platforms
NFR6	Deployment	Deployment of the near replica detector in a real cluster
NFR7	Scalability	The performance and the efficiency of the detector cannot be affected by increasing the size of the images handled
NFR8	Modularity	Implementation of the detector in a modular way so its components can be reused and easily miantained and debugged

Table 1.1: Non functional requirements of the replica detector

1.5.1 Non-Functional requirements

The non-functional requirements represent what our system must be able to accomplish. The list of non-functional requirements can be found in Table 1.1.

The requirements NFR1, NFR2 and NFR7 become critical points and must be accomplished and the rest are tagged as optional.

1.5.2 Functional requirements

The functional requirements of a system describe how the system should behave and support the non-functional requirements detailed. The list of the considered functional requirements of our system is shown in Table 1.2.

These functional requirements support the critical requirements pointed in the previous section.

Code	Title	Description
FR1	Real-time	The system must respond, on average, within 3 seconds for a query
FR2	Good performance	System must have a performance better than the random guess, expecting true positive rate above 50% and false positive rate below 1%

Table 1.2: Functional requirements of the replica detector

1.6 Scheduling

The projects is intended to last for 53 weeks, with 15 hours of dedication per week and considering 3 weeks of holidays. This makes a total of 750 hours, starting from May 15th 2014 and ending on May 31th 2015. The dedication per weeks is detailed as follows:

- Weeks from 25/05/14 to 16/08/14: Study and analysis of the state-of-the-art and documentation related. Selection of some existing replica detectors, study of the techniques that are common in this field and review of the evolution of the near replica detectors.
- Week from 17/08/14 to 23/08/14: Summer holidays.
- Weeks from 24/08/14 to 06/09/14: Continuation of the first task.
- Weeks from 07/09/14 to 27/09/14: Definition of the first approach of the overview of the system to implement as well as the technologies to be used and the revision of the planning.
- Weeks from 07/09/14 to 27/09/14: Setting up of the development environment (e.g. Eclipse, Maven and OpenCV) and creation of the code repository.
- Weeks from 05/10/14 to 01/11/14: Generation of a dataset to work with during the project and that will be also used for evaluation purposes.
- Week from 02/11/14 to 08/11/14: Coding of classes to extract image descriptors.
- Weeks from 09/11/14 to 29/11/14: Analysis of image features. Keypoint detectors existing are compared and one of them is selected to be used throughout the work.
- Week from 30/11/14 to 06/12/14: Set up of the HBase environment and coding of dummy examples.

- Weeks from 07/12/14 to 20/12/14: Implementation of the sketching algorithm. Generation of the Java code capable of converting image features into binary representations.
- Weeks from 21/12/15 to 10/01/15: Christmas holidays.
- Weeks from 11/01/15 to 30/01/15: Implementation of the indexing structure, declaration of HBase tables and implementation of the Java client to interact with this structure.
- Week from 01/02/15 to 07/02/15: Installation of Spark and coding of first simple examples.
- Weeks from 08/02/15 to 21/03/15: Integration of the functionalities implemented so far into Spark (batch and streaming). All tasks in our system that can be easily distributed are redesigned to be compatible with the framework.
- Weeks from 22/03/15 to 11/04/15: Execution of tests. In these weeks, all evaluation tests designed in the beginning of the project are launched and results are gathered.
- Weeks from 12/04/15 to 09/05/15: Writing of the first draft of the documentation.
- Week from 10/05/2015 to 16/05/2015: Revision of the documentation.
- Week from 17/05/2015 to 23/05/2015: Correction of the documentation.
- Week from 24/05/2015 to 31/05/2015: Last version and refining of the documentation.

1.7 Costs

The costs of the projects is the sum of three areas: human resources, hardware and software.

1.7.1 Human resources

Though all tasks in this work have been carried out by the same person we have estimated the cost as we were working in a development team. Table 1.3 show the detailed costs related to human resources.

Role	Dedication hours	Income per hour (€)	Total (€)
Research analyst	405	25	10, 125
Developer	345	18	6, 210
Total			16, 335

TABLE 1.3: Estimation of the costs derived from the human resources

1.7.2 Hardware resources

This project has been developed and tested on the same machine. However, in a real context we would also need a computing cluster where to deploy our work. Instead of buying and setting a server up we selected to estimate the costs of a cloud computing platform such as Amazon EC2. Hardware costs are summarized in Table 1.4, that are dependent on the computing time that we need to consume.

Resource	Cost (€)
Development platform	521.13
Amazon EC2	0.29 per hour
Total	521.13 + 0.29 per hour computed

TABLE 1.4: Estimation of the costs derived from the hardware resources

1.7.3 Software resources

As stated in the requirements, we decided to stick to open source tools and there is no cost originated by software licenses.

1.7.4 Total costs

The sum of total costs, shown in Table 1.5, is 16, 856.13 €, and adding 0.29 hours to each hour computed in the cloud.

Type of cost	Cost (€)
Human costs	16, 335
Hardware costs	521.13
Software	0
Total	16, 856.13

TABLE 1.5: Total estimation of the costs. Cloud computing time per hour not included

1.8 Summary

In Chapter 2 we review work done on near replica image detection. Chapter 3 describes the theoretical framework of our implementation, detailed in Chapter 4. Evaluation of our system can be found in Chapter 5 and conclusions and future work can be found in Chapter 6.

Chapter 2

Previous work

2.1 Early work

Approaches related to image content retrieval can be already found in the early 90s [5] [6], where systems were able to search for images depending on characteristics as shape, color or texture in small databases. Though they focused on a different problem, methodologies and use cases were close to the ones in near replica detection.

Works about near replica detection also appeared in the 90s. By then we could find iconic picture detectors [7] and replica detectors based on hand sketching in low-resolution images [8]. Work in [9] in 1995 already shows detectors for simple image transformations such as resampling using hash-table like indexing structures. All these detectors presented use wavelets or colour features.

Watermarking [10] also appeared to be a methodology to detect image replication but results impractical since it needed the original image to be manipulated. Other works, such as [11], integrate machine learning algorithms to build models trained offline for each reference image.

2.2 The growth of the Internet and the scalability curse

The overwhelming growth of data and users during the last decade (from 360 millions in 2000 to 2802 millions of users in 2014 [9]) and in the present forced new approaches to show up focusing on three main goals: ease to scale, high precision and efficiency.

2.2.1 State of the art: the bag-of-words

The bag-of-words approach became the state-of-the-art: image are represented as a vector of visual words belonging to a dictionary computed offline using k-means. This technique relies on the quantization of the images features into visual words, assigning each image feature to the closest one in the dictionary. This procedure achieves to reduce the dimensionality of the features by losing part of its discriminative power. Images are represented by a histogram of the appearance of those visual words in it. The similarity between images is estimated by the similarity of the set of their visual words. This information is usually structured using inverted index and measures as *tf-idf* or voting are used to evaluate the similarity of the sets.

[12] is an example of this approach, that uses bundled SIFT features (features are grouped into groups to build richer and broader ones) and a dictionary of a million visual words. Visual words are stored using inverted index structures and a voting scheme using TF-IDF is performed to retrieve the results. Extra geometric verification is done at the end of the process on the 300 top results and an embedded hamming representation of 24 bits is used for filtering out the high amount of false positives. Similar work can be found in [13].

2.2.2 Similarity by hashing

Another branch of works uses hashing procedures for similarity. A common approach in this type of systems is to implement several hash tables where indexed features are organized within buckets gathering features that are similar. For each descriptor vector belonging to an image, a bucket is computed for each table by independent hash functions using Local Sensitive Hashing [14]. The similarity of two images is measured by the similarity of their features. To find similar features to a query feature we must gather the features that are mapped in the same bucket it has been assigned in each hash table. Then, verification steps are performed to discard false positives.

Ke et al. presented the work in [15], becoming a very influencing one. They hash unary concatenated PCA-SIFT features into multiple hash tables and a voting strategy for similarity, using an extra geometric verification (RANSAC) at the end. Big issues of this implementation are the disk access bottleneck, the redundancy and high storage requirements of the data (each feature representation being larger than 8.000 bits) and again the extra verification stage. Works inspired by it tried to overcome these problems as the work by Yang et al., that used Local Difference Pattern descriptors (36 values) and applied a soft-quantization on them to reduce the dimensionality of their binary feature representations to 72 bits . After hashing, only a hash table is needed and then RANSAC verification is applied. [13] used also an LSH approach computing colour histograms using 36 hash tables and 536 bytes per image.

2.2.3 Other approaches

Different approaches to the ones presented here are available, as the work in [16], that combines spatial information and the bag of words design. It computes SURF histograms at different pyramids levels using different unique vocabularies for each level instead of a big single dictionary.

2.3 Recent approaches

All the works analysed so far have very high precision and recalls and responses within 2 or 3 seconds but they present several problems: the size of the dictionary in the bag of words becomes a very sensitive parameter (small dictionaries increase the number of false positives while big collections of words decrease the precision). Moreover, there is uncertainty when choosing which images conform the dictionary and the visual word quantization proves to be too discriminating for large-scale [17] environments. Hashing approaches usually need a high amount of tables that leads to higher computational time and waste of storage resources and only lead to a reduction of the similarity scanning problem to a portion of the original search space. In both cases, the final geometrical verification stage becomes a big computational burden. RANSAC is usually used for this final step: a subset of points is used to estimate the transformation between the query image and the candidate one and finally the transformation is evaluated using the rest of the points. However, work on reducing false positives cannot be skipped: even a 1% can be huge in large data sets (i.e 1% of 50 million are still $500,000$ false positive images). Regarding scalability, most of these works have not been tested by the authors in big-scale environments while some of them are just impractical in big environments of few million image.

Large-scale ready methodologies can be found on both sides to overcome these challenges. We select two of these works: [18] and [17]. The first extends the bag-of-words approach by using Vector of Locally Aggregated Descriptors (VLAD) and Product Quantization, showing the intractability of traditional bag-of-words approaches.

[17] uses hashing for building feature sketches assuming that a single feature match is enough to state that an image is a replica by ensuring that the features computed have a high quality. These sketches are binary representations of the features and their similarity

Paper	Features	Indexing	Distance	Verification	Performance	Year	Comments
[15]	PCA-SIFT descriptors, 36 bytes	LSH of unary SIFT, 20 tables	Thresholding	L2 distance, RANSAC	Recall: $90 - 99.85\%$, Precision: $90 - 100\%$	2004	-
[4]	Local Descriptive Pattern features, 36 bytes	LSH of 72 bits quantized features, 1 hash table	None	L2 distance, RANSAC	Recall: 98.9%, Precision: 99.9%	2004	-
[12]	Visual words from 1 million bundled SIFT features	Inverted-file index	Voting using TF-IDF	RANSAC on ranked images and Hamming verification	1.9 - 2.5 s per query. Mean precision average: 0.74	2009	-
[13] (CH-LSH)	Colour Histograms at different levels, 168 bytes	LSH	None	Optional (pruning, extra memory)	0.98% false negatives	2007	-
[13] (SF-mH)	Visual words from SIFT vocabulary	Min-hash, 724 bytes per image	Jaccard similarity	Thresholding	No ground truth provided	2007	Vocabulary of $2^{16} words$
[17]	SIFT, 128 bytes	Indexing of sketched features, 4 hash tables	Hamming distance	Query expansion on similarity graph	$2.4 * 10^-6$ false positives at 0.79 recall	2012	Entropy-based SIFT filtering and log scaling of features
[16]	Surf histograms computed at different pyramid levels	Not specified	Intersection Kernel	None	Precision above 95%	2013	Unique codebook of size 400 for each pyramid level.
[18]	VLAD vectors	Inverted file index	Intersection Kernel	Asymmetric Distance Computation	Precision around 0.70	2014	-

Table 2.1: Comparative between near replica detector techniques

is estimated by their Hamming distance. This distance is only computed on a small set of candidates identified by the the partitioning strategy in [19] for Hamming distance approximation.

To ensure that features have a good quality they both use feature filtering to discard low quality features and also avoiding extra verification steps. Both show good performances in large scale envinronments. Our implementation is a simplified generalization of the work from Dong et al. and is explained in full detail in 3.

Many other works have been published related to near replica detection, we just reviewed a tiny selection. A summary of the characteristics of some of the works analysed is presented in Table 2.1.

2.4 Applications related

Here is a list of found implementations similar to ours:

- LibPuzzle [20]. It is a lightweight C library (includes PHP bindings) for finding simple replicas from images. However, it is only capable of finding slight modifications and does not provide any indexing structure.
- TinEye [21] is a web search engine to find duplicates of images in the web. They have a private API for near replica detection of custom user images but it is only available under commercial license.
- Code from work in [18] is available in [22] and includes functions for feature extraction, descriptor computation and indexing but it does not conform a fully distributed framework and neither has any persistent storage capabilities.

No open public API was found for near replica detection meeting our scalability, efficiency and availability criteria.

Chapter 3

Workflow

As commented in the previous section, our implementation has been mainly inspired in the work by Dong et al. by simplifying and providing a generalization in a distributed way. Two main use cases have been defined:

- Indexing of images: Input images are processed and stored into our system and will be used to be queried against to find similar images.
- Querying images: images are matched against the ones already stored and the system retrieves the weighted results.

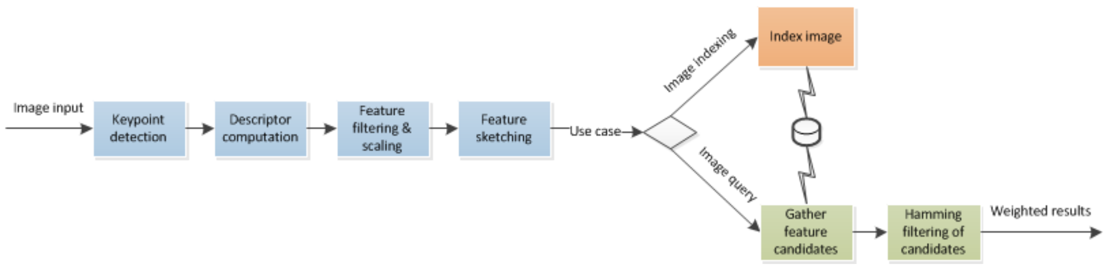


FIGURE 3.1: Overview of the two main use cases

3.0.1 Indexing of images

3.0.1.1 Overview

Once an image to index is received in our system, we follow these steps (Figure 3.1):

1. The points of interest of the image are computed.

2. Descriptor vectors are extracted around each detected point.
3. These vectors (features) are filtered to retain only the high quality ones. Log scaling can be applied to them to slightly modify their distribution.
4. Features are sketched into binary representations.
5. Sketched are stored by key in an indexing structure compound by several tables.

All these stages are described in the following subsections.

3.0.1.2 Detecting keypoints

Images are formed by a lot of information. Basically, we think of an image as a set of ordered pixels that themselves contain their color information. But we could also think that an image is identified by other structures such as sets of gradients, colour histograms, etc. Any of these characteristics that can be extracted from an image are called *features*. Features can be local and global. [1] defines a local feature as "an image pattern which differs from its immediate neighbourhood". On the other hand, global features extract measures from the whole set of pixels in the image area. We decide to use local features because they are semantically easier to interpret and have proved to be more useful in related computer vision fields such as object recognition [1].

In general terms, Tuytelaars and Mikolajczyk shows which ideal properties features should have:

- Repeatability: Given two similar images, a high ratio of similar features should be detect on both of them.
- Distinctiveness: Patterns detected by the features should vary enough from one to another in order to be dinstinguished from the rest.
- Locality: Features should be local (as to reduce probability of occlusion).
- Quantity: Number of features detected on an image should be large enough and adaptable using thresholding since its number highly rely on the specific application.
- Accuracy: Features must be localized with high precision.
- Efficiency: Features must be feasible to compute.

Keypoint detector	Rotation invariance	Scale invariance	Affine invariance
SIFT	$\surd$	$\surd$	$\times$
SURF	$\surd$	$\surd$	$\times$
ORB	$\surd$	$\times$	$\times$
MSER	$\surd$	$\surd$	$\surd$
BRISK	$\surd$	$\surd$	$\times$
HARRIS	$\surd$	$\times$	$\times$

TABLE 3.1: Invariance comparative between the selected keypoint detectors

All these properties should be maximized but in many cases maximizing one of them implies decreasing one of the other properties (e.g. the higher the quantity of features we have the more difficult to be distinguished). We should empower the properties that are most important in each specific application. Most of use cases require a high *repeatability*, that can be achieved by *robustness* (i.e. detectors less sensitive to slight changes) or *invariance* (i.e. detectors not affected by transformations). Here, *distinctiveness* and *efficiency* are also key aspects: we need features to truly represent the image they belong to, without much ambiguity, and we need to compute these features in a reduced amount of time.

Usually local features are based on intensity, color and texture measurements. They can represent many different things, such as points of interest, regions(blobs) or edges. Figure 3.2 shows the keypoints detected for an image.

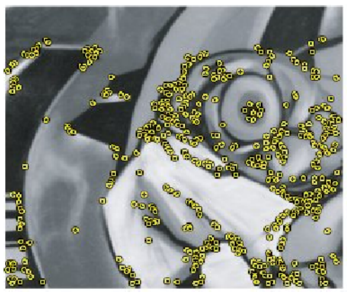


FIGURE 3.2: Example of interest points detected on an image, Source [1]

There are several feature detectors available. We select a subset from the ones implemented in OpenCV 2.4.9: Scale-Invariant Feature Transforms (SIFT), Speeded Up Feastures (SURF), Oriented Fast and Rotated Brief (ORB), Maximally Stable Extremal Regions (MSER), BRISK (Binary Robust Invariant Scalable Keypoints) and Harris Corner Detector (HARRIS). They have several levels of efficiency, repeatability, etc. Table 3.1 shows a summary of the invariance of the selected keypoint detectors and in the following paragraphs we give an overview of how they work.

SIFT keypoint detection SIFT is a rotation and scale invariant region detector that is widely used. For scale invariance the image must be scale filtered: a pyramid is built so for each octave of the scale space, the initial image is repeatedly convolved with Gaussians to produce a set of scale space images with different sizes. Then, adjacent Gaussian images are subtracted to produce difference-of-Gaussian images. After each octave, the Gaussian image is down-sampled by a factor of 2, and the process repeated. Keypoints are searched using local maxima across space and adjacent scales. After keypoints are detected, low-intensity contrast points and keypoints representing edges are discarded. Then orientation is computed on the remaining points to achieve orientation invariance using a histogram with 36 bins (for 360 degrees) with the neighbouring information.

SURF keypoint detection SURF is a rotation and scale invariant blob detector that follows similar steps as SIFT but differs in some of their details. Points are detected at scale-scape building a pyramid using Gaussian smoothing with square-shaped filters and using the integral version of the image. Instead of using smaller versions of an image in the scale space the filter mask is up-sized. The determinant of the Hessian matrix, that approximates the second derivative of the image at a point (i, j), is used as a measure of local change around the points and maximas are chosen. To localize interest points in the image and over scales non-maximum suppression (suppresses information that is not part of a local maxima) is applied. Scale and location of the detected points are finally interpolated.

ORB keypoint detection ORB was intented to computationally-efficient replace SIFT. The keypoint detection is an oriented variation of the existing FAST detector. ORB computes FAST points image using an intensity threshold between a pixel and its neighbours in a circular radius of 9 at different levels of a scale pyramid. Points detected eat each level are sorted using HARRIS measure (corner response measure) and top N are taken. Orientation is estimated by *intensity centroid*: it computes the offset vector to the center and estimates the orientation from the point to it within a radius r. As commented, it is invariant to rotation transformations but sensitive to scale changes.

MSER keypoint detection MSER regions are connected areas characterized by uniform intensity surrounded by contrasting backgrounds. Regions selected are those whose shapes remain unchanged after applying different sets of intensity thresholds. It can be computed by sorting all pixels by the grayscale intensity of their values, forming connected components. Each component is incrementally added pixels and their areas are monitored.

Those areas whose variation to different thresholds is minimal are considered maximal stable regions. It has proved to be scale and rotation invariant and highly invariant to affine transformations.

BRISK keypoint detection BRISK is another example of rotation and scale invariant region detector. It searches the scale-space of an image by building a pyramid with c_i octaves and d_i intra-octaves, where $i = 0, 1, .., n-1$ and typically $n = 4$. The original image is c_0 and each octave is computed by successive downsamplings. The first intra-octave is a downsampled version of c_0 by 1.5 and the rest of them are computed by half-sampling the first one. At each intra-octave and octave interest regions are identified when: there is a local maximum and FAST score s is greater than scales below and above. Location and orientation are finally interpolated.

HARRIS keypoint detection HARRIS detector is a corner detector that is rotation invariant but sensitive to scale changes. It works by computing a corner response at each pixel in the image and considers those that are above a threshold. The corner response is computed using a matrix built with the sum of products of derivatives at each pixel.

The decision of choosing one keypoint detector is very application dependent. The more invariant to transformations a detector is, the more computational time it needs. That means that the more invariant a detector is, the highest its repeatability but the lowest its efficiency. SIFT and SURF are widely used in computer vision because of its robustness and invariance to rotation, scaling and some illumination and viewpoint changes (they are partially affine invariant but cannot be labelled as it). Other faster detector such as ORB or FAST are more suitable for real-time purposes but they are much more affected to such image attacks. In order to choose a detector for our implementation we designed a benchmark to measure both the CPU time required to compute them and the quality of the keypoints found by each of them.

For this purpose we select 10 images from the UPCReplica dataset [23] and for each of them we generated 32 copies applying the following transformations with different parameters: rotation, cropping, JPEG compression, Gaussian noise addition, average filtering, gamma correction, channel modification, flipping, text insertion and occlusion. Then, for each image and transformation we match their SIFT descriptors (note that SIFT keypoint detection and SIFT descriptor extraction are two different concepts) using a Flann matcher, as described in the tutorial from the OpenCV site [24]. The test computes the transformation between an image and its modification and then measures the distance between the transformation model and the real transformation between the images. This test is called *homograpy*. We use this distance as an estimator of the *repeatability* of the

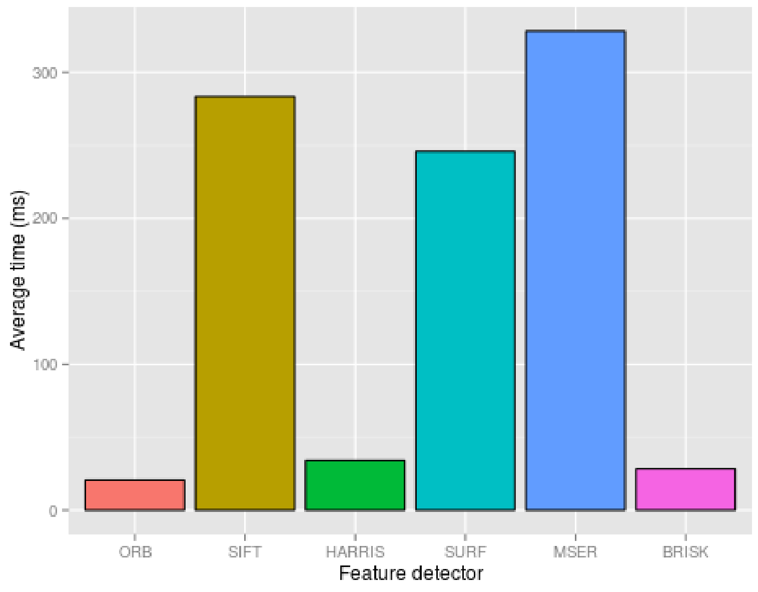


FIGURE 3.3: Average CPU time comparison of feature detectors on 309 images

keypoint detected. Low distances identify what we call *high quality keypoints* and large distances belong to normal/regular keypoints. All images areconverted into grayscale and resized, preserving their ratio, so their largest side is at most 350 pixels.

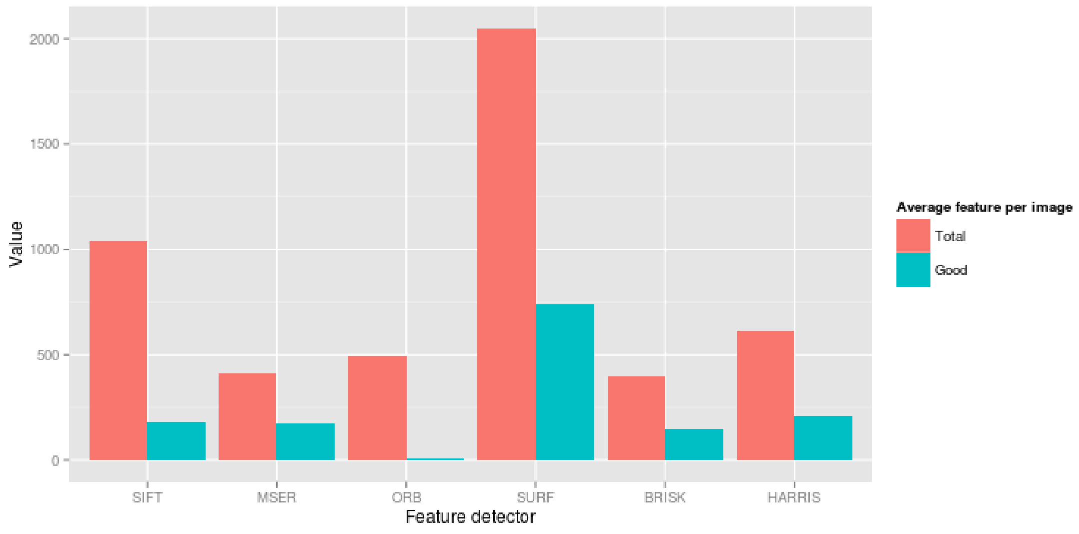


FIGURE 3.4: Features computed per detector on 320 pictures

Figure 3.3 confirms that the more powerful techniques (i.e. SIFT, SURF and MSER) are also the ones that need more computing time (around a third of a second) while the rest take only a few miliseconds. Figure 3.4 shows that all detectors compute enough keypoints for our needs: we will not be interested in extracting more than 250 features per image. ORB proves to be the worst detector and barely computes high quality features. SIFT also did not perform as expected and gets fewer high quality features ratio than the rest.

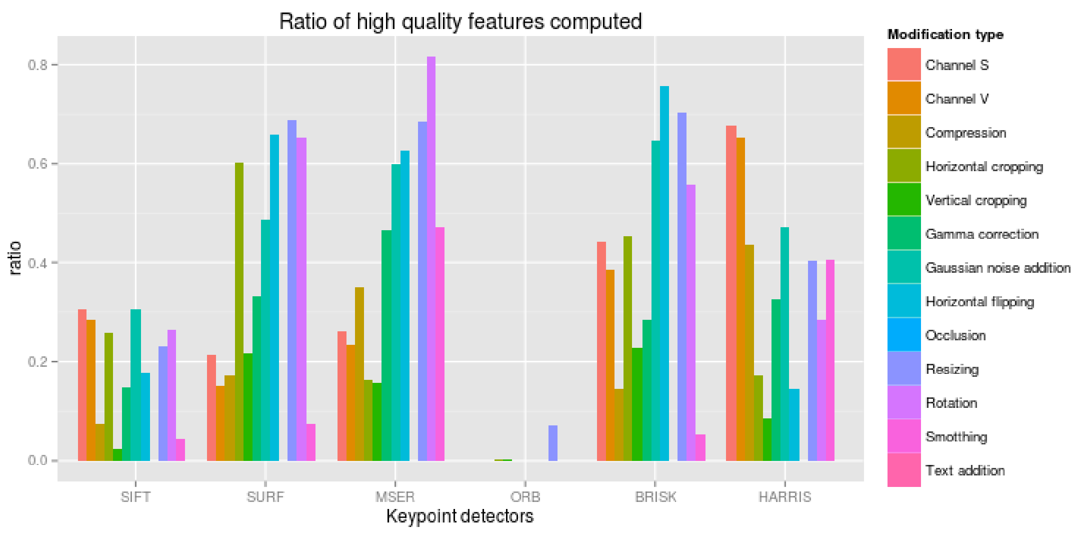


FIGURE 3.5: High quality features computed per each image modification type

Considering the significant features detected for each modification type (Figure 3.5), we see that we get very balanced results: some perform well in specific transformation and poor in others. *Occlusion* and *text addition* modifications get no significant keypoints from none of the detectors tested. We conclude using BRISK because it is the fastest detector (can be computed in 20 ms on average) and gets reasonable results, similar to MSER or SURF.

Keypoints detected are dependent on the image size. To avoid detecting huge amounts of points per image we resize each image before starting the process so its largest side is 350 pixels. Then, we compute the intensity image to map the three color channels into a single grayscale channel.

3.0.1.3 Descriptor extraction

Once we have all interest points detected we must extract some information from the neighbouring area to build local features. The numeric vectors representing the local information around a keypoint are called *descriptors*. In our implementation, these values are natural numbers. Once again, OpenCV gives us implementations of different techniques of descriptor computation. From all of them, we select Binary Robust Independent Elementary Features (BRIEF), Binary Robust Invariant Scalable Keypoints (BRISK), Oriented Fast and Rotated Brief (ORB), Scale-Invariant Feature Transform (SIFT) and Fast Retina Keypoint (FREAK). 3.6 shows SIFT descriptors examples.

Each of them have different characteristics, procedures and invariance levels. Deep analysis about these descriptors can be found in the bibliography and it is out of our work's scope.

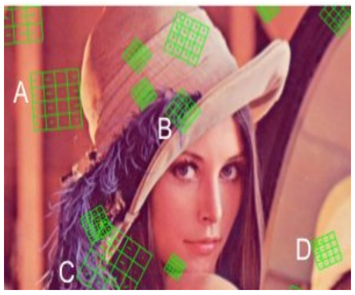


FIGURE 3.6: SIFT features on an image

Table 3.2 shows some basic characteristics of the descriptor extractors accounted and we can find a glance of how they are computed in the following paragraphs.

Descriptor extractor	Feature vector length	Minimum value	Maximum value
SIFT	128	0	255
ORB	32	0	255
BRIEF	32	0	255
BRISK	64	0	255
FREAK	64	0	255

TABLE 3.2: Number of values per feature and range

Most of the examples in the related work use SIFT techniques (Scale-Invariant Feature Transforms) such as [17] or [13], or SIFT-based detectors as the work by Wu et al., that group SIFT features to build broader and richer features.

SIFT descriptors SIFT computes descriptor by building a 16x16 grid around the key-point. The grid is then grouped into sub-blocks of size 4x4. For each sub-block, it creates an orientation histogram of 8 bins, resulting in 128 bin values represented as a vector.

BRIEF descriptors BRIEF defines binary tests around a set of image locations in a smoothed version of an intensity image. BRIEF descriptor is the concatenation of these binary tests. It is not invariant to rotation.

ORB descriptors ORB descriptor is a rotated version of BRIEF. It steers the points according to the keypoints orientation using a matrix product between the points matrix and a rotation matrix. Due to a loss of variance from the original BRIEF descriptor, an extra selection of high variance and less correlated points is made at the end.

FREAK descriptors FREAK is a descriptor extractor inspired by the human vision. Binary string is computed as the concatenation of the difference of pairs of receptive fields (areas detected) around a keypoint. Descriptor is formed by a coarse-to-fine ordering: first 16 bytes correspond to human peripheral vision (surroundings) and remaining 48 include finer details (center of the keypoint's area). Orientation is finally computed by selecting symmetric receptive fields with respect to the center.

BRISK descriptors BRISK defines N locations equally spaced on concentric circles around each keypoint and are Gaussian-smoothed with a deviation proportional to the distance to the center to avoid aliasing. Pairs of pixels around the center are classified between short and long distance and local gradient is computed for each pair in the long distance set. The feature orientation is computed as the average of sum of those gradients. The binary descriptor is computed from the rotation of the short distance pair.

We have evaluated the behaviour of all these descriptor extractors on the replica detection system.

3.0.2 Feature filtering

A feature vector is generated for each keypoint detected. As seen in Figure 3.4, hundreds of keypoints are computed per image and so feature vectors. This huge amount of features leads to the introduction of noise in the data base and its consequences: more storage and time consumption and increase of false positives, because noisy features tend to wrongly match other features easily. Works such as [15] or [4] need extra verification stages to discard all those matches using RANSAC algorithm. RANSAC (Random sample consensus) algorithm estimates the model transformation between a subset of the points from the original image and the matched image and then evaluates the model with the rest of the points. However, and as we already stated, this verification stage is prohibitive for real-time.

Dong et al. shows that this step can be skipped if the features computed have a high quality. After a proper filtering, they defend that a single feature match between two images is enough to identify them as near replicas. Our goal in this section is to be able to identify those feature vectors that have high *repeatability* and *distinctiveness*, discarding the rest of them.

For this purpose we define two measures of feature quality: entropy(3.2) and variance(3.1). Entropy measures the internal richness of the descriptor: the highest the entropy, the more information it contains. Variance indicates how the points of a vector are spread out from the mean and one another. Ideal feature vectors have high variance.

$$variance(F) = \frac{1}{n-1} \sum_{x,y \in F, x!=y} (x-y)^2 \tag{3.1}$$

Definition of variance

$$entropy(F) = -\sum_{i=min}^{max} p_i(F) \log_2 p_i(F), p_i(F) = \frac{|\{k|f_k = i\}|}{length(F)}, \tag{3.2}$$

Definition of entropy

In the previous section we estimated the *repeatability* of keypoints detected by different detectors. We want to to see if we can use the same test to identify any pattern to distinguish between *high quality* features and *regular* using entropy and variance measures, trying to identify thresholds between them. Homography test has been performed again using BRISK as keypoint detector and each of the presented descriptor extractors to compute the feature vectors. We try to identify any threshold to be able to classify features detected. We have found that the homography test does not provide useful information about the repeatability and distinctiveness of the descriptor values depending on entropy and variance. Figures corresponding to this experiment for each descriptor can be found in Appendix A.

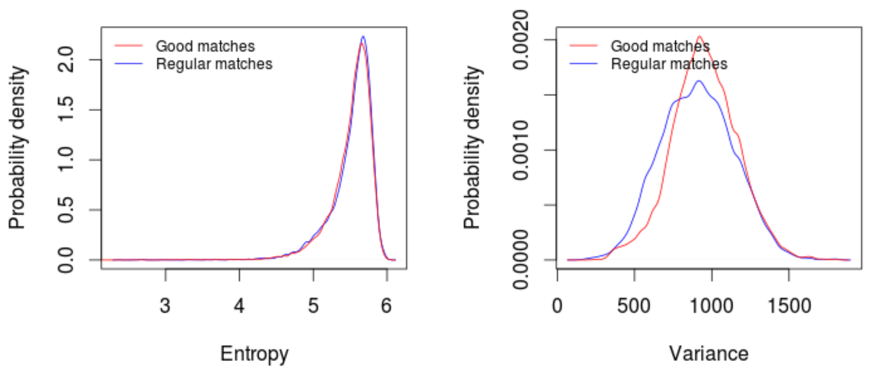


FIGURE 3.7: Distribution of SIFT descriptors depending on their entropy and variance for 320 images

The results show that both high quality and regular features have similar distributions: we can only appreciate some slight differences in SIFT variance (Figure 3.7, where high quality features are slightly shifted to the right) and in BRISK and FREAK (where high quality features are slightly shifted to the left). Thus, we cannot approximate the quality of a feature using those measures and there is probably no way to discard noisy features without discarding high quality ones. However, discarding some good features does not have to be dramatic for our detection rate: we only need a subset from the features to match. In our experiments we will evaluate the impact of feature filtering on the system's performance.

3.0.2.1 Log scaling

We assume that all possible values from a descriptor have the same probability of being computed, that feature vectors are uniformly distributed. Though we know this assumption is naive, we want to ensure that the probability density of these vectors is as uniform as possible. This is important because, in non-uniform distributed features, the same difference between two features has a different impact on the similarity estimation depending on the area of the distribution the difference falls in.

A simple approach to flat distributions is log scaling. For each feature computed in an image, we follow these steps:

1. Features scaled into range [0,1]
2. Log scale filtering is applied:

$$logScale(x_i) = log10(x_i + 1) \tag{3.3}$$

3. Features are rescaled into their original ranges.

Figure 3.8 show the impact of log scaling on the different descriptor distributions. SIFT and ORB are biased to the first values and show slight improvements using log scaling. This technique does not improve any of the other descriptor types. We conclude to use log scaling filtering for SIFT and ORB feature descriptors.

3.1 Feature sketching

After computing the descriptors, each feature is represented by a d-dimensional vector where d depends on the type of feature descriptor computed. To compute the similarity between two features is the same as computing the distance between them. A simple but effective distance measure is the Euclidean distance. However, it may be expensive to compute in high dimensional spaces (i.e. SIFT descriptors have length of 128) and Hamming distance becomes a good approximation of the L2-distance. To be able to compute it, we need to transform the descriptor vectors into a binary representation that preserves the original information. By doing this, we also help to reduce the dimensionality of the vectors.

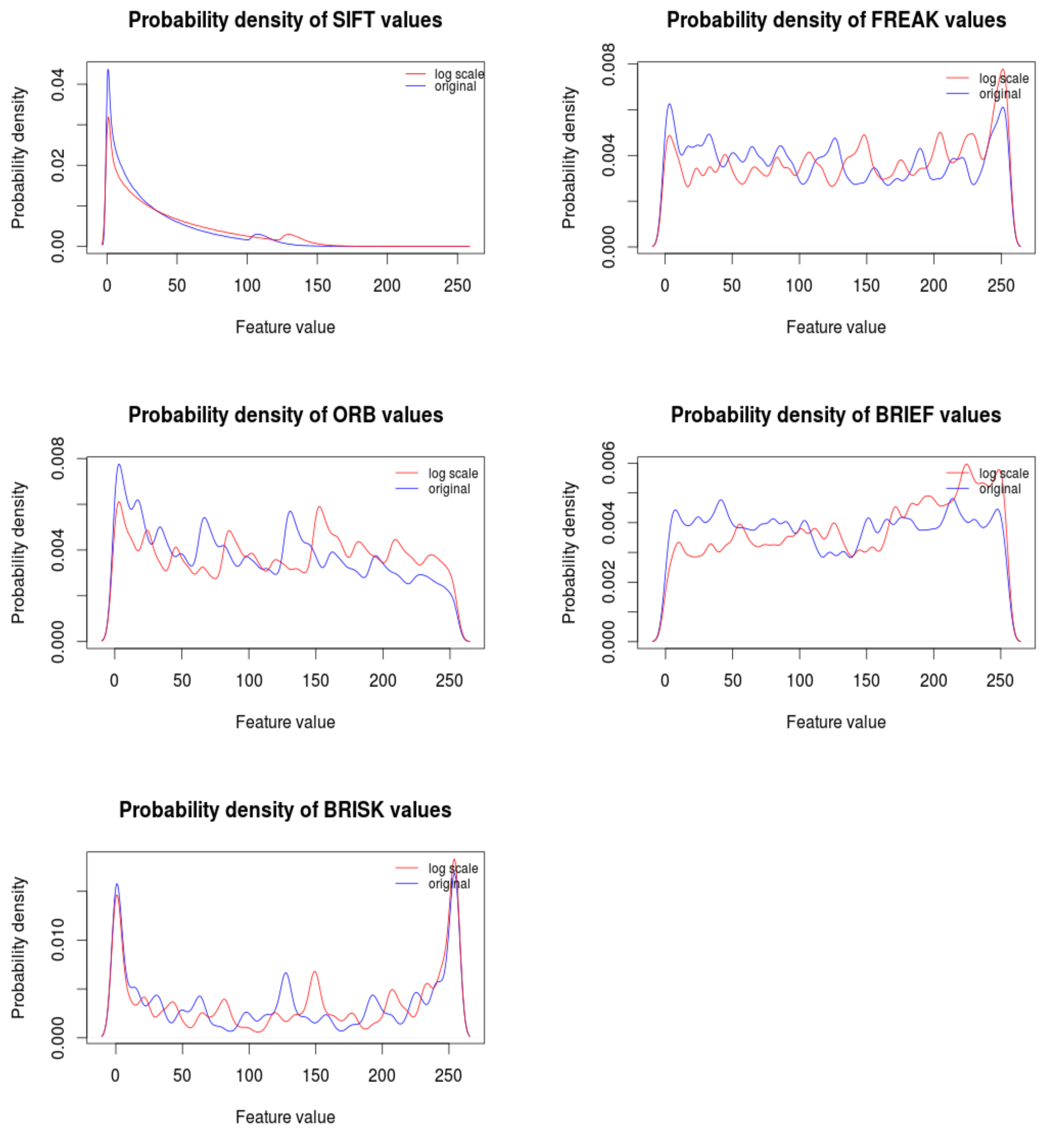


FIGURE 3.8: Distribution of descriptors and their log scaled values

Traditionally, similarity search with high dimensionality has been solved using mechanisms such as tree based structures or LSH. The first ended up becoming near brute force scans and the latter just restricted the whole search space to a portion of it using multiple tables and needing extra expensive computational stages. Sketch construction has proved to be a good way to approximate distance for near replica detection [17]. We decide to build sketches using the sketch construction technique presented in [2]. Sketching for similarity search differs from distance estimation in the way they measure the distance between vectors: sketching does not need to know the exact distance but the distance relationship (whether the distance is large or small). Instead of providing a uniform accuracy across all distances it shows a higher accuracy when distances are smaller (Figure 3.9). It is done by using a stripping technique that splits the space into gray and white stripes, where each color represents the binary sketch value: 0 for gray and 1 for white. Figure 3.9 shows an example of it in $\mathbb{R}^2$. Points that are closer fall in the same stripe and get mapped into the same values.

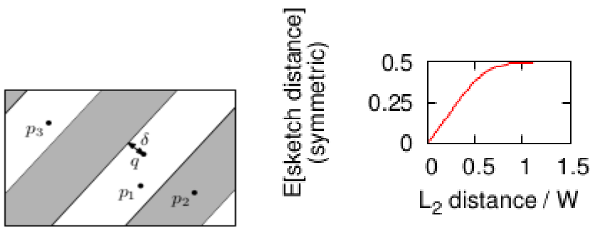


FIGURE 3.9: Stripes space partition for sketching in $\mathbb{R}^2$ (left) and relation between sketch similarity and L2-distance (right). Source: [2]

The function to construct the binary sketch is:

$$h_i = \left\lfloor \frac{A_i \cdot p_i + b}{W} \right\rfloor \mod 2 \qquad (3.4)$$

A is a random vector of length d where each component is independently generated from a Gaussian distribution [0,1], p_i is the $i-th$ element of the input feature vector, B i a real random variable uniformly drawn from the uniform distribution $[0, W)$ and W is the hashing sensitivity parameter. More precisely, W defines the width of the stripe partitioning: bigger width could make easier that similar points fall in the same stripe but also could lead to false positives to fall in.

This function may sound familiar because it is also used in LSH approaches (excluding the module operation). However, there is a big difference between the hashing techniques and the sketching one. Hashing techniques usually need a lot of tables (e.g. [15] needed around 20) and require extra geometrical verification steps. Instead of mapping each feature into buckets in multiple tables we map each value form the feature vector into a new space using a single hash function. In this new space, and as we will see in the following sections, we will look for the nearest neighbours of the query features retrieving only a small subset and being able to filter out false positives by setting a threshold on the Hamming distance with the candidate sketches (low level operation). The Hamming distance between two sketches is computed as the cardinality between their XOR bit operation (the number of ones in their XOR).

3.1.0.2 Sketch indexing

Computing the Hamming distance of a feature against all others is still very expensive. All the sketches must be cleverly indexed in order to compute this distance only on a small subgroup of candidates and be able to response within few seconds. Dong et al. uses a simplified version of the work in [19] based on document retrieval contexts to index the image sketches. They split the binary sketch into equally sized blocks and build a hash table using each block as a key. In that work, matching features differ at most 3 bits, that means that at least one block out of the four is identical. A feature is a candidate for a query feature if it matches at least one of its blocks. By searching all hash tables, we make sure that we find all possible candidates. Then Hamming verification is proceeded with all candidates to get the resulting feature matches. This process is detailed in Section 3.1.1.2.

Our approach is the generalization of this process for m tables and Hamming distance between matched features of at most h, so candidate match feature must have at least $h - m$ matching hash tables (blocks). This design is inspired by the traditional inverted-file structures in text retrieval.

After a feature f of length d is converted into a d-length binary sketch, it is divided into m blocks (tables) of the same size. Each block is represented using an integer so each one have at most 32 bits. To take advantage of this candidate approximation for Hamming distance we also need to set another requirement: number of tables has to be strictly bigger than the Hamming distance. Only in that case, a match from the key of a block would approximate the Hamming distance.

A summary of the whole process can be found in Figure 3.10.

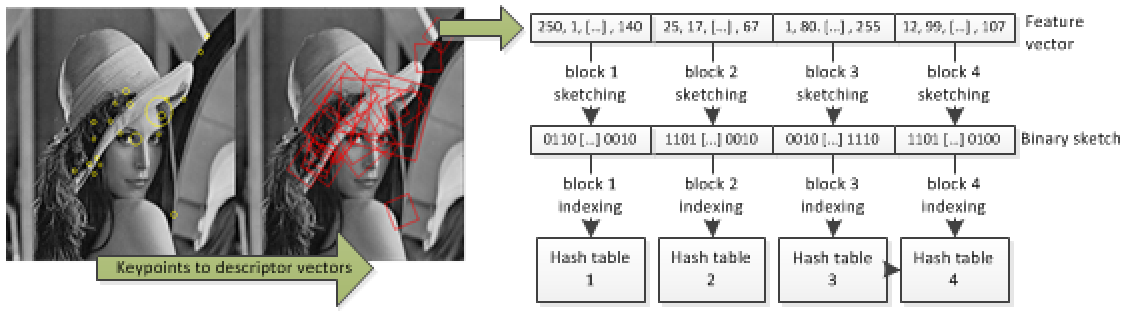


FIGURE 3.10: Summary of the indexing process for 4 hash tables

3.1.0.3 Limiting the scope of indexing parameters

In the image indexing process there are 4 important parameters: the number of hash tables, the Hamming threshold, the feature length and the sketching sensitivity W. The feature vector length depends on the descriptor type, the Hamming threshold is limited by the number of tables used and W will be adjusted in following experiments. Higher number of tables leads to a wider range of hamming thresholds. However, it also increases the probability of collision between features and may lead to bottlenecks.

Considering d as the feature length and so the feature sketch, there are 2^d possible values for a feature sketch, that we also call *fingerprint*. Assuming these values are distributed uniformly, each block feature have the same probability of falling into each of the row keys of a hash table. Having d total bits and m number of tables, we have $b = d/m$ bits per block. Each hash table has at least 2^b possible values, where each value may contain many different image features related to it. If we index n features and assuming that there are r average features computed per image, we have to index $t = n \cdot r$ fingerprints. On average, each possible value in a hash table (out of the 2^b possible ones) will be linked with, at least, $s = t/2^b$ sketched features. This also means that each query to a hash table would retrieve s candidates.

The more candidates retrieved, the more candidates that need to be processed, increasing the computing time. Our design should try to avoid high number of collisions between query features and indexed ones. These s collisions depend on both the feature length d and the number of tables m.

Assuming a large scale environment where we need to index $n = 50 \cdot 10^6$ images each of them generating $r = 250$ features, we will compute at most (without feature filtering) $t = 50 \cdot 10^6 \cdot 250 = 12.5 \cdot 10^9$ features to index. We decide to limit the size of the blocks to 32 bits (4 bytes), though all bits do not need to be used if $b < 32$. It is a requirement that feature blocks must have the same length.

Using $b = 32$ bits and t indexed features, each queried feature would retrieve at most $s = 12.5 \cdot 10^9/2^{32} = 2.91$ candidate features per table. Using $b = 16$ bits out of the 32 bits of the block, would retrieve $s = 12.5 \cdot 10^9/2^{16} = 190,734.86$ features per table. This amount is already difficult to handle and may be prohibitive in larger environments. We fix the minimum number of bits to be used in a block to 16.

Using this limitation and considering that a candidate must match have at least $m - h$ tables to the query feature, we have the following set of configurations for each descriptor types:

- SIFT descriptors: They are 128-length vectors so sketches have $d = 128$ bits. We can use up to 8 hash tables and a hamming threshold up to 7.
- BRISK and FREAK descriptors: All these features have a length of $d = 64$, being able to use up to 4 tables and hamming thresholds from 1 to 3.

- BRIEF and ORB descriptors: Their length is $d = 32$ and can only be split into two tables, using a hamming threshold of 1.

3.1.0.4 Structures

In order to be able to persist the information related to the feature blocks and the feature and the image metadata an indexing structure must be designed. We define 3 tables:

- Hash table: Contain the hashing information of the sketched features. The *ith* hash table is related to the *ith* block of the binary representation of a feature stored. Each row is formed by:
  - Hash: *ith* block of the original feature that is used as a key.
  - Feature identifier: link to the feature table.
- Feature table: Contains basic information related to a feature. That is:
  - Feature identifier: Unique identifier of the feature in the system, used as key.
  - Hamming blocks: Columns belonging to each of the blocks that compound the binary feature sketch.
  - Image id: Identifier of the image the feature belongs to.
- Image table: Contains the meta data information of the images stored:
  - Image identifier: Unique identifier of the image.
  - Path: URL or local path where to find the image.
  - Resource id: If the image comes from any specific platform (e.g. Twitter, Instagram), this field contains the identifier of the external resource containing the image.
  - Image provider: Provider of the image (e.g. Disk or Twitter).

In tables Table 3.3, Table 3.4 and Table 3.5 we can see the schemas for all tables.

Block key	Feature identifier
key_1	$feature\_identifier_1$
[...]	[...]
key_n	$feature\_identifier_n$

TABLE 3.3: Schema of one of the hash tables in the system

Feature identifier	Hamming block 1	[...]	Hamming block M	Image identifier
$feature\_id_1$	$hamming\_code_{11}$	[...]	$hamming\_code_{1m}$	$image\_id_1$
[...]	[...]	[...]	[...]	[...]
$feature\_id_t$	$hamming\_code_{t1}$	[...]	$hamming\_code_{tm}$	$image\_id_t$

Table 3.4: Schema of the feature table

Image identifier	Image path	Image resource	Provider
$image\_id_1$	$image\_path_1$	$image\_src_1$	$image\_provider_1$
[...]	[...]	[...]	[...]
$image\_id_r$	$image\_path_r$	$image\_src_r$	$image\_provider_r$

Table 3.5: Schema of the image table

3.1.1 Query by image context

3.1.1.1 Overview

In this use case the goal is to find all those images in the data base that are similar to a query one. The steps required for this process are summarised in Figure 3.1, and are:

1. The points of interest of the image are computed.
2. Descriptor vectors are extracted around each detected point.
3. These vectors (features) are filtered to retain only the high quality ones and log scaled is performed if needed.
4. Features are sketched into binary representations.
5. Sketches are queried against the stored features and similar ones are retrieved.
6. Similar features are filtered using the hamming distance and a list of matched images is built.
7. Image matches are weighted by the number of features that matched for the specific image and can be filtered using a weight threshold.

Steps 1 to 4 are the same as the ones for the indexing use case, explained in Section 3.0.1. The rest of steps are detailed in the following sections.

3.1.1.2 Querying sketches

Each block a sketched feature is partitioned into produces a query to the corresponding hash table, retrieving a list of candidates. Those candidates from the database that matched at least $m - h$ tables (blocks) from the feature are preserved. This algorithm is shown in 1.

Once all candidate are gathered, the Hamming distance between each of them and the query feature must be computed. Matches above a threshold h are discarded and the ones remaining become the list of matches between the database and the query image. All images that matched at least one feature from the query are considered near replicas of it.

Algorithm 1: Feature candidates query algorithm

```
input  : A feature sketch $f$, the number of tables $m$ and Hamming threshold $h$
output: Feature candidates for the input feature sketch
matches ← $\emptyset$
result ← $\emptyset$
for $i \leftarrow 1$ to $m$ do
    blockKey ← block $i$ from feature sketch $f$
    candidatesBlock ← list of matched blocks for $blockKey$ in hash table $i$
    for $c \in candidatesBlock$ do
        candidateFeature ← feature identifier from feature $c$
        if $candidateFeature$ not present in $matches$ then
            set $candidateFeature$ value in $matches$ to 1
        else
            increase $candidateFeature$ value in $matches$
        if $candidateFeature$ value in $matches$ greater than $(m - h)$ then
            add candidate $c$ to $result$
return $result$;
```

3.1.1.3 Gathering results

Matches are finally grouped by the image they belong to and a voting strategy is performed by assigning a weight to each match: the weight of a matched image is the number of feature matches with the query. We can perform an extra step by filtering results by its weight in case false positives are still high.

Chapter 4

Implementation

The previous section described the theoretical framework of our system. In this section we focus on how to implement it using modern and open source tools. We decide to implement two versions of our system: a persistent disk based implementation and a memory intensive implementation.

4.1 Architecture overview

The technologies stack used can be seen in Figure 4.1. The input channel is generalized to both file systems (local or distributed) and also streaming endpoints as Twitter Streaming API. In order to be ready to support large-scale environments we use Apache Spark, a computational distributed framework that is presented in Section 4.3. Sketching information and image metadata is stored in disk using the distributed database HBase, detailed in Section 4.4 and in memory using Spark persistence options.

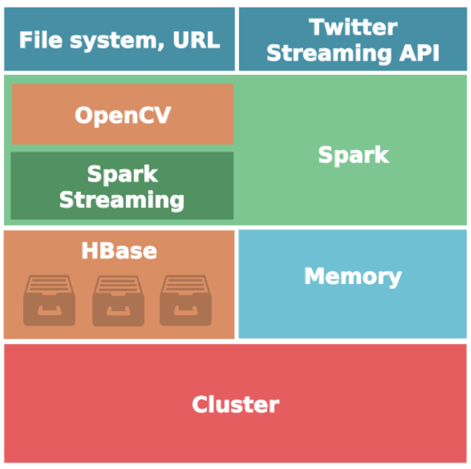


FIGURE 4.1: Overview of replica detector system stack

4.2 Image processing: OpenCV

Some of the operations required in our workflow rely on very specific image processing procedures. Several image processing libraries exist for Java, such as ImageJ[25] or Marvin[26] but they are focused on simple basic low level operations, that do not fit our needs.

A robust library able to cope with all image transformations in the project and able to efficiently compute the descriptor vectors is in need: OpenCV. OpenCV[27] is a widely used image processing open source C/C++ library. It can take advantage of multi-core processing and also have interfaces for GPU hardware acceleration and support for different programming languages, including Java. It has a spread community of users as well as a complete documentation. Nonetheless, the documentation available is focused on the C++ version and examples are sometimes difficult to port into Java. We have also encountered some errors that have been difficult to track because they are caught somewhere in the native C++ code, printing ambiguous error messages.

We use OpenCV for four purposes: detect feature keypoints in the images, evaluate the keypoint detectors, compute descriptors around the features and generate replicas from images. Figure 4.2 shows an example of how to compute the feature vectors of an image using OpenCV for Java.

4.3 Distributed approach: Apache Spark

Apache Spark is an open-source cluster computing framework. It is the evolution of Hadoop's MapReduce approach, expanding the types of primitive data operations available, and has proved to be x100 times faster on memory and x10 on disk for some specific appplications [28].

4.3.1 Architecture

Spark applicatons are independent processes that run on a cluster and coordinated by a SparkContext object. A cluster is formed by *worker nodes*, that provide a set of *executors* that run computations and store data for the submitted applications. A summary of the architecture of Spark can be found in Figure 4.3.

When an application is submitted into spark, the Spark Context inside the driver program connects to the cluster. Then, the application is given a set of processes(executors) that will be used thoughrout all the application lifetime. This means that applications are

```java
// ComputeDescriptor.java
import org.opencv.features2d.DescriptorExtractor;
import org.opencv.features2d.FeatureDetector;
import org.opencv.highgui.Highgui;
import org.opencv.core.Mat;
import org.opencv.core.MatOfKeyPoint;

public class ComputeDescriptor extends JApplet {
    public static void main(String[] args) {

      String path = args[0];

      // Read image
      Mat img = Highgui.imread(path, Highgui.CV_LOAD_IMAGE_COLOR);

      // Compute keypoints
      MatOfKeyPoint points = new MatOfKeyPoint();
      FeatureDetector detector = FeatureDetector.create(FeatureDetector.BRISK);
      detector.detect(img, points);

      // Compute descriptor
      DescriptorExtractor ext =
    DescriptorExtractor.create(DescriptorExtractor.SIFT);
      Mat descriptor = new Mat();
      ext.compute(img, points, descriptor);
    }
}
```

FIGURE 4.2: Example of how to compute SIFT descriptor vector around BRISK keypoints

isolated from each other and that they do not share any computational resource or data. Once executors are acquired, the application code (i.e. JAR or Python files) is sent to the executors and the Spark Context sends tasks (units of work) to them.

Each application is formed by Spark jobs (e.g. map, filter), that are divided into stages. It is recommended that the machine launching the application lies *close* to the cluster worked nodes.

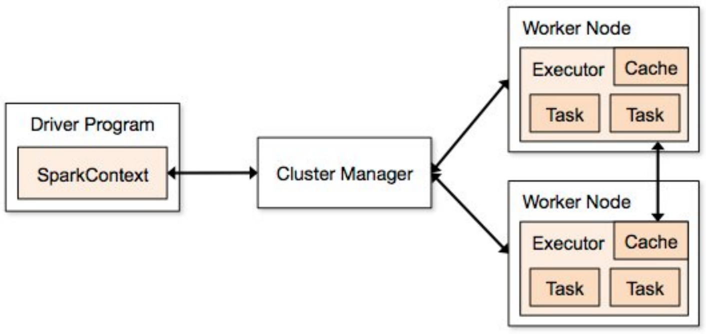


FIGURE 4.3: Spark cluster overview from the Spark documentation

Spark can be deployed in three differenet cluster managers:

- Mesos: general cluster manager supporting Hadoop MapReduce and service applications.
- Yarn: resource manager in Hadoop2.
- Standalone: cluster manager embedded in Spark. It only requires to place the same version of Spark in each of the working nodes.

For the ease of the development we decide to use the standalone cluster included in the Spark distribution.

4.3.2 Resource allocation and scheduling

By default, the allocation of resources is done by *static partitioning.* This means that each application is given the maximum set of resources it can use and they remain held by it until it finishes. In standalone mode, FIFO (First In First Out) is the default policy for CPU allocation: applications take the maximum nodes from the available ones though maximum number of nodes and maximum amount of memory used by an application can be limited by the properties *spark.cores.max* and *spark.executor.memory* respectively. Jobs within an application are also scheduled in a FIFO fashion: a job only gets priority until all stages from the previous one have finished.

Dynamic allocation of resource across applications is only available on Yarn so far. For multi user servers it is also possible to enable fair scheduling using a Round Robin policy. It can be done by setting the *spark.scheduler.mode* property to FAIR when configuring a *SparkContext.* Pools with different priority and scheduling policies can also be configured. We will stick to the FIFO fashion because our server is a private dedicated machine.

4.3.3 Spark stack

On the top of Spark, some domain specific applications (Figure 4.4) have been developed to take advantage of the Spark potential: ML lib, GraphX, SparkSQL and Spark Streaming.

From all the Spark applications we only used Spark Streaming, that is detailed in Section 4.3.5.3.

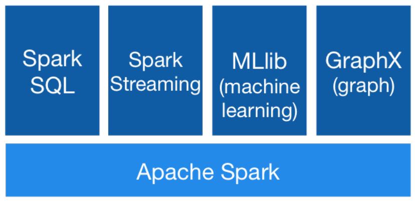


FIGURE 4.4: Spark application stack

4.3.4 Data

Spark uses *resilient distributed dataset* (RDD) to manipulate data. They are fault-tolerant immutable distributed collection of objects that can be operated in parallel. RDDs are split into multiple partitions on different nodes across the cluster. When performing operations on the data, Spark will independently work on each partition using different tasks per CPU. Partitions are automatically set based on the cluster but they can also be set manually when a RDD is created. It is recommended to use between 2 and 4 partitions per CPU in the cluster. Partitioners are the objects that split the data among different nodes. They can be explicitly set (RangePartitioner and HashPartitioner are built-in partitioner) or custom partitioners can be implemented. In our experiments we will use the default partitioning from Spark.

RDDs can contain basic types in Python, Java and Scala but also user defined objects. PairRDD also exists and are formed by a key and a value.

4.3.4.1 Creating RDD

RDDs can be created by loading data from external sources (e.g. text files or databases) or by distributing a collection of objects (e.g. list) in the driver program.

Figure 4.5 shows different ways of loading data into an RDD. The first example shows how to distribute and load a collection of integers and the second one does the same with a set of strings, specifying the number of partitions to be used. The third example shows how to load lines from a document into an RDD. The last example parallellizes data read from a HBase table. More details on HBase are explained in Section 4.4.

4.3.4.2 Operations on RDD

RDD operations can be divided into *transformations* and *actions*. Transformations are operations that return another RDD while actions return a result to the driver program or write a result into an output. All operations (jobs) in an RDD work in the same way: they apply a function to each element in the RDD to generate a result or output. Being

```java
// LoadData.java

   // sc - Spark context

   // Parallelize collection of data
   List<Integer> dataInt = Arrays.asList(1, 2, 3, 4, 5);
   JavaRDD<Integer> distData = sc.parallelize(dataInt);

   // Parallelize collection of data using 2 partitions
   List<String> dataStr = Arrays.asList("this", "is", "an", "example");
   JavaRDD<String> distData = sc.parallelize(dataStr, 2);

    // Read data from text file
    JavaRDD<String> lines = sc.textFile(path);

   // Read data from HBase
   Configuration conf = HBaseConfiguration.create();
   conf.set(TableInputFormat.INPUT_TABLE, table);
   conf.set(TableInputFormat.SCAN_ROW_START, start);
   conf.set(TableInputFormat.SCAN_ROW_STOP, end);
   JavaPairRDD<ImmutableBytesWritable, Result> content = sc.newAPIHadoopRDD(
         conf,
         TableInputFormat.class,
          ImmutableBytesWritable.class,
          Result.class));

    }
}
```

Figure 4.5: Example of how to load data into RDDs

an RDD distributed among different nodes, each job is computed by as worker nodes as partitions a collection is divided into.

The transformations we used the most in our work were:

- *groupByKey*: used in PairRDDs, gathers together all values for each different key.
- *reduceByKey*: used in PairRDDs, combines values with the same key using a custom function.
- *map*: maps each object in the RDD into another object.
- *mapToPair*: map each object in the RDD into a PairRDD.
- *mapPartition*: map each object in the RDD into a object, using a user defined task for each partition the data is split into. This is useful when an external connection needs to be open for a job but we do not want to open one for each object in the RDD. Instead, it opens one connection for each partition.

- *flatMap*: maps each object in the RDD into multiple new objects.
- *flatMapToPair*: maps each object in the input RDD into multiple new pair objects.
- *filter*: erases from the RDD the objects that do not satisfy a user criteria.

These transformations are executed *lazily*: when they are called, they are not immediately computed. Instead, sparks records the meta-data indicating that the operation has been requested. All transformation requested on a RDD are stored in the *lineage graph*, that is a directed acyclic graph (DAG). Therefore, RDDs are conceived not as a set containing data but as a result of several transformations. Some of these transformations, such a *reduceByKey* or *groupByKey*, need partitions to be recomputed and can force the worker nodes to exchange data across partitions (e.g. distribution of keys after mapping pairs into new keys may need to gather values that were originally far from each other). This is called *shuffling* and must be avoided because it may force Spark to write data on disk (independently whether it is computed on disk or memory) and generate bottlenecks. When several design options are available we must choose the ones that best preserves the partitioning.

The list of the most common action operations we used in this work are:

- *foreach*: applies a function for each element of the RDD. It has no output and does not modify the collection of objects.
- *foreachPartition*: applies a function for each element of the RDD, using a user defined task for each partition the data is split into.

Once a action is called, the transformations contained in its lineage graph are computed in the order they were requested and then the action requested is performed. If we want transformations to be computed, we need to call at least one action.

Examples of how to use some of these operations can be found in Figure 4.6. The example uses arbitrary input data and maps each integer in the collection into a new one. Then, filters out all values in the resulting collection using a threshold. In the end, it counts the results by calling the action *count*. Once the action is requested, Spark computes the lineage graph in Figure 4.7 and executes it from top to bottom.

4.3.4.3 Persistence

Every time we need to perform an action on a RDD, Spark computes all of its dependencies in the lineage graph. When two actions are performed in a row, RDDs need to be computed

```
// LoadData.java

      // threshold - Threshold of the application
      // values - JavaRDD<Integer> containing a set of values

      // Operation 1
      JavaRDD<Integer> newValues = values.map(new Function<Integer, Integer>() {

            public Integer call(Integer v) throws Exception {
               Integer result = v * new Random().nextInt();
               return result;
            }
      });

      // Operation 2
      JavaRDD<Integer> filtered = newValues.filter(new Function<Integer,
    Boolean>(){

            public Boolean call(Integer v) throws Exception {
               return v > threshold;
            }

      });

      // Operation 3
      int count = filtered.count();
    }
}
```

FIGURE 4.6: Examples of operations on RDDs using Java 7

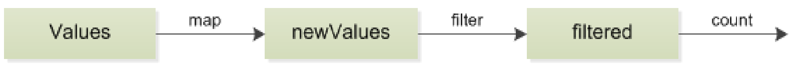


FIGURE 4.7: Lineage graph from the example

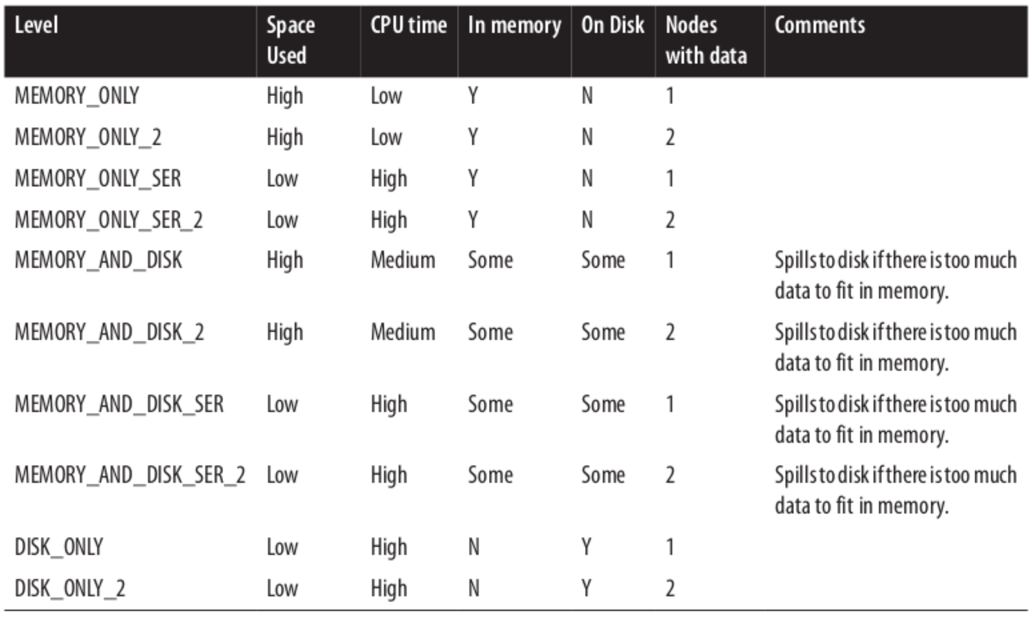
twice, being computationally expensive. To solve this we can ask Spark to persist the data into memory using the *cache* and the *persist* operators, so each node stores their partitions on memory.

```
// Persistence.java

// values - rdd containing data
values.cache();
values.count();
values.count();
```

FIGURE 4.8: Code showing how to cache data into memory

Figure 4.8 shows an example of data caching. Let's assume that several transformations haven been applied on data represented by the RDD *values*. When *cache()* is called Spark is notified to persist the content of the RDD into memory. When the first *count()* (action) is performed it computes the dependencies of the RDD and caches the computed elements into memory. The second *count()* can compute that value from the cached data instead of recomputing them all. We can also use the more general *persist* operator, that needs the level of persistence of the data to cache as input. There are different levels of persistence that satisfy different space and CPU requirements. The complete list can be found in Figure 4.9. *Cache* is a specific case of *persist* where the RDD is stored into memory. By default, Java will persist data as unserialized objects in the JVM heap.



Level	Space Used	CPU time	In memory	On Disk	Nodes with data	Comments
MEMORY_ONLY	High	Low	Y	N	1	
MEMORY_ONLY_2	High	Low	Y	N	2	
MEMORY_ONLY_SER	Low	High	Y	N	1	
MEMORY_ONLY_SER_2	Low	High	Y	N	2	
MEMORY_AND_DISK	High	Medium	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_2	High	Medium	Some	Some	2	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER_2	Low	High	Some	Some	2	Spills to disk if there is too much data to fit in memory.
DISK_ONLY	Low	High	N	Y	1	
DISK_ONLY_2	Low	High	N	Y	2	

FIGURE 4.9: Spark persistence levels

If the cached data does not fit in memory, Spark releases old partitions using Least Recently Used (LRU) policy. Depending on the persistence selected, the system will recompute these partitions or write them to disk. Note that caching unnecessary data can also be negative for the performance of the application.

To release the data from the cache we can call the *unpersist* operator.

4.3.5 Batch use cases

In this section we describe the two use cases implemented for batch indexing and querying of images, one for each mode of persistence considered.

4.3.5.1 Batch indexing

This use case processes a bulk of images, extracts their features, computes the sketches and indexes them in a distributed way. First, images are loaded into RDDs and then a set of Spark actions are performed following the schema in Figure 4.10. Each vertical stage in the flow diagram represents a level of parallelism while the horizontal stages represent the different jobs to compute. Let's detail them:

1. Job 1. Image loading. Images are loaded from an external resource (e.g. dataset, local file system or HDFS).
2. Job 2. Descriptor extraction. Each image is loaded, resized and converted into grayscale. Then, keypoints in the image are detected and feature vectors are computed. Each feature vector is mapped as an object into a new RDD.
3. Job 3. Feature filtering. Feature vectors in the RDD are filtered out using entropy or variance so only a fraction of them remain.
4. Job 4. Feature scaling. Optionally, a log scaling process can be applied on the feature vectors in the RDD.
5. Job 5. Feature sketching. Each feature vector of length d is mapped into a d-dimensional binary sketch.
6. Job 6. Image indexing. Each pair of image metadata and related feature vectors is stored. This stage is different on the memory based and on the disk based approach:
   - Disk (HBase). Contains two steps:
     (a) Image grouping: Feature belonging to the same image are grouped.
     (b) HBase connection: Instead of opening a connection to the database for each element in the RDD, we use a *foreachPartition* action that allows to schedule specific tasks for each of the partitions the data is split into. Using this solution we avoid the overhead of creating too many connections to the database, that may lead to bottlenecks and slow performances. The number of partitions is an important parameter here though we used the default partitioning. Each partition opens a connection to HBase and stores sketched images it contains.
   - Memory. It is compound by several jobs:
     (a) Block mapping: Creates an array with m positions where m are the number of tables in the system. For each table i in the system it maps the input features to the $i-th$ block of their sketches and assigns them to the $i-th$ position in the array.

(b) Image extraction: Image metadata is extracted from the whole set of sketched features.

(c) Image unique: Repeated image identifiers are erased.

(d) Identifier extraction: Image identifier is extracted from the image metadata.

(e) Feature indexing: For each position i from the sketched blocks array we append them to the respective RDD of blocks already indexed.

(f) Image indexing: Image metadata computed is appended to the RDD of indexed images. At the end of the process we request that data must be persisted and we perform an action (e.g. *count()*) on each RDD to ensure that they are computed and cached on memory. Data is only stored on disk if it does not fit on memory. It is important, before using memory based use cases, that we must ensure beforehand that data is going to fit into the cluster's memory. Otherwise, data will be split between memory and disk and it may slow down the performance by orders of magnitude.

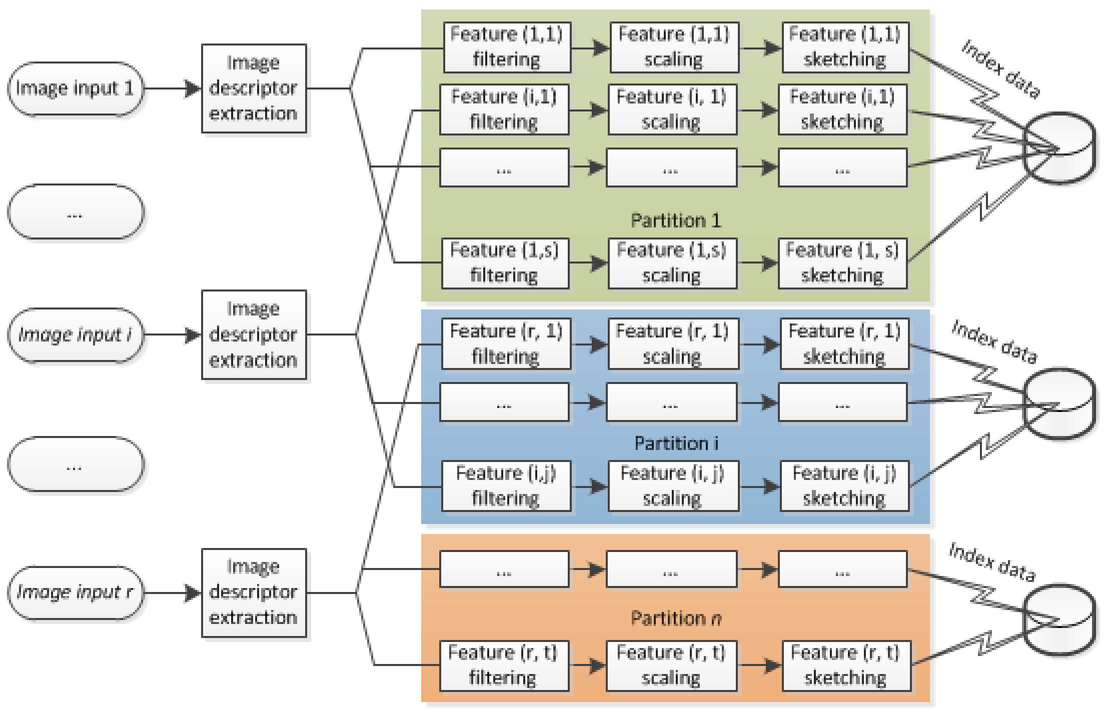


Figure 4.10: Spark implementation overview of the batch indexing use case

Lineage graphs in Section B.1 show the lineage graphs for both implementations. It is remarkable that the memory intensive scenario needs to schedule more Spark jobs and some of them are likely to produce data shuffling (as *union*), possibly leading to slower performances.

4.3.5.2 Batch querying

The process of querying for replicas into the database shares the first steps with the indexing use case. Once the sketches are computed, candidates are searched for and they are finally filtered to build weighted reliable results (Figure 4.11). The jobs required in this use case are described as:

1. Job 1. Descriptor extraction. Image/s are loaded into Spark in the same way as in the indexing scenario.
2. Job 2. Feature filtering. Same as indexing.
3. Job 3. Feature scaling. Same as indexing.
4. Job 4. Feature sketching. Same as indexing.
5. Job 5. Query for candidates. This step must search for feature candidates for the queried features. This, again, depends on the specific implementation:
   - Disk (HBase). Similar to what we did in the indexing case, a connection is opened for each partition where the sketches are distributed into and the database is queried retrieving all candidates for each feature sketch.
   - Memory. In the memory implementation we need several jobs:
     (a) Mapping: for each table i, a new RDD is computed containing the block i from their sketches. Then, it is *joined* with the $i - th$ indexed blocks data so the result represents the block matches for table i. When all table matches have been computed, they are gathered using *union* operations.
     (b) Weighting: each block match is given a weight of 1.
     (c) Group by match: matches belonging to the same feature are grouped.
     (d) Filter by feature match: feature matches that do not match at least the minimum number of tables are discarded.
     (e) Map to match: candidates are mapped into a new RDD.
6. Job 6. Hamming filtering. Each feature match in the candidates RDD is filtered discarding those whose Hamming distance is above the threshold.
7. Job 7. Image identifier extraction. This job extracts the identifiers of the unique matches and assigns a weight of 1 to each of them.
8. Job 8. Weight accumulation. The sum of weights for each image matched is performed.

9. Job 9. Weight filtering. The resulting RDD from the previous job contains pairs of image matches and the corresponding weight of the match. If weight filtering is enabled, the matches are filtered and only those with weights above a threshold are kept.

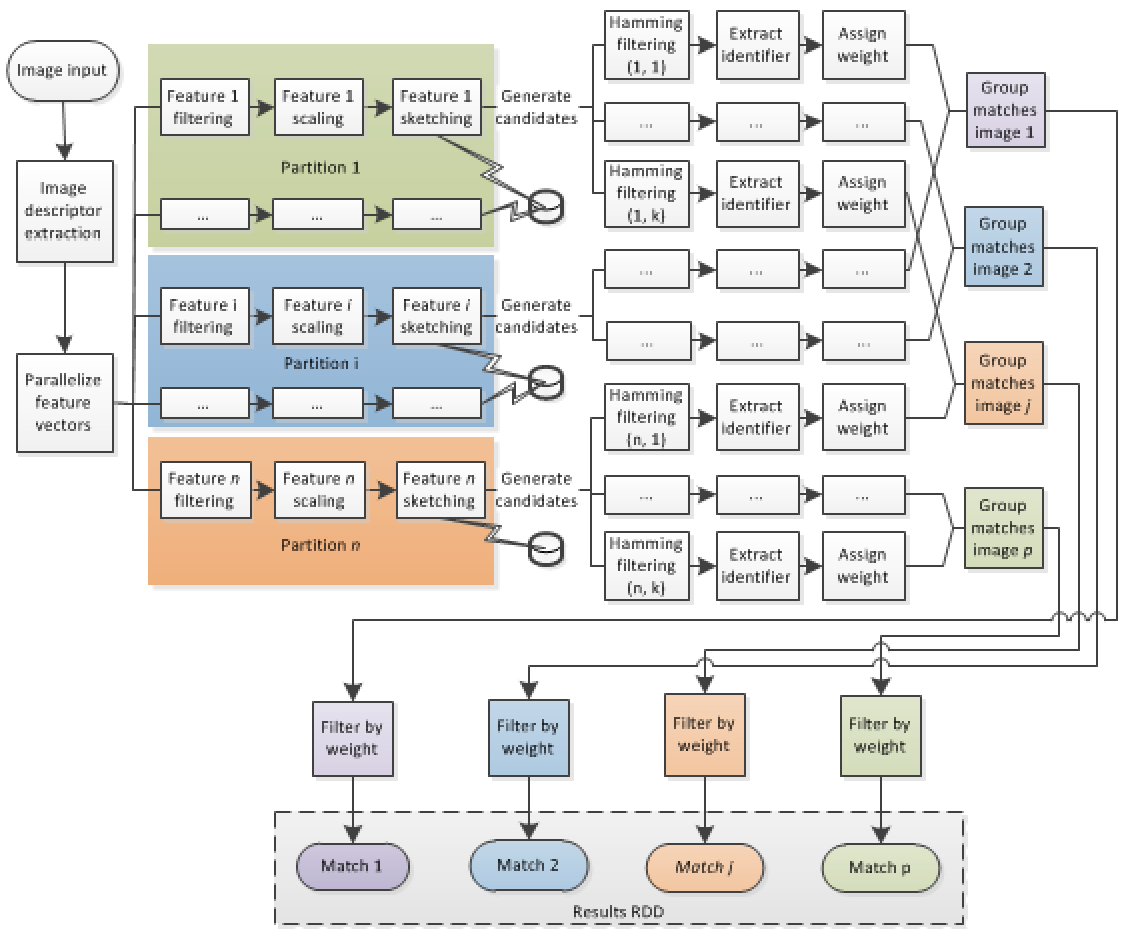


FIGURE 4.11: Spark implementation overview of the image query use case for N images where N = 1

Lineage graphs from both implementations can be found in Section B.2 showing, again, the overhead introduced by the memory approach that makes it far more complex than the disk-based one.

4.3.5.3 Streaming

Spark Streaming enables processing of live data streams. The abstraction of continuous data streams provided by Spark Streaming are called *discretized streams* or *DStreams*. They are represented as sets of RDDs belonging to a certain interval of time or time window, as seen in Figure 4.12. Transformations on DStreams generate other DStreams and they are computed by executing operations on each of the RDDs contained.

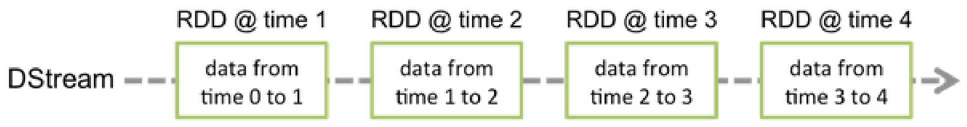


FIGURE 4.12: Input data in Spark Streaming. Source: Apache Spark documentation

Spark streaming can receive input data from multiple sources such as Kafka queues, Kinesis, Flume or Twitter, the endpoint we have used in this work. Each receiver in an application needs to have an extra executor to receive the data.

The worflow for the streaming use cases is the same for both disk (HBase) and memory cases: we apply the already described steps on each of the RDDs that conform each DStream.

The rate of records received per second can be limited by setting the configuration parameters "spark.streaming.receiver.maxRate" to a positive value.

4.4 Persistent data management: HBase

Indexing massive amounts of image information require high storage requirements and a way to handle data in a scalable and efficient way. Assuming an average of 250 features are extracted per image and indexing 50 million images, information related to at least 2500 million elements needs to be stored and retrieved easily. In these cases, traditional relational database management systems are not enough to satisfy the requirements: they do not scale for such big amounts of data.

The way we need to insert and retrieve data is by using values as a key (feature identifiers and sketch block keys), as seen in Section 3.1.0.2. We have focused our research on key-value based database managers with special attention on their scalability. Several options were found: Redis, Cassandra and HBase. Though Redis is a very fast it is not distributed and it is suitable to be used with volatile data.

Cassandra and HBase have very similar characteristics: key-value based, NoSQL and HDFS integration. They are both an implementation of the groundbreaking BigTable paper from Google [29]. Though Cassandra's documentation is more complete and the set up is easier than in HBase, we decide to use the second because we are a little familiar with it.

To make sure we have chosen the proper technology, we check that our application fits all the requirements to be implemented with HBase following the guidelines on their reference guide [30]:

- In a real context we expect to handle hundreds of millions of data.
- We do not need to use extra features from traditional database management systems such as transactions or triggers.
- For a proper deployment at least 6 nodes need to be used. This last requirement is not important for our development because we are using a single machine. However, it must be considered for further deployment in real clusters and it is a feasible requirement.

4.4.1 Data

Data in HBase is stored in tables, identified by a table name. Tables have several columns that are grouped by *column families*, that can have multiple *column qualifiers*. Related qualifiers should be stored in the same column family because they are stored close to each other. Column names are formed by using the column family as the prefix and a qualifier, separated by a colon. Column families are defined at the table schema definition while column qualifiers are defined at runtime, and both amounts are recommended to be kept low. HBase data can be considered as sparse: not all table entries must have values for all columns and they do not need to share the same column qualifiers.

A table entry (row) is identified by a unique key, a set of uninterpreted bytes. Rows are lexicographically sorted in an ascending way, so lower order keys are in first place. HBase can keep track of the modifications of a column value in time, keeping a configurable limit of them. Each of these modifications is called a *version*, and has a timestamp assigned. The default number of versions stored is 1 and is the suitable for our system: features and images are constant in time. The union of a *row*, a *column* and a *version* is called *cell*.

Tables (HTable) are organized in regions and may be split among different ones. Initially and by default, all tables start in a single region and when they reach a given size, they are split by the middle. Though data is stored in disk, HBase has several configurable levels of caching that help avoid seeking into disk for recurrent data requests.

Similar to other data base management systems, common CRUD operations (Creation, Retrieval, Update and Deletion) are implemented. The basic operations in HBase are:

- Get: return the column of a specific row. Subset of columns, qualifiers, versions and timestamps can be provided for a more accurate result.
- Put: adds or updates (if the key exists) the content of the row. It always creates a new cell with a specific timestamp. If the maximum number of versions is 1 and the row already exists, it is overwritten.
- Scan: iterates over rows and returns their columns. Different filters can be applied as in get operations. Scan operations can be done by searching row prefixes or by intervals of rows, being the latter faster. It is important to be concerned about the limitations of HBase: it is intended to search through small intervals of rows and scan operations of whole tables can take a lot of time.
- Delete: deletes a specific version, column family or column from a row in a table.

There is no modification operation in HBase: the need to modify a cell implies deleting and recreating it.

4.4.2 Architecture in HBase

The architecture model (Figure 4.13) follows a master-slave approach. There are some important elements in the HBase architecture:

- Master node (HMaster): manages and monitors the working cluster and balances the load of work.
- Region Server node (HRegionServer): manages the access to its regions and and holds the communication with the client. A region is every component the storage space is divided in.
- .ROOT. table: table storing the location of .META. table.
- .META. table: table containing the assignment of the data to each Region Server.

Each Region in a Region Server has a MemStore, an in-memory data store where information is saved before being written to disk. MemStore flushes data into disk when it is full.

There are two types of files: StoreFiles and WALs. WAL (Write-Ahead Log) is a file shared by all regions in a Region Server that stores new data that has not been persisted. It is useful in case of server failure because it can recover data contained in the MemStore

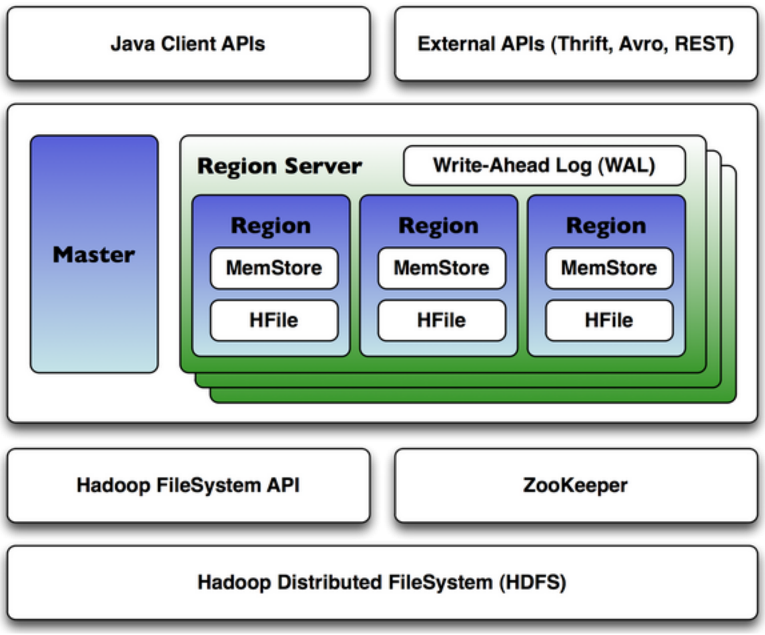


FIGURE 4.13: Summary of HBase architecture. Source: DZone [3]

that has not been persisted into disk. The implementation of WAL in HBase is called HLog.

One key aspect to ensure the scalability of HBase is the region splitting. Each region is automatically configured to hold a range of row keys. By default, when region exceeds a configurable size, HBase splits the region in two child regions. These splits can be manually configured by pre splitting the tables or manually, but it is only recommended for advanced users and only when the user knows the distribution of the row keys beforehand: regions split poorly may affect the cluster load distribution and decrease its performance. We decide to rely on the default splitting configuration from HBase.

HBase works on the top of Zookeeper, a cluster manager. Zookeper is up to tasks such as the master selection, the region lookup or the coordination between nodes. HBase manages already the Zookeper cluster but it can also point to existing clusters accessible from all the clients.

4.4.3 Data workflow

Here we describe the workflow when a client tries to write a value:

1. Client contacts Zookeeper cluster to find a specific cell by providing its key and table. Zookeeper retrieves the server hosting the .ROOT. table. Now client can query the .META. table. This information is requested once and cached afterwards.

2. Client can now query the meta table to get the server that holds the key for the given table.
3. The client caches the information of the server and does not need to query the meta table again. From now and on, the client directly contacts with the corresponding HRegionServer.
4. When the HRegionSever receives the connection and opens the region, it creates the corresponding HRegion object. The HRegion itself creates *Store* instances, one per each Colum Family for each table. The Store instance can also have one or several *StoreFiles*, that are wrapper around the actual distributed multi-indexed files called *HFiles*.
5. Client pushes data into the server, that is first written into the WAL (HLog) file. By default it is enabled and it is recommended to do it to recover information in case of server failure. Afterwards, data is written into the MemStore. In case it gets full, data inside is flushed into the HFiles.

4.4.4 Configurations

HBase has 3 possible working configurations:

- Distributed mode: it works on a real distributed cluster where each node is a HMaster or a HRegionServer (or both).
- Pseudo-distributed: it works on a single host machine but each needed daemon (HMaster, HRegionServer and Zookeeper) works on a separate process.
- Local: everything works on a single host and different processes are created in the same Java Virtual Machine. It is only used for development purposes.

Local mode has been adopted in this work because of the lack of time and hardware resources and the lack of expertise in cluster management.

HBase is widely configurable and adaptable to user needs. Some of its parameters can be tuned in the *hbase-site.xml* inside the configuration folder. We identified some parameters that could be useful in a real cluster deployment for our specific problem:

- hbase.client.max.total.tasks: Maximum number of concurrent tasks a table instance will send to the cluster (default 100).

- hbase.client.max.perserver.tasks: Maximum number of concurrent tasks a single HTable instance will send to a single region server (default 5).
- hbase.client.max.perregion.tasks: maximum number of concurrent connections the client will maintain to a single region. If this number is reached, next connections will have to wait until some writes finish (default 1).
- hbase.client.scanner.caching: Number of rows fetched when calling next on a result scanner object when not served from memory (default 100).

4.4.5 Data structure

4.4.5.1 Design of a row key

One of the key aspects in the design of a HBase schema is the row key definition, the unique value that identifies a row in a table. There are some ideas we should take into account when designing a row key:

- Row keys are unique.
- Row keys should be as short as possible and also be designed to be accessed easily. Names of column qualifiers and their values must be kept short as well.
- Rows are sorted alphabetically.
- Rows with similar keys are stored in the same region.

These characteristics are important when designing the tables in HBase. Note that autoincremented identifiers or timestamp values are not recommended to be used as keys: successive values would be pushed into the same region and we would not take profit of the distributed behaviour, easily blocking the Region Server containing the rows. Thus, row keys must be as random as possible.

The following sections explain the HBase implementation of the data structure design described in the previous chapter.

4.4.5.2 Hash tables

Each hash table relates to a slice of the Hamming representation of a feature and uses this block as key. Matching features must have at least one identical block, so there may be two features with the same block key. Though, row keys must be unique, so our design appends a separator (i.e. an underscore) to the integer representation of the block key followed by the unique image identifier and embeds it into a string, building a unique identifier that uses the block key as a prefix. This approach sacrifices disk storage (string representations are variable and longer than numeric ones) by gaining ease of access and guarantees that there can be multiple features from different images with the same block identifier.

One column family is defined for this table: column family *hf* (i.e. hamming family). A qualifier *fq* (i.e. feature qualifier) is defined within to represent the reference to the feature a block belongs to. An example of a hash table can be found in Table 4.1. The table shows two cells from one out of 4 hash tables, containing two feature blocks with respective keys 1,546 and 84,476 belonging to the same image 1,450.

Row key	Column family 'hf'
1546_1450	$hf : fq =$ "1546_11478_4568_55442_1450"
84476_1450	$hf : fq =$ "84476_9887_0002_2145_1450"

Table 4.1: Example of a hash table containing two blocks from different features belonging to image 1,450

4.4.5.3 Feature table

This table maps each feature with the image they belong to and contains the full feature sketch. The feature identifier or row row key for a feature is built by appending the sketch block values separated by an underscore with the image the feature belongs, both separated again by an underscore. It has one column family called *fi* (i.e. feature info) with qualifiers:

- *id*: links the feature with the image unique identifier.
- *hc*: long representation of the feature sketch.

Row key	Column family 'fi'
1546_11478_4568_55442_1450	$fi : fid =$ "1450"
1546_11478_4568_55442_1450	$fi : hc =$ "1546_11478_4568_55442"
84476_9887_0002_2145_1450	$fi : fid =$ "1450"
84476_9887_0002_2145_1450	$fi : hc =$ "84476_9887_0002_2145"

Table 4.2: Rows (cells) for two indexed features belonging to the same image 1,450

4.4.5.4 Image table

This table contains the basic metadata of the image indexed. The image identifier is used as row key and and it contains one family *imf* (i.e. image metadata family) with qualifiers:

- *pq*: Path to the image file or URL.
- *rq*: The resource identifier the image belongs to (e.g. if it belongs to a *Tweet* it contains the tweet identifier).
- *prq*: Source the image comes from. Some examples: disk, Twitter, Instagram, etc.

Row key	Column family 'imf'
1450	$imf : pq = "https : //pbs.twimg.com/media/CCvbbxpW8AAmClP.jpg"$
1450	$imf : rq = "588805469394886656"$
1450	$imf : prq = "Twitter"$
65	$imf : pq = "https : //pbs.twimg.com/media/CCvFBbJWgAEjKyC.jpg"$
65	$imf : rq = "588780804651757568"$
65	$imf : pq = "Twitter"$

TABLE 4.3: Rows containing information from two images

4.4.5.5 Parameters table

This table stores the settings defined by the user at the initialization. It contains one only row identified by the key '1'. Three column families are defined:

- *df*. Parameters related to the feature vectors extraction. Qualifiers used are:
  - *kyq*. Keypoint extraction technique (values: 'SIFT', 'SURF', 'HARRIS', 'ORB', 'BRIEF', 'BRISK' and 'MSER').
  - *dtq*. Descriptor computation method (values: 'SIFT', 'FREAK', 'ORB', 'BRIEF' and 'BRISK').
  - *msq*. Maximum size of the largest side of the images to be resized before computing their feature vectors.
- *ff*. Parameters related to the pre-processing of the feature vectors. Qualifiers used on this column family are:
  - *ftq*. Type of filtering to apply on features (values: 'NONE', 'ENTROPY' and 'VARIANCE').

  - *tq*. Threshold of the feature filtering, if any.
  - *lsq*. Whether log-scale should be applied.

- *if*. Parameters belonging to the hash function used to build the sketches. The following qualifiers are used:
  - *aq*. *A* vector from the sketch construction function. Values are separated by commas.
  - *bq*. Parameter *b* from the sketch construction function.
  - *cq*. Whether compression should be enabled on the rest of the tables.
  - *dbeq*. Whether block encoding should be used on the column families of the rest of the tables.
  - *hdq*. Hamming distance threshold to identify matching features.
  - *ntq*. Number of hash tables.
  - *ttlq*. Time-to-live of cells. It is the time (seconds) the images remain alive in the database. It may be useful for streaming scenarios where we only want to evaluate the most recent data and when storage is a restriction. It is set to negative when it is disabled.
  - *wq*. Sketching sesnsitivity parameter W.

Table 4.4 shows an example of a parameters table. In memory-based implementations this table is replaced by a properties file.

Row key	Column family 'df'	Column family 'ff'	Column family 'if'
1	$df : keyp = "BRISK$		
1	$df : dt = "SIFT"$		
1	$df : msq = "350"$		
1		$ff : ft = "VARIANCE"$	
1		$ff : tq = "900"$	
1		$ff : lsq = "true"$	
1			$if : aq = "0.019, [...], 0.24"$
1			$if : bq = "8"$
1			$if : cq = "true"$
1			$if : hdq = "3"$
1			$if : ntq = "4"$
1			$if : ttlq = "120"$
1			$if : wq = "10"$

Table 4.4: Rows containing the parameter configuration for a replica detector

4.4.5.6 Storage optimizations

Storage can also be optimized taking profit of specific characteristics of our implementation and its redundancy. In the previous subsections we have seen that each block of a hash table references the entry in the *feature table* of the feature it belongs to. Moreover, we have seen that this reference is identified by the feature id, that is built using the sketch representation of the feature and the image identifier. When querying a feature to the database, we need to query all hash tables and for each candidate, query also the feature table to get the sketch representation. However, this representation is already embedded into the feature identifier. It is straightforward that, in our specific implementation, deleting the feature table would save us time (as we save a query to a table) and also storage.

After testing this new optimization using 510 images indexed, that generated around $110,000$ features and querying 570 images to them, we conclude that:

- 18 MB were saved in the indexing process.
- The average query time got speeded up by a 2.3%, which is not very significant.

Though the average query time was not substantially reduced, we see that we get some benefits that could be significant in large scale environments. We decide adopt this approach for the rest of the work.

4.4.5.7 Data encoding and compression

There are two mechanisms that can be applied to reduce storage in HBase: compression and data block encoding. Compression reduces the size of byte array cells while data block encoding takes advantage of design aspects of HBase (e.g. sorted row keys) to limit the duplication of information. They both can be used together on a Column Family.

Different choices exist in HBase for both data block encoding and compression and it is important to choose the right ones according to the specific application: wrong choices could even lead to higher storage sizes and performance burdens. The criteria to decide which to apply relies on the key and value length for the block encoding and the frequency of access for the compression. We can consider data as *hot data* (accessed frequently) because the data to access depends on the query one, that is random, and all information has the same probability to be accessed. Snappy and LZO are the compressors available for *hot data*, being Snappy the default since 2011 and proving to be faster. We decide to use Snappy compression on the hash tables.

Data block encoding approaches are only recommended when keys are longer than values or when tables have many columns. Considering that our tables have few columns and that usually values are longer than the keys, we discard using data block encoding.

```
HTableDescriptor desc = new HTableDescriptor("t");
HColumnDescriptor hcd = new HColumnDescriptor("f");
hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
Configuration c = HBaseConfiguration.create();
HBaseAdmin hbaseAdmin = new HBaseAdmin(conf);
hbaseAdmin.createTable(desc);
```

FIGURE 4.14: Creating a table t with FAST DIFF block encoding enabled on the column family f

```
HTableDescriptor desc = new HTableDescriptor("t");
HColumnDescriptor hcd = new HColumnDescriptor("f");
hcd.setCompressionType(Algorithm.SNAPPY);
Configuration c = HBaseConfiguration.create();
HBaseAdmin hbaseAdmin = new HBaseAdmin(conf);
hbaseAdmin.createTable(desc);
```

FIGURE 4.15: Creating a table t with Snappy compression on the column family f

Our experiments using 510 indexing images and 570 image queries showed that Snappy compression saves 1.69% of storage preserving the indexing and querying time. We decide, then, to adopt the compression in our system. Though, again it is not significant in a small context it may be in large scale environments.

4.4.5.8 Size

Storage is an important aspect part of our implementation though it is not critical. We want to measure the space needed to store each feature and image but HBase does not provide a mechanism to size the data stored. Instead, we check the average size of the HBase data folder after performing several probes, havint the results in Table 4.5.

Element	Size (bytes)
Image metadata	$150 - 250$
Feature per table	53

TABLE 4.5: Average sizes on HBase

Considering these results and assuming that we extract an average of 250 features (without feature filtering) per image using 4 hash tables, we would need at most $(53 * 4 * 250 + 250) * 50 \cdot 10^6 = 2.66$ TB of data to index 50 million images. Usually HBase will use more data than that because of its overhead (e.g.: WAL files increase the size of the folder and old WALs are also kept). HBase can be configured to replicate data but we disabled this option to save disk space.

Chapter 5

Results

Once our framework is defined and built a way to validate its outcomes must be designed. In this section we detail the data we used to evaluate our implementation, the evaluation criteria and the experiments performed.

5.1 Data sets

Though a lot of work has already been done on near replica image detection, there is no related open dataset where to evaluate our approach from. Most of the works analysed built they own datasets getting their images from search engines or general data bases and then applied transformations on them using a variety of tools (e.g. ImageMagick [31]), but none of them are public. [15] published the data set used in their work but it is no longer available online.

There is a related replica detection dataset called Affine Covariant Regions dataset [32]. It contains 8 groups of 6 images where 5 of them are modifications from the original one. Transformations are complex, sometimes combined (zoom + rotation) and some of them are out of the scope of our detector (viewport change). However, we need a more complete dataset containing more examples and types of transformations. To test our system we decide to build our own data set: the UPCReplica dataset.

5.1.1 UPCReplica data set

For the evaluation and the tuning of the parameters of our the implementation we define an annotated data set of 3.925 images, consisting of:

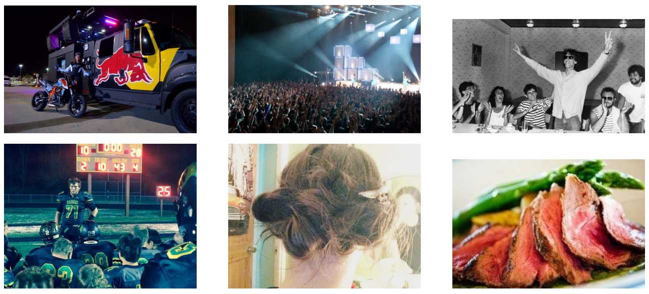


FIGURE 5.1: Examples of background images from the UPCReplica dataset

- $2,275$ 'background' (BI) images. Those are assumed to be images that have no replica within the data set. Figure 5.1 shows some examples.
- 50 'source images' (SI). They are the set of images we compute replicas from.
- $1,600$ 'replicas' (R). Modified versions of the *SI* images.

Twitter Streaming API [33] have been used to get the 2,325 images that conform both the *BI* and *SI* sets and have been manually checked to delete all replicas. A set of 14 transformations (Table 5.1) has been defined to generate modified versions (replicas) of the *SI* and 32 versions of each image have been generated using OpenCV.

The *UPCReplica* data set has been made public to be used under GPL 2.0 license in order to empower research on this topic and can be accessed online [23].

5.1.2 Data set 2: Desigual dataset

This second dataset contains 16,000 random images extracted from *Instagram* containing the tag "Desigual", a popular Spanish clothing brand. These unlabelled images have been used for the scalability tests performed.

5.2 Experiments design

Different experiments must be designed to test all parameters seen so far. The first one is intended to measure the capability of finding near duplicates using different descriptor types and the impact of using different sketching sensitivities. The second experiment evaluates how false positives can be reduced by feature filtering. The third experiment covers the impact of the weight filtering of the replica matches on both the precision and the recall. The fourth experiment is intended to evaluate the performance of the selection of parameters from previous experiments. Fifth experiment compares the performances of

Operation	Parameters	Number of replicas
Rotation	degrees=30, 90, 180, 270	4
Horizontal cropping	keep 75 % and 50%	2
Channel V modification	+10 % and -10%	2
Channel S modification	+10 % and -10%	2
Vertical cropping	keep 75 % and 50%	2
JPEG compression	quality=0.1, 0.3, 0.5, 0.7	4
Resizing	scale=0.5, 0.8, 1.2, 1.5	4
Gaussian noise addition	mean=0.0, variance=0.1; mean=0.0, variance=0.25	2
Averaging filter	kernel size=2,3	2
Horizontal flipping	none	1
Gamma correction	gamma=0.25, 0.6, 1.5, 1.8	4
Text addition	length=15	1
Occlusion using black circles	circles number=3, size=10%; circles number=2, size=15%	2
Total		32

TABLE 5.1: List of image modifications

$$precision = \frac{truePositives}{truePositives + falsePositives} \tag{5.1}$$

Definition of precision

$$recall = \frac{truePositives}{truePositives + falseNegatives} \tag{5.2}$$

Definition of recall

memory-based and disk-based implementations. Sixth experiment focus on the scalability of the system. Seventh experiment faces the detector to real life image modifications and the eight and last experiment is performed on the Affine Covariant Regions dataset.

In each experiment we evaluated the elapsed time of both indexing and querying, and the precision (5.1) and the recall (5.2) of each result.

All experiments have been tested on our development machine: a single host with a Xubuntu 12.0.4 operative system, 4GB of RAM memory and a Pentium(R) Dual Core T4300 2.10 Ghz and local Apache Spark and HBase configurations. All experiments except from Experiment 6 have been tested using disk persistence of the data (HBase). A consideration must be done before evaluating the results: a different random hash function has been used in each probe to generate the sketches and may explain some random behaviours in the results.

5.2.1 Experiment 1: Descriptors

For this experiment we define two sets of images from the UPCReplica dataset:

- *Base set*: set of 10 *source images* and 500 *background images.*
- *Query set*: set of 250 *background images* and all 320 *replica images* generated from the source images in the base set.

This experiment consists of indexing the base set and then match the query set against the indexed data using different descriptors and modifying the sketching sensitivity parameter W. Keypoint detection is made using BRISK and images are resized to 350 pixels its largest side and converted into grayscale. Results can be found in Figure 5.2 and Figure 5.3.

Looking at both the query and indexing times we see how it is directly affected by the number of tables used to approximate the hamming distance: being the SIFT using 8 tables the slowest, then probes using 4 tables (and being the SIFT one the slowest among them because of being more computationally expensive) and last probes using 2 hash tables.

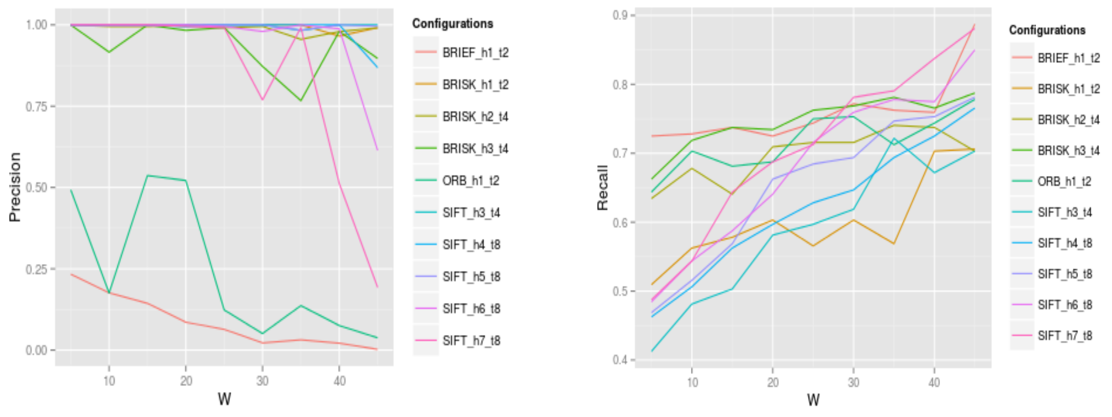


FIGURE 5.2: Precision and recall for variable sketching sensitivity

ORB and BRIEF show very low precisions at all levels (below 50 % most of the time) and will be discarded in future experiments. Overall recall grows as W does but at high values the precision starts to fall dramatically, specially in configurations with higher hamming thresholds.

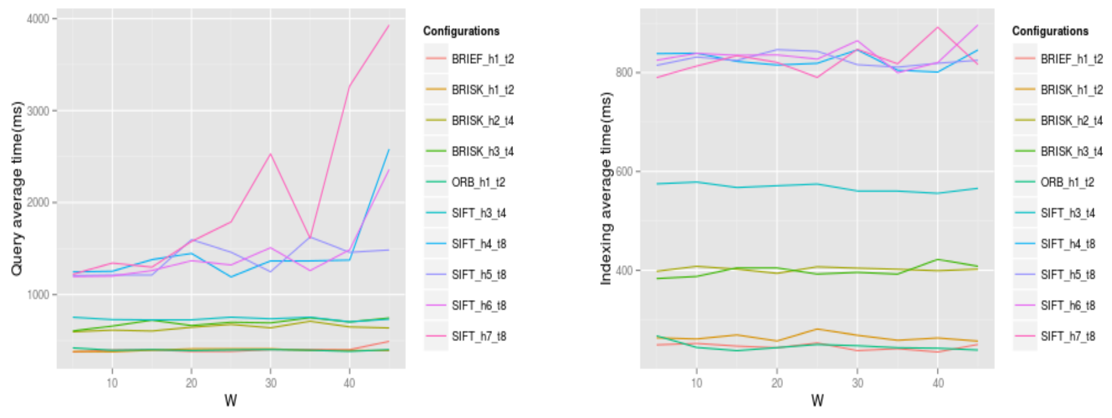


Figure 5.3: Indexing and query average time for variable sketching sensitivity

From remaining configurations we want to select those that have best performance using less hash tables and thus being faster. SIFT probes with 8 tables show very good true positive rate at high W but poor precisions. At smaller values of W where SIFT precision is still high, BRISK configurations outperform them by similar recalls and smaller response time. We select the best configuration from the ones using 16 bits per sketched block and the only two configurations using 32 bits per block. This is an important fact because smaller block utilization may lead to higher collisions rate, as commented in Section 3.1.0.3, and therefore less chances to scale. The three configurations selected are:

- BRISK descriptors using 4 tables, hamming threshold of 3 and $W = 25$ (BRISK4t3h).
- SIFT descriptors using 4 tables, hamming threshold of 3 and $W = 30$ (SIFT4t3h).
- BRISK descriptors using 2 tables, hamming threshold of 1 and $W = 45$ (BRISK2t1h).

FREAK descriptors could not be tested due to unexpected errors with OpenCV.

5.2.2 Experiment 2: Feature filtering

In Section 3.0.2 we saw how it was not possible to distinguish between high quality and regular feature vectors by their entropy and variance. Nevertheless, considering that only one match is enough for finding a replica, it is possible to discard some of the strong features in order to increase precision and reduce response time and still preserve all (or most of) feature matches. Configurations selected are the ones from the previous experiment, named as SIFT4t3h, BRISK4t3h and BRISK2t1h.

The results from applying different variance and entropy thresholds are shown in Figure 5.4 and in Figure 5.5. We observe that precision increases in BRISK2t1h configuration while BRISK4t3h improves using entropy and shows a random and poor behaviour using variance. SIFT4t3h remains without false positives in both filtered and non-filtered configurations.

In both cases, as the filtering threshold rises, the indexing and querying time averages decrease. However, the recall starts to fall at high levels of filtering. We must find a balance between high precision and good recall using the highest filtering threshold to obtain the smallest response time. Table 5.2 shows the selected threshold for each of the 3 configurations tested.

Configuration	Filtering type	Threshold	Precision(%)	Recall loss (%)	Indexing speed up (%)	Querying speed up (%)	Features discarded (%)
BRISK4t3h	Entropy	4.6	100.0	2.5	5.31	12.78	27.28
BRISK2t1h	Variance	7,500	100.0	4.71	2.35	3.68	7.41
SIFT4t3h	Variance	900	100.0	0.32	25.75	23.37	52.63

Table 5.2: Selected feature filtering thresholds and impact on performances

After feature filtering BRISK4t3h processes around 210 features per image, BRISK2t1h around 250 features per image and SIFT4t3h around 105 features per image.

5.2.3 Experiment 3: Filtering weighted results

This experiment tests how the weight assigned to each matched image can be used to improve precision and preserve a decent recall by setting a threshold on it. First of all, we test the configurations from the previous experiment on a subset of the UPCReplica dataset. We index 650 images and query 3,276 images against them: 1,600 replicas and 1,676 background images.

Configuration	Precision(%)	Recall (%)
BRISK4t3h	98.15	76.50
BRISK2t1h	77.77	70.63
SIFT4t3h	97.69	63.00

Table 5.3: Results for the UPCReplica dataset

The results from Table 5.3 show good recalls but precisions still not the expected.

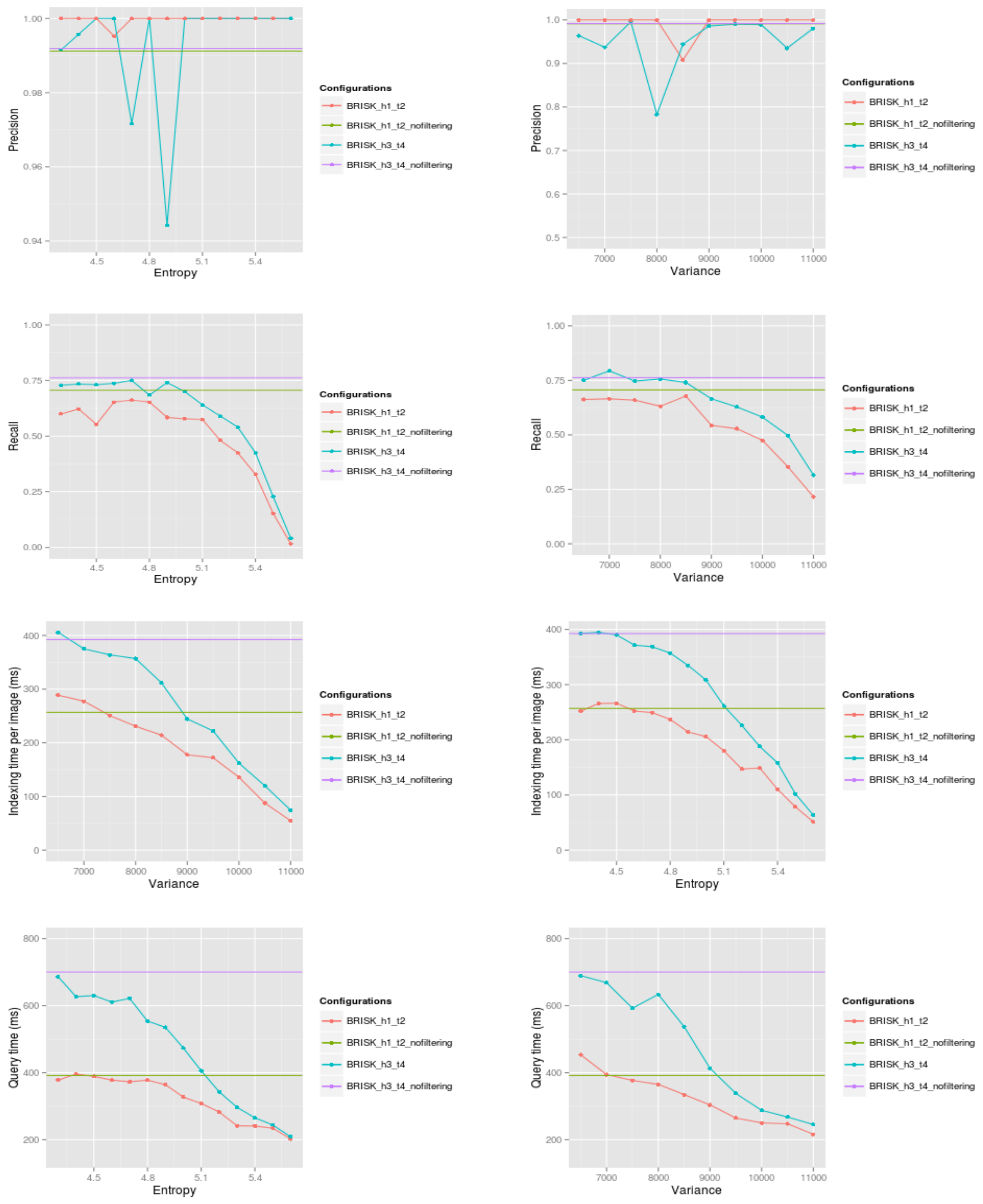


FIGURE 5.4: Performance results for entropy and variance filtering on BRISK sketches

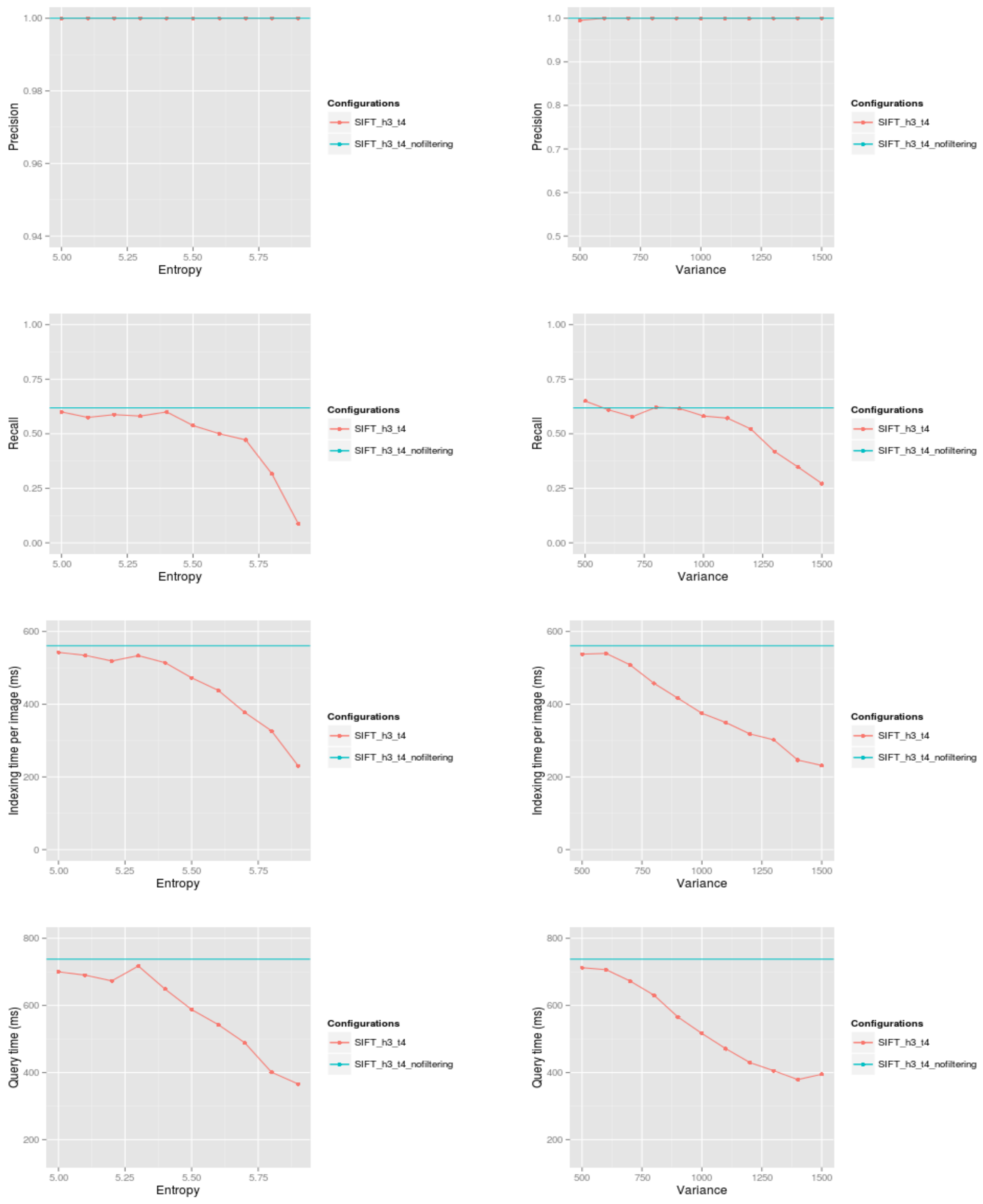


FIGURE 5.5: Performance results for entropy and variance filtering on SIFT sketches

Configuration	Precision(%)	Recall (%)
BRISK4t3h	98.15	76.50
BRISK4t3h + WF (thresh = 2)	99.82	69.13
BRISK4t3h, $W = 20$	99.41	74.86
BRISK4t3h, $W = 15$	99.24	74.25
BRISK4t3h, $W = 10$	99.57	73.06
BRISK2t1h	77.77	70.63
BRISK2t1h + WF (thresh = 2)	99.89	59.13
BRISK2t1h, $W = 40$	97.69	68.75
BRISK2t1h, $W = 35$	98.19	64.43
BRISK2t1h, $W = 30$	99.90	62.56
SIFT4t3h	97.69	63.00
SIFT4t3h + WF (thresh = 2)	99.32	54.44
SIFT4t3h, $W = 25$	99.78	56.81

TABLE 5.4: Results for weight ranking

Figure 5.6 show the density of weights for both false positives and true positives for the three configurations. We observe that most of the false positives have a small weight of but a big percentage of true positives also fall in that interval. Again, it is not possible to decrease false positives without affecting the true positive rate.

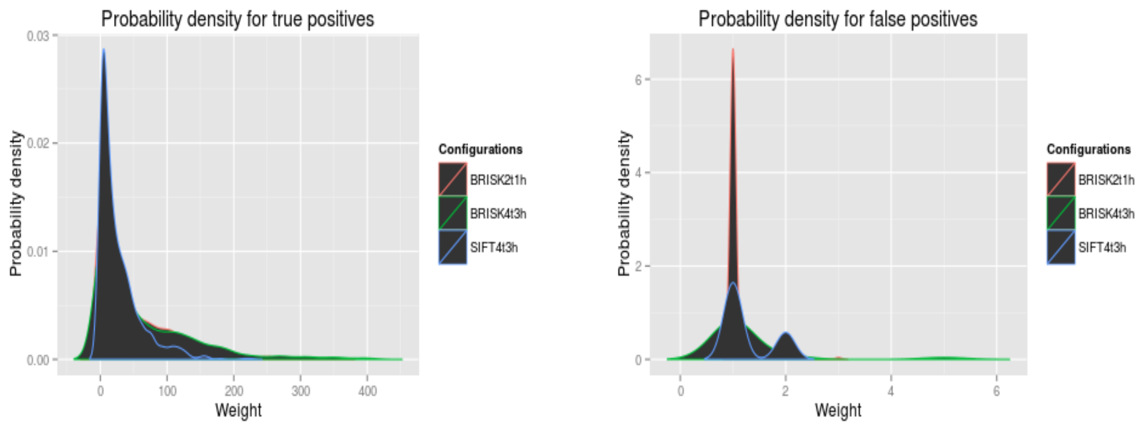


FIGURE 5.6: Distribution of weights

The same experiment is performed, using weight filtering (WF) and also different levels of W. Results can be found in Table 5.4, showing that WF achieves to increase precision but penalising the recall. In both BRISK2t1h and SIFT4t3h reducing the hashing sensitivity W obtains the same effect on precision and preserves better the true positives rate.

We confirm that filtering by the weight is a good option to improve precision but in many cases reducing the sketching sensitivity performs similar or even better.

5.2.4 Experiment 4: UPC Replica dataset

After all parameters have been tuned we want to see how the resulting configurations perform on the complete UPCReplica dataset, indexing $2,076$ images and querying $1,600$ replicas and 250 background images against them.

Configuration	Precision(%)	Recall (%)	Indexing average time (ms)	Query average time (ms)
BRISK2t1h, $W = 30$	98.00	67.56	168.99	394.59
BRISK2t1h + WF (thresh = 2)	99.80	67.34	176.80	399.21
BRISK4t3h, $W = 10$	97.78	74.36	298.60	827.51
BRISK4t3h + WF (thresh = 2)	98.56	72.50	298.16	1078.50
SIFT4t3h, $W = 25$	93.03	62.56	334.49	613.66
SIFT4t3h + WF (thresh = 2)	99.53	53.06	331.382	584.20

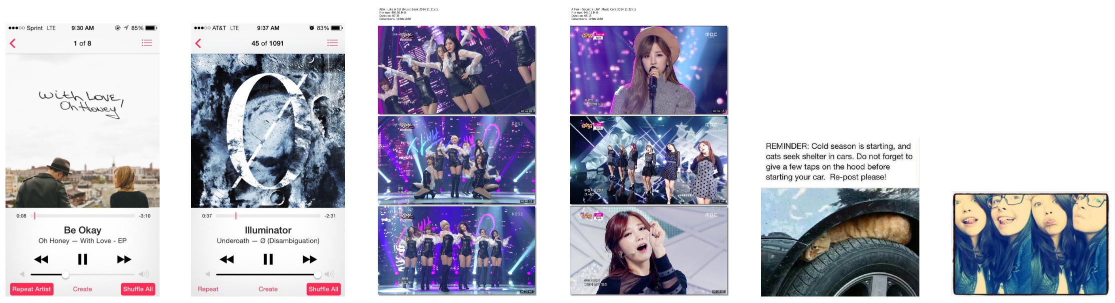
Table 5.5: Results for the UPCReplica dataset

Table 5.5 shows the results, proving that some precisions are still far from the 99% the expected we set as requirement. A deeper analysis of the false positives encounters that not all of them could be considered in the same way and we identify 3 categories of false positives:

- Replica match: Though the dataset was manually checked several times some images detected are identified as resized replicas of others. These images have been removed from the final dataset that is available online.
- High similarity match: Images that do not come from the same source but share a lot of traits. All of them belong to sreenshots of the same mobile application where the common interface is an important element of the structure of the picture.
- Similar match: Images do not share significant characteristics but their structures are similar and are easily mismatched by the detector.
- False positive match: Images matched are completely different from each other.

Figure 5.7 show examples of the different levels of similarity between false positives. We assign all false positives detected to one of the categories defined and identify their weight relative to the total amount of false positives.

Considering the numbers in Table 5.6 we see how precisions are highly improved when replica matches are erased and how most of them lie close to the 100 % when we also consider the very similar ones (High Similarity).



REMINDER: Cold season is starting, and cats seek shelter in cars. Do not forget to give a few taps on the hood before starting your car. Re-post please!

Figure 5.7: Example of high similarity (left), similarity (center) and non-similarity (right) between false positive matches

Configuration	Replicas(R)	High similarity(HS)	Similar(S)	False positives(FP)	Precision + (R)	Precision + (R) + (HS)
BRISK2t1h, $W = 30$	9.09	18.18	4.55	68.18	98.18	98.54
BRISK2t1h + WF (thresh = 2)	50.00	50.00	0.00	0.00	99.90	100.00
BRISK4t3h, $W = 10$	69.23	15.38	3.85	11.54	99.31	99.59
BRISK4t3h + WF (thresh = 2)	89.47	10.53	0.00	0.00	99.85	100.00
SIFT4t3h, $W = 25$	18.98	33.78	40.54	6.76	94.35	96.71
SIFT4t3h + WF (thresh = 2)	0.00	25.00	75.00	0.00	99.53	99.65

Table 5.6: Percentage of false positive matches depending on its transformation type for UPCReplica dataset

We are interested in knowing which image transformations are easily tracked and those harder to detect. The detection ratio of each image modification can be found in Figure 5.8. We can extract several conclusions from it:

- Image attacks based on color and occlusions are the easiest to detect.
- Resizing, gamma correction, compression and smoothing have also very good results and are not detected in extreme conditions.
- Vertical cropping has also high ratio detection but horizontal cropping barely trespasses the 50% threshold. It may happen because usually background elements are placed in both top and bottom of images rather than in the sides.
- Rotation shows a poor ratio around 20%.

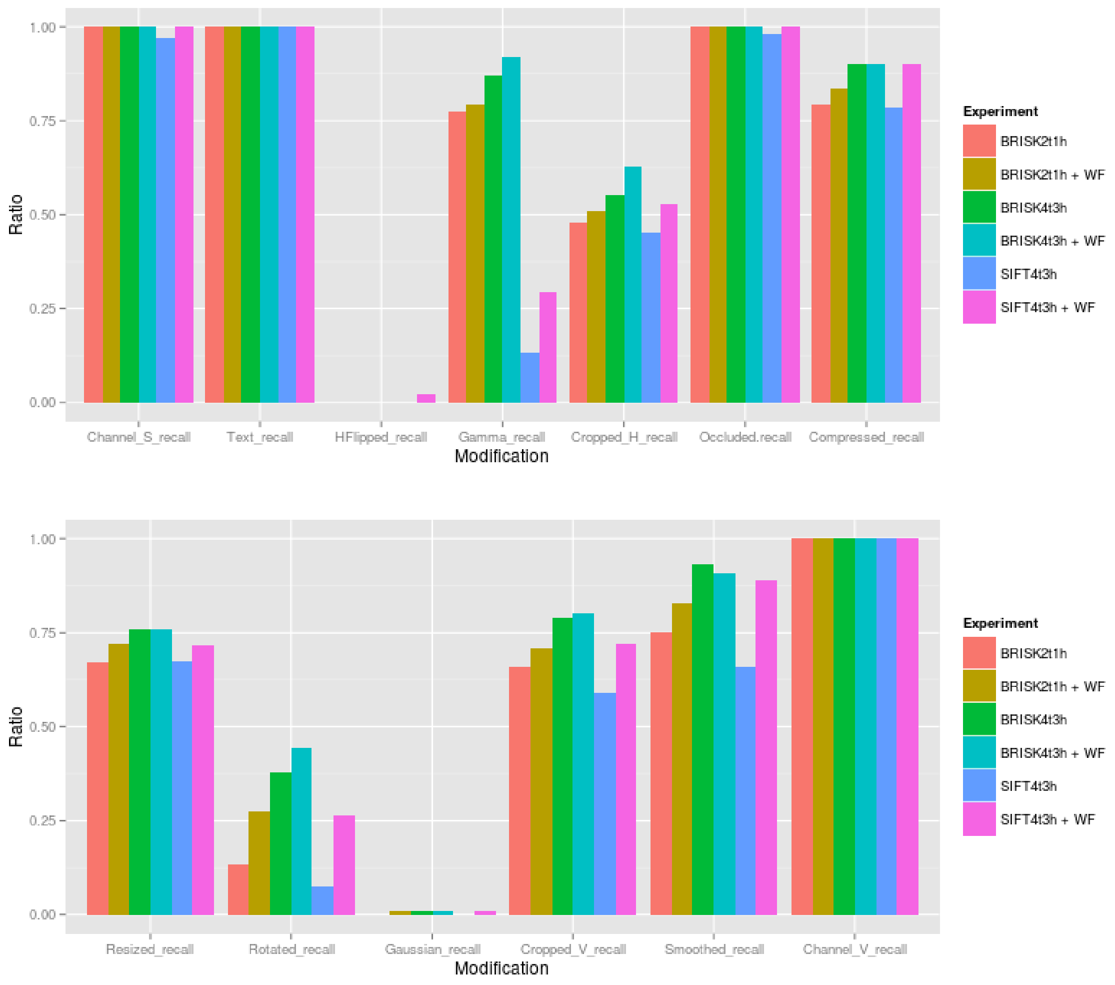


FIGURE 5.8: Ratio of detected replicas for all image transformations in the UPCReplica dataset

- Horizontal flipping is rarely detected. It may be explained because it is a global change of the image that cannot be detected by local descriptors: the orientation bins cannot detect the flipped orientations.
- Gaussian noise addition are also never detected. It is expected because the noise it applies to the image generates loads of intensity changes that are misclassified as keypoints.

For the rest of experiments we will just consider those configurations that had more than 99% of precision in the UPCReplica dataset, that are:

- BRISK2t1h using weight filtering of 2 (BRISK2t1hWF).
- BRISK4t3h with $W = 10$ (BRISK4t3hW10).

- BRISK4t3h using weight filtering of 2 (BRISK4t3hWF).
- SIFT4t3h using weight filtering of 2 (SIFT4t3hWF).

5.2.5 Experiment 5: Memory vs Disk

This experiment tries to compare both memory-based and disk-based implementations and observe how we can speed up the detector by caching the content on the main memory. Nonetheless, results showed that, in small scales, the overhead introduced by the multiple number of Spark jobs in the memory-based implementation is too expensive to compute and forces it not to scale and lose the real-time behaviour. Because of that, further experiments in this work will only focus on the disk-based approach. Further work must be done to observe the behaviour of both implementations in large scale contexts.

5.2.6 Experiment 6: Proving scalability

A key element of our design is to be able to scale to large environments with millions of images. However, our development environment is reduced to a single development machine and it only allows us to perform small-scale tests up to a few thousands of images. Cloud computing platforms such as Amazon EC2 were not used for large scale testing because that would need extra time that was not considered in our initial planning.

The goal of this experiment is to see the evolution of query and indexing time after indexing different sets from the *Desigual* dataset with increasing sizes. In each case 250 random images from the dataset have been queried. The results in Figure 5.9 show that indexing time not only remains constant but also decreases. The query time remains constant for BRISK2t1hWF and SIFT4t3hWF but starts to increase at few thousands of images for BRISK4t3hW10 and BRISK4t3hWF. This can be explained because these two configurations use BRISK sketches (binary vectors of length 64) and 4 blocks, and each block represents 16 bits out of the whole sketch, leading to a higher probability of collision between the features and therefore processing more candidates per feature.

From this part of the experiment we can conclude that, as we expected, our implementation scales in small contexts for configurations using sketch blocks of 32 bits.

5.2.7 Experiment 7: Real life examples

This test evaluates the performance of the detector on synthetic replicas handmade generated inspired by quotidian image transformation: *memes*, image collages and handmade

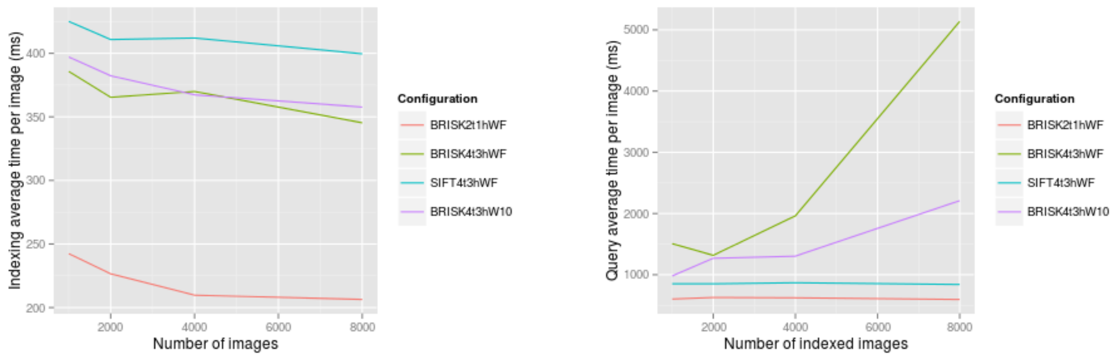


FIGURE 5.9: Average indexing (left) and query (right) time for increasing number of images

drawings. Figure 5.10 are some of these examples. 5 images have been queried against 2,283 indexed images using the configurations resulting from the previous experiments. The conclusions have been:

- No false positives are found
- Meme images are detected as replicas of the original ones.
- Drawing are detected as replica of the original one.
- Only those images that are predominant in the collages are detected as replicas.

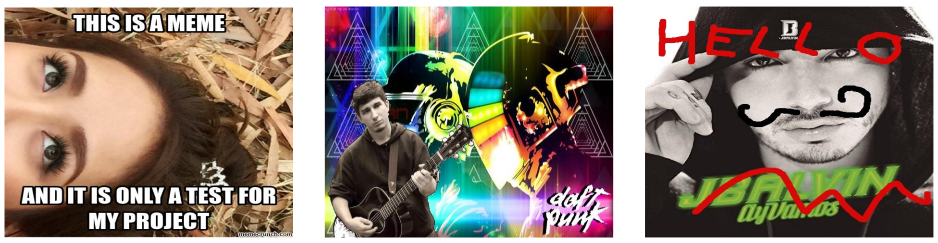


FIGURE 5.10: Examples of quotidian image transformations

5.2.8 Experiment 8: Affine Covariant Regions dataset

The last experiment of the set faces the detector to aggressive transformations in the Affine Covariant Regions dataset. This dataset contains 8 groups of 6 images: 1 source image and 5 replicas from a specific image modification type. These groups are:

- *Bikes* and *Trees*: blurred replicas. Examples of this group can be found in Figure 5.11.

- *Graffiti* and *Wall*: viewpoint changes.
- *Boat* and *Bark*: zoom and rotation changes.
- *Leuven*: light modifications.
- *UBC*: compressed replicas.

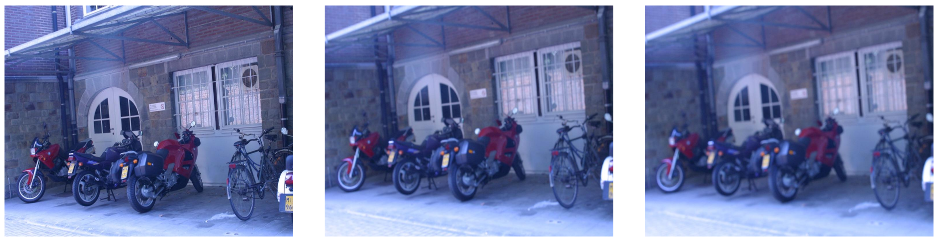


FIGURE 5.11: Examples of images from the Bike group of the Wall dataset

The results in Table 5.7 show that our detector is still far from detect replicas under hard conditions of light and blur but works fine with compression.

Configuration	Bikes (blur)	Trees (blur)	Leuven (light changes)	UBC (compression)
BRISK2t1hWF	0.0	0.0	0.0	0.4
BRISK4t3hW10	0.2	0.0	0.2	0.8
BRISK4t3hWF	0.2	0.0	0.0	0.6
SIFT4t3hWF	0.0	0.0	0.2	0.6

TABLE 5.7: Recall ratio for each group tested from the Affine Covariant Regions dataset

Chapter 6

Conclusions

6.1 Conclusions

We have presented a distributed implementation of image near replica detection system based on the generalization of the work in [17] for both batch and streaming use cases. In the evaluation stage we achieved precisions above 99 % in most of the probes, peak recalls around the 70% and average real-time responses within 3 seconds. We summarize some of the conclusions from this work:

- BRISK keypoint detectors and BRISK descriptors have shown the best balance between time efficiency and good performance.
- Scalability of the system has been partially proved (only in small scale) and would need further work to be fully demonstrated.
- We have observed the incapability of fully distinguish high quality features from the feature set using two measures (entropy and variance) and the benefits of feature filtering speeding up the time responses but penalising the detection rate.
- Sketching sensitivity W has been identified as an important parameter of the implementation that directly influences the true detection rate but generates poor precisions at high values.
- Filtering results by weight have proved to be efficient to improve precision but at the same time penalises a lot the true positive ratio. In some cases, similar results have been obtained by reducing W.
- Memory-based implementation has shown a poor performance due to the overhead introduced by Spark but needs further analysis.

- Our implementation is robust to color based modifications (e.g. channel modification, gamma correction), occlusions, compressions and smoothed replicas, partially robust to resizing and croppings. It also performs well for quotidian image attacks such as *memes* and drawings and partially to collages.
- On the other hand, it is weak to rotations and not valid for Gaussian noise addition and flipping transformations.

6.2 Outcomes

This near replica detection implementation is available on [34]. The specific dataset built for the evaluation is also public and can be accessed here [23]. We are currently working on a publication for the Special Issue on Multimedia Analytics of the Signal Processing Journal [35].

6.3 Scheduling

Through all the year we have been keeping the pace of the starting scheduling. However, due to problems with the development environment, the testing stages took more than the expected and we had to work extra hours during the last month of the project, doubling the time spent on it per day. That means that 60 hours of development were added to the initial bag of hours computed in the planning. The updated costs of the project are presented in Table 6.1 and Table 6.2.

Role	Dedication hours	Hours needed	Income per hour (€)	Initial estimation (€)	Updated cost (€)
Research analyst	405	405	25	10,125	10,125
Developer	345	405	18	6,210	7,290
Total				16,335	17,415

TABLE 6.1: Updated human resources costs

6.4 Future work

There are still a lot of areas of improvement to be further analysed. We gave support to a limited set of features (e.g. SIFT, ORB) and work could be done to integrate new and more distinctive features such as PCA-SIFT.

Type of cost	Estimated (€)	Updated (€)
Human costs	16, 335	17, 415
Hardware costs	521.13	521.13
Software	0	0
Total	16, 856.13	17, 936.13

Table 6.2: Updated total costs

OpenCV can be speeded up using the native CUDA for taking advantage of the GPU processing. Nonetheless, OpenCV showed several problems: lack of Java documentation and instability during the the evaluation phase. By the end of this work we found a library called *OpenImaj* that could replace the role of OpenCV.

Full scalability evaluation still has to be performed and further work could emphasize on deploying the detector on a real cluster. Memory-based implementation also need deeper analysis.

Another idea that appeared during the development process was to serialize similarities between the indexed images and to have a graph-like structure persisting this information.

Finally, further analysis of how this replica detector could be also used for image similarity detection could be performed, using public datasets such as INRIA Holidays dataset [36] or IND dataset [37].

Appendix A

Descriptor values distributions

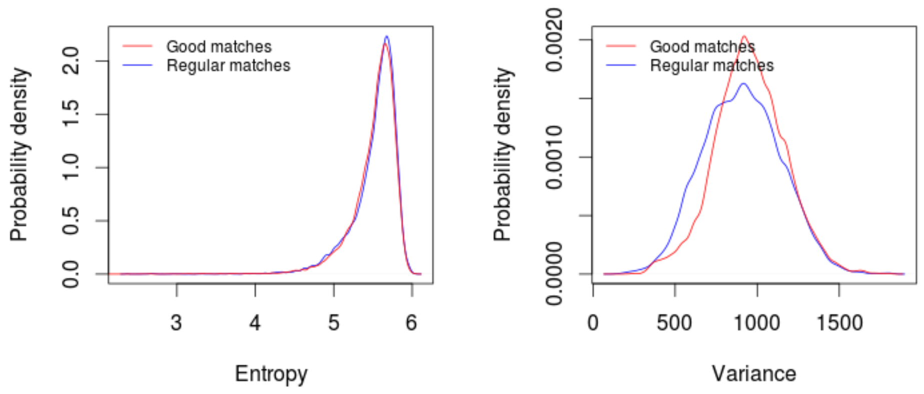


FIGURE A.1: Distribution of SIFT descriptor values and relation between entropy and variance for 320 images

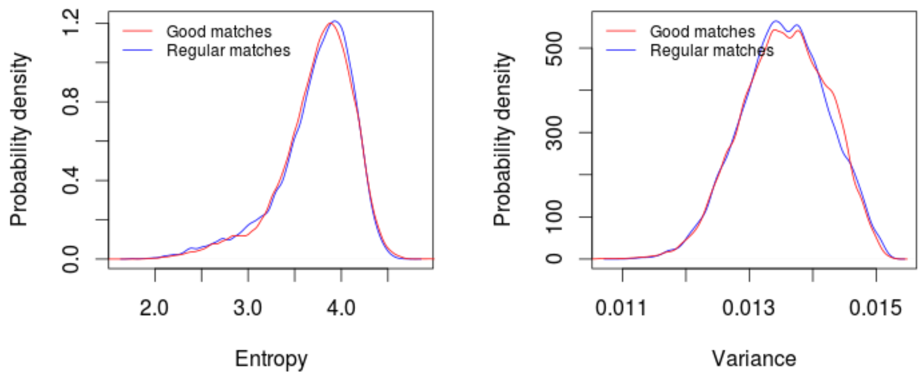


FIGURE A.2: Distribution of SURF descriptor values and relation between entropy and variance for 320 images

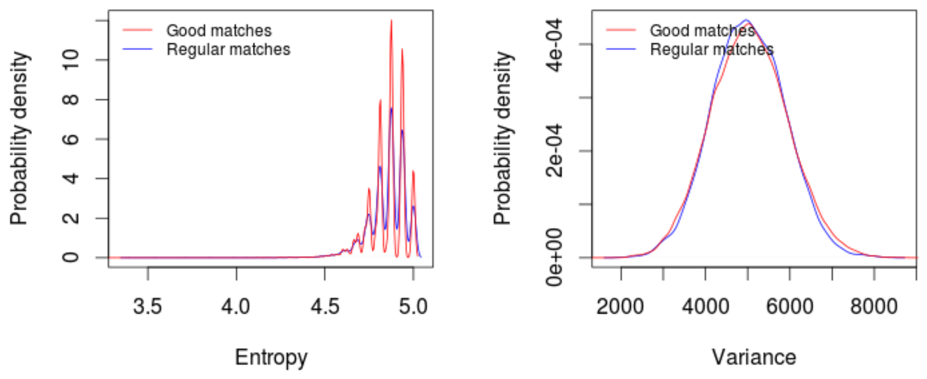


FIGURE A.3: Distribution of ORB descriptor values for 320 images

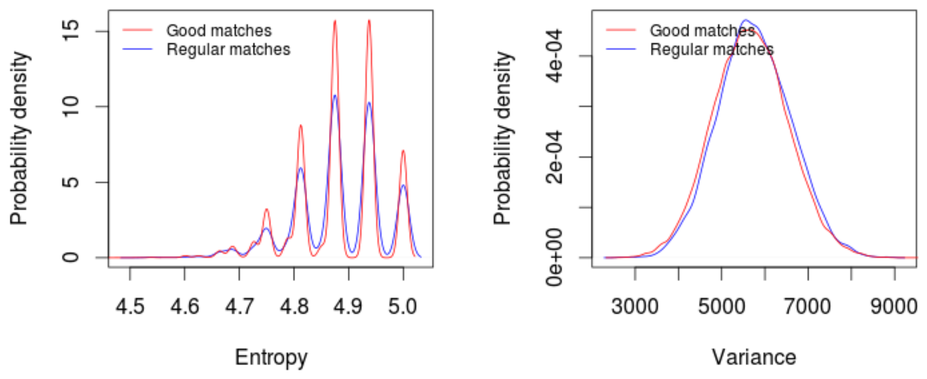


FIGURE A.4: Distribution of BRIEF descriptor values for 320 images

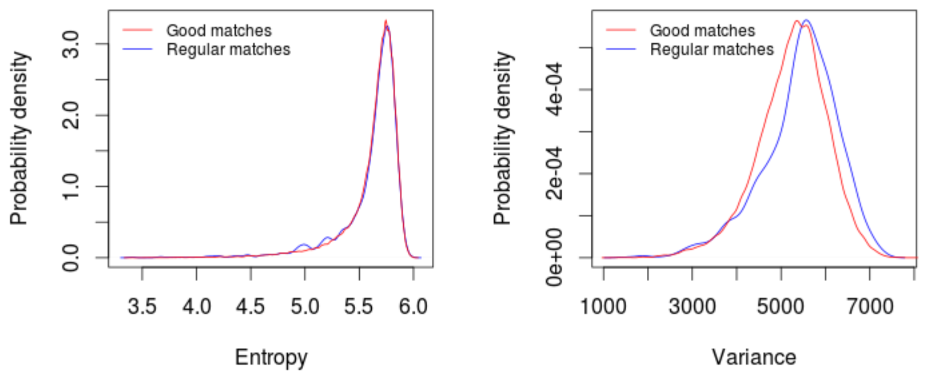


FIGURE A.5: Distribution of FREAK descriptor values for 320 images

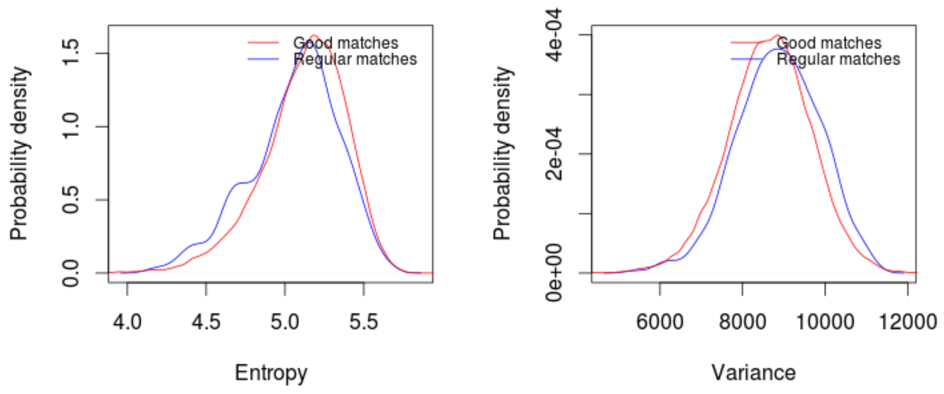


Figure A.6: Distribution of BRISK descriptor values for 320 images

Appendix B

Descriptor distributions

B.1 Spark lineage graphs for batch indexing use case

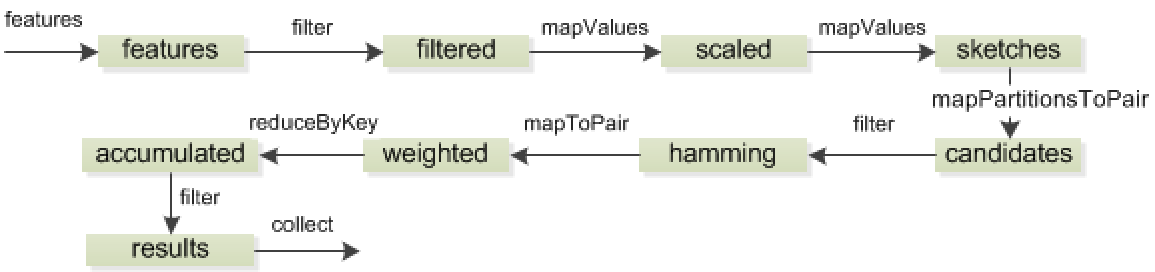


FIGURE B.1: Lineage graph for the disk-based implementation of the batch query use case

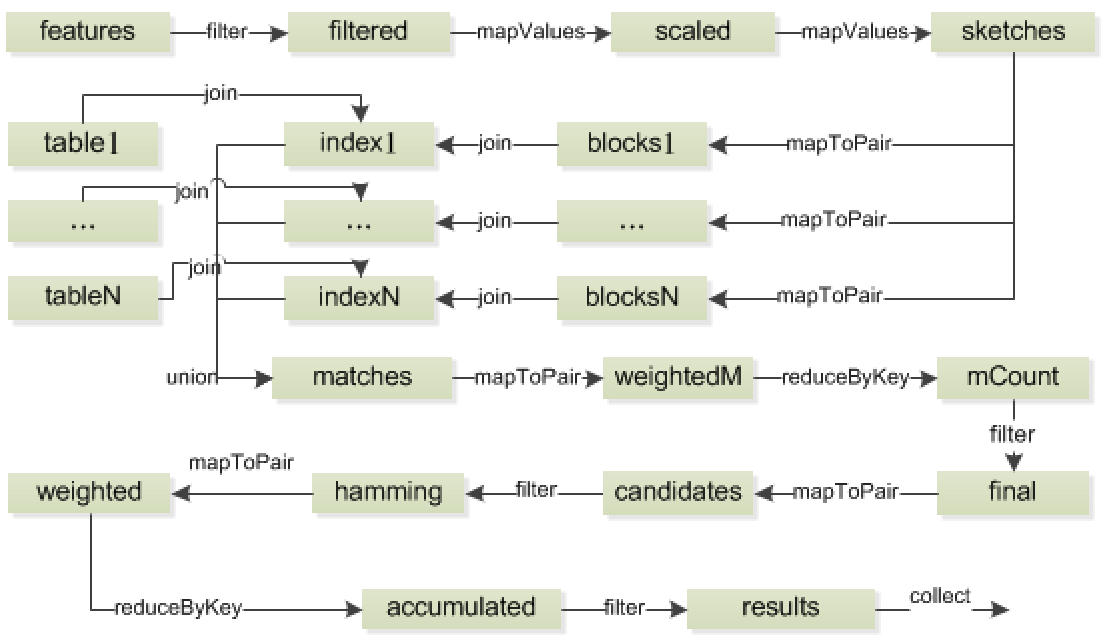


FIGURE B.2: Lineage graph for the memory-based implementation of the batch query use case

B.2 Spark lineage graphs descriptor for batch query use case

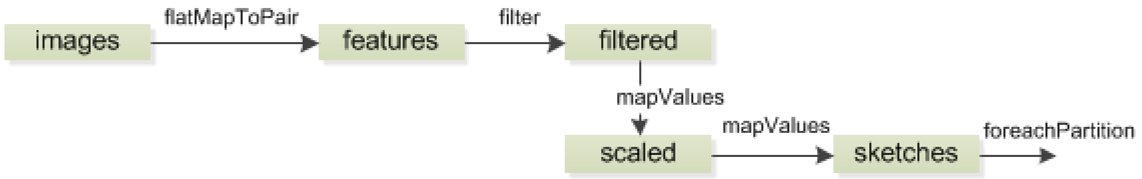


FIGURE B.3: Lineage graph for the disk-based implementation of the batch indexing use case

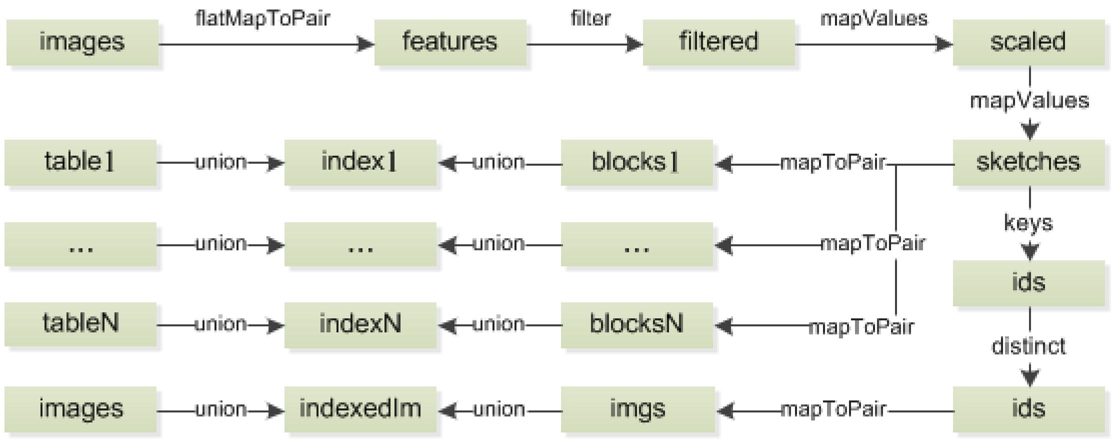


FIGURE B.4: Lineage graph for the memory-based implementation of the batch indexing use case

Bibliography

[1] Tinne Tuytelaars and Krystian Mikolajczyk. Local invariant feature detectors: A survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, July 2008. ISSN 1572-2740. doi: 10.1561/0600000017. URL `http://dx.doi.org/10.1561/0600000017`.

[2] Wei Dong, Moses Charikar, and Kai Li. Asymmetric distance estimation with sketches for similarity search in high-dimensional spaces. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 123–130, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-164-4. doi: 10.1145/1390334.1390358. URL `http://doi.acm.org/10.1145/1390334.1390358`.

[3] Scott Lebernight. Handling big data hbase, part 3. URL `http://java.dzone.com/articles/handling-big-data-hbase-part-3`.

[4] Xin Yang, Qiang Zhu, and Kwang ting Cheng. Near-duplicate detection for images and videos. In *in Proc. ACM LSMMRM Workshop*, pages 73–80, 2009.

[5] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[6] Carlton W. Niblack, Ron Barber, Will Equitz, Myron D. Flickner, Eduardo H. Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. Qbic project: querying images by content, using color, texture, and shape. volume 1908, pages 173–187, 1993.

[7] Chin-Chen Chang and Tzong-Chen Wu. Retrieving the most similar symbolic pictures from pictorial databases. *Information Processing and Management*, 28(5):581 – 588, 1992. ISSN 0306-4573.

[8] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast multiresolution image querying. pages 277–286, 1995.

[9] Edward Y. Chang, James Ze Wang, Chen Li, and Gio Wiederhold. Rime: A replicated image detector for the world-wide web. In *PROC. OF SPIE SYMPOSIUM OF VOICE, VIDEO, AND DATA COMMUNICATIONS*, pages 58–67, 1998.

[10] F. Hartung and M. Kutter. Multimedia watermarking techniques. *Proceedings of the IEEE*, 87(7):1079–1107, Jul 1999. ISSN 0018-9219. doi: 10.1109/5.771066.

[11] Y. Maret, F. Dufaux, and T. Ebrahimi. Image replica detection based on support vector classifier. pages 173–181, 2005.

[12] Zhong Wu, Qifa Ke, M. Isard, and Jian Sun. Bundling features for large scale partial-duplicate web image search. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 25–32, June 2009. doi: 10.1109/CVPR.2009.5206566.

[13] Ondrej Chum, James Philbin, Michael Isard, and Andrew Zisserman. Scalable near identical image and shot detection. In *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*, CIVR '07, pages 549–556, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-733-9. doi: 10.1145/1282280.1282359. URL `http://doi.acm.org/10.1145/1282280.1282359`.

[14] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM. ISBN 0-89791-962-9. doi: 10.1145/276698.276876. URL `http://doi.acm.org/10.1145/276698.276876`.

[15] Yan Ke, Rahul Sukthankar, Larry Huston, Yan Ke, and Rahul Sukthankar. Efficient near-duplicate detection and sub-image retrieval. In *In ACM Multimedia*, pages 869–876, 2004.

[16] Shan Zhou, Jun Li, Junliang Xing, Weiming Hu, and Jinfeng Yang. Non-negative sparse coding using independent multi-codebooks for near-duplicate image detection. In Changyin Sun, Fang Fang, Zhi-Hua Zhou, Wankou Yang, and Zhi-Yong Liu, editors, *Intelligence Science and Big Data Engineering*, volume 8261 of *Lecture Notes in Computer Science*, pages 152–159. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-42056-6. doi: 10.1007/978-3-642-42057-3_20. URL `http://dx.doi.org/10.1007/978-3-642-42057-3_20`.

[17] Wei Dong, Zhe Wang, Moses Charikar, and Kai Li. High-confidence near-duplicate image detection. In *Proceedings of the 2Nd ACM International Conference on Multimedia Retrieval*, ICMR '12, pages 1:1–1:8, New York, NY, USA, 2012. ACM. ISBN

978-1-4503-1329-2. doi: 10.1145/2324796.2324798. URL http://doi.acm.org/10.1145/2324796.2324798.

[18] E. Spyromitros-Xioufis, S. Papadopoulos, I.Y. Kompatsiaris, G. Tsoumakas, and I. Vlahavas. A comprehensive study over vlad and product quantization in large-scale image retrieval. *Multimedia, IEEE Transactions on*, 16(6):1713–1728, Oct 2014. ISSN 1520-9210. doi: 10.1109/TMM.2014.2329648.

[19] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 141–150, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242592. URL http://doi.acm.org/10.1145/1242572.1242592.

[20] libpuzzle. URL http://www.pureftpd.org/project/libpuzzle.

[21] Tineye. URL http://tineye.com/.

[22] multimedia-indexing: A framework for large-scale feature extraction, indexing and retrieval. URL https://github.com/socialsensor/multimedia-indexing.

[23] Upcreplica data set. URL https://github.com/DaniUPC/UPCTwitter-social-data-set.

[24] Features2d + homography to find a known object. URL http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html.

[25] Imagej, . URL http://imagej.nih.gov/ij/.

[26] Marvin project. URL http://marvinproject.sourceforge.net/en/index.html.

[27] Opencv. URL http://opencv.org/.

[28] Apache spark. URL https://spark.apache.org/.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2): 4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816.

[30] Hbase reference guide. URL http://hbase.apache.org/book.html#_namespace.

[31] Imagemagick, . URL http://www.imagemagick.org/.

[32] Affine covariant regions dataset. URL http://www.robots.ox.ac.uk/~vgg/data/data-aff.html.

[33] Twitter streaming api. URL `https://dev.twitter.com/streaming/overview`.

[34] Near replica detection implementation. URL `https://github.com/DaniUPC/image-near-replica-detection`.

[35] Special issue on big data meets multimedia analytics. URL `http://www.journals.elsevier.com/signal-processing/call-for-papers/special-issue-on-big-data-meets-multimedia-analytics/`.

[36] Inria holiday dataset. URL `https://lear.inrialpes.fr/~jegou/data.php`.

[37] Ind dataset (detecting image near-duplicate for linking multimedia content). URL `http://www.ee.columbia.edu/ln/dvmm/researchProjects/FeatureExtraction/NearDuplicateByParts/INDDetection.html`.