Carl-Hauser - Documentation

Vincent FALCONIERI

February 2019 - August 2019

Contents

Chapter 1

Introduction

Goal The goal of this document is to provide an overview of how the carl-hauser library is built, from API to core computation.

Methodology A State of the Art overview had been performed and is available on the project page at `https://github.com/Vincent-CIRCL/carl-hauser`

In the following, we expose :

- ...

Problem Statement [Cevikalp et al.,] states the Image Retrieval problem as "Given a query image, finding and representing (in an ordered manner) the images depicting the same scene or objects in large unordered image collections"

Please, be sure to consider this document is under construction, and it can contain mistakes, structural errors, missing topics .. feel free to ping me if you find such flaw. (Open a PR/Issue/...)

Chapter 2

API

2.1 Calls

In the first version of carl-hauser, following calls are available. Note that this is the minimal API given the problem.

- PING : Allow to quickly check if the API is alive
- ADD PICTURE : Store a picture in the database, that could later be fetched if close to a request picture. Returns the ID of the added picture, as stored in the database.
- REQUEST SIMILAR PICTURE : Performs a request on the database, to fetch similar pictures. Returns a request id to later fetch results.
- GET RESULTS : Given a request id, returns a formatted JSON of results (list of similar pictures ids).

2.2 Design

Flask manages the API endpoints listed previously.

2.3 Ressources

- Flask documentation : `https://www.tutorialspoint.com/flask/flask_http_methods.htm` or `http://flask.pocoo.org/docs/0.12/quickstart/` e.g.
- Extensive tutorial to build REST API with Flask : `https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask`

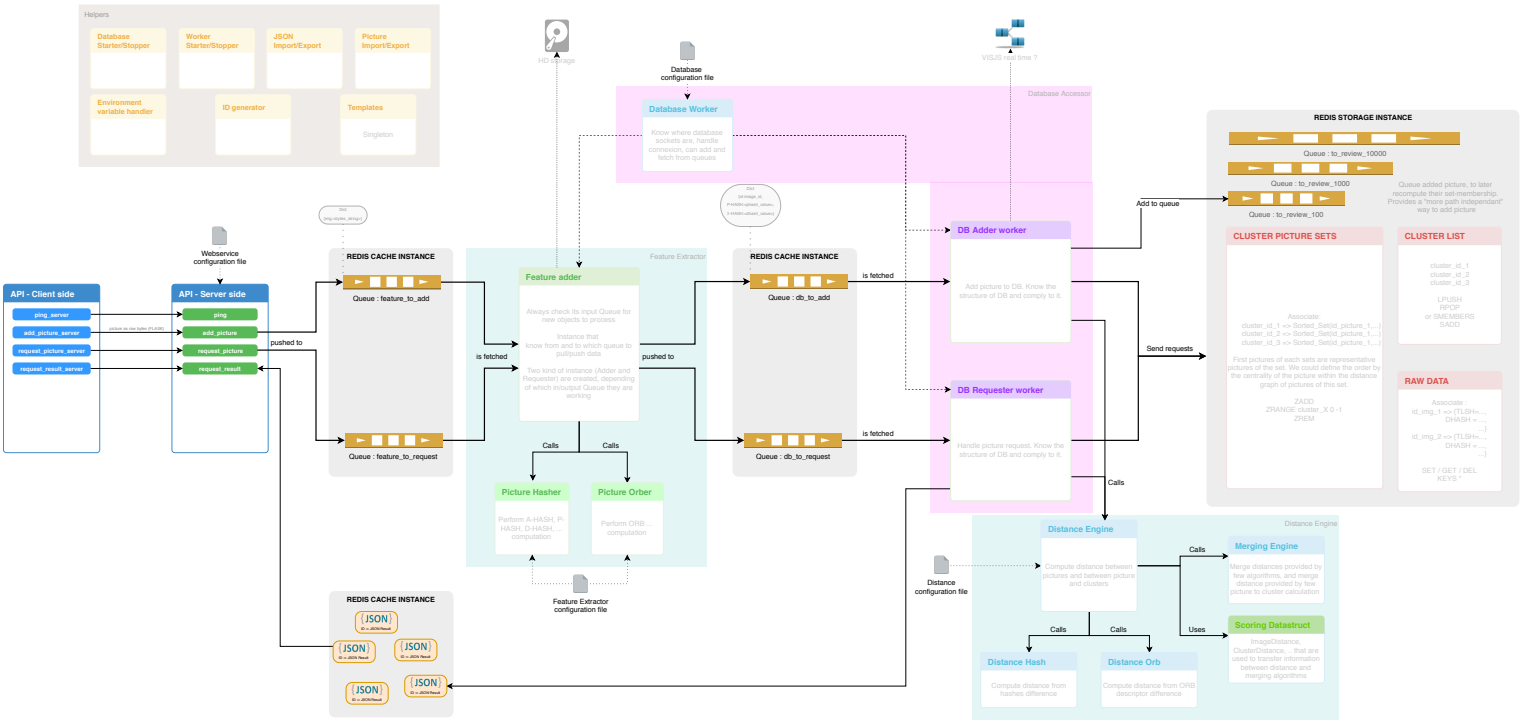
Chapter 3

Components and libraries

Carl-hauser library is split in two : the server side and the client side. The client side is only an accessor of server-side functions.

The application is split in few packages, see Figure **??** :

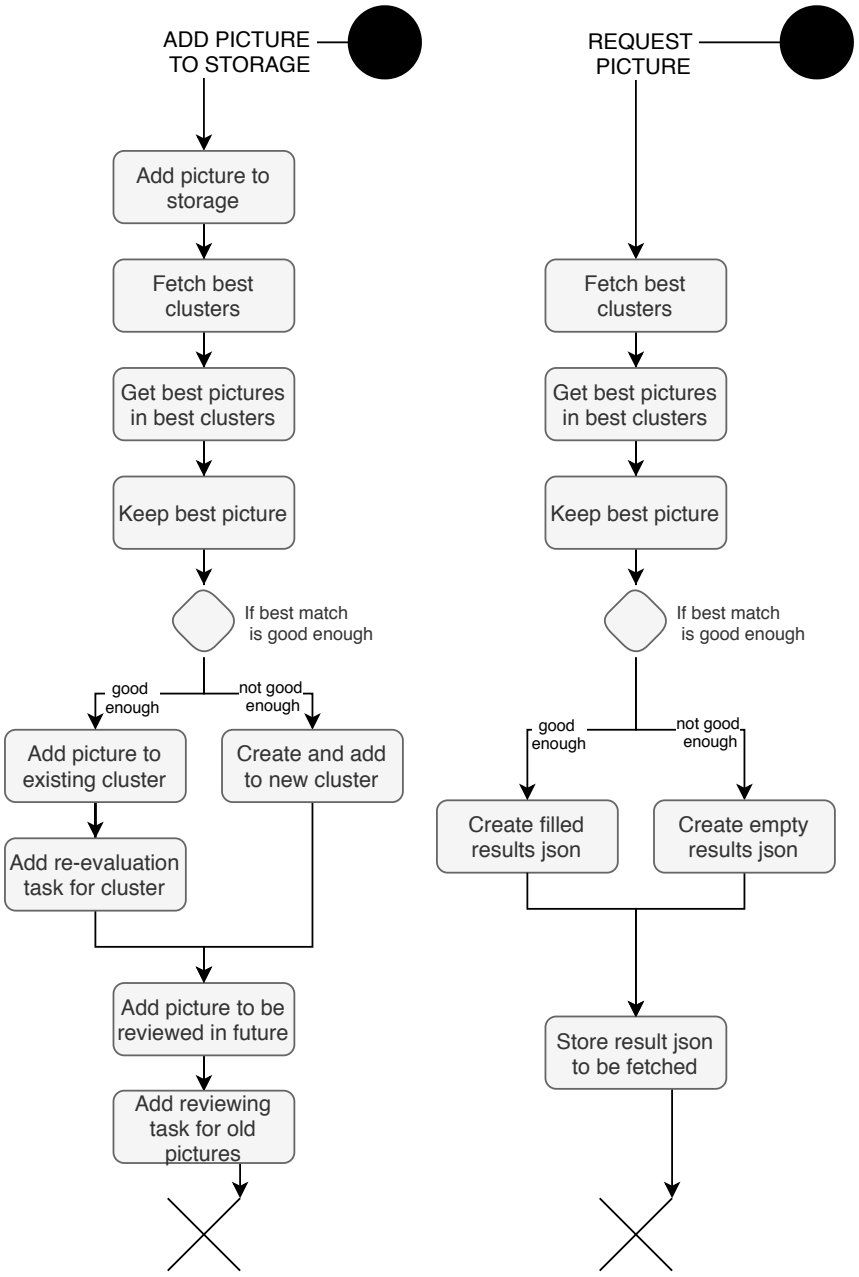
- **API** : The server's endpoint which received client's requests. Mainly running on *Flask*.
- **A Redis cache instance** : it stores current client's requests. For example, a picture to be processed, or a bunch of calculated values to be stored in the redis storage instance.
- **A Redis storage instance** : it stores long-term values, as picture descriptors, or computed datastructure.
- **Feature extractor** : combines feature adder, feature requester, picture hasher, picture orber, ... It computes image representation, as for example, ORB descriptors of a picture, HASHs, ...
- **Database accessor** : combines Database worker, DB Adder, DB Requester, ... It computes and fetch data to and from the redis storage instance. It knows how the database is structured and how to perform request on it.
- **Distance engine** : combines a merging engine, a distance hash, orb distance, scoring datastructure, ... It provides a way to computes the distance between pictures, between picture and clusters, etc. This is where the core computation are performed.



(a) Software architecture

Figure 3.1: Software Components

Let's focus on how a picture is added and how a picture is requested, to the database, see Figure **??**.



(a) Processus to add a picture to database

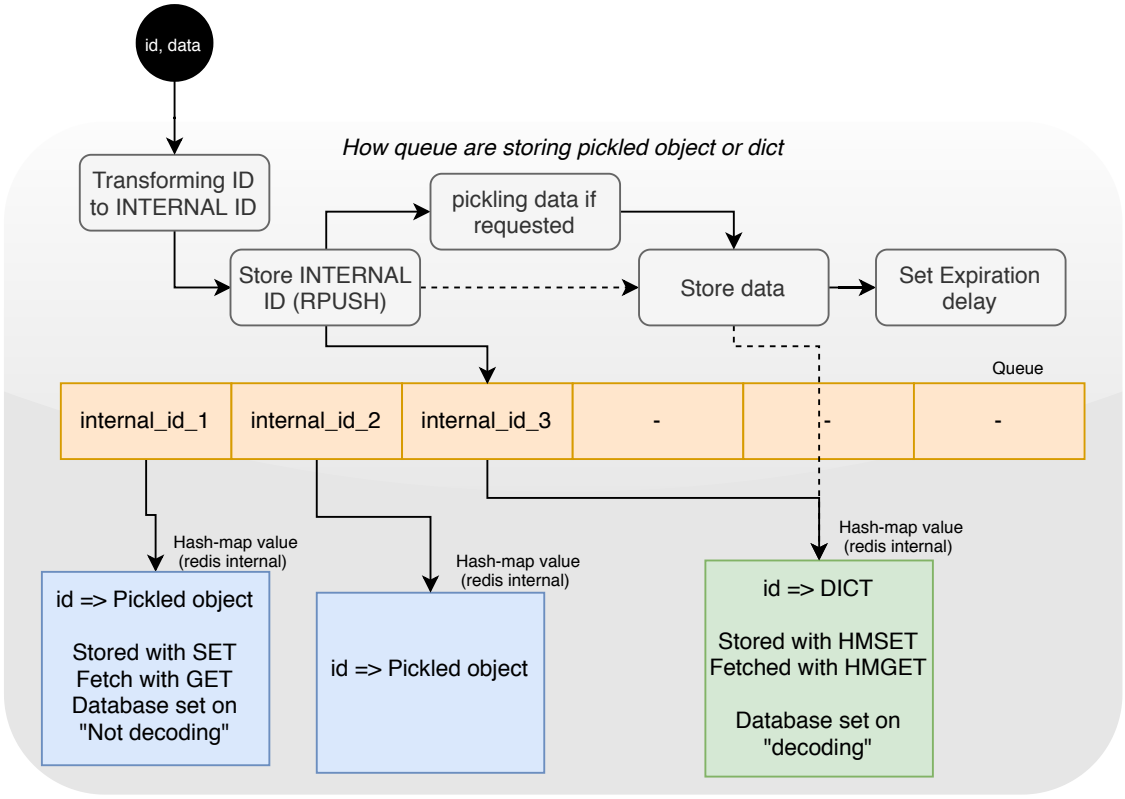
(b) Processus to request a picture's similar pictures

Figure 3.2: Processus

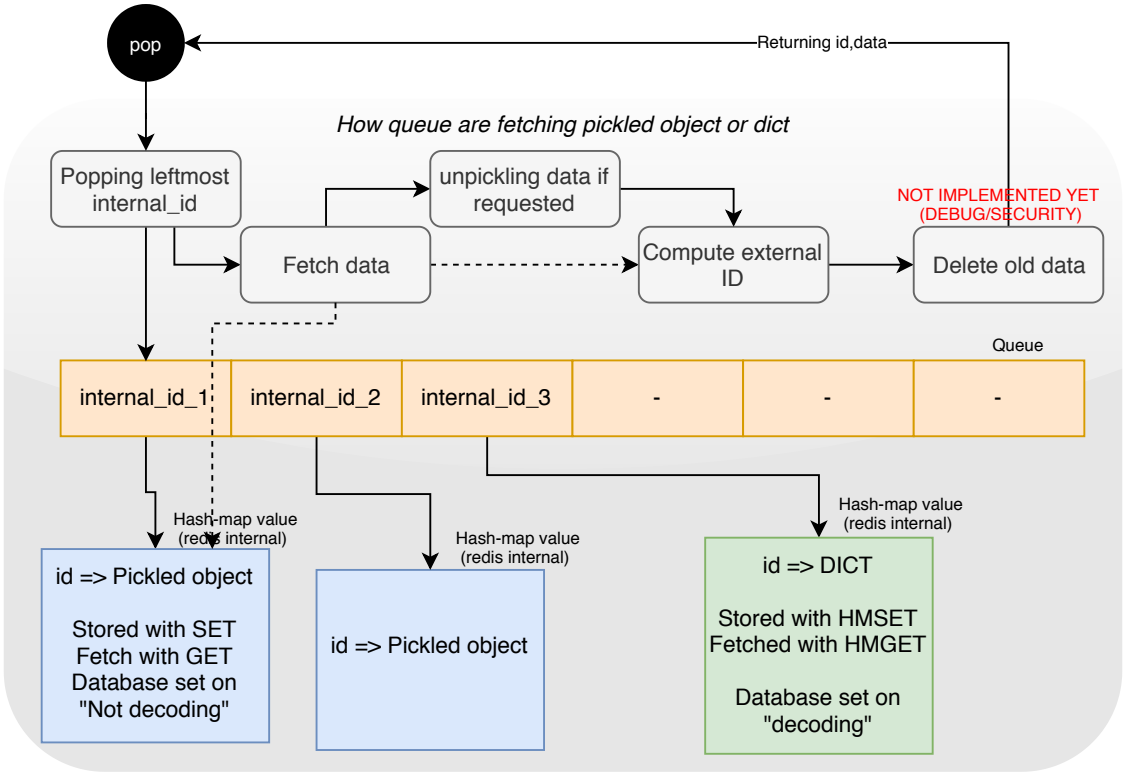
Chapter 4

Parallelism

Queues are heavily used in the application to handle asynchronous tasks. These queues are storing any object in redis cache instance. An easy push/pull interface is implemented, and its internal actions are presented in Figure **??**.



(a) Storing objects in queue



(b) Fetching object from queue

Figure 4.1: Queue management

Chapter 5

Database operations

5.1 Descriptor storage and serialization

We met an important issue to store descriptors of pictures computed by OpenCV, for example. Each interest point in a picture is represented as an object with attributes about this interest point. This data-structure can be called, for example, *cv2.Keypoint*.

When we store string into the database, we could use a HMSET, which is equivalent to storing a python dictionary in redis. HMSET are only handling 1-depth dictionary. However, these objects and the nesting that they imply can't reasonably be stored as a 1-depth dictionary. See Figure 5.1a.

If a HMSET, and so a direct access to any member of the data-structure, is not reasonably possible, then a solution is to "bundle" this data-structure by serializing. We have some choice : JSON, pickle, own datastructure ..

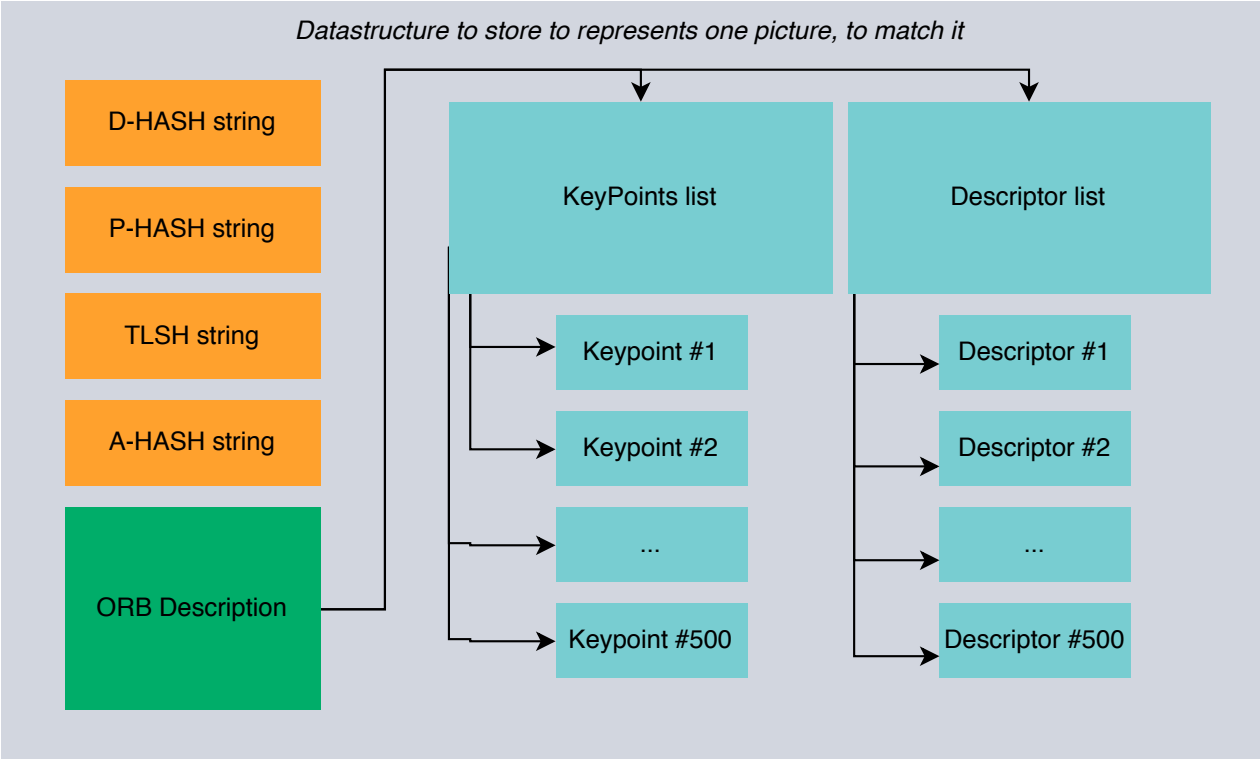
On a performance side, *CPickle* (usable after Python 3.4) seems to provide best results with last version of the protocol. Therefore, we use this implementation. See Figure 5.1b

However, most *OpenCV* objects are not pickle-serializable. More precisely, "pickle stores references to classes and functions, not their definition, because that's way more efficient" [Pyt, a]. Therefore pickle only store data of an object instance, and the type of object it originally was. It does not store class hierarchy nor method definitions. At one point, it "double-checks that it can use that name to load the same class or function back again." [Pyt, a].

For *OpenCV* object, this test fails, because each object name is declared twice : as an object and as a function (due to underlying C++ needs). Therefore, it could not unpickle the data it would pickle, and abort the pickling, leading to an Exception.

A workaround is to register a small method to overwrite the data loading. If such method is set, then Pickle does not perform its sanity-check and so allow the pickling.

Therefore an interesting fix can be built. See Figure 5.1c [Pyt, b]. Then, descriptors "bundle" is pickled and store as a simple key-value pair (*SET/GET* in Redis), retrieved and unpickled. A full bundle for one picture is about *44,5 Ko*.



(a) Representation of one picture

```
pickle          :   0.847938 seconds
cPickle         :   0.810384 seconds
cPickle highest:   0.004283 seconds
json            :   1.769215 seconds
msgpack         :   0.270886 seconds
```

(b) Speed difference for dumping data. Similar results for loading.

```
def patch_Keypoint_pickling(self):
    # Create the bundling between class and arguements to save for Keypoint class
    # See : https://stackoverflow.com/questions/50337569/pickle-exception-for-cv2-boost-when-using-multiprocessing/50394788#50394788
    def _pickle_keypoint(keypoint): # : cv2.KeyPoint
        return cv2.KeyPoint, (
            keypoint.pt[0],
            keypoint.pt[1],
            keypoint.size,
            keypoint.angle,
            keypoint.response,
            keypoint.octave,
            keypoint.class_id,
        )
    # C++ : KeyPoint (float x, float y, float _size, float _angle=-1, float _response=0, int _octave=0, int _class_id=-1)
    # Python: cv2.KeyPoint([x, y, _size[, _angle[, _response[, _octave[, _class_id]]]]]) → <KeyPoint object>

    # Apply the bundling to pickle
    copyreg.pickle(cv2.KeyPoint().__class__, _pickle_keypoint)
```

(c) Speed difference for dumping data. Similar results for loading.

Figure 5.1: Challenge - Datastructure to store in Redis

Chapter 6

Core computation

Matching

Prerequisites : Clusters populated with candidate pictures and an input_picture

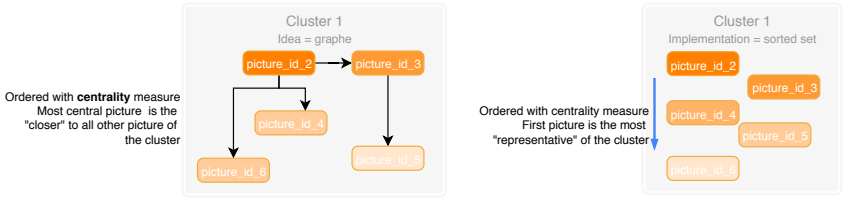
1. Evaluating proximity from input_picture to each cluster
= compute distance between input_picture and *N1* "best representative pictures" of each cluster, and merge the result in one unique "picture_to_cluster" distance.

2. Find best matching clusters
= Keep *N2* best clusters regarding "picture_to_cluster" distance.

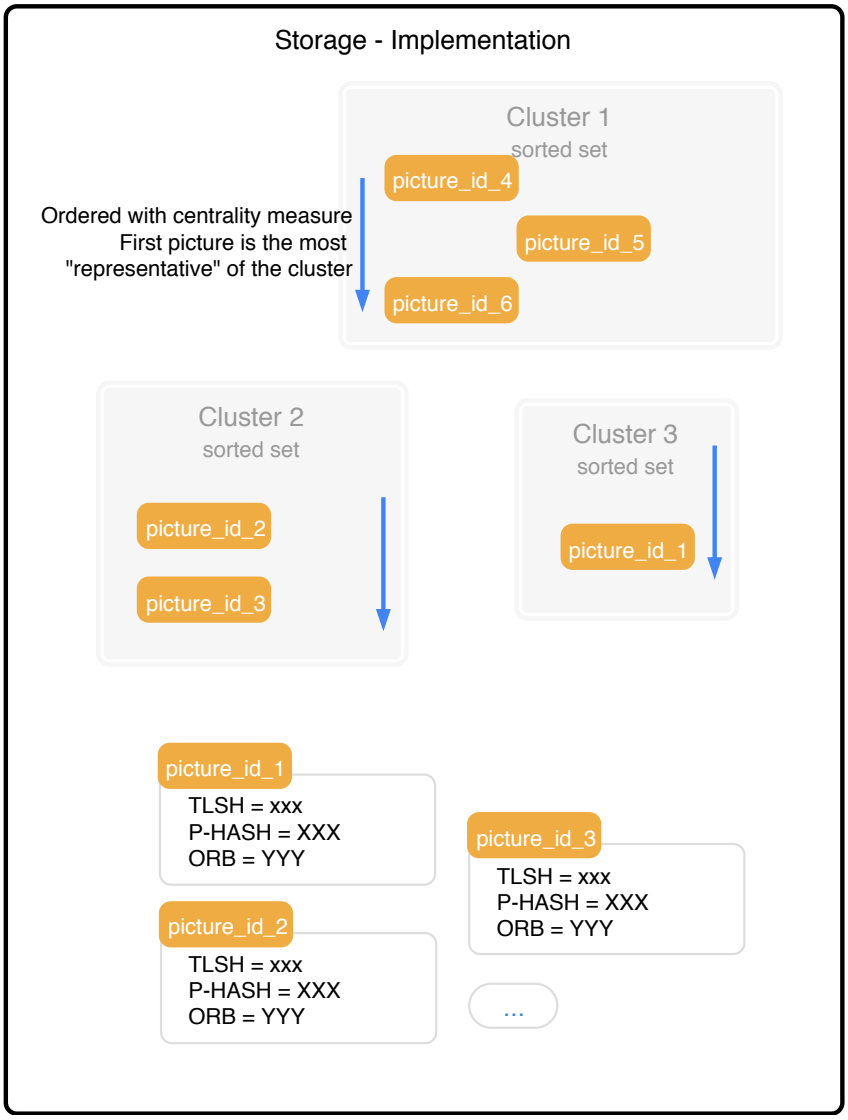
3. Evaluating proximity from input_picture to all candidate picture
= Compute distance between input_picture and each picture of these *N2* clusters.

4. Find best matching pictures
= Keep *N3* best pictures regarding input_picture to candidate picture.

(a) Principle of a similarity search in the redis storage instance



(b) Conceptual view versus Implementation view



(c) Redis storage instance structure

Figure 6.1: Datastructure and search

Chapter 7

Visualization

A visualization tool had been built for the occasion and is available at `https://github.com/Vincent-CIRCL/visjs_classificator`

Bibliography

[Pyt, a] Python - Pickle exception for cv2.Boost when using multiprocessing.

[Pyt, b] Python - Pickling cv2.KeyPoint causes PicklingError - Stack Overflow.

[Cevikalp et al.,] Cevikalp, H., Elmas, M., and Ozkan, S. Large-scale image retrieval using transductive support vector machines. 173:2–12.