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 manage 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

Chapter 4

Parallelism

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 ..

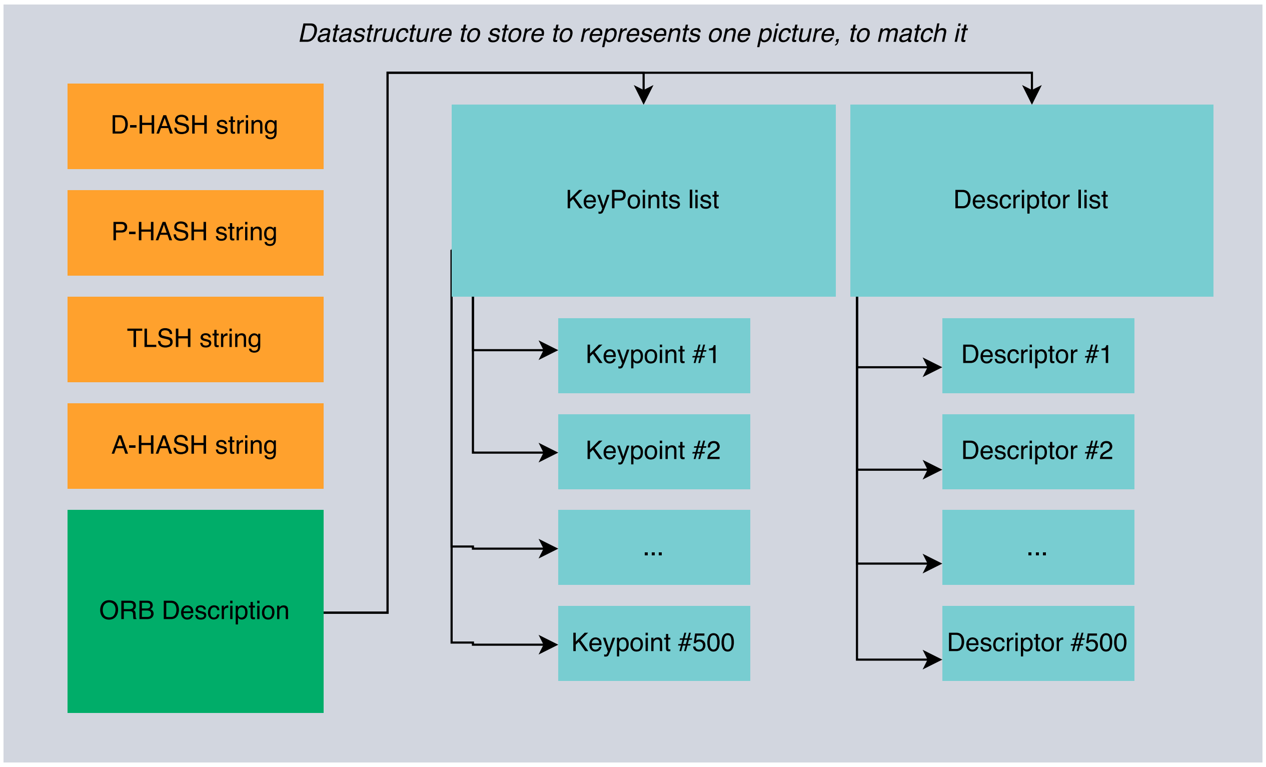
On a performance side, *CPickle* (usable after Python 3.4) seems to provide best results with last version of the protocol. Therefore, we use this implementation. See Figure 5.1b

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" [**?**]. 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." [**?**].

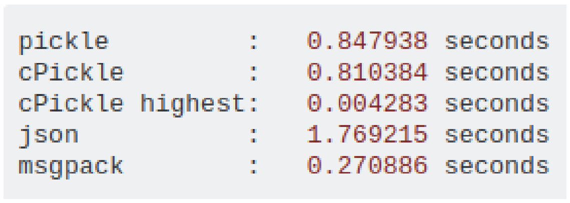
For *OpenCV* object, this test fails, because each object name is declared twice : as an object and as a function (due to underlying C++ needs). Therefore, it could not unpickle the data it would pickle, and abort the pickling, leading to an Exception.

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 [**?**]. 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.*

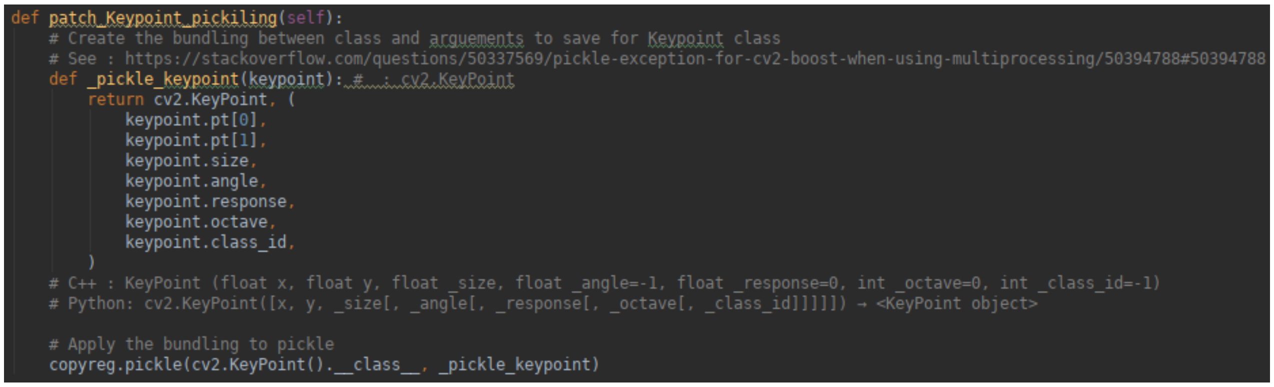


(a) Representation of one picture



```
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_pickiling(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

Chapter 7

Visualization

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

Bibliography

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