Douglas-Quaid - Ex-Carl-Hauser - Documentation

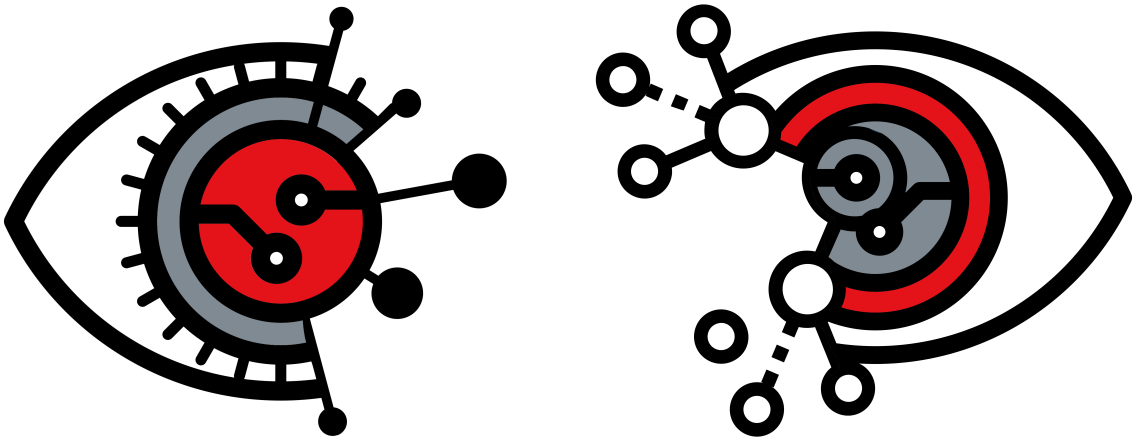
Vincent FALCONIERI

February 2019 - August 2019

Contents

Chapter 1

Introduction



(a) Carl-Hauser (b) Douglas-Quaid

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.
- **WAIT FOR ADD** : Blocking call to wait for an adding to be done
- **REQUEST SIMILAR PICTURE** : Performs a request on the database, to fetch similar pictures. Returns a request id to later fetch results.
- **WAIT FOR REQUEST** : Blocking call to wait for a request to be answered
- **GET RESULTS**: Given a request id, returns a formatted JSON of results (list of similar pictures ids).
- **REQUEST DB**: Request a copy of the database as stored by the Redis storage server

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

Code base

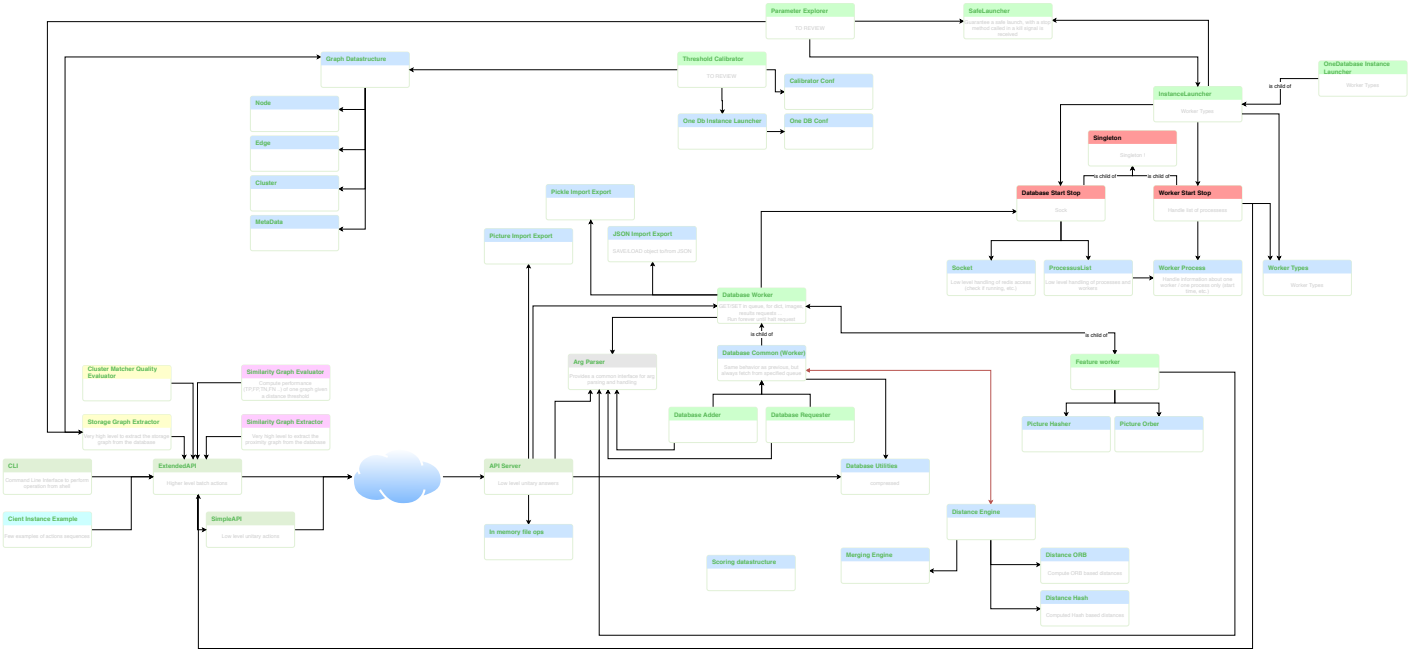


Figure 3.1: CLI, API and extractor structure on client side

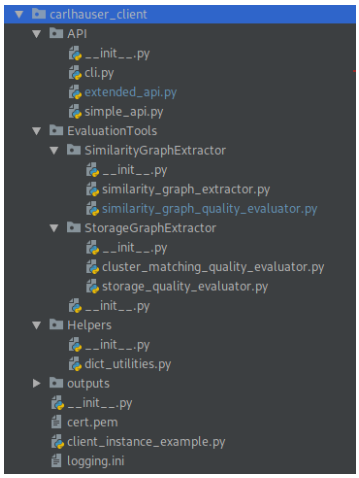
3.1 Client

Client side of the application. With few majors blocs : API (Application programming interface), CLI (Command-line interface), Similarity Graph extractor and Storage graph extractor.

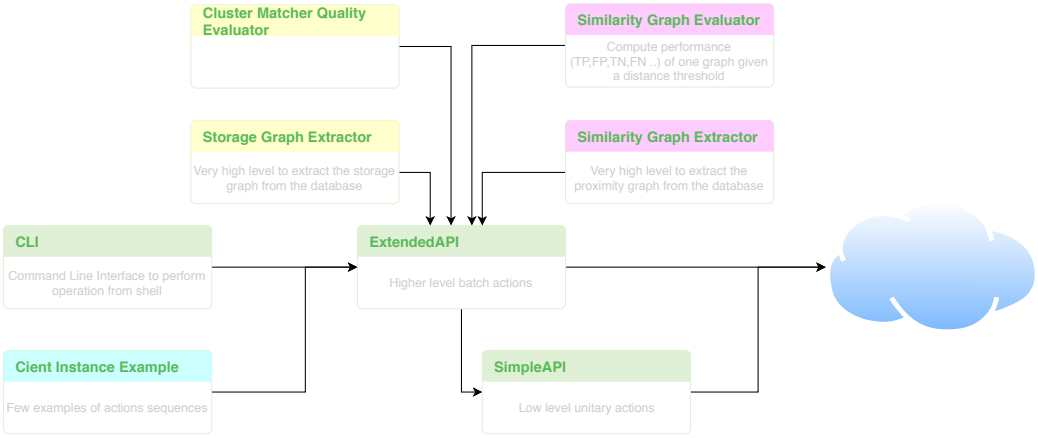
Two denomination are used in the library : similarity and storage graph. Both are different and should not be mistaken. If you are lost, your are new, and you don't know about which graph you should look for, it's most probably the similarity graph. As a newcomer you probably don't even need to know - for now - why the storage graph exist.
Two kinds of graphs can be referenced :

- **Storage graphs** : Graphs and clusters as it is stored in Redis. (Datastructure to improve requests performance)

- **Proximity graphs** : Graphs computed from many requests, that show which picture is close to which picture. This is not the way it is stored in the database, but a condensed view of all requests that can be made on the database.



(a) Client classes



(b) CLI, API and extractor structure on client side

Figure 3.2: Client

3.1.1 API - Client Side

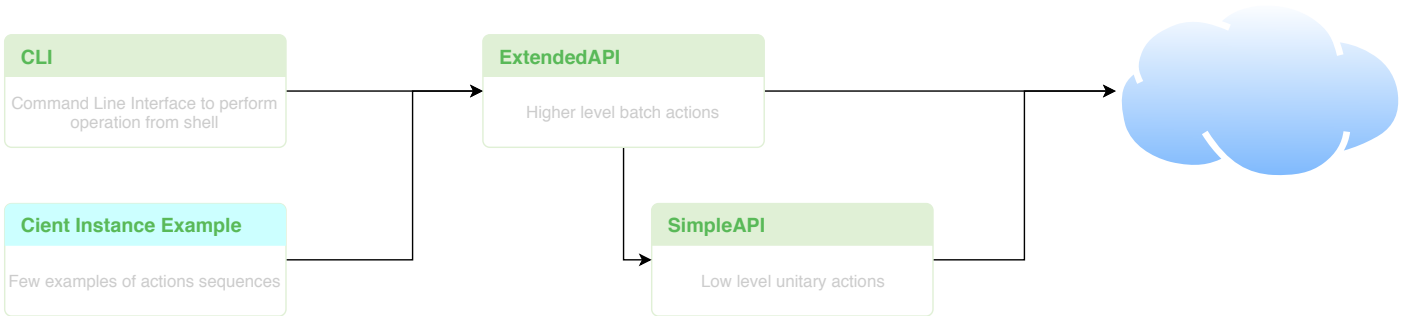


Figure 3.3: API structure on client side

Simple API

Essential elements of the API, low level, to talk with the server. Give small and essentials methods to exchange information with the server-side API.

For example :

- Static method to return an instance of the API (SimpleAPI type)
- Ping the server to check if he is alive with both GET and POST requests.
- Add one picture to the server
- Regularly ask the server if the adding of a picture had been performed
- Ask the server if the adding had been performed
- Request similar picture of one picture to the server
- Fetch the results of a request previously sent.
- Regularly ask the server if the results of the provided (requestid) are ready.
- Ask the server if the results of the provided (requestid) is ready.
- Ask the server a copy of the database

Extended API

Higher level methods to interact with the server-side API.

For example :

- Static method to return an instance of the API (ExtendedAPI type)
- Add a picture to the server, wait for the adding to be performed.
- Add all the pictures of the provided folder to the server (direct children, not recursive)
- Add all the pictures of the provided folder to the server (direct children, not recursive) and wait for each of them to be added (one after the other)
- Add all the pictures of the provided folder to the server (direct children, not recursive) and wait for ALL of them to be added (as a batch). Faster than "add many pictures and wait for each"
- Request similar picture of one picture to the server, wait for an answer.
- Request similar picture of all pictures of the provided folder to the server (direct children, not recursive) wait for each of them (one after the other) and store all the result in one unique list
- Request similar picture of all pictures of the provided folder to the server (direct children, not recursive) wait for them AS A BATCH (wait the last one only), fetch and store all the results in one unique list

- Ask the server a copy of the database, convert it as graph and returns it
- Send pictures of a folder, request all pictures one by one, construct a list of results, revert the mapping to get back pictures names

CLI

Command line interface for client side. Python script that can be launched with arguments (-h to get the help for details) to perform each action without full custom python script.

- Ping the server to check if he is alive.
- Perform the upload of all picture in the provided folder (args.path) and save the mapping (original file name)-(id given by server)
- Request the similar pictures of the provided picture (args.path) if we get an answer before timeout (args.waittime). Translate back the provided ids of the server with the filenames to id mapping saved previously (args.mapfile)
- Dump the database and transmit it to the client, and save it in a file(args.dbfile) Translate back the provided ids of the server with the filenames to id mapping saved previously (args.mapfile). Can duplicate id of picture to their "image" and "shape" attributes. Allows to visualize the database with visjs-classificator (args.copyids)

Client Instance Example

A simple example as provided in the git README page. The below extract is just to understand what you can find. Please refer to the current file to get the last version of this code example.

Listing 3.1: Code example of a client side client

```
@staticmethod
def example():
    # Generate the API access point link to the hardcoded server
    cert = (get_homedir() / "carlhauser_client" / "cert.pem").resolve()
    api = Simple_API(url='https://localhost:5000/', certificate_path=cert)

    # Ping server, and perform uploads
    api.ping_server()
    api.add_one_picture(get_homedir() / "datasets" / "simple_pictures" / "image.jpg")
    # (...)

    # Request a picture matches
    request_id = api.request_similar(get_homedir() / "datasets" / "simple_pictures" / "image.bmp")[1]
    # (...)

    # Wait a bit
    api.poll_until_result_ready(request_id, max_time=60)

    # Retrieve results of the previous request
    api.get_results(request_id)

    # Triggers a DB export of the server as-is, to be displayed with visjsclassificator . Server-side only
        operation.
    api.export_db_server()
```

3.1.2 Similarity Graph

The similarity graph refers to the distance between pictures, and the graph its constitutes, if we extract each picture of the database and its nearest neighbor, as a client. It is a visualization of "what the database says" and can be analyzed to see if the library output is correct or not. Classes on client side with this name are able to extract this graph from the server.

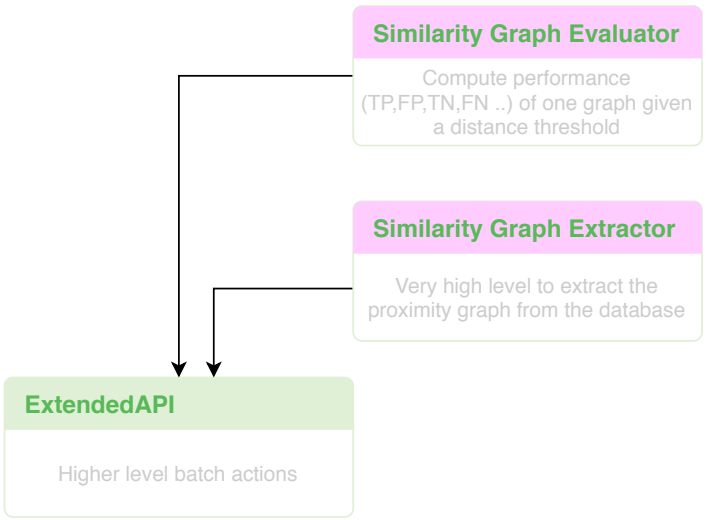


Figure 3.4: Similarity graph structure on client side

Similarity Graph Extractor

Construct a graph from from a folder of pictures, sent to DB and requested one by one. Performs the operation to extract the similarity graph from server.

Similarity Graph Evaluator

Extract a list of performance datastructures from a list of results, previously extracted. Allows to evaluate the Similarity graph extracted from server.

3.1.3 Storage Graph

The storage graph refers to the set of clusters and their content, used in the database, to store pictures. It is a visualization of "how is the database" and can be analyzed to see if the database will be efficient or not. Classes on client side with this name are able to extract this graph from the server.

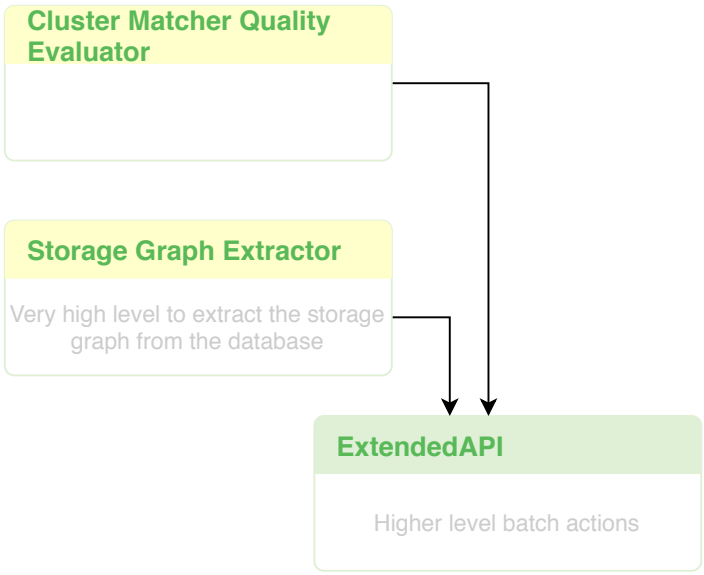


Figure 3.5: Storage graph structure on client side

Storage Graph Extractor

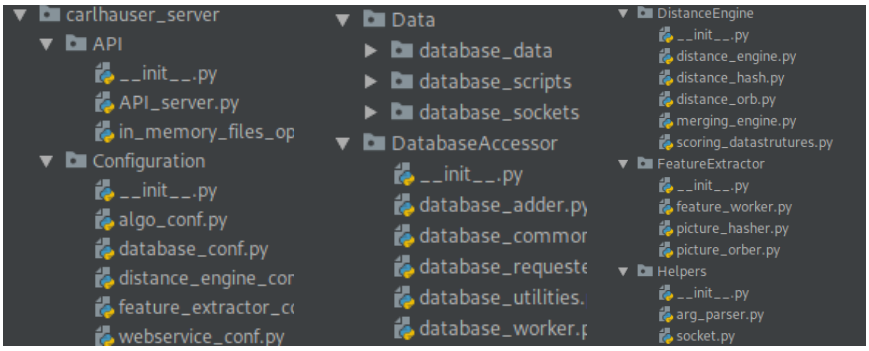
Construct a storage graph from a folder of pictures, sent to DB and a dumped from the DB. Performs the operation to extract the Storage graph from server.

Cluster Matcher Quality Evaluator

Compute statistic about each cluster pairs, about their members. Check the True positive, False positive, etc. rates. Allows to evaluate the Storage graph extracted from server.

3.2 Server

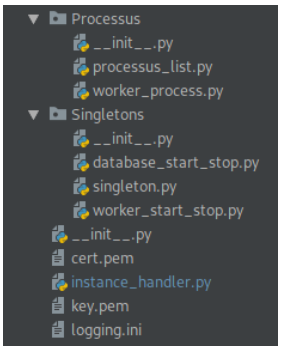
A few main blocks compose the Server. Some classes to handle API calls, some to handle how to interact with the database, some to compute and extract features out of pictures, some to extract distances out of features, some to handle workers and processes, some to handle export and import ... Each is described in the following.



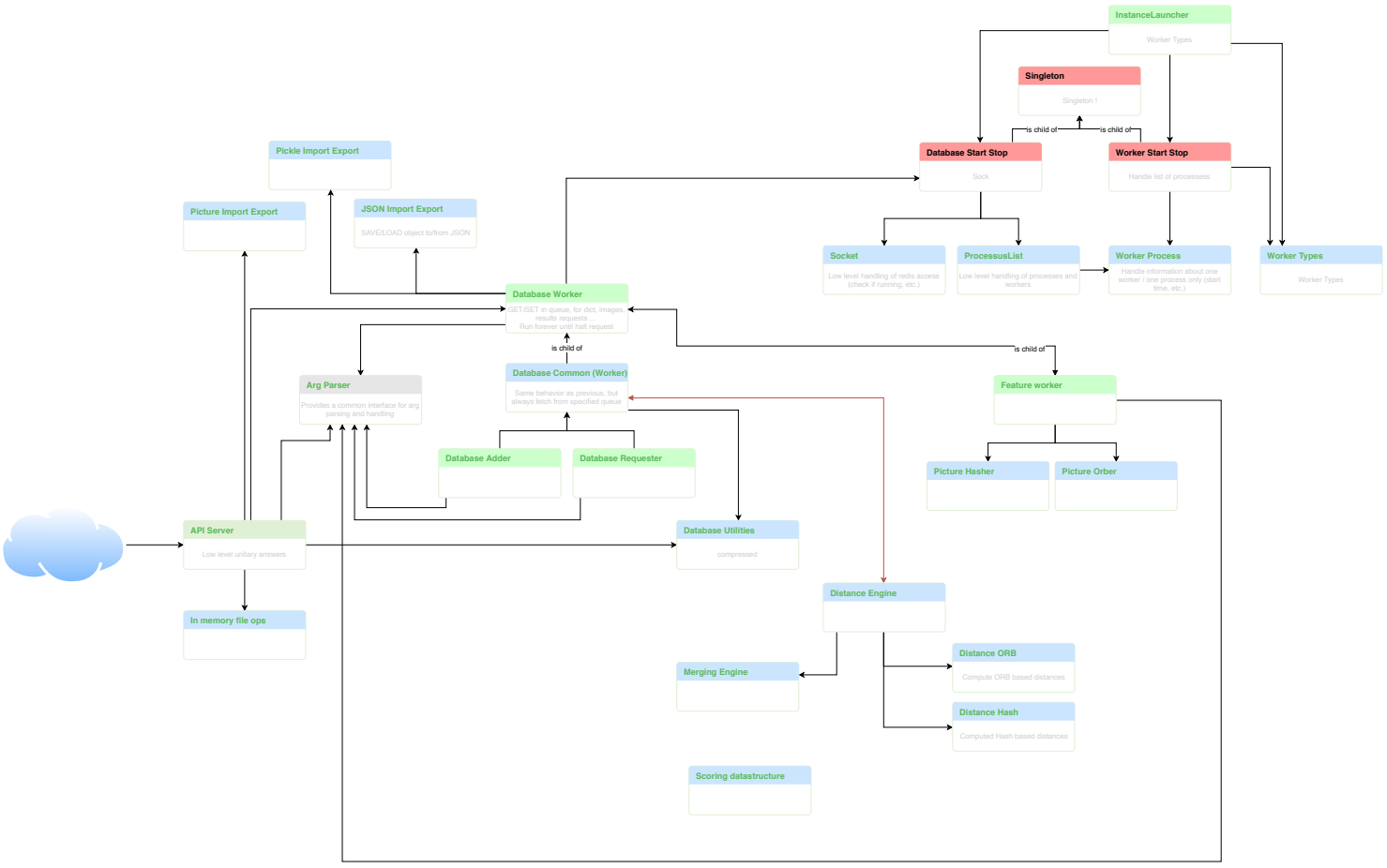
(a) Server classes

(b) Server classes

(c) Server classes



(d) Server classes



(e) Classes structure on server side

Figure 3.6: Server

3.2.1 API - Server Side

This set of classes handle the reception and the processing of user's requests. It answers to API call.

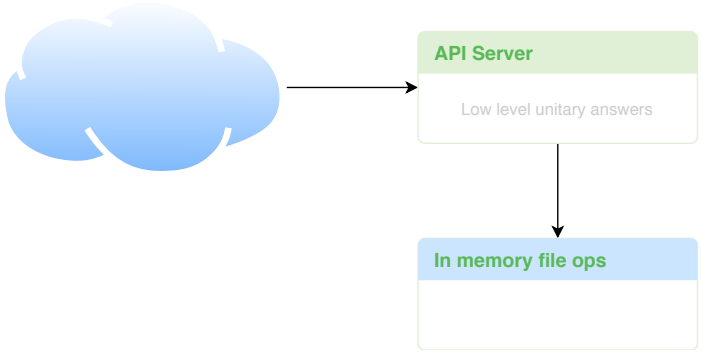


Figure 3.7: API classes on Server side

API

Define and instanciates server side endpoints for each API call. Uses a Flask server. Handle SSL. Can interact and call relevant functions elsewhere on the server-side.

For example :

- Add all endpoints = all callable URL of the API, and link them to the good functions
- Add one endpoint to the flask application. Accept GET, POST and PUT.
- Handle a ping request on server side. Creates a pong answer
- Handle an adding of a picture request on server side.
- Handle a check of emptiness on the pipelines (list of queues)
- Handle a request of one picture, to return a list of similar pictures
- Handle a retrieval of results
- Handle a check of a request readiness
- Handle a check of a add readiness
- Handle a dump of the database request
- Hash the picture, generate a BMP file, enqueue it, return the result json with relevant values

In Memory Files Operations

Allow to perform in memory operations on fetched files.

For example :

- Get SHA1 hash of the file, directly in memory
- Get the BMP version of the file, directly in memory

3.2.2 Database Workers

Green classes are standalone executable : many instances in parallel can be launched, even on a hot-running server. These launch-able classes uses ArgParser to manage in an unified way the way to set launching arguments.

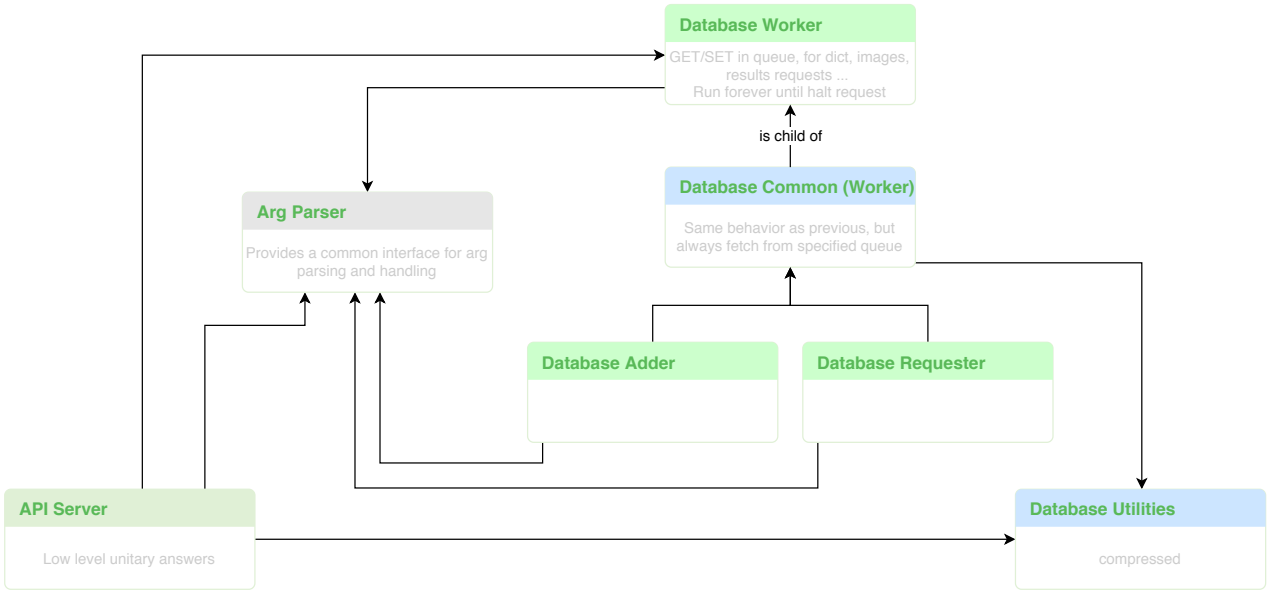


Figure 3.8: Database worker classes

ArgParser

Allows to manage in an unified way standard arguments for all executables's server's classes.

For example :

- Add argument to the argument parser (Configuration file, etc.)
- Parse args to configuration files, if they exist.

Database Worker

Represents the parent class to access databases. Has access to storage and cache databases, with decoded and undecoded (raw) channels. Know where the databases sockets are, and give the standard common methods to interact with the database.

For example :

- Push data to a specified queue, with a specific id. Wrapper to handle queuing of id(s) and separated storage of data linked to this id(s). Transparent way to push data to a queue
- Fetch data from a specified queue. Wrapper to handle queuing of id(s) and separated storage of data linked to this id(s). Transparent way to pull data from a queue
- Set a dict of values, pickled or not, to a key
- Retrieve a dict of values, pickled or not, from a key
- Store images as pickled dict in the provided storage
- Retrieve images as pickled dict in the provided storage
- Store results as pickled dict in the provided storage with a configuration-defined expiration time

- Retrieve results as pickled dict in the provided storage
- Check if all queues (TO ADD, TO REQUEST, etc.) are empty
- Check if the specified Queue in the specified storage is empty
- Print all keys of the storage
- Check if a halt had been requested
- Run indefinitely except if the worker have received a stop signal. Fetch from database queue and call a process function on it.
- Method to overwrite to specify the worker. Called each time something is fetched from queue. Has to be defined in the children of this class.

Database Common

Factorize common part between Database Adder and database Requester. No other purposes than prevent code duplication.

For example :

- Extract the list of top matching pictures and the list of top matching clusters from a result dict.
- Check if a match is good enough (at least one match, not None ..)

Database Adder

Worker (background running process) that add picture's set of features to the database.

For example :

- Add picture to storage, evaluate near-similar pictures, choose a good cluster and add the picture to this cluster.
- Re-evaluate the representative picture of the cluster ¡cluster id¿, knowing or not, that the last added and non evaluated picture of the cluster is ¡fetched id¿
- Returns centrality of a picture within a list of other pictures.

Database Requester

Worker (background running process) that compare picture's set of features to the ones stored in the database.

For example :

- Perform calculation on the database to fetch near pictures. Does not add the requested picture to the database

Database Utilities

Set of utilities functions, to access the database in an unified way. Can add, get, remove clusters, pictures, edges, ...

For example :

- Get the list of pictures associated with a given cluster
- Store a cluster's name in the list of cluster's names
- Remove a cluster's name of the list of cluster's names
- Add a picture to a cluster (already existing)

- Add a picture to a cluster (freshly created)
- Update the set "ranking" value of a picture into a cluster
- Generate cluster id
- Retrieve the set of pictures ids from a cluster
- Generate the key of the set of pictures
- Add the picture to be reviewed in few time (100 queue, 1000 queue, ...)
- Export the current state of the database as a graph datastructure. This represents the storage graph of the server.

3.2.3 Distance Engine

The first and main entry point is the Distance Engine, which handle distance calculation between set of features.

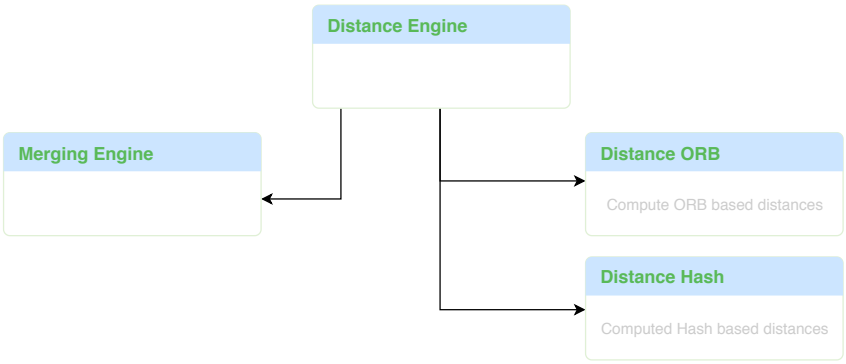


Figure 3.9: Distance Engine classes

Distance Engine

Handle distance computation between pictures, and more precisely between their features. Handles the distance between cluster and picture, between picture and picture, etc.

For example :

- Compute a list of distance and decision from two images representations, with activated (conf) algorithms.
- Compare two pictures by their features, to return a one unique distance and a one unique decision. All activated algorithms are involved.
- Check if a match is good enough. Compare the distance between pictures with the threshold between clusters. Usable for storage graph.
- Evaluate the similarity between the given picture and each cluster's representative picture. Returns a list of the N closest clusters
- Evaluate the similarity between the given picture and all pictures of cluster list. Returns a list of the N closest pictures and cluster, with distance
- Go through N first picture of given cluster, and test their distance to given image. Merge the results into one unified distance

Distance ORB

Handles distance computation on the specific case of ORB features.

For example :

- Distance between two provided pictures (dicts) with ORB methods
- Add results to answer dict, depending on the algorithm name we want to compute
- Compute hash difference for ORB
- Keep only "good" matches. Filter on distance, hardcoded at 64 (known as best ratio TP/FP)
- Compute the distance from the list of all matches and the list of good matches.
- From a distance between orb distance, gives a decision : is it a match or not ? Or maybe ?

Distance Hash

Handles distance computation on the specific case of Hashes.

For example :

- Distance between two provided pictures (dicts) with fuzzy hashing methods
- Add results to answer dict, depending on the algorithm name we want to compute
- Compute hash difference for A-HASH, P-HASH, etc.
- Compute hash difference for TLSH only
- From a distance between hashes, gives a decision : is it a match or not ? Or maybe ?

Merging Engine

Handles the merging of distances. For example, handles the way to merge distances of pictures to picture, to a distance of cluster to picture. Also handles the merging of algorithms distances together into one unique distance. Does the same for decisions.

For example :

- Merge distances from many algorithms outputs
- Merge decisions from many algorithms outputs
- Get a max out of distance list
- Get the most prevalent decision
- Get the weighted per algorithm type mean of distances
- Output a decision if more than 80
- Output the most prevalent decision type
- Output the most prevalent decision type, weighted by algorithm type
- Utility function to get the most prevalent decision out of a list of decision
- Output only one decision out of a list of decision. Start by most truthworthy algorithms. If unsure, go down in the algorithm trust hierarchy, until one algorithm is sure of its decision.
- Utility function to compute the number of decision out of a list of decision. Can be weighted or not.

Scoring Datastructure

Provides many classes to handle matches, and so threshold, distances, decision, source and target id, in an unified way.

For example :

- Provides a Decision type (YES/MAYBE/NO) list, which are possible answers to the question "Are these pictures the same ?"
- Provides an Algorithm match, which is a datastructure to handle the returned values of a "distance evaluation" between two hashs, Orb ...
- Provides a Cluster match, which is a datastructures to handle a match, a distance/decision, from a picture to a cluster.
- Provides an Image match, which is a datastructures to handle a match, a distance/decision, from a picture to another picture.
- Provides a TOPN datastructure, which is a datastructure to handle the top N matches of some type.

3.2.4 Feature Worker

The main entry point is the Feature Worker, which handles the transformation of a picture into a set of features.

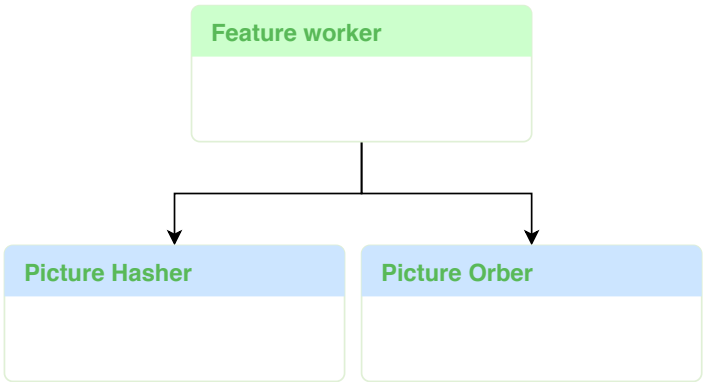


Figure 3.10: Feature Worker classes

Feature Worker

Transform a picture into a list of features, depending on activated algorithms. Heritate from the database accessor, and so has already built in access to cache, storage .. but overwriting the existing fetching function, to prevent PICKLING when fetching

For example :

- From a fetch id and a fetch asssociated dict, compute the hashes and orbing of current picture and add it to next queue

Picture Hasher

Handles the creation of hash-like features.

For example :

- Hash a picture and returns the hash value
- Check if the provided hash is none, if so, provide a "null version" of the hash.

Picture Orber

Handles the creation of orb-like features.

For example :

- Orb a picture and returns the orb value

3.2.5 Instance Launcher

Classes to handle a full instance launch, in a safe way, with a safe stop even with KeyBoard Interruption or signal received.

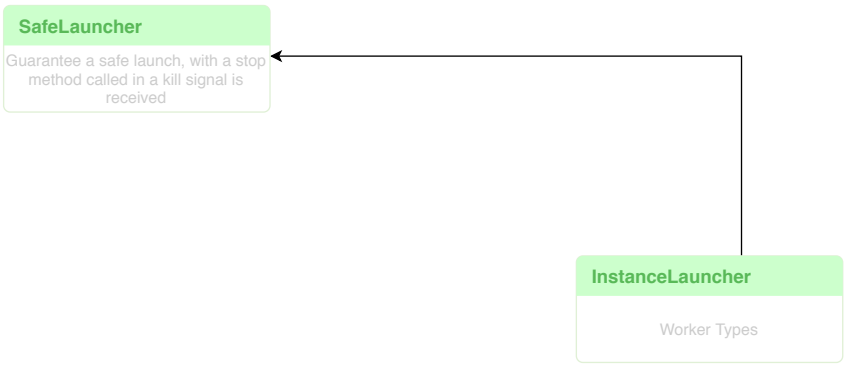


Figure 3.11: Instance Launcher classes

Safe Launcher

Handle a class launch with fallback method and clean exit even if brutally stopped.
For example :

- Launch provided instance with specified launching method
- Stop provided instance with specified stopping method
- Modify and restore the original signal handler as otherwise evil things will happen

Instance Launcher

Handle a server instance
For example :

- Launch a full server : Databases, workers (webservice to adder), wait for startup and check status of everything
- Stop everything. Webservice, workers, and database
- Create a Singleton instance of DB handler if none is present
- Create a Singleton instance of DB handler if none is present
- Start the database, with all redis and wait until running
- Stop the database, with all redis, and wait until stopped
- Flush the database. Use at your own risks ! You will loose data !
- Start adder workers (which add pictures from queue to the database)
- Start requester workers (which request pictures from queue to the database)
- Start feature workers (which compute features from pictures from queueto an other queue)
- Start API webservice (Flask) to handle client requests
- Stop the webservice which handle clients requests
- Check workers status
- Stop workers and wait for their shutdown
- Remove the halt key from the database.
- Flush all workers. Kill them all and forget it. Goes away from the past.

3.2.6 Singleton and Process Management

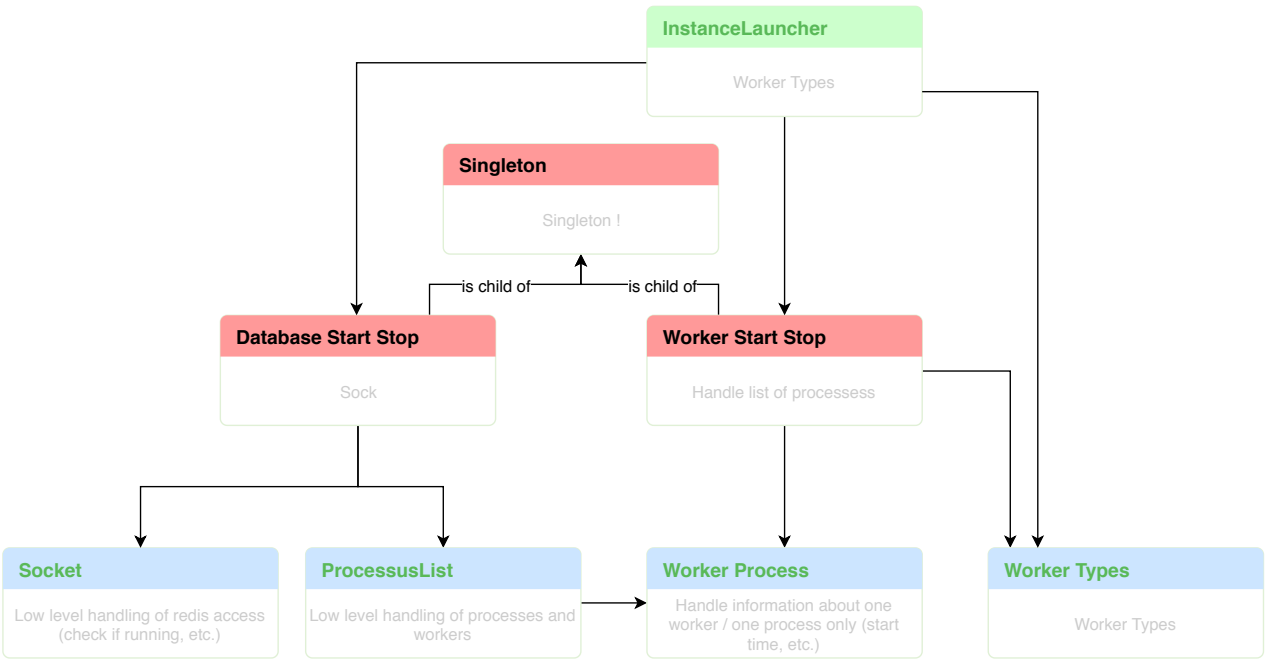


Figure 3.12: Singleton and Process Management classes

Singleton

Is a Singleton metaclass that handle the creation of one unique instance of any of its children.

Database Start Stop

Singleton class that handle database access. His behavior is largely influences by database configuration file. Handle launch, verification and stop of databases. Uses a list of socket to keep track of databases.
For example :

- Launch the cache, storage and test instance of redis
- Stop the cache, storage and test instance of redis
- Flush the cache, storage and test instance of redis
- Blocking function that wait until all database are launched.
- Blocking function that wait until all database are stopped.
- Remove HALT key in all redis instance. Worker then can't stop themselves on launch
- Post a HALT key in all redis instance. Worker should react "quickly" and stop themselves

Socket

Socket handler to a redis database. Handle one connection to one redis database, via its socket.
For example :

- Create a socket handler which know scripts path and socket path for a specific database
- Launch the database behind the given socket

- Stop the database behind the given socket
- Flush the database behind the given socket
- Get an access to the redis database behind the given socket
- Check if the database is running behind the given socket. Peform only one test
- Wait until the database behind the given socket is launched (= answer to a ping) Put timeout -1 if you don't want to function to timeout
- Wait until the database is stopped (= does not answer to a ping) Put timeout -1 if you don't want to function to timeout
- Remove "Halt" key in database to prevent workers to stop on launch
- Put "Halt" key in database to notify workers to stop
- Run a given script file (bash)

Worker Start Stop

Singleton class that handle workers and processes management. Defines worker types as an enumeration. Handle launch, verification and stop of workers. Uses a list of processes list to keep track of workers.

For example :

- Returns the workers list of the asked worker type
- Returns all lists of workers lists
- Add ¡nb¿ workers to the ¡list to add¿ workers lists, by launching ¡worker path¿ as a subprocess and giving ¡XX conf¿ as parameters (many at once is possible)
- Stop the worker of the provided type
- Send signal to all processes to stop (via redis database). Wait while processes are still running, in the limit of a maximal amount of time. Send back a boolean to notify if all workers had been stopped, or not.
- Returns the list of currently running workers (all types)
- Check if workers are alive, and return True if all workers are down
- Kill each worker and then empty lists. Very violent. Use at your own risks :)

ProcessusList

Handle a list of process of one type. Keep track of these processes. Is using a list of Worker processes.

For example :

- Flush the list of processes
- Remove all the proccesses that are not detected as curently running.
- Try to kill all workers of the given list, waiting ¡grace time¿ for each processus to finish (2 sec per default)
- Provide a sublist of the list of processus, which are currently running
- Check if workers are alive, and return True if all worker are down
- Wait until all the workers are stopped (= terminated) Put timeout -1 if you don't want to function to timeout

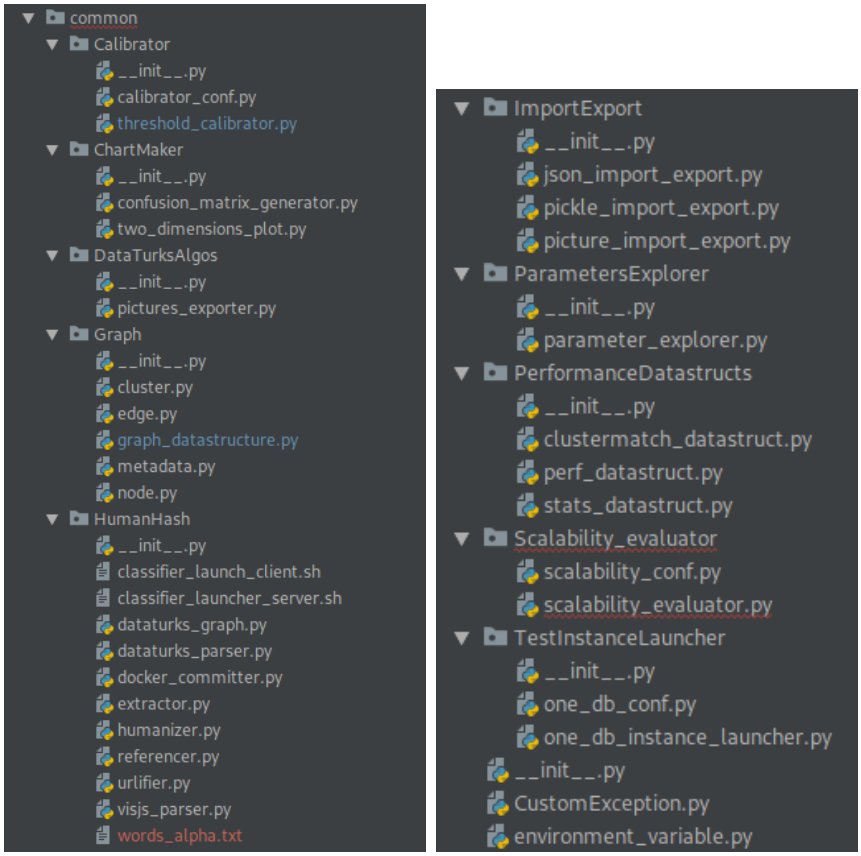
Worker Process

Keep track of information about one worker, which is one subprocess.

For example :

- Construct an argument list and launch the process.
- Try to stop within ¡gracetime¿ seconds the current processus/worker
- Check and display status on logging output
- Check if current processus is running
- Wait until the worker is stopped (= terminated) Put timeout -1 if you don't want to function to timeout

3.3 Common



(a) Common classes (b) Common classes

Figure 3.13: Common

3.3.1 Calibrator

Set of classes used to output a relevant configuration files and thresholds.

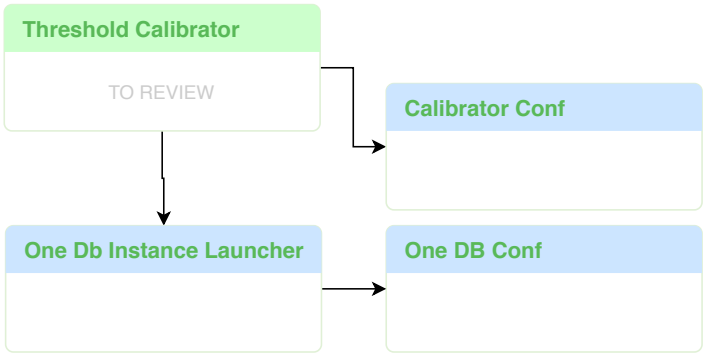


Figure 3.14: Calibrator classes

Threshold Calibrator

From a set of pictures, a ground truth file (a JSON specifying the clustered version of these pictures) and some directives (acceptable rate of true positive, false positive, true negative, false negative), create a configuration file with thresholds as close as possible to the target performance expected. Probably one of the most complex class of the application.

For example :

- Create an instance of douglas quaid for each algorithm and for each threshold : send provided pictures, get quality statistics compared with provided ground truth file
- Outputs a configuration file optimized for the provided dataset and ground truth file, given expected threshold TP/FP goals specified in calibrator configuration
- Verify if input paths are valid. Throw exception if not.
- Evaluate each possible algorithm on the specific dataset, returns the "best" configuration file for each. Uses ground truth files and provided pictures list to optimize goals setted in calibrator conf
- Returns a distance configuration file calibrated to optimize given metrics on ground truth files and provided pictures list.
- Create an instance of a server and client only working on a temporary database (test db). Calibrate distance threshold given provided configuration file (an so which algorithms to activate together) and ground truth files and provided pictures list. Modify calibrator configuration and returns a list of performance object as results.
- Generate a feature configuration object with only one algorithm activated
- Generate a distance configuration object which is less performant but with maximal score in quality
- Compute the best threshold to apply to distance to have an optimized number of False Positive, True Positive, etc.
- Extract the threshold for a specific kind of value
  - rightmost, higher = Get the rightmost higher value of the attribute
  - rightmost, lower = Get the rightmost lower value of the attribute, with attribute greater than Tolerance of the graph evaluator
  - leftmost, higher = Get the leftmost higher value of the attribute, with attribute lower than Tolerance of the graph evaluator
  - leftmost, lower = Get the leftmost lower value of the attribute

Calibrator configuration

Configuration file of the calibrator. Specify acceptable rate of true positive, false positive, true negative, false negative, the number of points to evaluate on the threshold, etc.

3.3.2 Scalability Evaluator

Set of classes used to evaluate the scalability of the application. Add and request pictures, measuring response time. It stores results in JSON and save it as a graph.

Scalability Evaluator

Evaluate the scalability of the database as-is, with current default configuration.

-
- TO COMPLETE

Scalability with threshold evaluator

Evaluate the scalability of the database by changing the threshold for cluster creation in the configuration file.

-
- TO COMPLETE

Scalability configuration

Configuration file of the scalability evaluator. Specify the number of pictures to request to evaluate time, the size of databases to test, etc.

Scalability datastructures

Defines datastructures objects, that store feature, adding, and request time, as well as number of pictures added and requested. Store threshold used for computation.

3.3.3 TestInstance Launcher

Provides an unified way to create a test instance of the server. Point the storage and cache database (and sockets) to a test database, than won't modify production data. Allows to overwrite default configuration files.

OneDataBase configuration

Is an overwritten version of database Default Configuration, modifying path of scripts and sockets.

OneDataBase Instance Launcher

Create a running instance of douglas-quaid, all linked on a unique test database. Modify the behavior of the core launcher handler, to use only one database.

- Construct a full Database instance (Database, workers etc.)
- Construct a Database only instance (Database, no workers etc.)
- Create a database handler (start/stop), modify its configuration and launch the DB
- Replace all attributes of db handler by test database values
- Create a core launcher and overwrite its configuration

3.3.4 ChartMaker

Set of classes used to create chart and graphs. Can print these charts into files.

Confusion Matrix Generator

- Create and export a confusion matrix from two list of clusters. The confusion matrix display the number of pictures in common between any pair of two cluster, taken in both lists.
- Create a matrix (a picture/chart) with specific size, etc.
- Create a heatmap from a numpy array and two lists of labels.
- A function to annotate a heatmap.

Two Dimension Plot

- Print a graph with the TPR,TNR,FPR,FNR ... on one unique chart
- Print a graph with the TPR,TNR,FPR,FNR ... with thresholds provided on one unique chart
- Print a graph of the performance results

3.3.5 Environment Variables

Set of classes accessible from all other classes of the application.

Custom Exception

Defines special custom exceptions.

Environment variable

- Provide the home folder path where douglas-Quaid is installed
- Loading configuration files for logging
- Print big or small lines on standard output
- Verify path validity
- Defines Queues Names in cache database
- Defines End points names for Server side API

3.3.6 Graph

Set of classes to handle operation on graphs

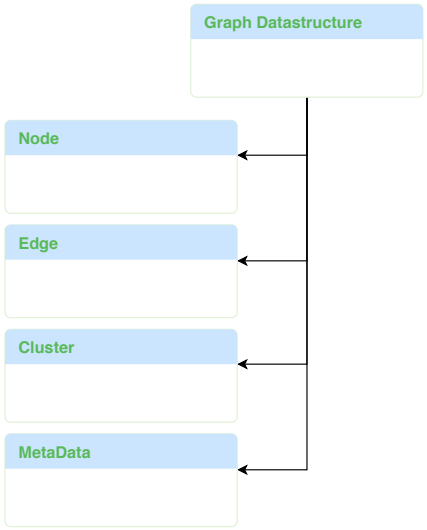


Figure 3.15: Graph classes

Graph Datastructure

Handle the complete graph.

- Add a node to the graph (a picture, not a cluster)
- Add a cluster to the graph (not a picture)
- Add a cluster to the graph with its nodes
- Add an edge to the graph
- Return True if both nodes ids are in this cluster (by ids)
- Return True if both nodes id are in this cluster (by name). Translate names to id and fallback to check if ids in same cluster
- Replace olds id by new ids, if provided in the input list (filename to id). Ex : "toto":id=2 input with Edge : "toto":id=1 input =¿ Edge : "toto":id=2 output
- Get the list of cluster
- Get the cluster of one node
- Returns a list of edges as a dict in the order (node id =¿ has edge to =¿ cluster id)
- Copy "id" values of nodes to their "image" field. (useful if we have no image, and id are actual path)
- Export the current graph as a dict
- Load/ Import a graph from a dict
- Merge two graphs into one unique graph. Uses the list of clusters matches to merge each cluster with each other. Merge a visjs and db graph to produce only one visjs graph, with colors depending on good/bad matches
- Merge a edges of two graphs with colors depending on good/bad matches
- Load VisJS to graph datastructure

Cluster

Handle a cluster of the graph

- Modify an id in the list of members. Replace old by new.
- Return True if both nodes id are in this cluster
- Load/ Import a Cluster object from a dict

Node

Handle a node of the graph

- Load/ Import a Node object from a dict
- Copy id value to image value

Edge

Handle an edge of the graph

- Modify an id in the list of members. Replace old by new
- Load/ Import a Edge object from a dict

MetaData

Handle metadata of the graph / Handle a serie of information related to the current element

- Load/ Import a Meta object from a dict

3.3.7 Import Export Package

Handle exportation and importation of files, objects, pictures, pickled object, etc. in a consistent and unified way.

(a) Import Export Package classes

```
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.

Figure 3.16: Import and Export package information

Pickle Import Export

Handle importation and exportation of object to their pickled version. Pickling version had been chosen regarding its performances.

For example :

- Very specific file to overwrite method to pickle some specific object types.
- Return an object from the pickle version of it
- Return a pickle version of an object
- Create the bundling between class and arguments to save, for Keypoint class. Allow Keypoints from CV2 to be pickled successfully.

Picture Import Export

Handle importation and exportation of pictures in bytes version.

For example :

- Save a bytes representation of a picture to a file
- Return a bytes representation of a picture

JSON Import Export

Define a Custom JSON Encoder class to store Enum and custom configuration objects (for example) of the framework, which handle how to encode specifics object.

For example :

- Save an object as JSON
- Loading an object/data from json.

3.3.8 Parameter Explorer

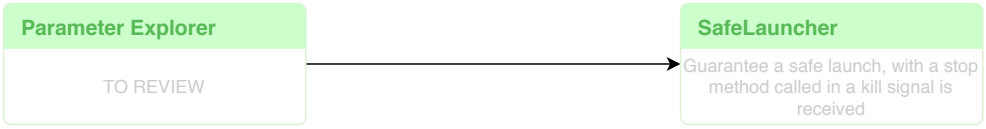


Figure 3.17: Parameters explorer classes

3.3.9 Performance DataStructs

Cluster Match

Stores :

- the first cluster
- the second cluster
- the score between both clusters

Perf Datastruct

Stores :

- A score
- A threshold

Stats Datastruct

Stores :

- Positive elements number
- Negative elements number
- True Positive elements number
- False Positive elements number
- ...

3.3.10 HumanHash

Dataturks Graph

-
-
-
-
-
-
-

Dataturks Parser

-
-
-
-
-
-
-

Docker commiter

-
-
-
-
-
-
-

Extractor

-
-
-
-
-
-
-

URLifier

-
-
-
-
-

-
-

Referencer

-
-
-
-
-
-
-

VISJS Parser

-
-
-
-
-
-
-

3.3.11 DataTurksAlgos

Picture Exporter

Export 4000 pictures not already present in output folder, from a dataturks classification JSON. Do not copy "to delete" items.

-
-
-
-
-
-
-

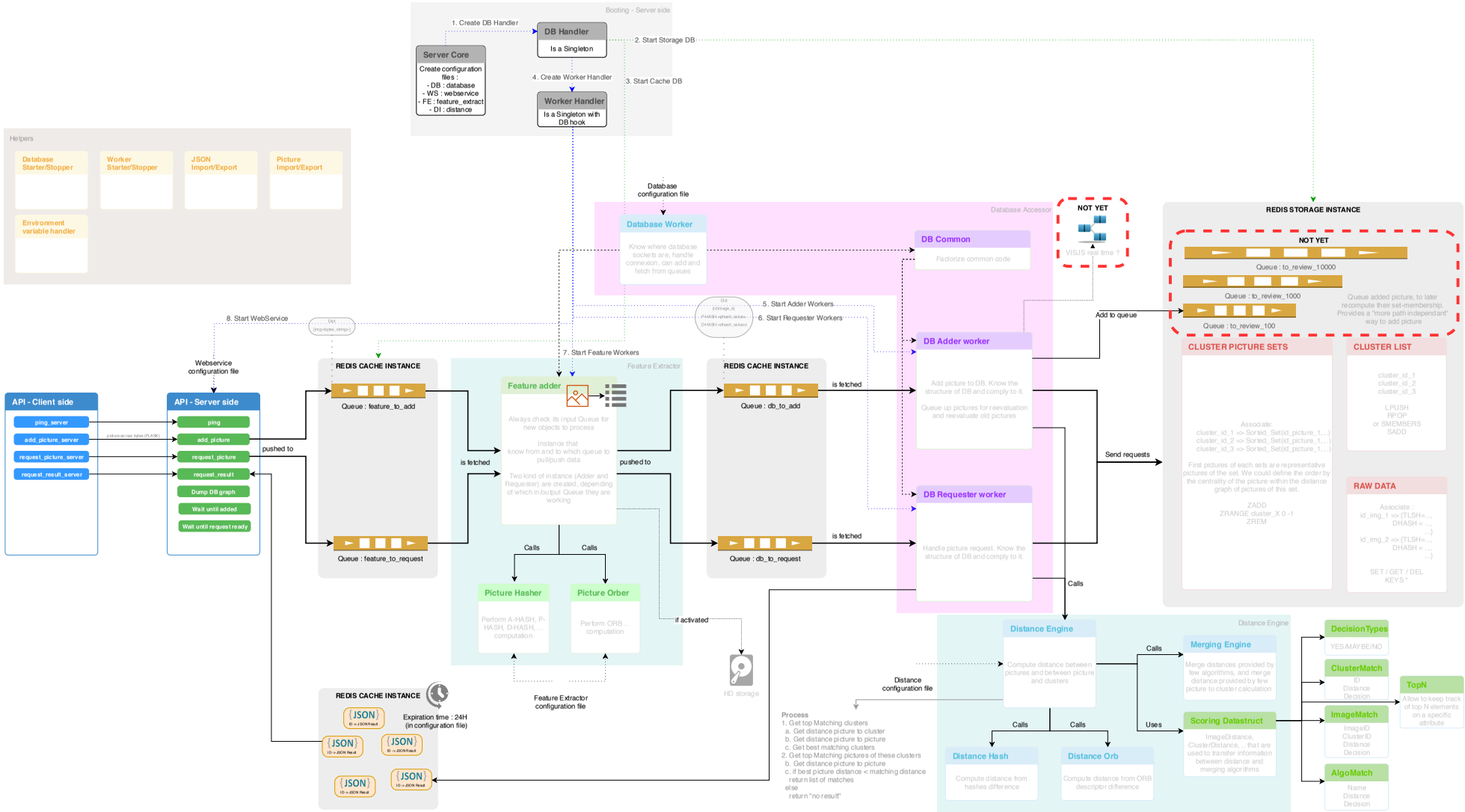
Chapter 4

Components and libraries interaction

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 **??** :

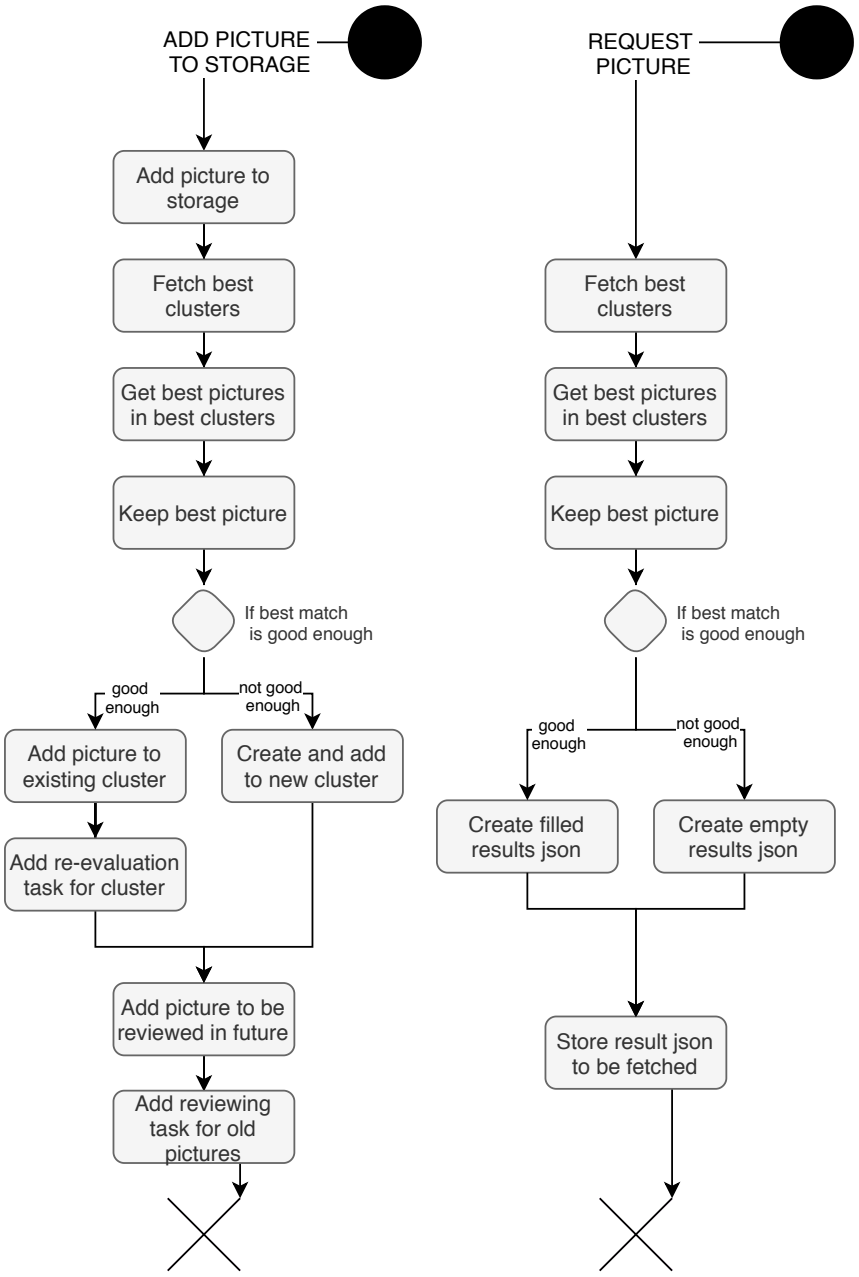
- **API** : The server's endpoint which received client's requests. Mainly running on *Flask*.
- **A Redis cache instance** : it stores current client's requests. For example, a picture to be processed, or a bunch of calculated values to be stored in the redis storage instance.
- **A Redis storage instance** : 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 4.1: 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

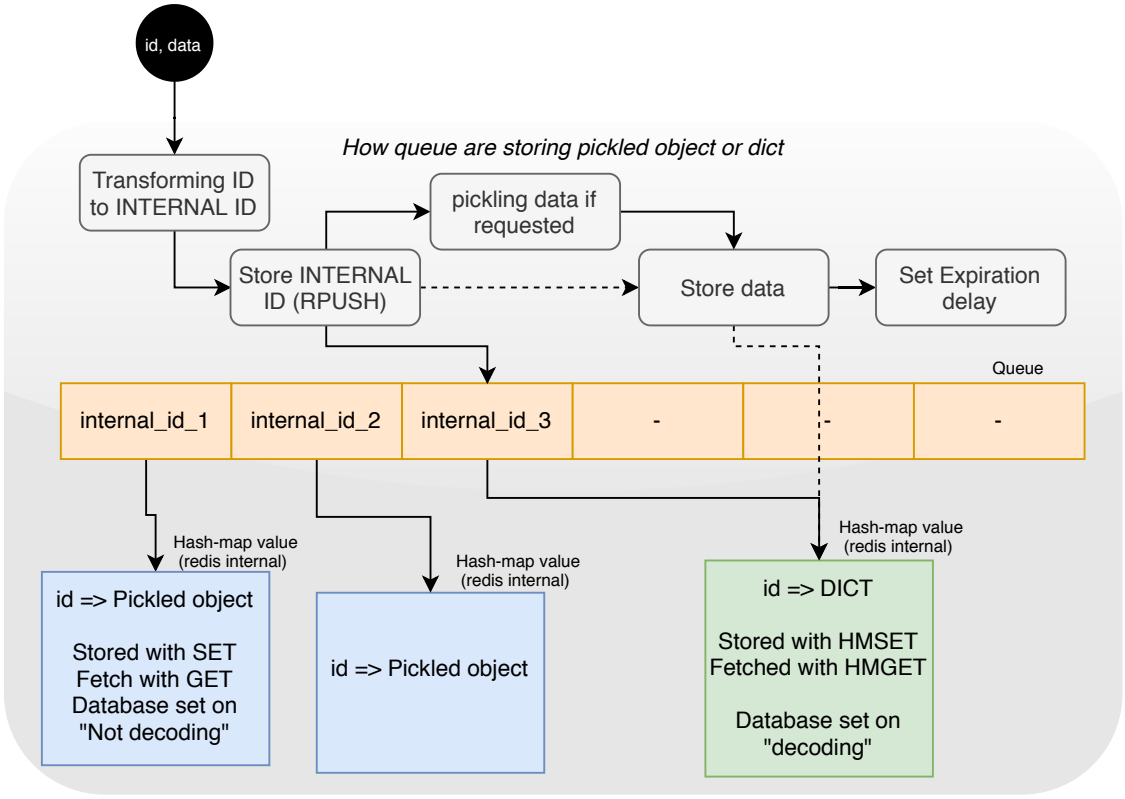
(b) Processus to request a picture's similar pictures

Figure 4.2: Processus

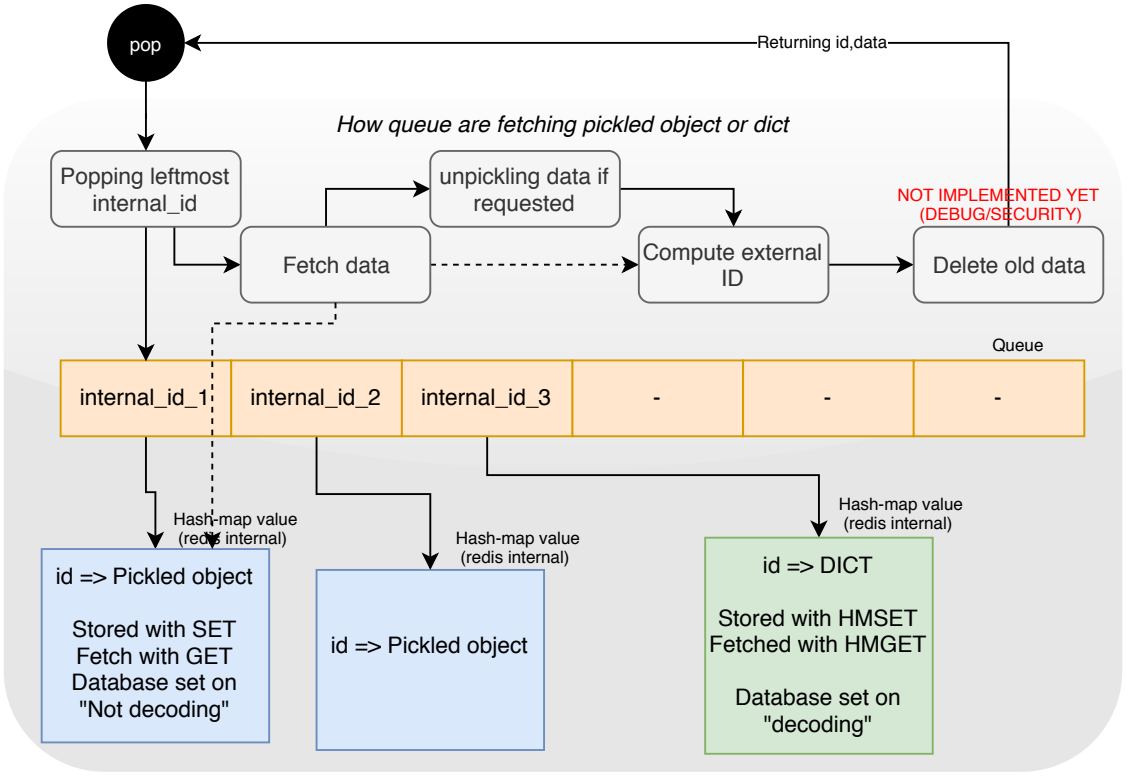
Chapter 5

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 5.1: Queue management

Chapter 6

Database operations

6.1 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 6.1a.

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 ..

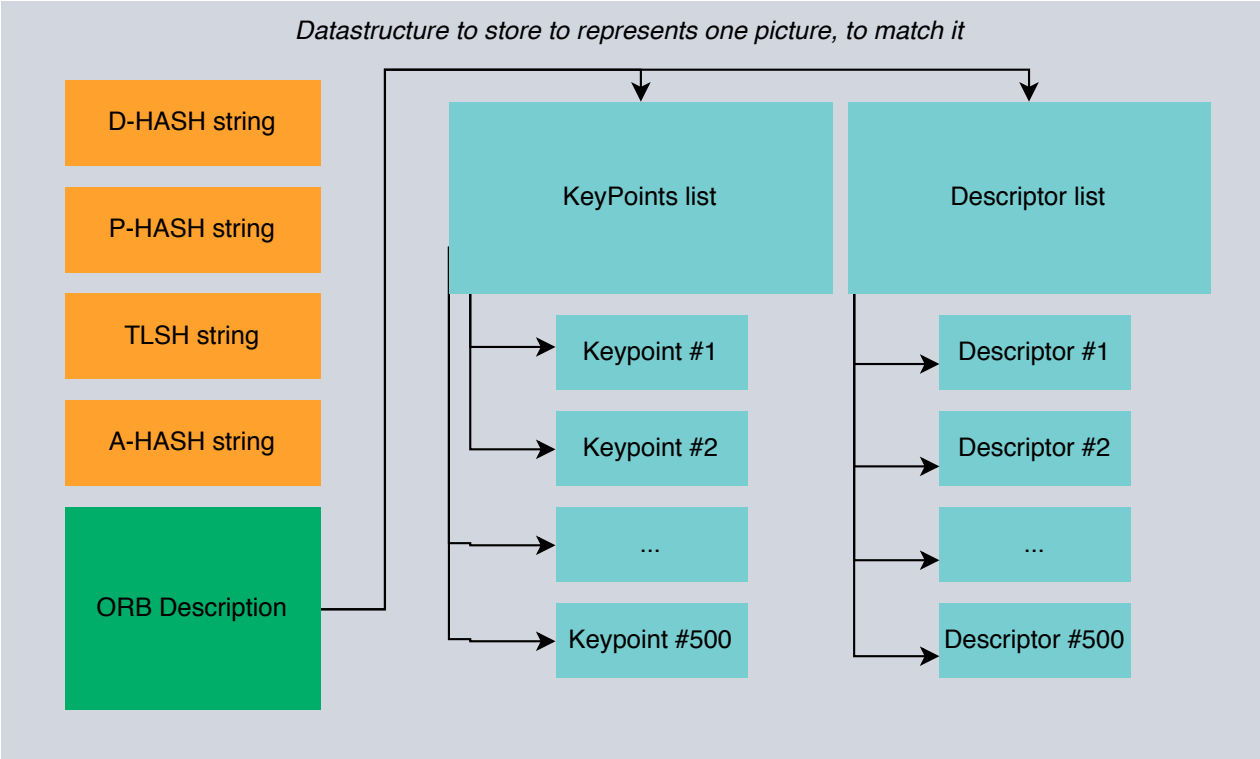
On a performance side, *CPickle* (usable after Python 3.4) seems to provide best results with last version of the protocol. Therefore, we use this implementation. See Figure 6.1b

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 6.1c [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 6.1: Challenge - Datastructure to store in Redis

Chapter 7

Core computation

7.1 Distance precision

I'm doing thresholding at 0.01 at the end of the calculations, no more precise. Mainly because it would be much more tricky to evaluate the "good threshold" in a real-number space. This is discrete and this is easier.

True : it depends on used algorithms. Hashes have 64 up to 400 bits (TLSH, I believe), compared with Hamming distance. So roughly (1/400)th precision, assuming a best case where all hashing algorithms generate 400 bits-length hashes. ORB has 500 descriptors, compared 2 by 2. Giving at maximum 500 matches. So (1/500)th precision. So, we would not need more than a 1/250th precision. (I guess this would need to be applied : `https://en.wikipedia.org/wiki/Propagation_of_uncertainty`) so roughly 0.004 precision

So, true, default 17 digits are not needed.

Matching

Prerequisites : Clusters populated with candidate pictures and an input_picture

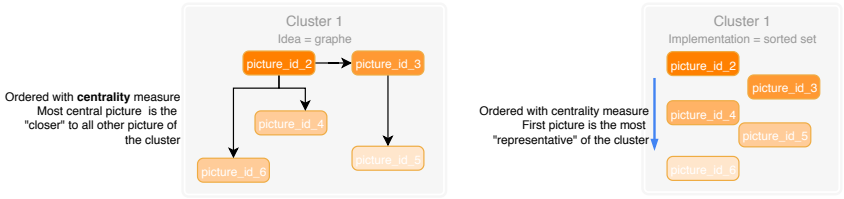
1. Evaluating proximity from input_picture to each cluster
= compute distance between input_picture and *N1* "best representative pictures" of each cluster, and merge the result in one unique "picture_to_cluster" distance.

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

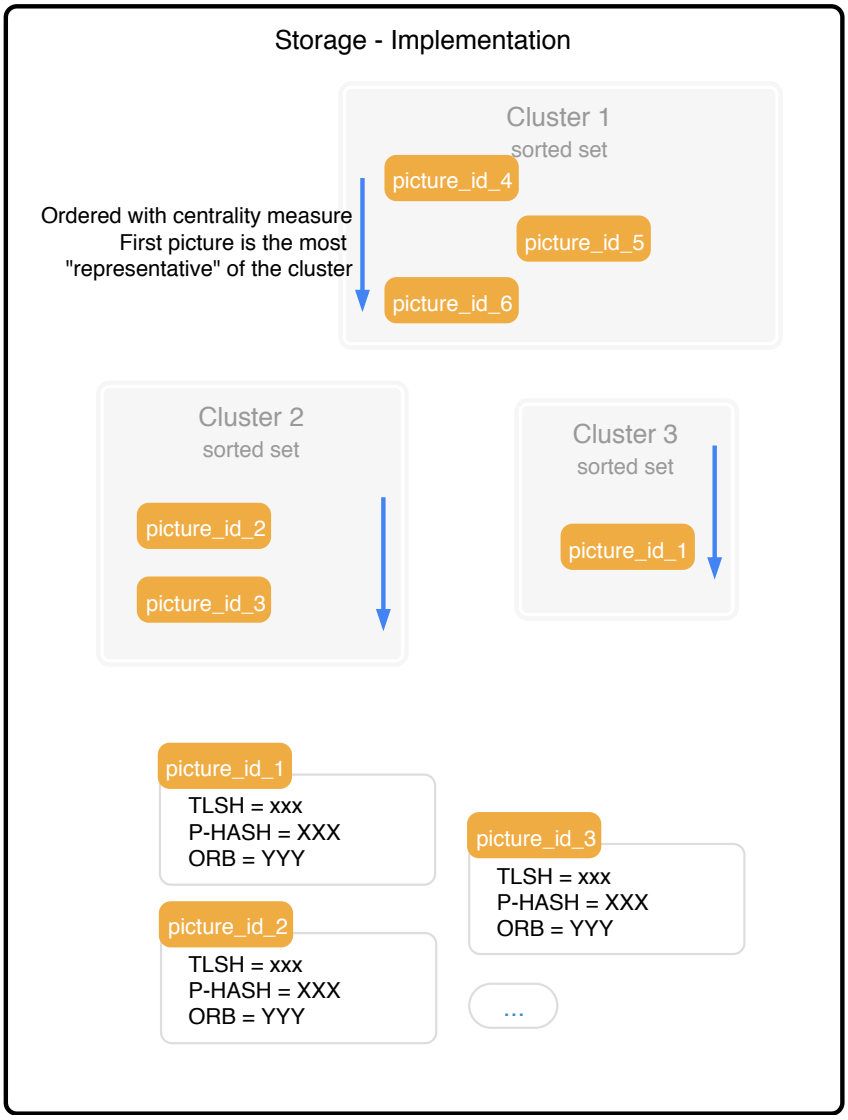
3. Evaluating proximity from input_picture to all candidate picture
= Compute distance between input_picture and each picture of these *N2* clusters.

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 7.1: Datastructure and search

Chapter 8

Tests and Examples

8.1 Performance and Stats datastructures

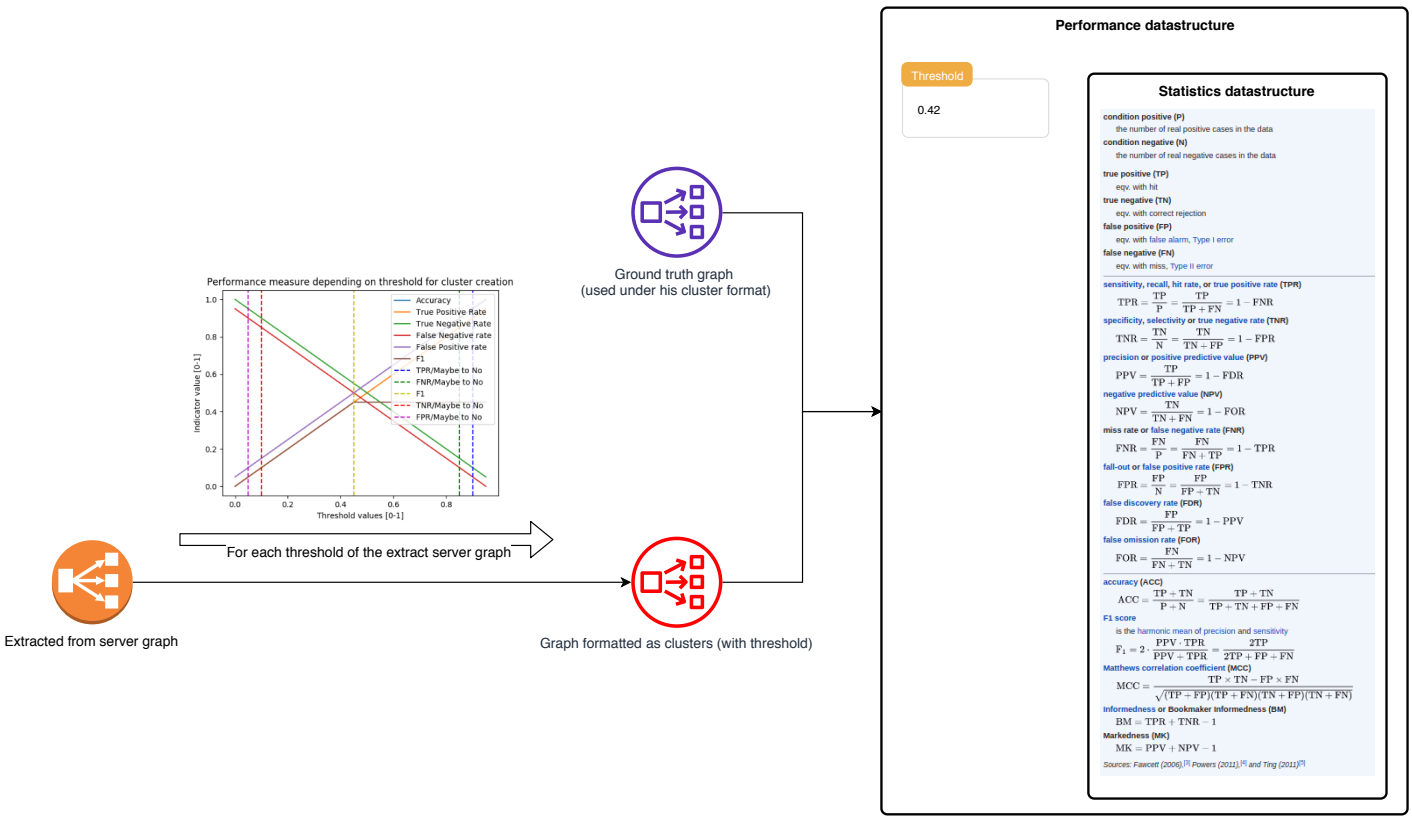


Figure 8.1: Performance and Stats structure, and how it is used to evaluate a full graph for dozens of thresholds

8.2 Graph evaluation and matching quality

We can evaluate the quality of matching made by the library regarding a batch of input files and a ground truth file, that can be constructed with visjs-Classificator, for example.
On one side, we have a clustered version of pictures (output of visjs-Classificator) and on the other

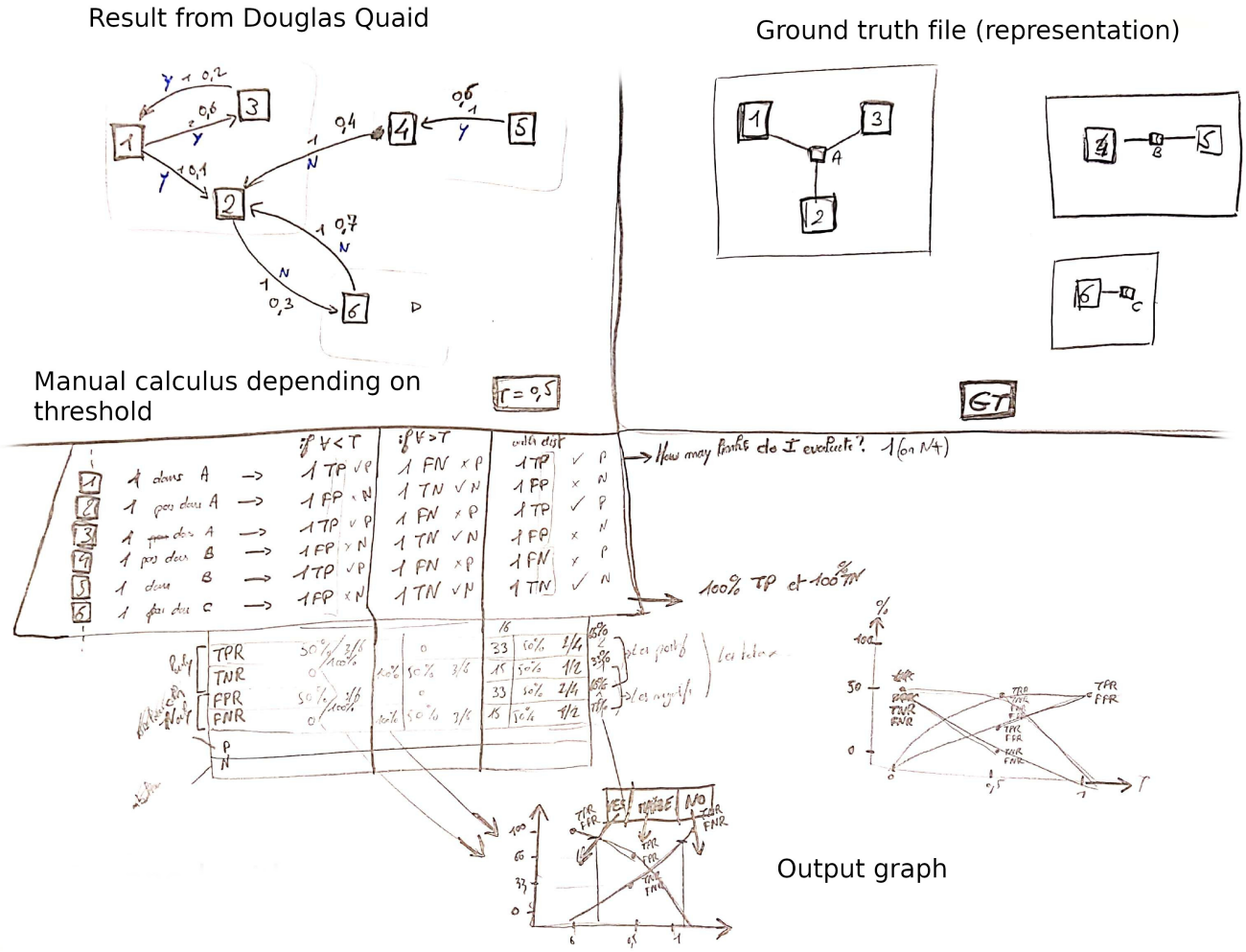
side, we have a a graph representation (output of the client-side graph extractor, which is requesting all pictures of the database and its nearest neighbors).
Quality evaluation of a graph regarding a clustered version of the data is not trivial and so may need some explanation. Starting from the output graph, we evaluate each link (each matches) in an ascending order. So, we go through the best match to the worst match, for each picture of the database. We should evaluate each link as a True Positive, False Positive, True Negative, False Negative, etc.

Essential elements are :

- If the current picture and the matched picture are in the same cluster in the ground truth file, this match is counted as a positive data (P)
- If the current picture and the matched picture are NOT in the same cluster in the ground truth file, this match is counted as a negative data (N)
- If the current match of the current picture is lower than the threshold, this is counted as a positive match (xP) as follow :
  - If then, both pictures were in the same cluster in ground truth file, this is a True Positive match. (TP)
  - Otherwise, if both pictures were NOT in the same cluster in ground truth file, this is a False Positive match. (FP)
- If the current match of the current picture is greater than the threshold, this is counted as a negative match (xN) as follow :
  - If then, both pictures were in the same cluster in ground truth file, this is a False Negative match. (FN)
  - Otherwise, if both pictures were NOT in the same cluster in ground truth file, this is a True Negative match. (TN)

Then other indicators, as True Positive Rate (TPR), True Negative Rate (TNR) etc. can be computed.

Ground truth file format and requests outputs from Douglas-Quaid API are displayed below.



(a) Graph evaluation - Manual verification

Figure 8.2: Test concerning output sgraph evaluation

Listing 8.1: List of requests and result for each image of the database

```
[{' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.1,
                       'image_id': '2'},
                      {' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.6,
                       'image_id': '3'}],
  'request_id': '1',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.3,
                       'image_id': '6'}],
  'request_id': '2',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.2,
                       'image_id': '1'}],
  'request_id': '3',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.4,
                       'image_id': '2'}],
  'request_id': '4',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.6,
                       'image_id': '4'}],
  'request_id': '5',
  'request_time': 0,
  'status': 'matches_found'},
 {' list_pictures ': [{' cluster_id ': 'XXX',
                       'decision': 'YES',
                       'distance': 0.7,
                       'image_id': '2'}],
  'request_id': '6',
  'request_time': 0,
  'status': 'matches_found'}]
```

Listing 8.2: VisJS-Classificator output format ground truth file

```
{'clusters': [{'group': '',
               'id': 'A',
               'image': 'A',
               'label': 'A',
               'members': ['1', '2', '3'],
               'shape': 'image'},
              {'group': '',
               'id': 'B',
               'image': 'B',
               'label': 'B',
               'members': ['4', '5'],
               'shape': 'image'},
              {'group': '',
               'id': 'C',
               'image': 'C',
               'label': 'C',
               'members': ['6'],
               'shape': 'image'}],
 'edges': [{'color': 'gray', 'from': '1', 'to': 'A'},
           {'color': 'gray', 'from': '2', 'to': 'A'},
           {'color': 'gray', 'from': '3', 'to': 'A'},
           {'color': 'gray', 'from': '4', 'to': 'B'},
           {'color': 'gray', 'from': '5', 'to': 'B'},
           {'color': 'gray', 'from': '6', 'to': 'C'}],
 'meta': {'source': 'DBDUMP'},
 'nodes': [{'id': '1', 'image': '1', 'label': '1', 'shape': 'image'},
           {'id': '2', 'image': '2', 'label': '2', 'shape': 'image'},
           {'id': '3', 'image': '3', 'label': '3', 'shape': 'image'},
           {'id': '4', 'image': '4', 'label': '4', 'shape': 'image'},
           {'id': '5', 'image': '5', 'label': '5', 'shape': 'image'},
           {'id': '6', 'image': '6', 'label': '6', 'shape': 'image'}]}
```

8.3 Threshold extraction from quality Graph

One ability of the library would be to define its own thresholds from an input dataset provided along with a ground truth file.
We've seen that a scoring mechanism can allow us, for a given threshold, to extract true positive, false positive, true negative, false negative rates and more. This is not enough for the library to work : we need to extract thresholds out of these measures.
A simple mechanism compute these bunch of values for all possible thresholds in some range. For example, we will compute TP,TR,FN,FN for a distance threshold of 0.1, 0.2, 0.3, etc. A threshold at X means that all links with a distance below X are marked as "Good link" thanks to the library and all links with a distance upper than X are marked as "Bad Link" thanks to the library. Therefore, we have more links marked as "good" if the threshold is increased. Therefore, we have more false positive : link that should not have been marked as "good" but were marked as "good".
For each case (True positive, false positive, true negative, false negative) we want to have the "best" threshold. As this definition is related to what is to be expected from the library, we want to configure it by a percentage of acceptable false negative, or acceptable false positive, or minimum true positive, or minimum false negative. We could also want a mean value, if we don't want to have a YES/MAYBE/NO threshold, but only a YES/NO threshold.

Therefore, we defined 4 thresholds obtainable from a score graph. If we setup a 10% acceptable rate of false negative, false positive, etc. we can have for instance :

- A TPR threshold : upper to this threshold, there is more than 90% true positive
- A FNR threshold : upper to this threshold, there is less than 10% false negative
- A TNR threshold : below to this threshold, there is more than 90% true negative
- A FPR threshold : below to this threshold, there is less than 10% false positive

We are then able to know which region of the graph, we label as "YES, it's a match"/"MAYBE, it's a match"/"NO, it's not a match", regarding the current evaluated distance between two pictures .

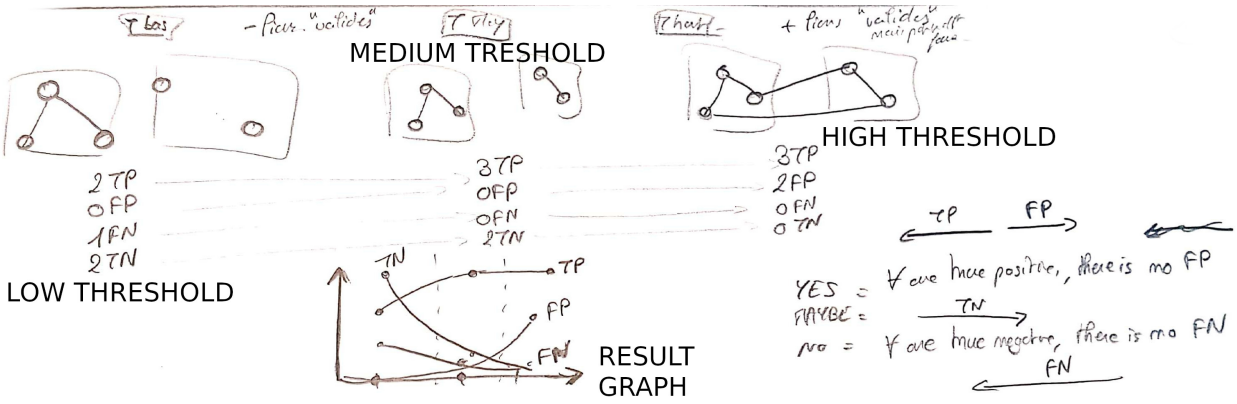
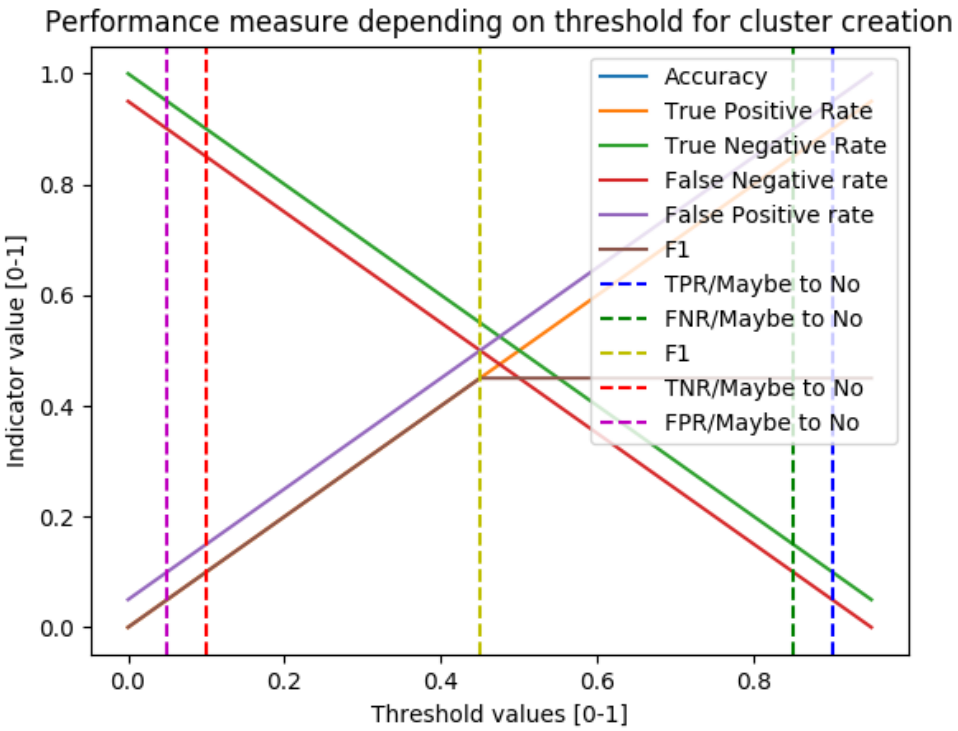


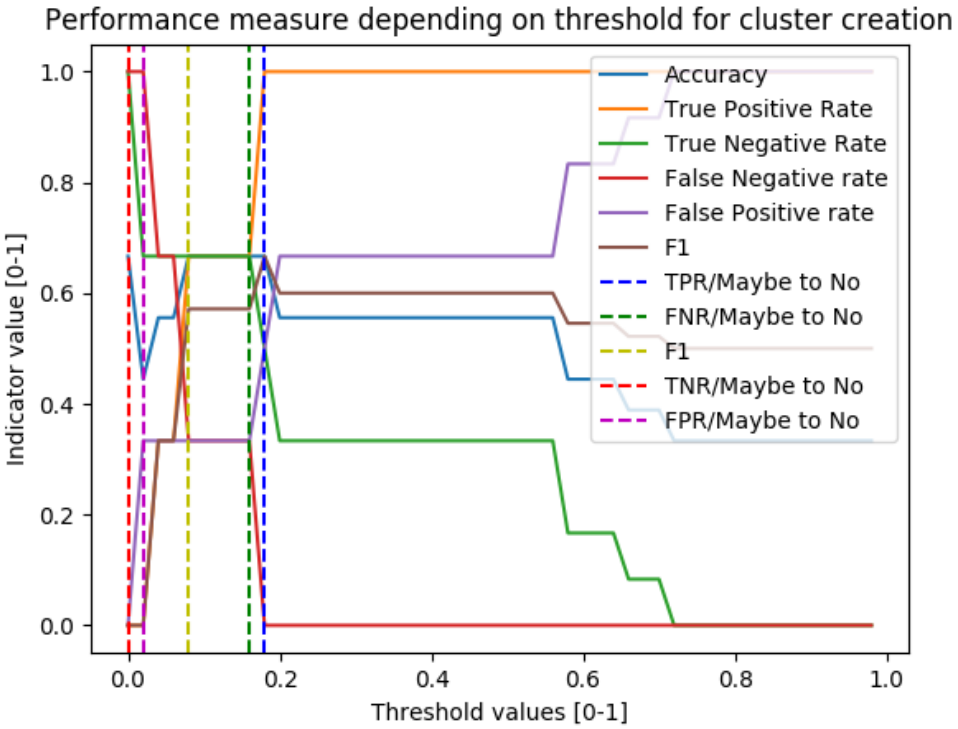
Figure 8.3: Thresholds examples

You can then understand where the areas a placed on 8.4b : the leftmost area (lower than 0.05) is a YES area, between 0.05 and 0.19 a MAYBE area, and upper a NO area.

Implementation details : A tradeoff between maintainability and quantity of code was made. In particular, the algorithm which finds the "best threshold" out of a graph, is not complex. However, the numerous possibilities offered to the user (specifying TX and TY, or TY and NZ, etc.) as input of the calibrator complexify the code.



(a) Simulated values



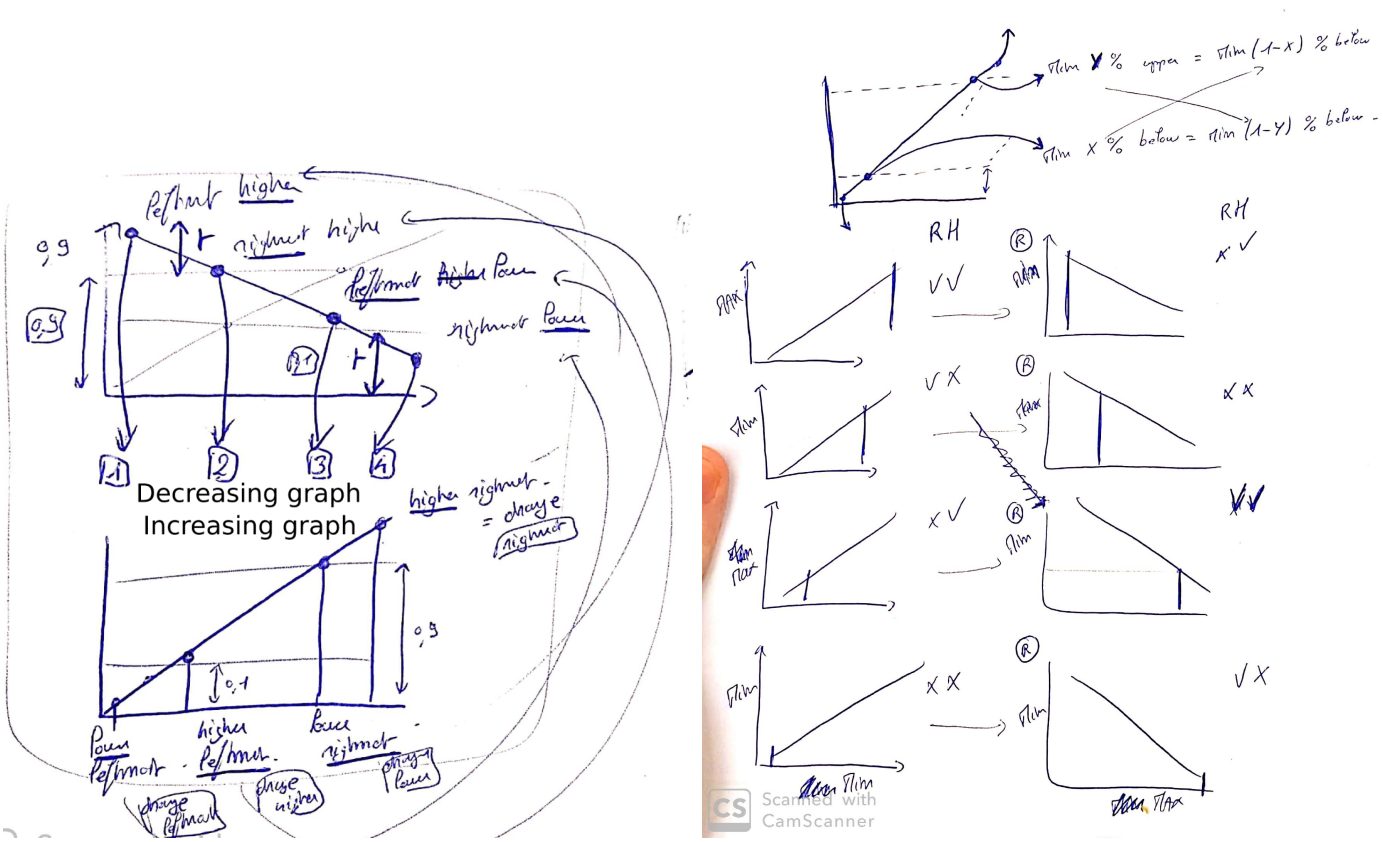
(b) Real values - ORB

Figure 8.4: Thresholds extraction from a scoring graph

Therefore, to factorize the code, all graphs are reduced to decreasing graphs only. 4 major cases are met :

- Finding the most extreme (lowest, for example [4] on fig. 8.5a) point of the graph
- Finding the most other extreme (highest, for example [1] on fig. 8.5a) point of the graph
- Finding the rightmost upper point ([2] on fig. 8.5a) given some threshold
- Finding the leftmost lower point ([3] on fig. 8.5a) given some threshold

To transform a "human-like" command like "**I want no more than 10% of false positive in the automatically classified pictures**" into one threshold is not that easy.
We need to figure out if the attribute to optimize will have an increasing or decreasing graph. (False positive are increased as threshold is increased. True Negative is decreased as threshold is increased). Then we need to understand the threshold. Do we need a maximum or minimum value of 10% ? Because it's a true positive here, this is a maximum value of 10%). Then we need to find on the graph the threshold at which this condition is found, while optimizing the threshold in the "good way". (A threshold at 0% would satisfy a "10% false positive maximum" condition. However, we want to reduce the amount of work needed, and so maximize the threshold within the area where the condition is verified).



(a) Thresholds look-up equivalence (b) Mapping from increasing cases to decreasing cases

Figure 8.5: Explanation and drafts to compute with the less code possible thresholds over TP,TN, etc. graph

Chapter 9

Visualization

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

Chapter 10

Results

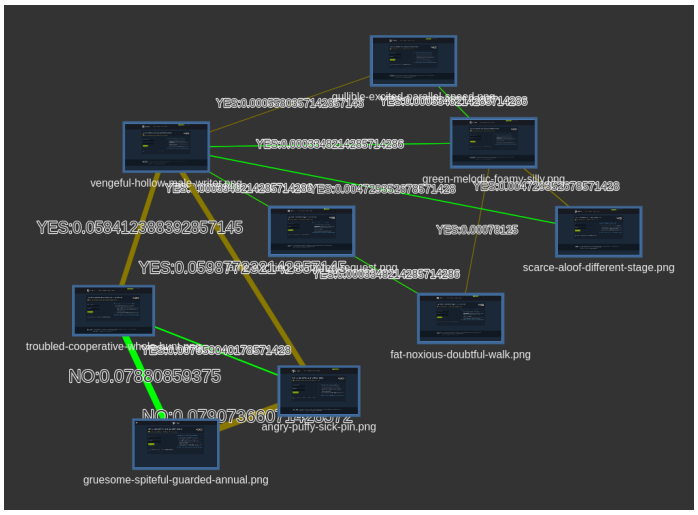
Execution of Douglas-Quaid on test dataset were recorded, evaluated, and are now presented in the next sections.

10.1 Execution 1

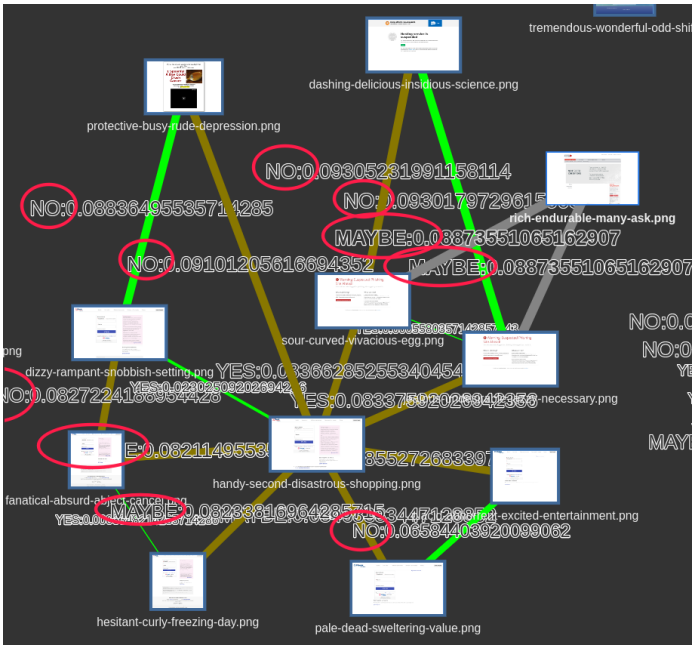
This was the first version of the "pipelined" Douglas-Quaid library.

10.1.1 Quality

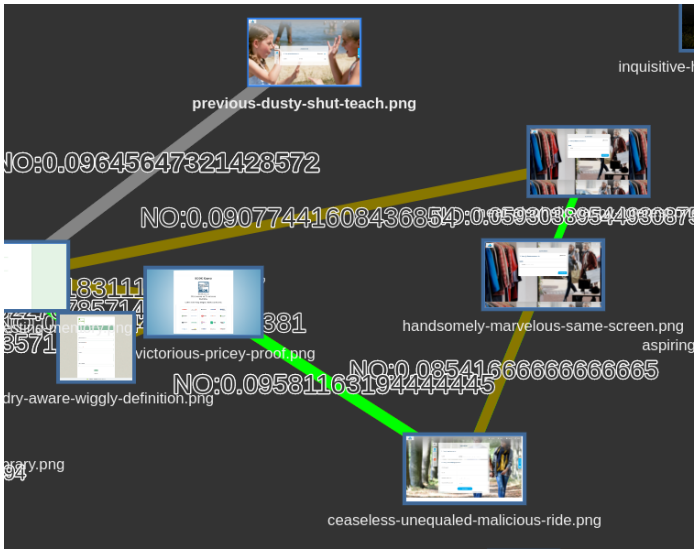
I'll try to evaluate wih Yes/No/Maybe tag, to see what result we can achieve. (Note : the calibration had been performed on 40 pictures. Final dataset evaluated of 470+ pictures)



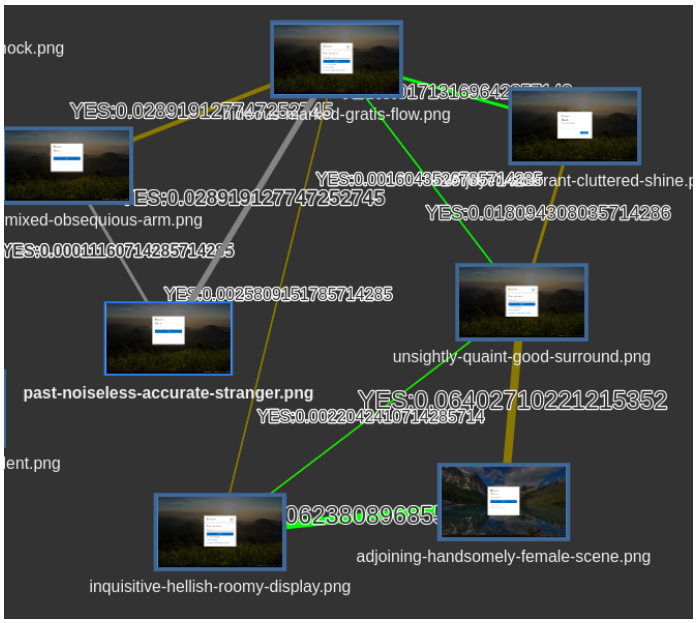
(a) Good for steam



(b) Interesting matches on Fibank. Quite good to remove non-matches

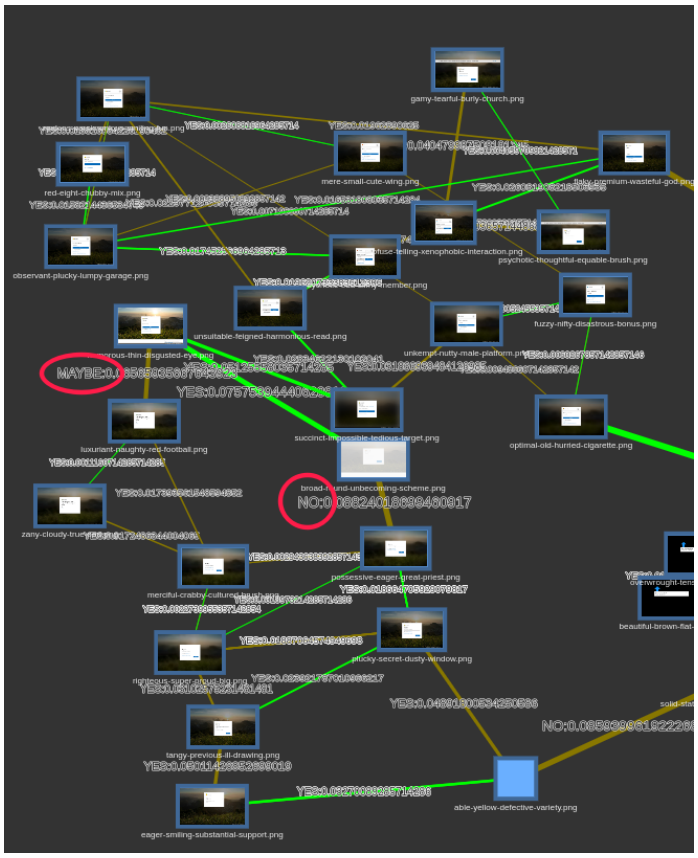


(c) KBC is (in this configuration) not detected as "YES" matches. Need to investigate

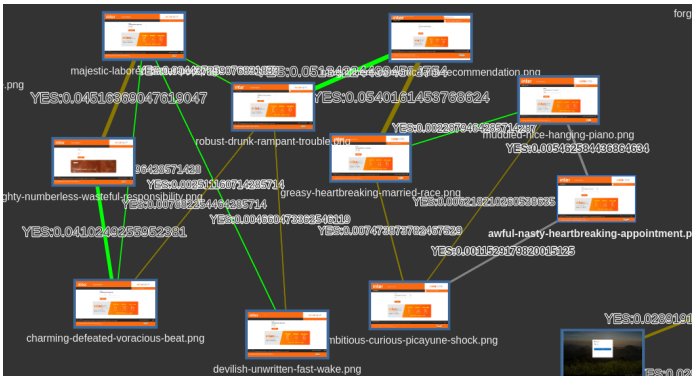


(d) No problem for microsoft, even with colors, etc.

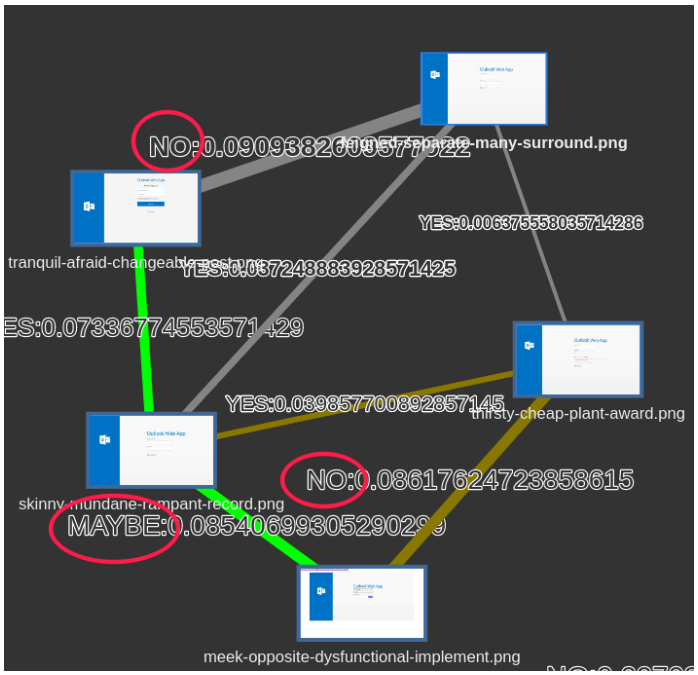
Figure 10.1: Experiment N1 overview



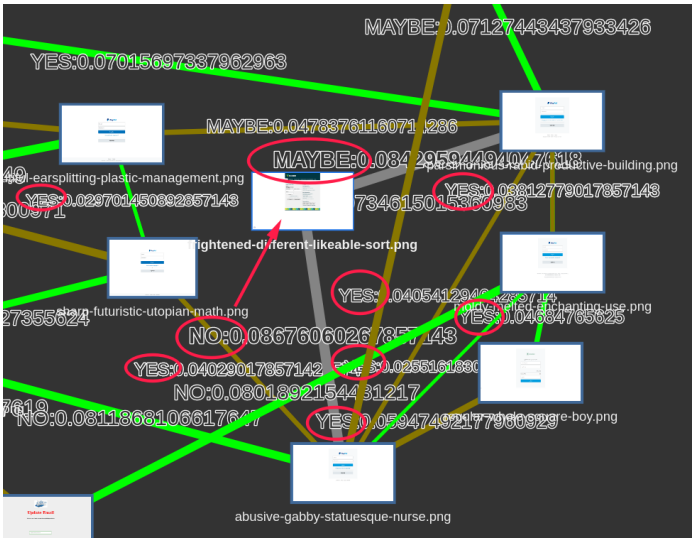
(a) No problem for microsoft, even with colors, etc.



(b) Good for uniform picture, even with adds

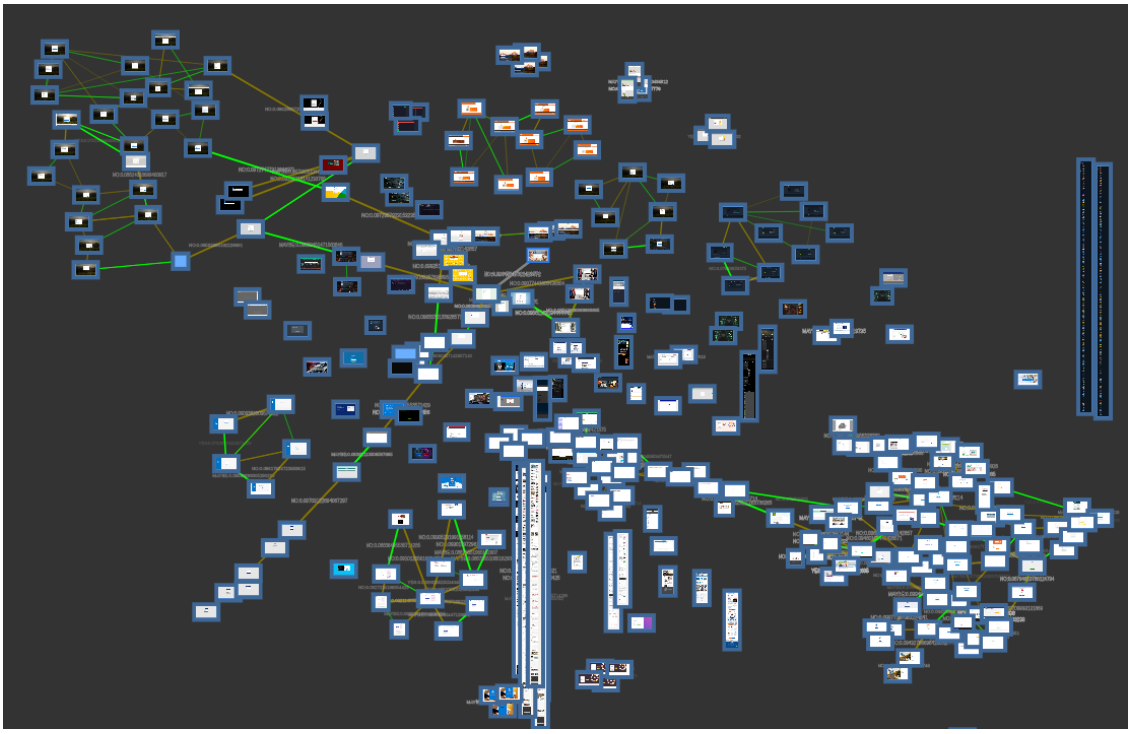


(c) Good with outlook, even with modifications

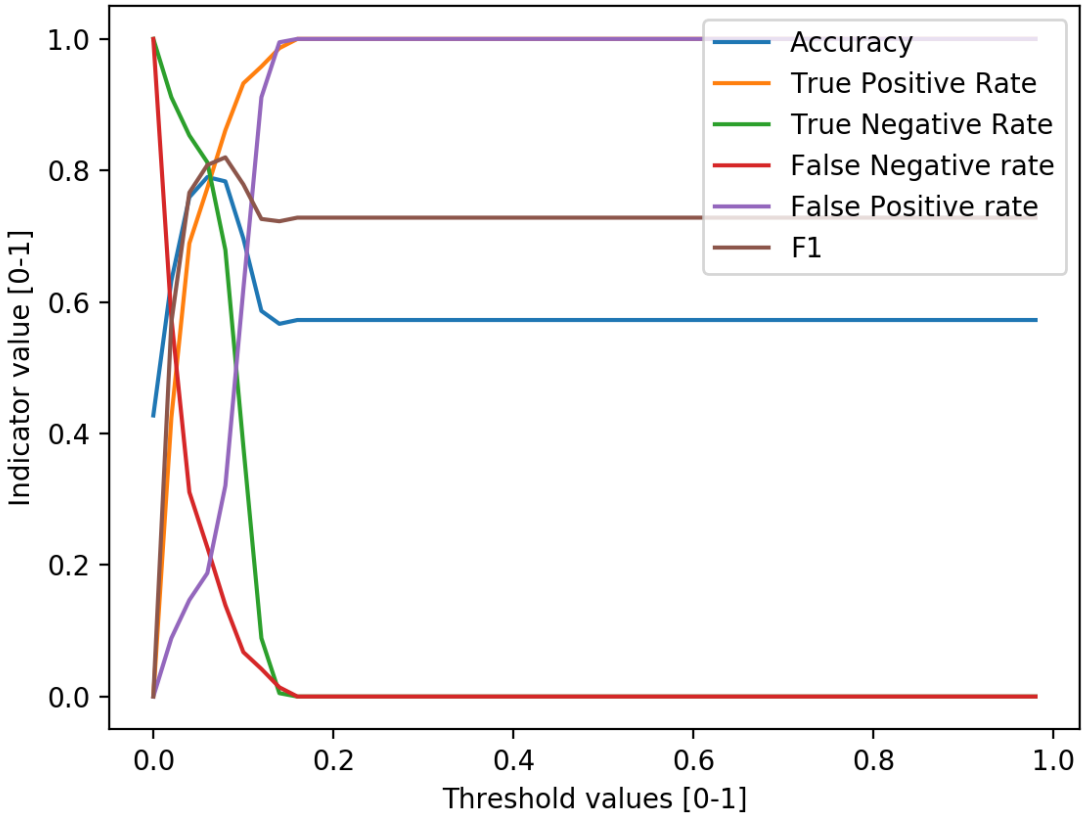


(d) And we can spot the real "good thing" with Yes/-Maybe/No : with distance it's are to tell anything, but with Y/M/N its here possible to differentiate Paypal and non-paypal pictures

Figure 10.2: Experiment N1 overview

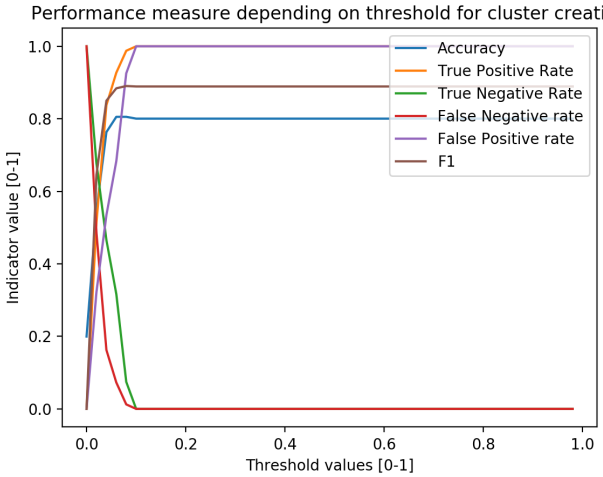


(a) Global results overview

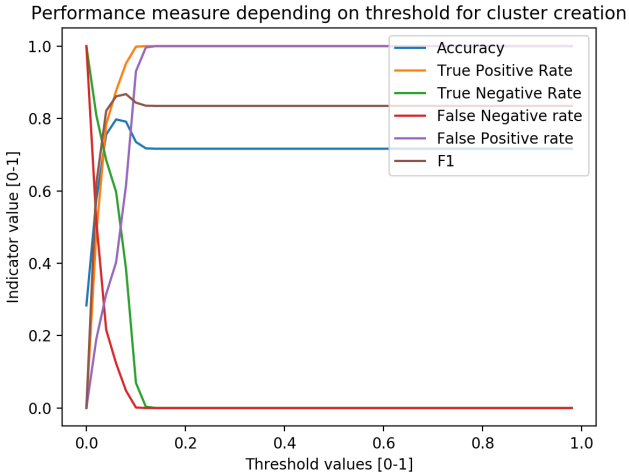


(b) Matching quality : if we rely only on distances, and put a threshold to evaluate the "goodness" of matches, we can at best reach a 80% accuracy, 80% F1 score with 80% true negative, 80% True positive, 20% false positive and 20% false negative (threshold at 0.1, roughly)

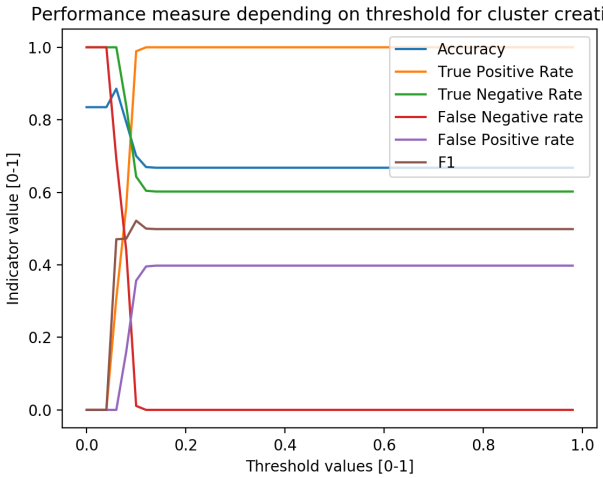
Figure 10.3: Experiment N1 overview



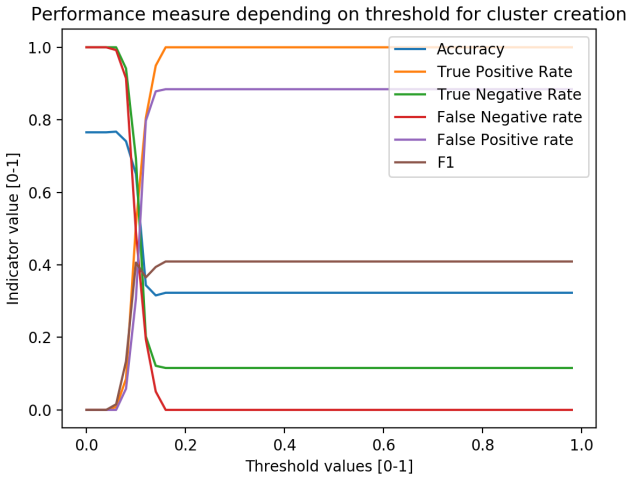
(a) YES matches only. A monotonic function indicates that there is no "Good choice" to make about the threshold. All "YES" matches seems good. None has to be removed. (to check) True positive and False positive are so aligned that they is no good threshold to choose to optimize one versus the other. (to check interpretation)



(b) YES and MAYBE matches only. A non-monotonic function indicates that there is some "better choice" to make about the threshold. Some matches have to be removed

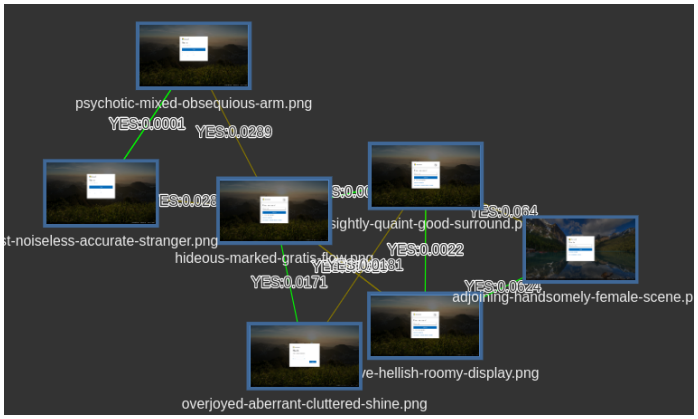


(c) MAYBE matches only



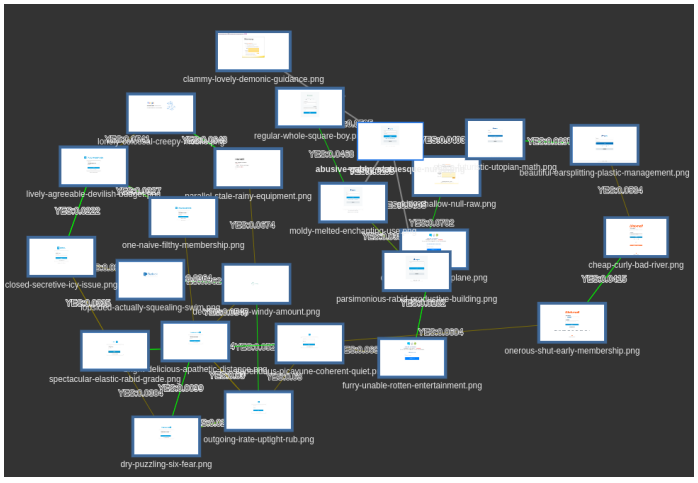
(d) NO matches only

Figure 10.4: Quality depending of threshold filtered by match type. These graph can be interpreted as "When the library is a good match, this is a good match."

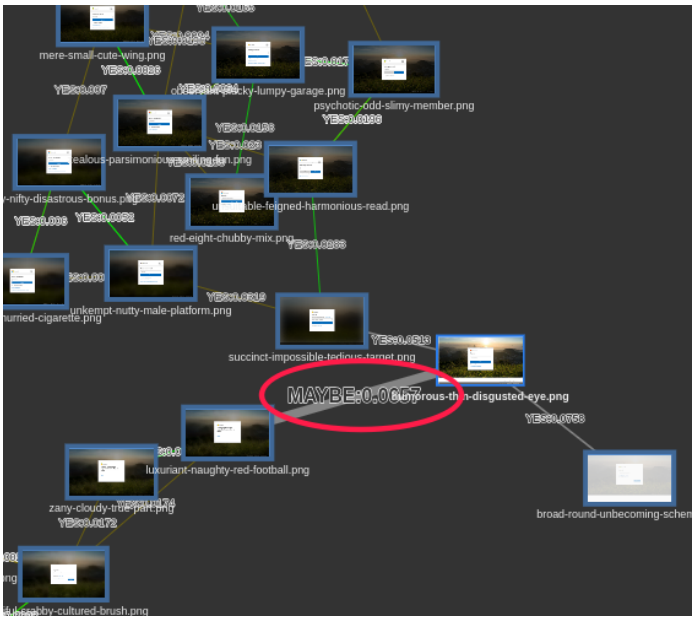


(a) Cluster without false positive on Microsoft with YES decisions only

(b) Error pages matches with YES decisions only

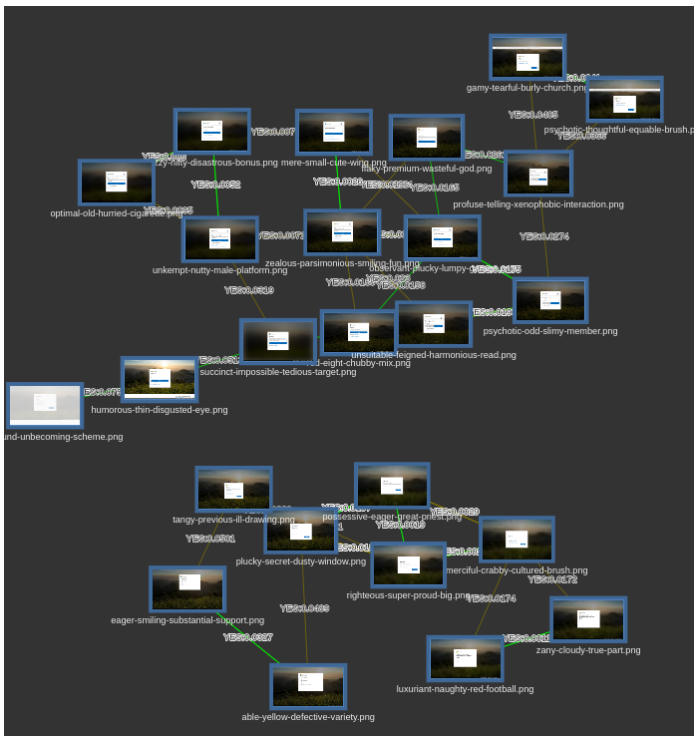


(c) Forms from Paypal, Outlook and Circl are messing up together with YES decisions only

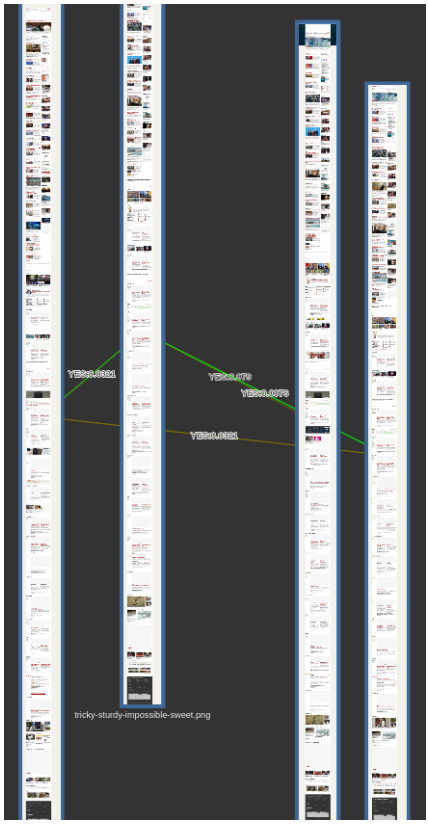


(d) A MAYBE decision is merging two Microsoft Clusters.

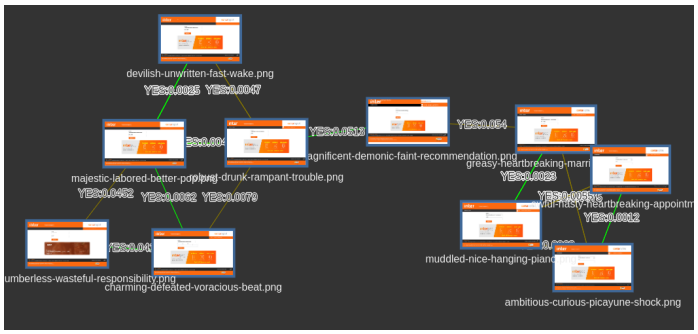
Figure 10.5: Experiment N1 overview



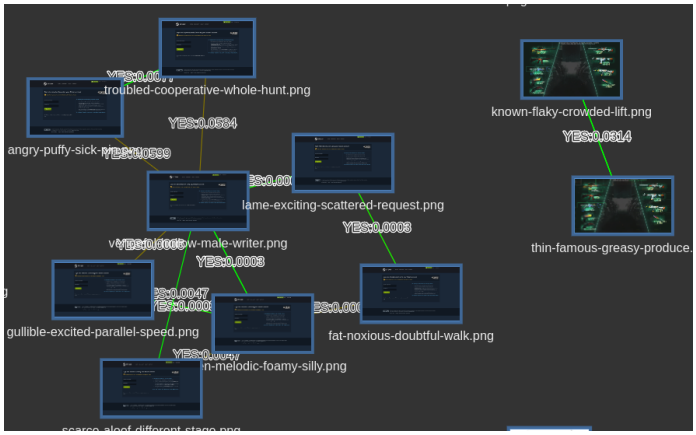
(a) Two microsoft clusters that are not linked with YES decisions only



(b) Big pictures is not an issue

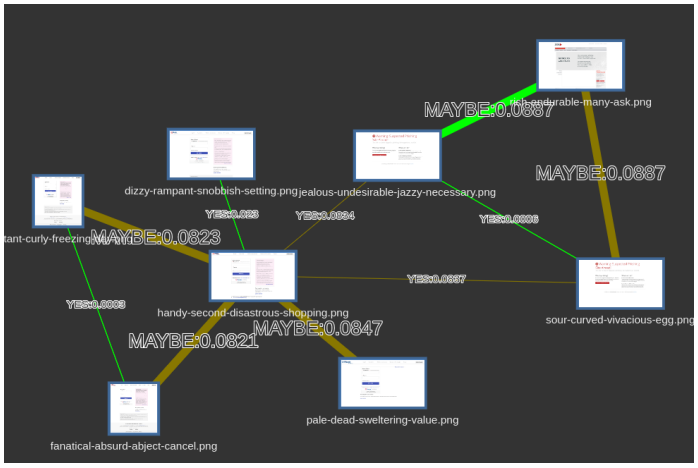


(c) Cluster without any false positive with YES decisions only

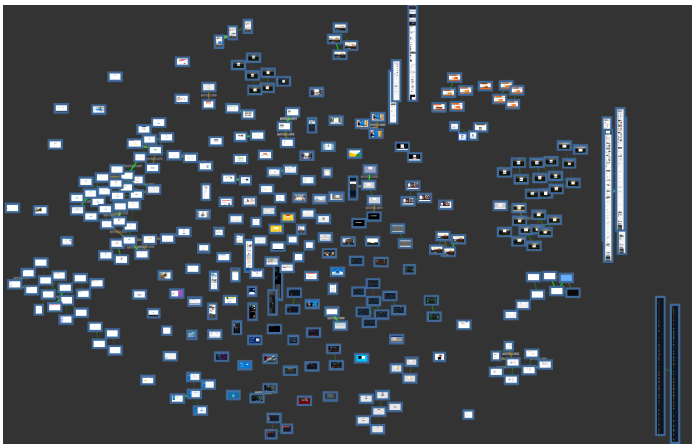


(d) Cluster without any false positive with YES decisions only

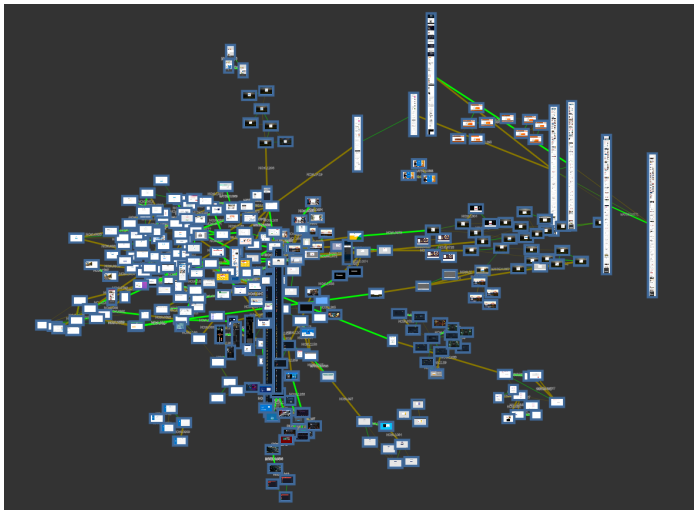
Figure 10.6: Experiment N1 overview



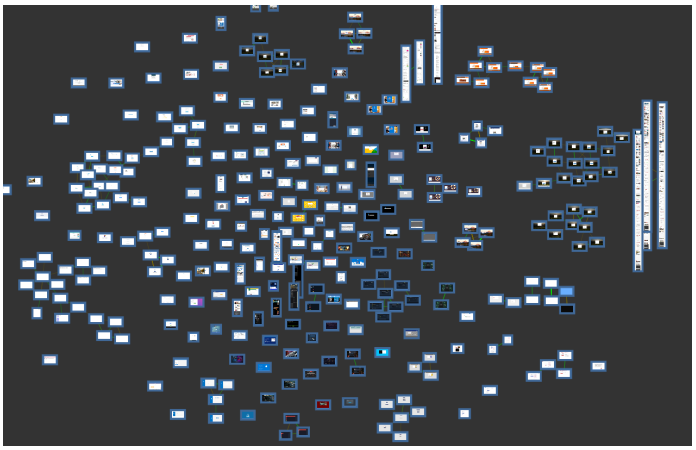
(a) FiBank cluster matching together and mixed with another form. YES and MAYBE decisions



(b) YES AND MAYBE graph together



(c) YES, MAYBE and NO decisions graphs together



(d) YES only decision graph

Figure 10.7: Experiment N1 overview

10.1.2 Speed

On a performance side, there is a rising issue : scalability. A scalability test had been performed quite late in the development of Douglas-Quaid, waiting to have a working version of the library. Few issues were detected, as tests were performed :

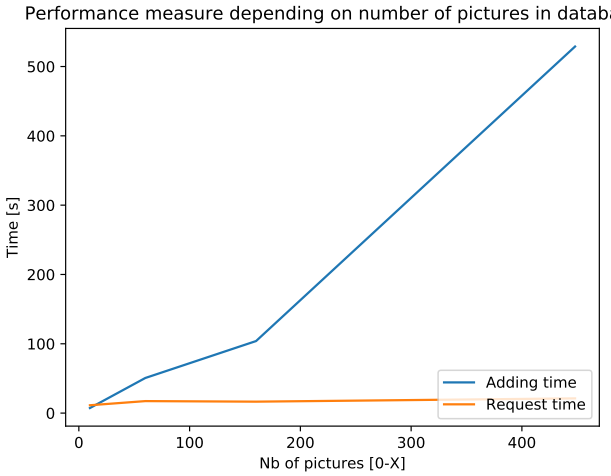
- While adding pictures, the rate at which pictures are added is irregular globally but very regular worker-wise.
- The scalability of the 'clustering' solution seems not to perform as well as expected. Adding time is at least constant while database grows, and seems above linear in number of pictures already stored.

Few sources of problems were brainstormed :

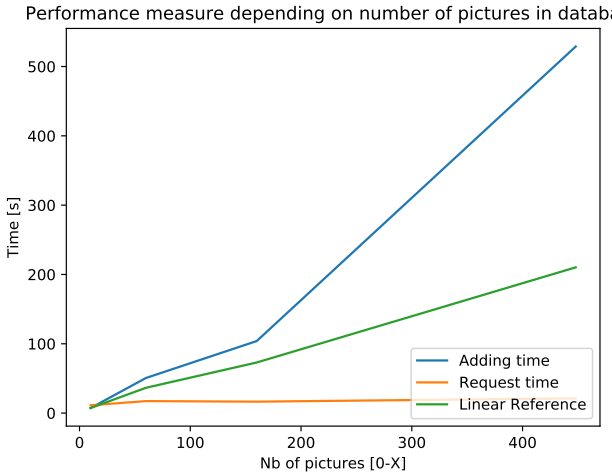
- Redis capability : with more than 40 workers, each having 4 connections to Redis Database, performing requests all together, we can legitimately think that Redis is the bottleneck. CPU Usage of Redis staying "normal" (not 100% CPU) showed this was not source of the problem. This was double checked, as redis called were not the place where most of the DataBase Adder Function is spent.
- Context switches, too much workers : however, the number of used processors is still below its maximum (5 cores free, steady)
- A CProfile Analysis of one worker showed that most of the calculation came from ORB OpenCV2 calculations. The number of comparisons, leveraged by the time of one OpenCV2 computation, blew up the time taken to add a picture.

Few solutions are envisioned :

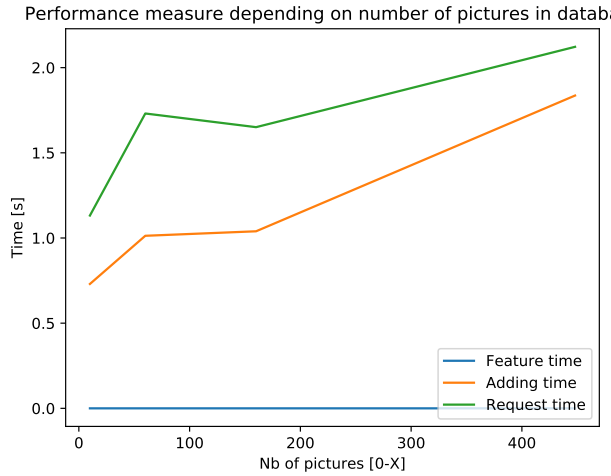
- Replace ORB by a BoW ORB. This would imply to manually trigger the dictionary creation. However, this would lower the matching speed to one comparable to hash's one. Dictionary could be created on the go, automatically, in a V2.
- Removing ORB is a solution. Not satisfying however.
- Using ORB as a second-class verification algorithm only, when hashes had already been checked. This solution is not that much a good idea, as match from ORB and from Hashes are really complementary.
- Use a Flann matcher/LSH with ORB. Even if noted as faster, it seems that on a low number of features (500 in our case) gives advantage to BruteForce Matcher. This was verified with CarlHauser, were matches were better (expected) with BruteForce Matcher (as this is an exhaustive search) in front on FLANN LSH search (as it constructs an index to navigate the features); and speed was greater (unexpected) with BruteForce than FLANN LSH. Please note that options and parameters may influences this conclusion. More digging would be needed to verify this statement.



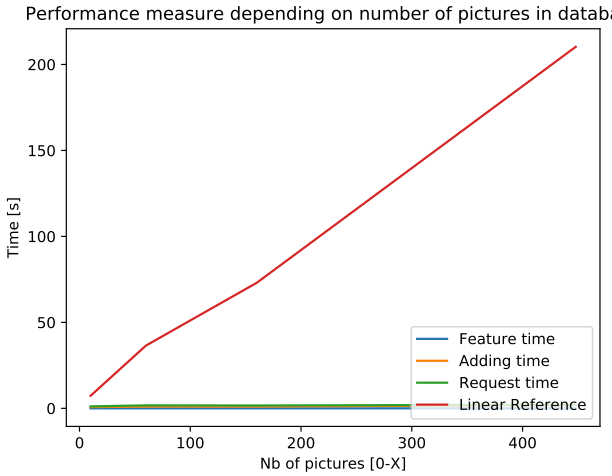
(a) Total time to add pictures in database - No normalization and no reference line



(b) Time to add pictures in database - No normalization and reference line

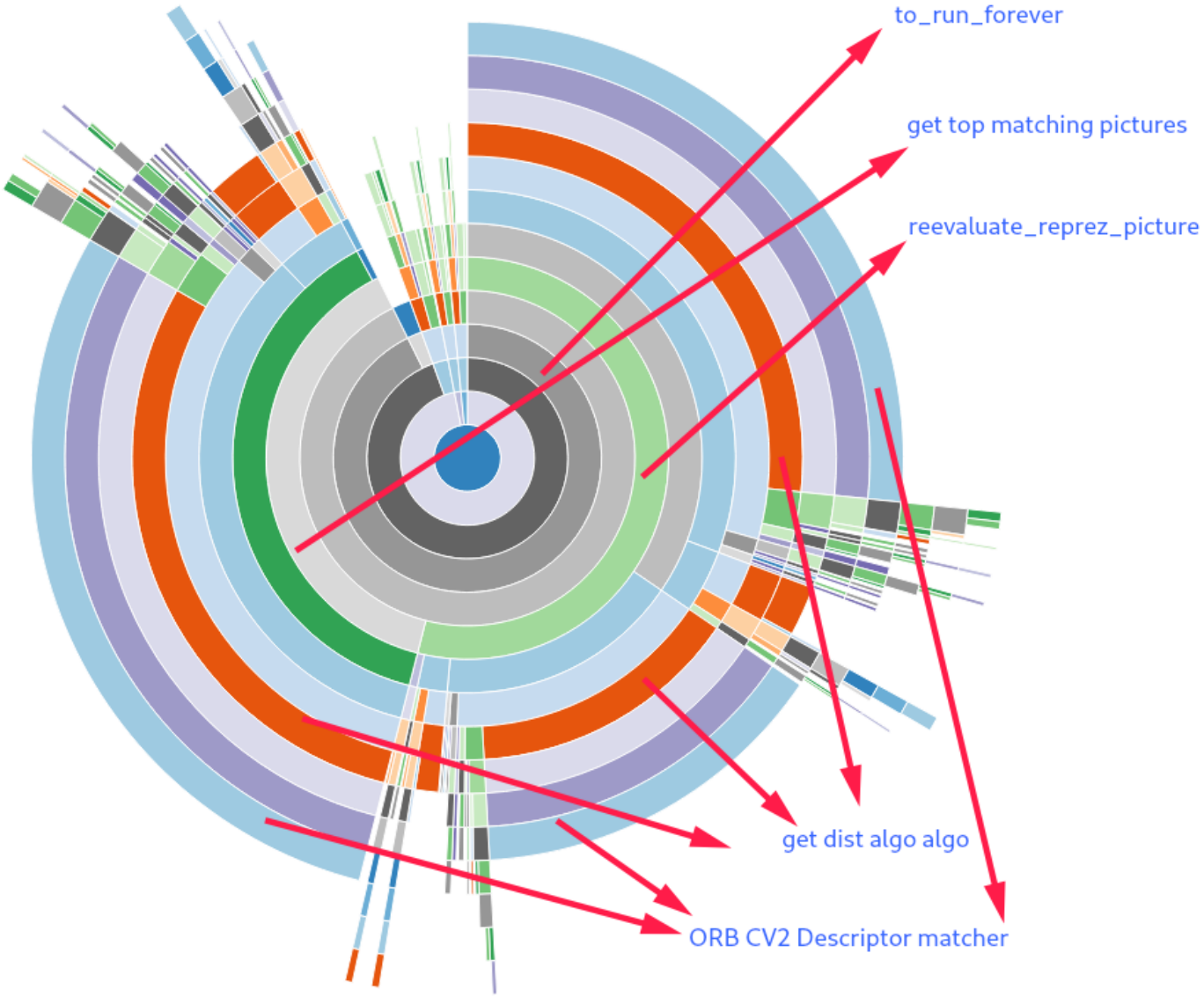


(c) Time to add pictures in database - normalization and no reference line



(d) Time to add pictures in database - normalization and reference line

Figure 10.8: Time performance to add pictures



(a) Time was mainly consumed by two things : re-evaluate representative picture of a cluster in O(nb pics in this cluster) complexity and the time taken by ORB evaluation directly in OpenCV2.

Figure 10.9: Cluster creation threshold at 0 (worst case) - Time repartition in database adder - CProfile of DataBase Adder Worker displayed with Sphinx

Speed is an issue, and was at test step sometimes even worse than a linear lookup in the database. How to explain that ? The scalability evaluator extract few metrics, which are shown in Frame 10.1. We notice that the reparition of cluster size if very inequal. Which means that we have a "*long tail distribution*" of cluster sizes : few very big clusters and numerous small clusters.

This generates a very inefficient datastructure. Assuming - this is somewhat proven by the performance results at this time - that biggest clusters tends to grow even bigger, we can depicts why the lookup is slow. It follows such pattern :

- We iterate over all clusters's head (representative pictures). As we have a lot of small clusters, we iterate over 7 clusters (following numbers from Frame 10.1).
- We choose best clusters among these evaluated clusters. As we made the hypothesis that biggest clusters tends to become bigger (reason unknown, but proven by experience), we can surely keep only the biggest clusters (247, 21 and 1 sizes).
- Each of the pictures in the cluster is evaluated. Here, $247 + 21 + 1$ comparisons are performed.
- Only the top N (usually 10) pictures is kept.

In total, in this scenarios, we performed $247 + 21 + 1 + 7 = 276$ comparisons, 3 more than a standard linear search (273 comparisons).

Listing 10.1: Performance Evaluation for one threshold and one number of pictures in database


```
{
    "feature_time": null,
    "adding_time": 347.4904863834381,
    "request_time": 0.00011372566223144531,
    "nb_picture_added": 96,
    "nb_picture_total_in_db": 273,
    "nb_picture_requested": 0,
    "iteration": 4,
    "nb_clusters_in_db": 7,
    "clusters_sizes": [
        1,
        1,
        1,
        1,
        1,
        21,
        247
    ]
}
```

So, what is the perfect target of pictures repartition and cluster number to have a best time ? Some quick maths :

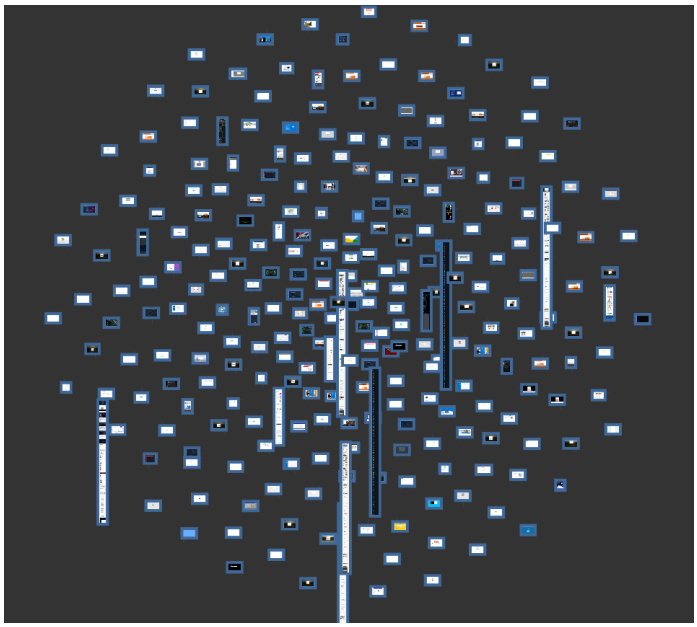
- The number of comparisons is defined by : *number of clusters+number of pictures in chosen clusters* considering we keep only one cluster.
- Let's note X as the number of clusters and Y as the number of pictures in these clusters (making the hypothesis that all clusters should have the same size, as we have equal probability to ends in one or the other. We do not consider a probabilistic approach) and S the number of comparisons performed. Then we have a first equation : $X + Y = S$
- We have the relation $X * Y = N$ with N being the total number of pictures in the stored dataset. We have this relation because each picture is attributed to only one cluster.
- We can then solve this very tiny system for X, minimizing S. We consider the function $f(X) = X + Y = X + \frac{N}{X}$ *when* $x! = 0$
- To find the extrema, we derivates f : $f'(X) = 1 - \frac{N}{X^2}$
- We solve the derivated function for 0 : $1 - \frac{N}{X^2} = 0 <==> X^2 = N <==> X = \sqrt{N}$ *or* $X = -\sqrt{N}$
- We obviously keep the positive solution : $X = \sqrt{N}$. That is to say that the "perfect situation" would be to have $\sqrt{N}$ clusters in database with $\sqrt{N}$ pictures in each

Let's see what would be the gain in such perfect scenario :

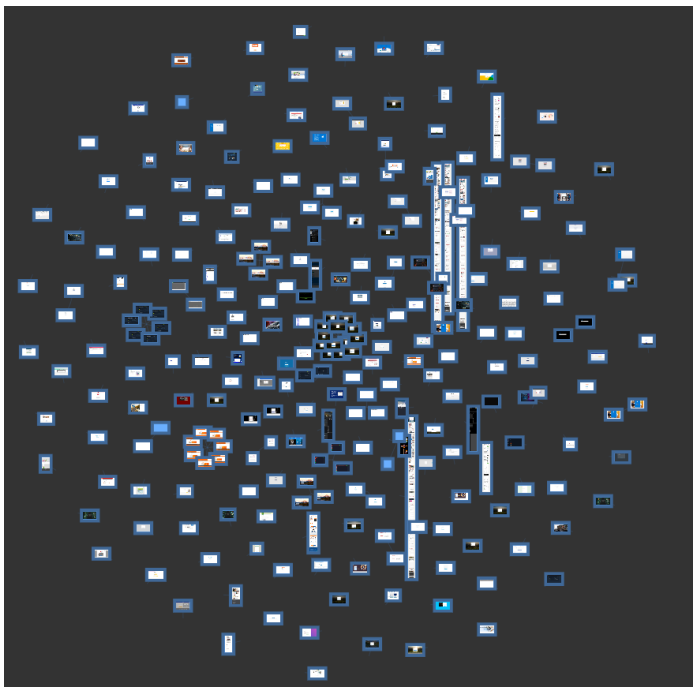
- We iterate over all clusters's head (representative pictures) : $\sqrt{N}$ which would here be $\sqrt{273} = 17$
- We choose best clusters among these evaluated clusters. We keep the 3 biggest clusters (17, 17 and 17)
- Each of the pictures in the cluster is evaluated. Here, $17 + 17 + 17$ comparisons are performed.
- Only the top N (usually 10) pictures is kept.

In total, in this scenarios, we performed $17 * 4 = 68$ comparisons, 205 less than a standard linear search (273 comparisons). Such perfect case would take 25% of time of a linear look-up.

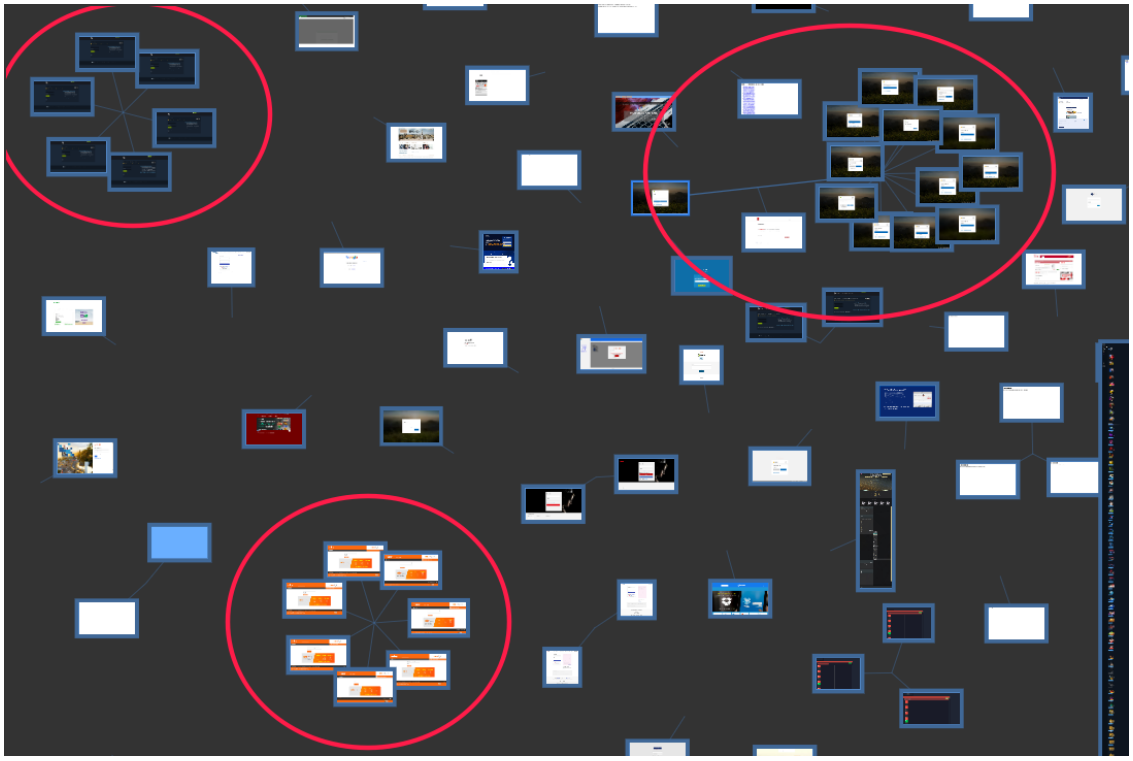
Figures **??**, **??**, **??**, **??** are presenting storage graph as they evolve with internal threshold "*MAX DIST FOR NEW CLUSTER*". We see that few big clusters emerges, but are homogeneous and "good". However, after Figure **??**, we see a big, totally wrong, cluster. This cluster generates a high number of calculation for "nothing". Please note, that in this configuration, we only check the matching between THE most representative picture of the cluster and the picture to be added.



(a) Global view of the clusters with threshold at 0.0



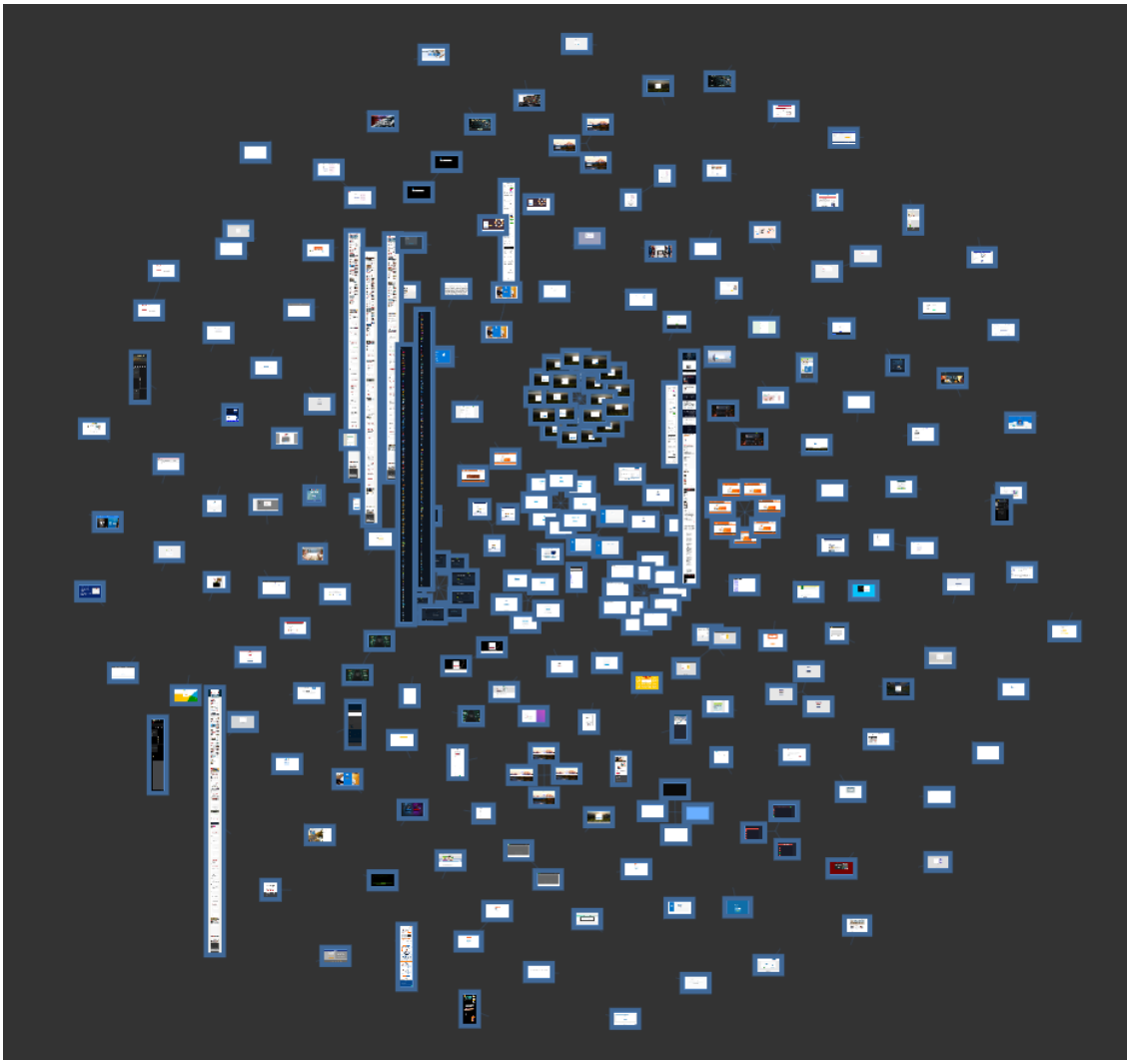
(b) Global view of the clusters with threshold at 0.0195



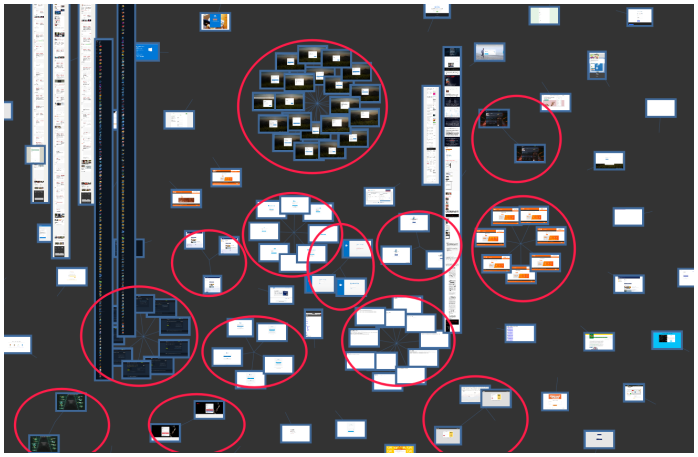
(c) Main clusters with threshold at 0.0195

Figure 10.10: Threshold at 0.0 and 0.0195 - Storage graph view - 1 picture checked

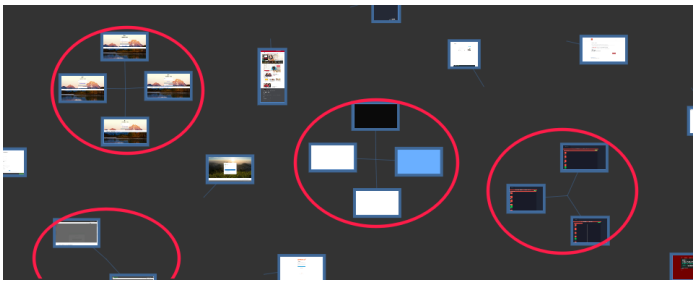
Figures **??**, **??**, **??** are also presenting storage graph as they evolve with internal threshold "*MAX DIST FOR NEW CLUSTER*". Please note, that in this configuration, we check matching between the three first best representative pictures of the cluster and the picture to be added. We see that even if setting more pictures to test within one cluster to have a match with this cluster, does not modify



(a) Global view of the clusters with threshold at 0.039



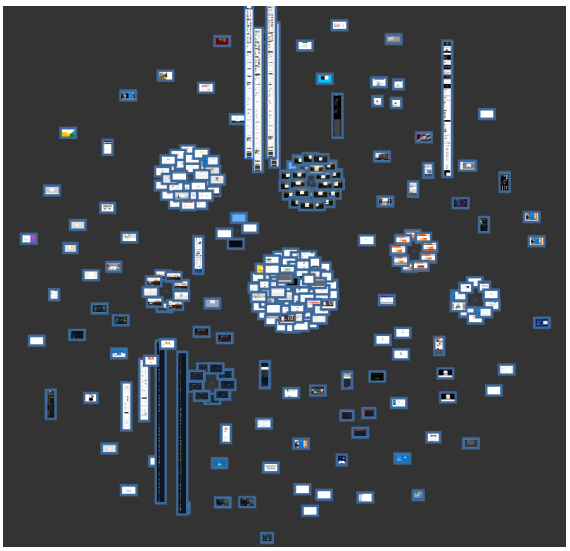
(b) Main clusters with threshold at 0.039



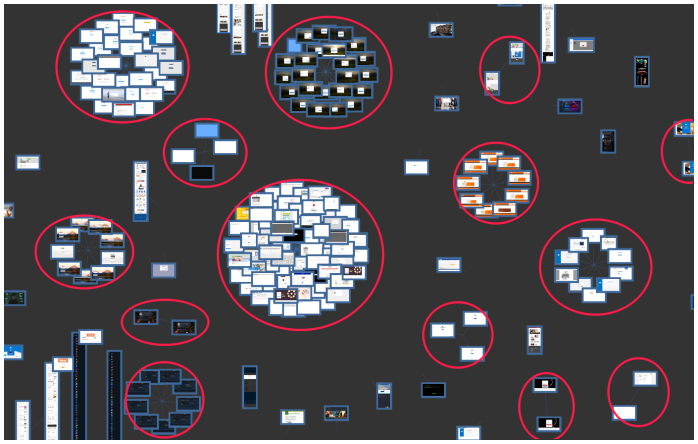
(c) Smaller clusters with threshold at 0.039

Figure 10.11: Threshold at 0.039 - Storage graph view - 1 picture checked

deeply the behavior of the library. However, it "push further" the distance at which the problematic behavior happen.



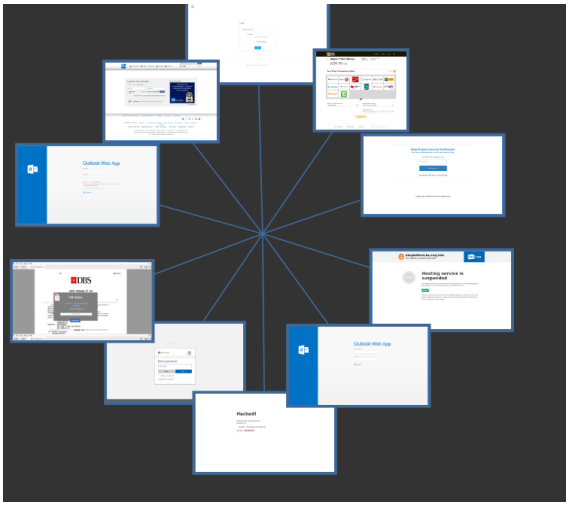
(a) Global view of the clusters with threshold at 0.058



(b) Main clusters with threshold at 0.058

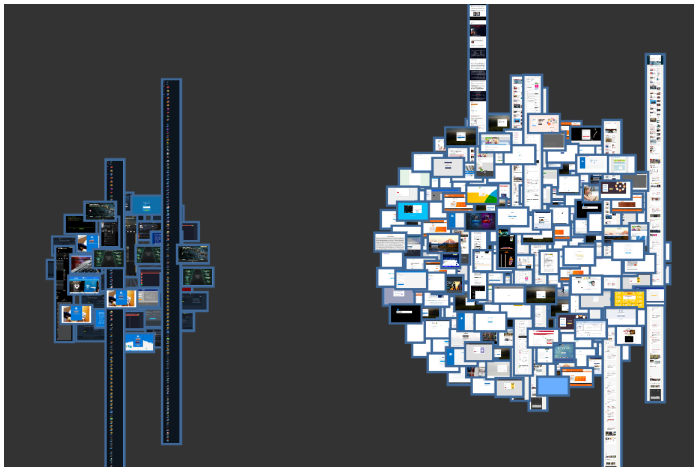
(c) Biggest -wrong- cluster with threshold at 0.058

(d) Biggest clusters with threshold at 0.058



(e) Biggest clusters with threshold at 0.058

Figure 10.12: Threshold at 0.058 - Storage graph view - 1 picture checked



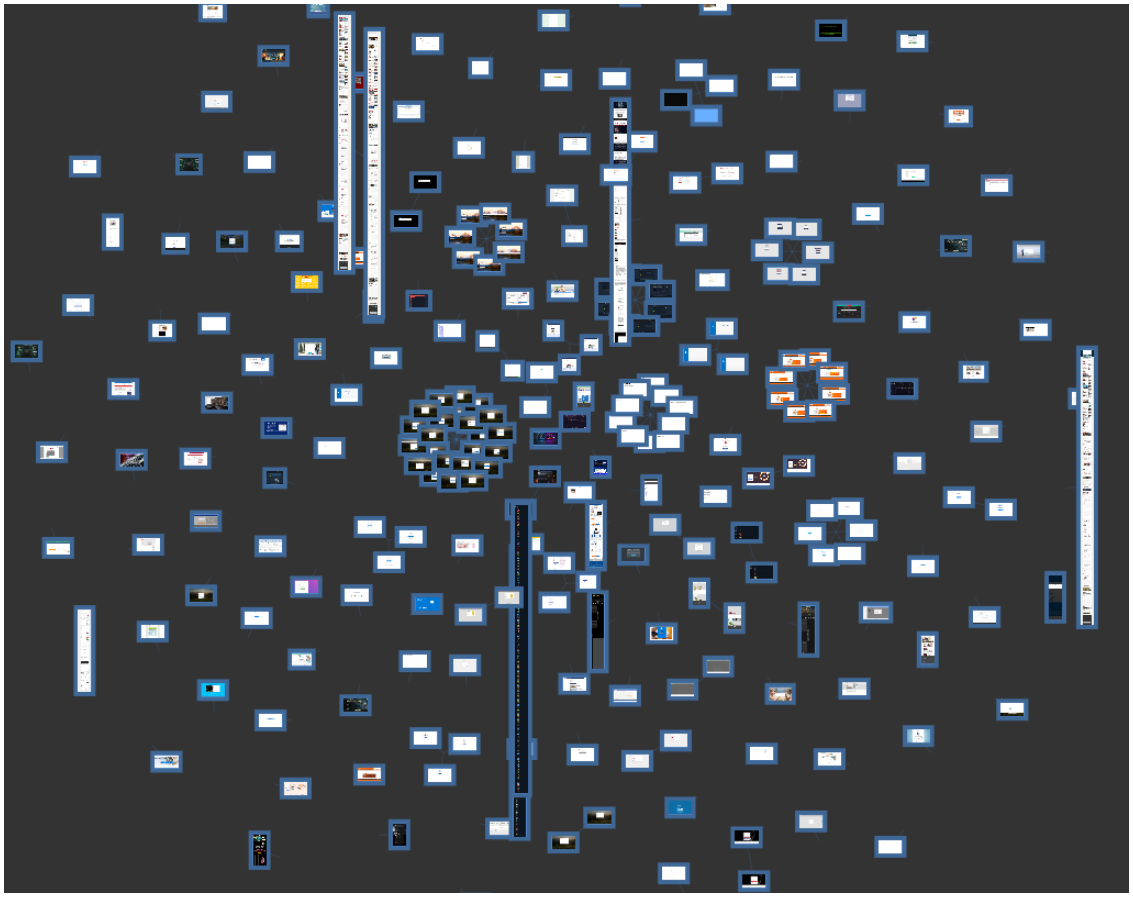
(a) Global view of the clusters with threshold at 0.25

Figure 10.13: Threshold at 0.25 - Storage graph view - 1 picture checked

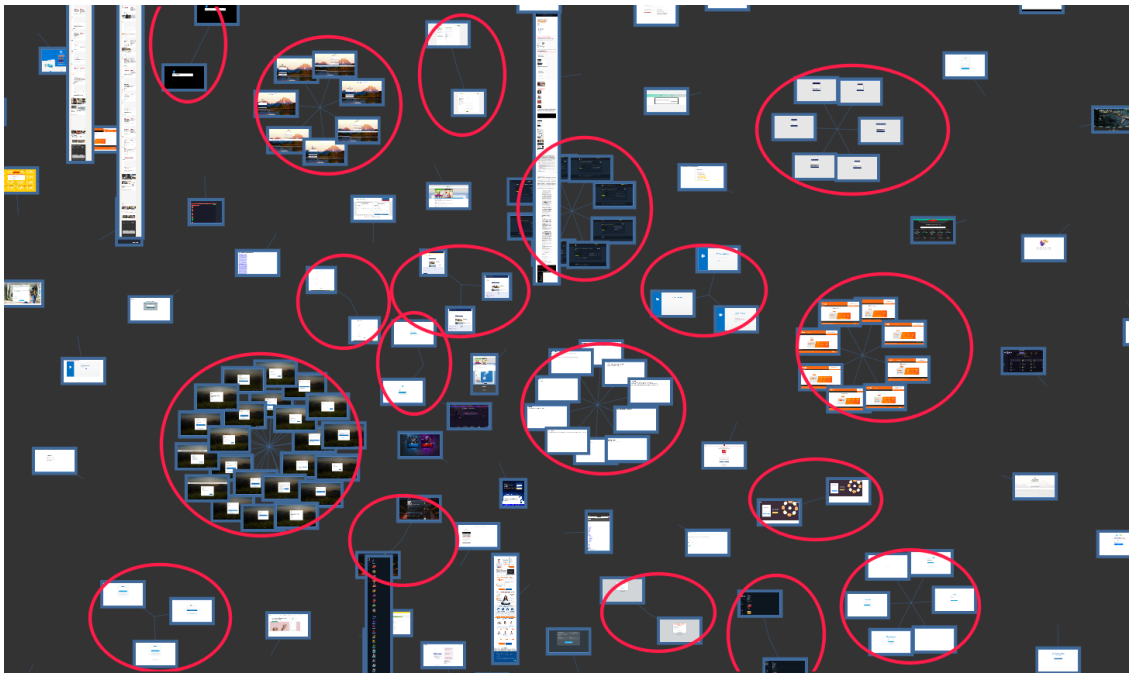
Few solutions can be considered :

- Implementing the re-evaluation queue (a picture added to the database will be re-evaluated after X other addition)
- Deeply changing the way the library store its pictures. Using a tree, or a hash table. Something really good, but restrictive with the way we could compute features of pictures, is by using FLANN-databases-like (based on hash-table with vector prefix). But this would imply that any value store can be sorted (alphabetically, for instance)
- Statistically, if a cluster is bigger, it seems legitimate that a picture can "fall into this one". As the cluster is bigger, there are more potential pictures with which it could match. Therefore, ensuring that no cluster "overgrow" could be a solution. (Without precise good idea on how to enforce it)

Evaluate the parameters

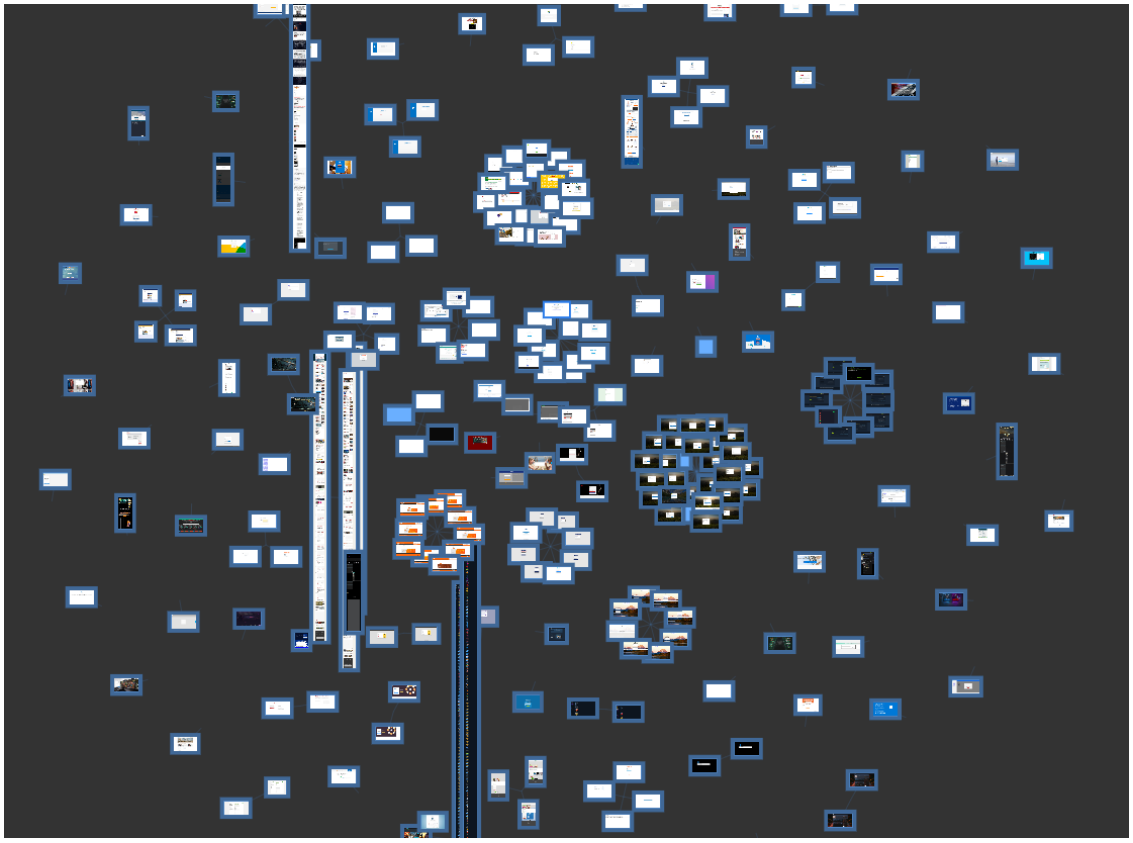


(a) Global view of the clusters with threshold at 0.039

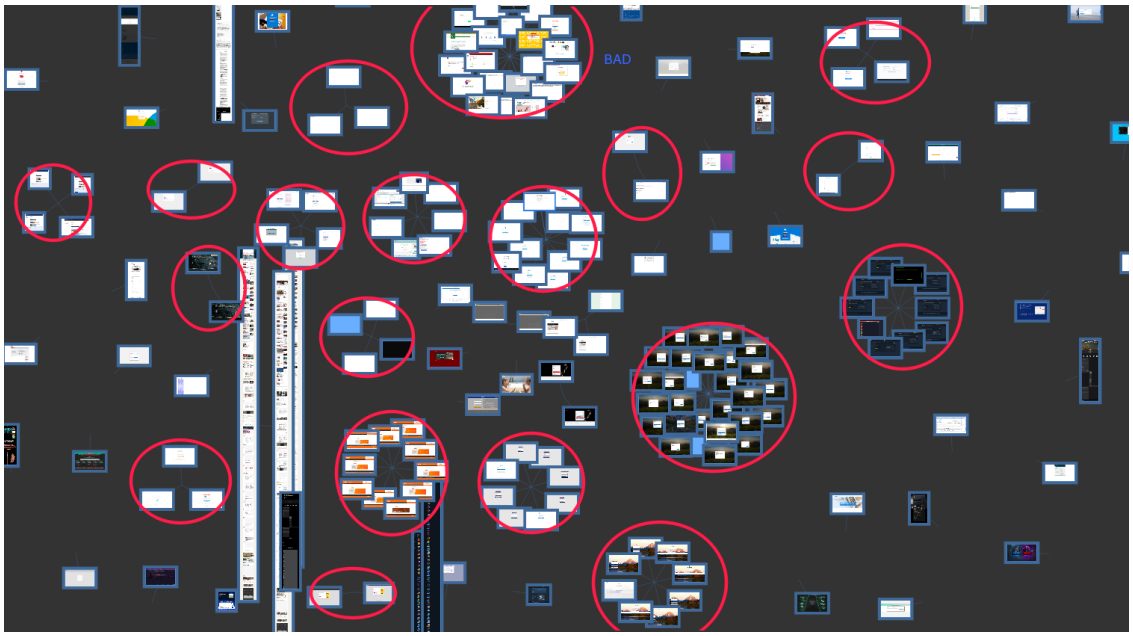


(b) Main clusters with threshold at 0.039

Figure 10.14: Threshold at 0.039 - Storage graph view - 3 pictures checked

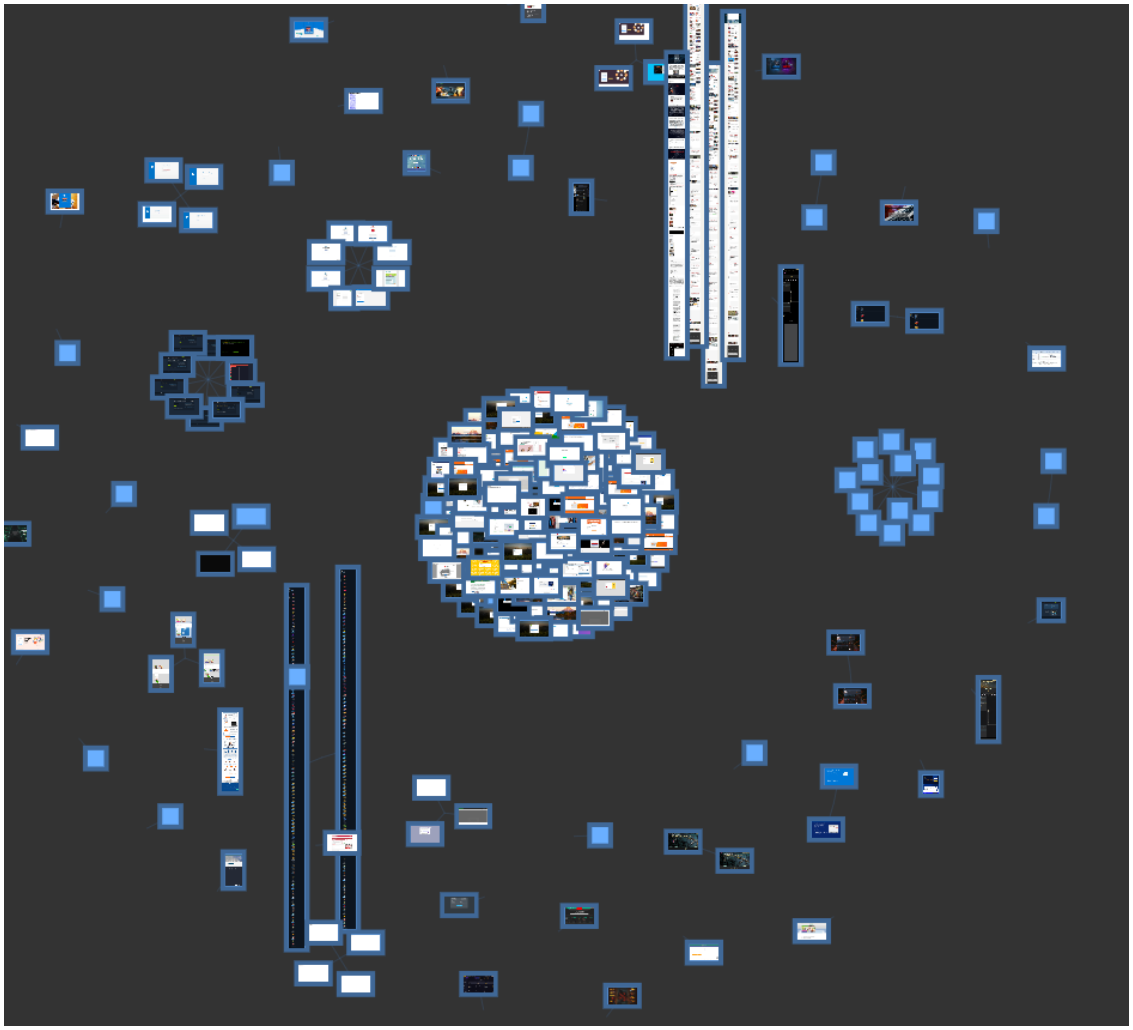


(a) Global view of the clusters with threshold at 0.058

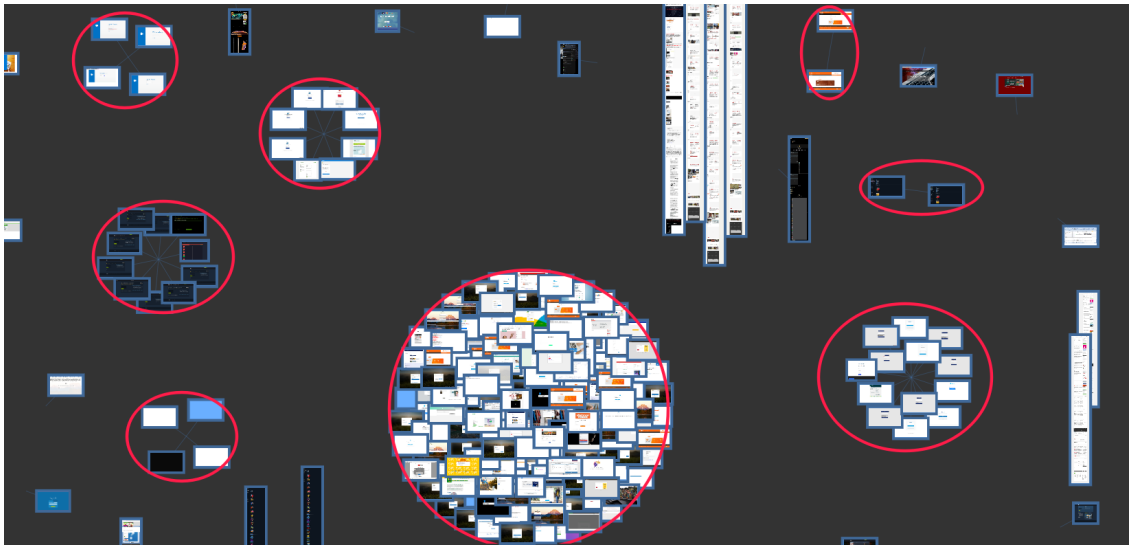


(b) Main clusters with threshold at 0.058

Figure 10.15: Threshold at 0.058 - Storage graph view - 3 pictures checked



(a) Global view of the clusters with threshold at 0.078



(b) Main clusters with threshold at 0.078

Figure 10.16: Threshold at 0.078 - Storage graph view - 3 pictures checked

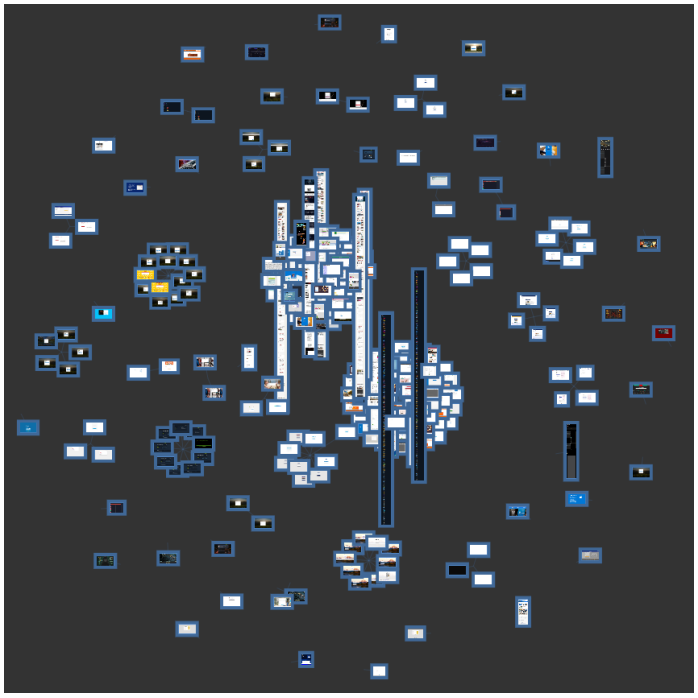
Findings

A few findings were made :

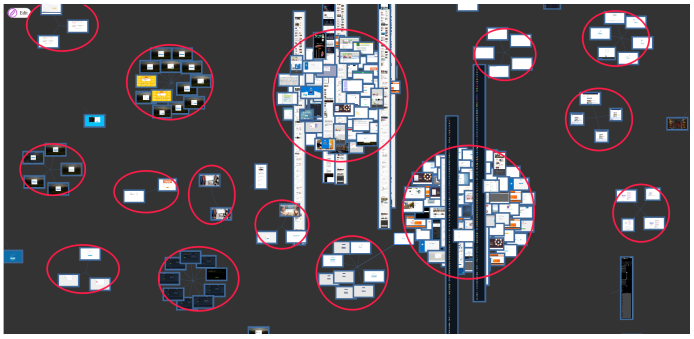
- There is a "long tail" repartition of the clusters sizes. Very few big clusters, a lot of very small clusters.
- Statistically, the bigger a cluster is, the more probable it is to attract new pictures. The bigger it is, the bigger it become (depends on the exact method to attach pictures to clusters, however.)
- Some repartition of clusters/pictures in cluster can be worst that linear.
- The Matching quality (algorithms quality performances) directly influences the speed performances. If algorithms are good to cluster pictures together, they can produce relevant clusters in database, and so, generate good performances.

When we want to add a picture to the database, we get two list : a list of pictures matches (regardless of their cluster) and a list of cluster matches (only with N first representative pictures). Methods to create or not new clusters are :

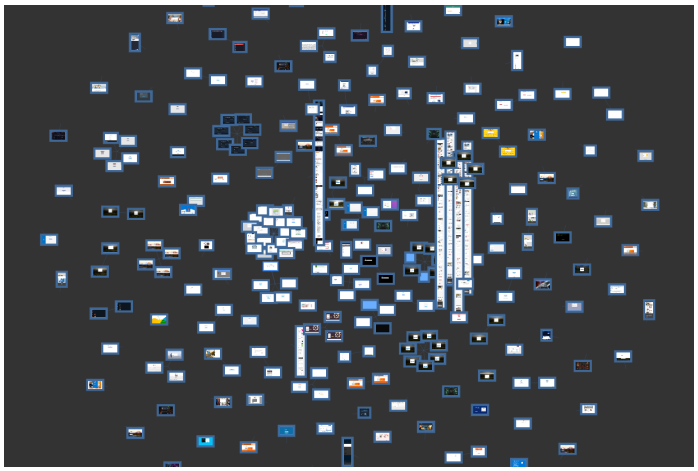
- Simply veryfying if the best match of the list of pictures matches is good enough. If yes, put the picture in the same cluster. Otherwise, put in a new cluster. We can check for 1 or 3 most representative pictures per cluster.
- Check clusters matches : the first cluster with a distance below threshold, and with a "YES" decision is taken. Otherwise, we put the picture in a new cluster. We can check for 1 or 3 most representative pictures per cluster.
- Check the pictures matches and normalize distances : the first picture with a distance + (relative size of its cluster to square-root of (nb pictures in database)) is below threshold, and with a "YES" decision is taken. Otherwise, we put the picture in a new cluster.



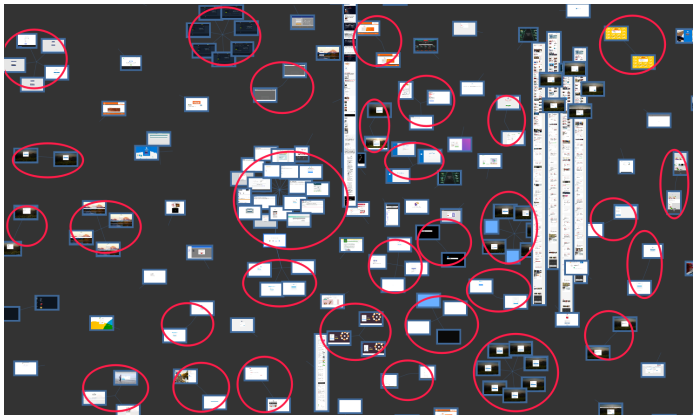
(a) Overview - Threshold at 0.35



(b) Main clusters - Threshold at 0.35



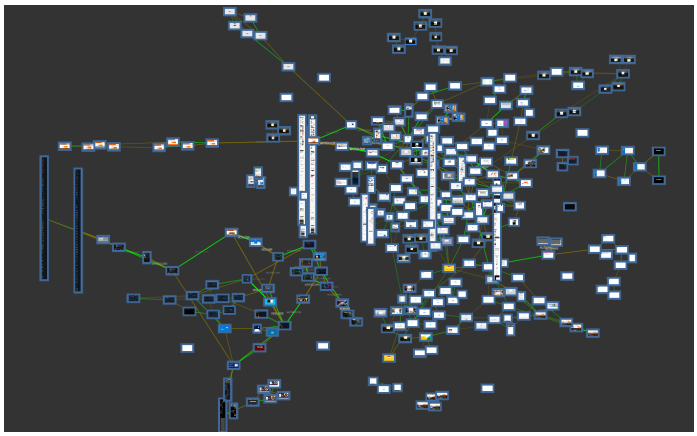
(c) Overview - Threshold at 0.1755



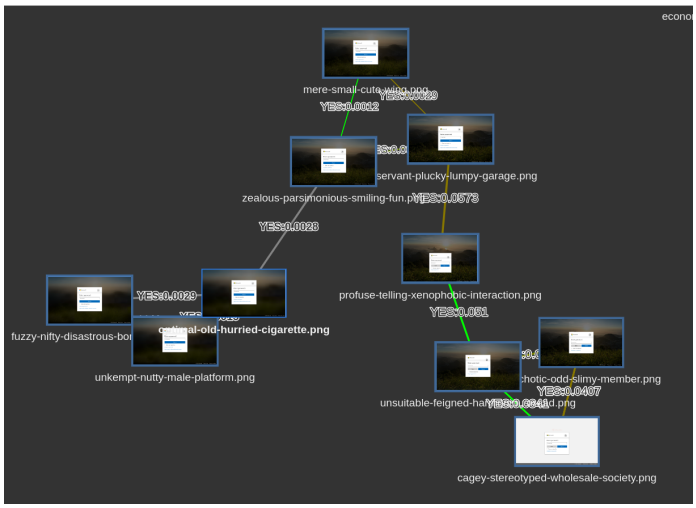
(d) Main clusters - Threshold at 0.1755

Figure 10.17: Storage graph view - 1 most representative picture checked per cluster

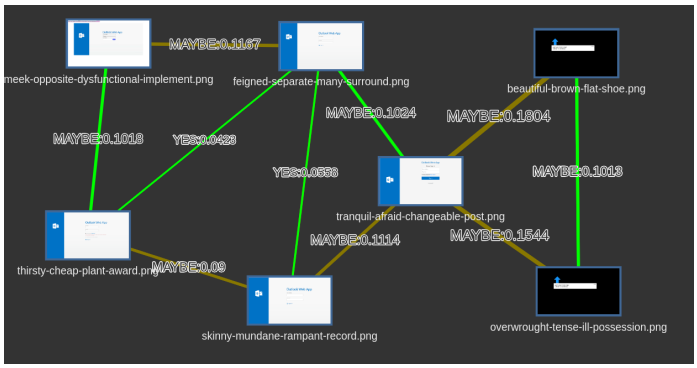
10.1.3 BoW ORB - Bag Of Words Approach



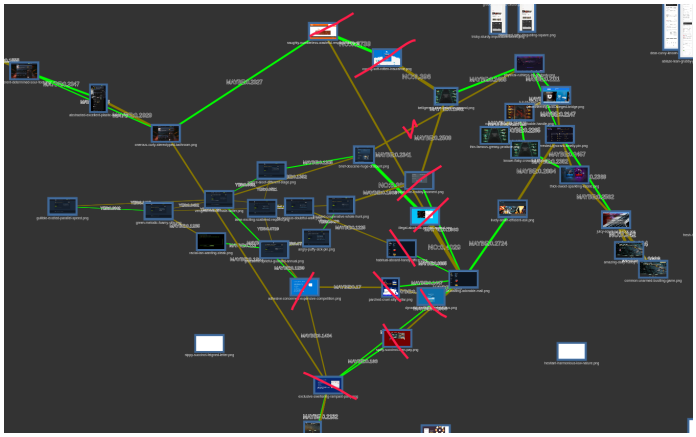
(a) Overview - No threshold (at 1)



(b) Microsoft cluster - No threshold (at 1)

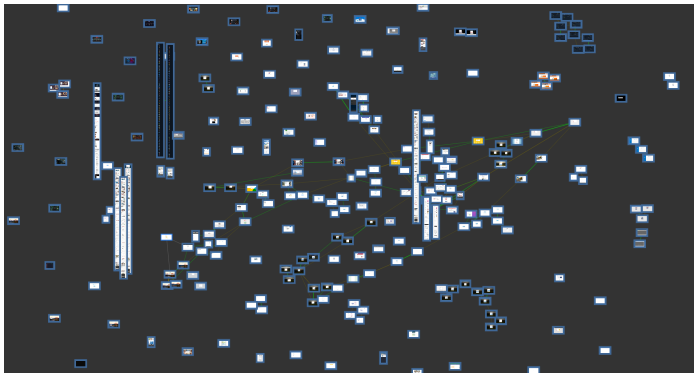


(c) Outlook cluster - No threshold (at 1)



(d) Steam cluster - No threshold (at 1)

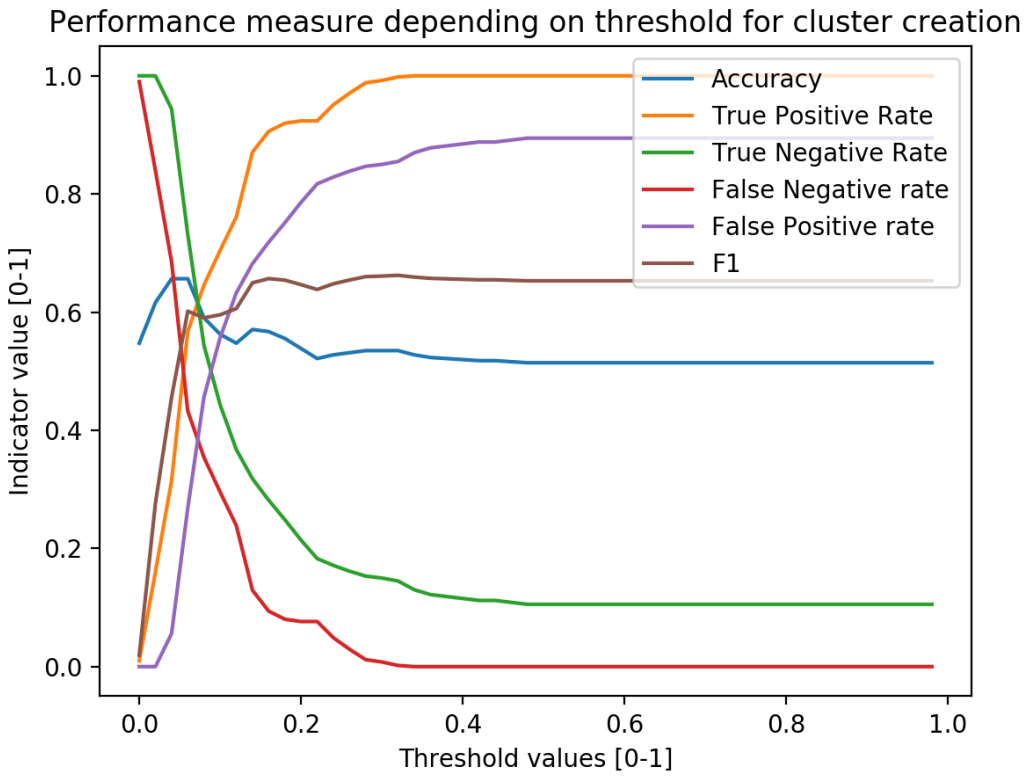
Figure 10.18: Similarity Graph view - BoW ORB with a 100 elements (bad) vocabulary/dictionnary



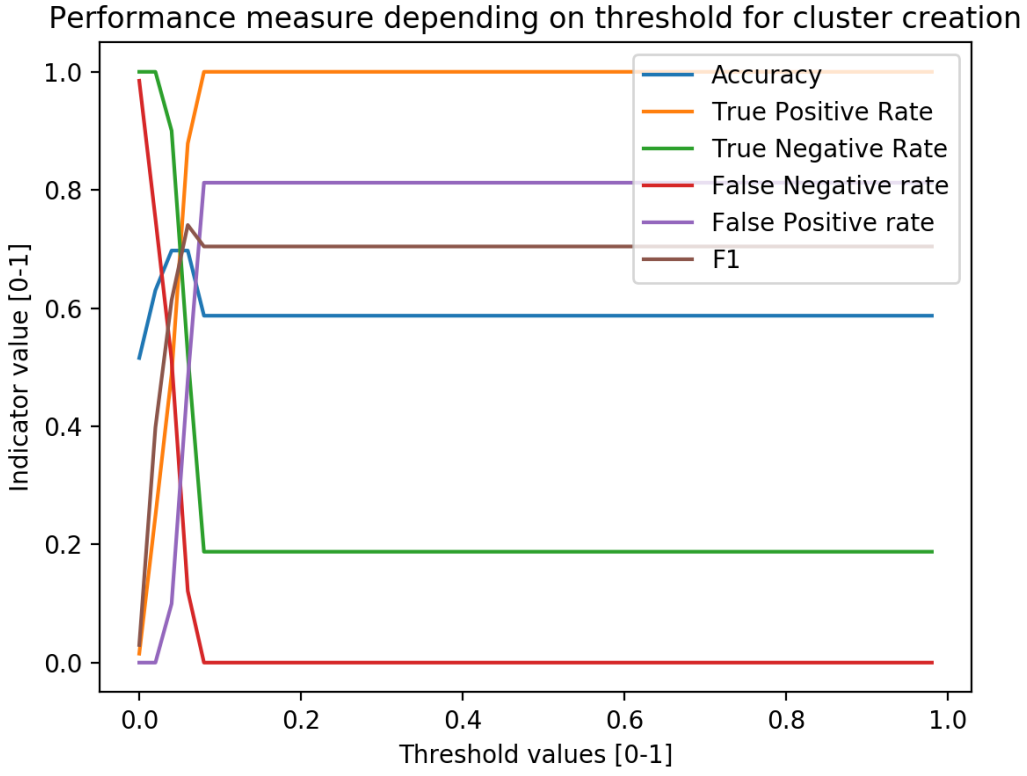
(a) Overview - No threshold (at 1)

(b) Cluster examples - No threshold (at 1)

Figure 10.19: Similarity Graph view - YES matches only - BoW ORB with a 100 elements (bad) vocabulary/dictionnary

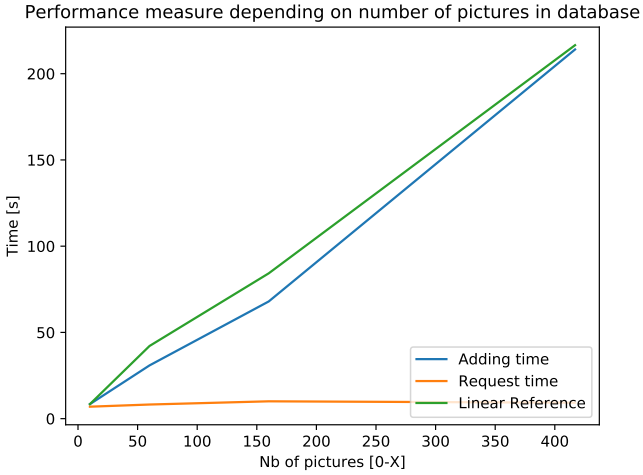


(a) Quality with Yes and MAYBE and NO matches of the configuration

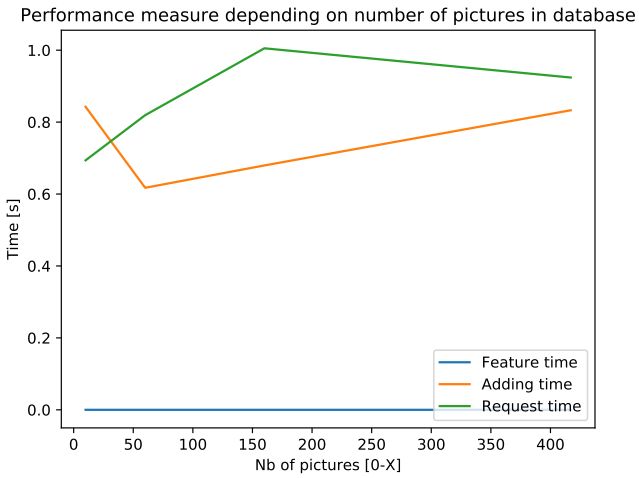


(b) Quality with Yes matches only of the configuration

Figure 10.20: Similarity Graph view - BoW ORB with a 100 elements (bad) vocabulary/dictionnary



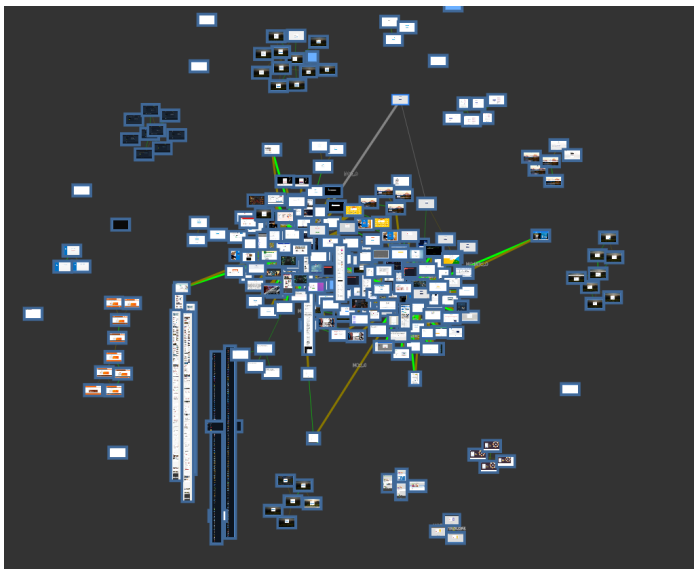
(a) Not normalized (increment is normal, as each time we are adding more pictures)



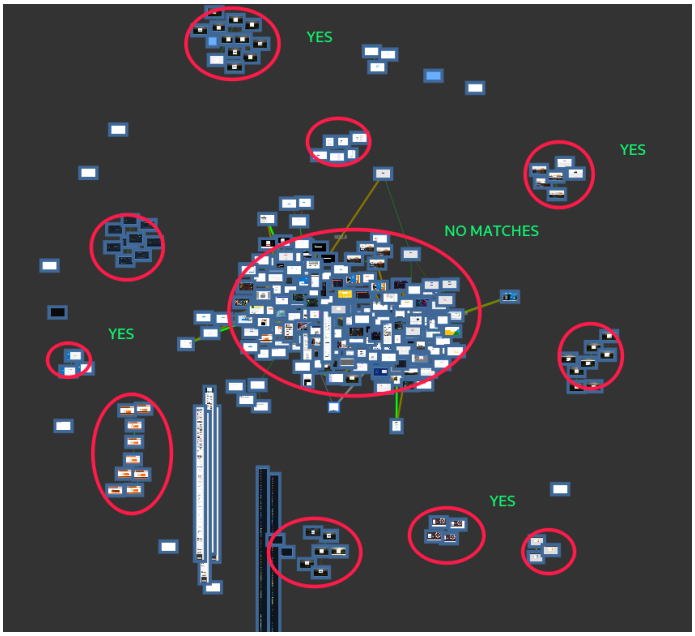
(b) Normalized by picture added

Figure 10.21: Scalability measures

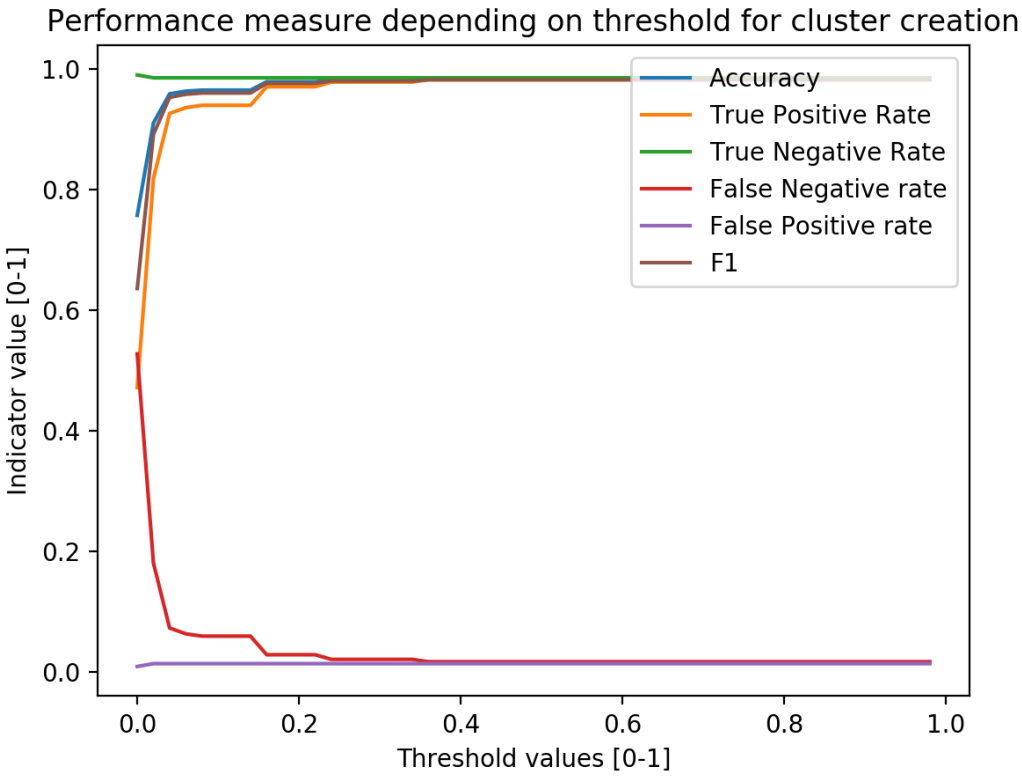
10.1.4 RANSAC ORB



(a) Overview - No threshold (at 1)

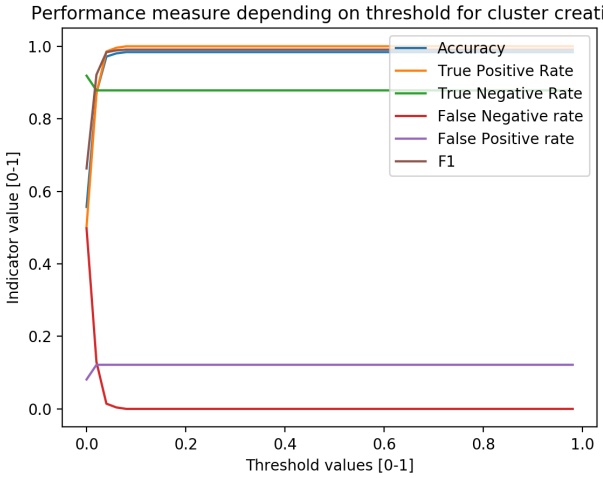


(b) Clusters - No threshold (at 1)

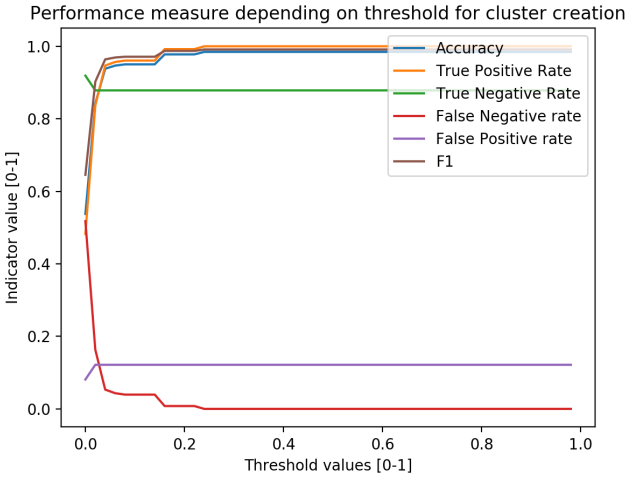


(c) Overview of quality measures relative to ground truth file

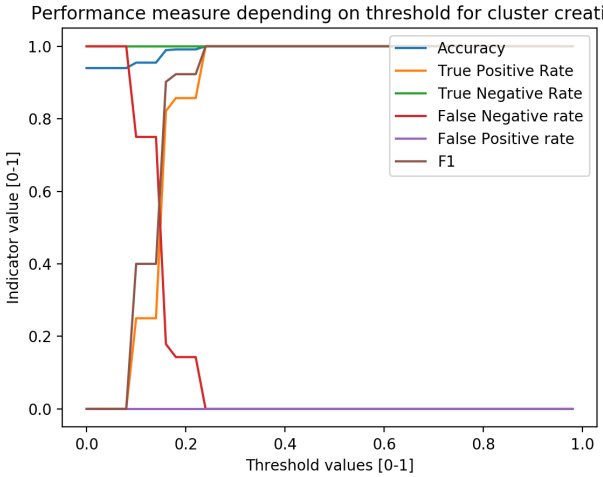
Figure 10.22: Similarity Graph view - RANSAC ORB



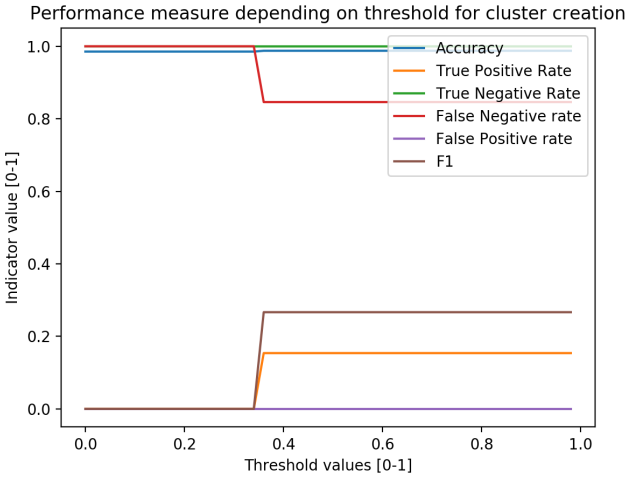
(a) Quality measures - YES only



(b) Quality measures - YES and MAYBE only

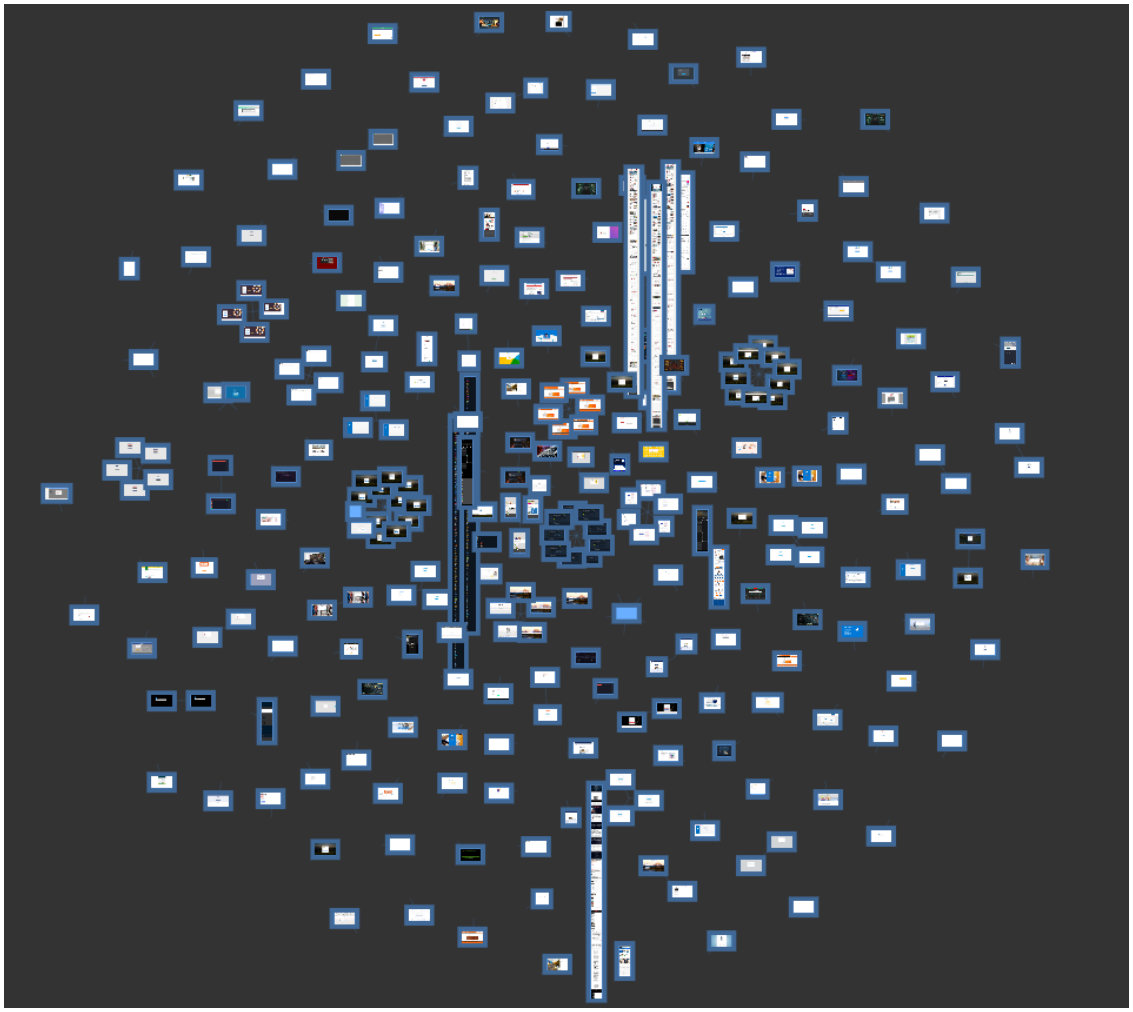


(c) Quality measures - MAYBE only



(d) Quality measures - NO only

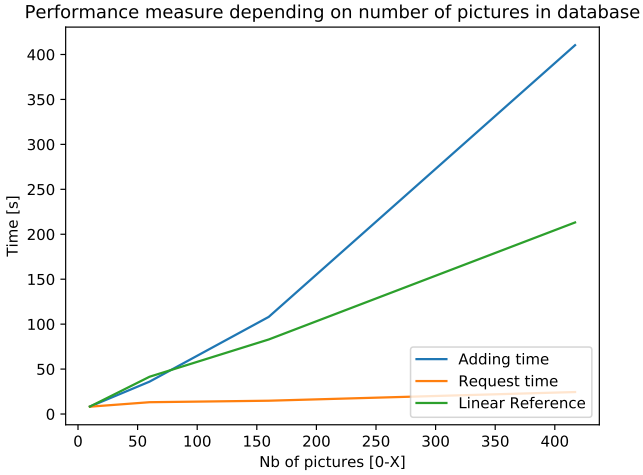
Figure 10.23: Similarity Graph view - RANSAC ORB



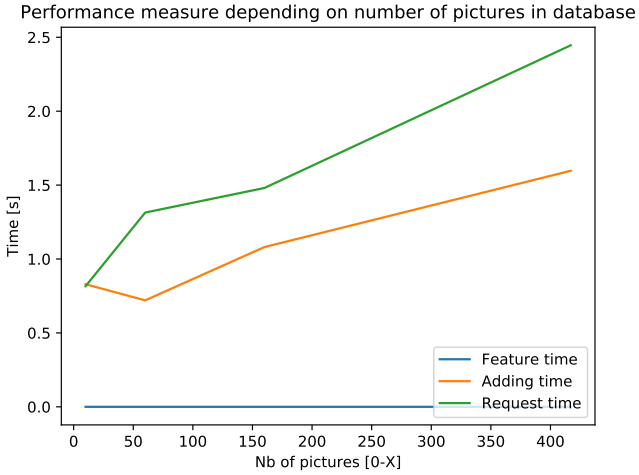
(a) Overview

(b) Clusters

Figure 10.24: Storage Graph view - RANSAC ORB



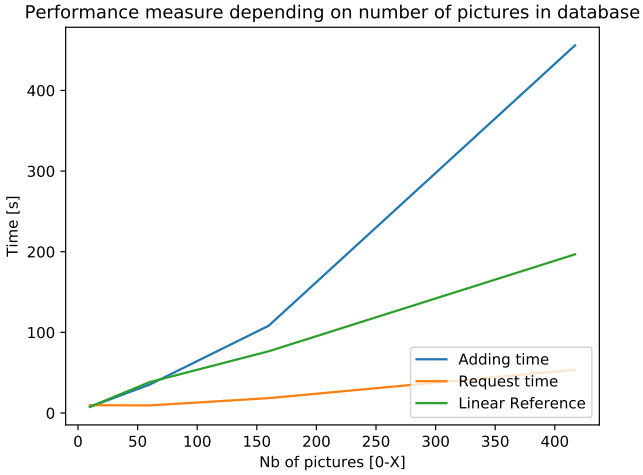
(a) Not normalized (increment is normal, as each time we are adding more pictures)



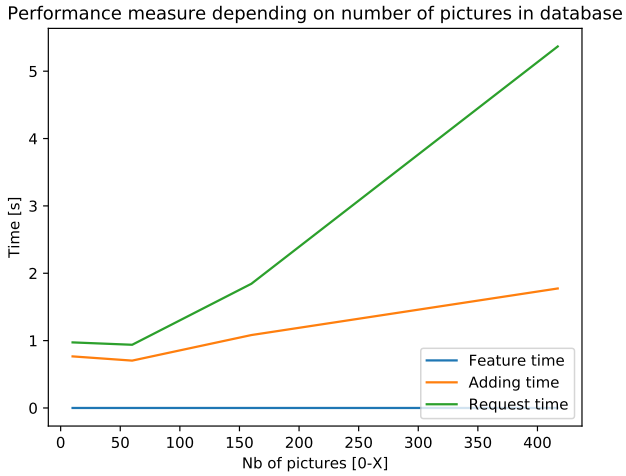
(b) Normalized by picture added

Figure 10.25: Scalability measures

10.1.5 Algos combination



(a) Not normalized (increment is normal, as each time we are adding more pictures)



(b) Normalized by picture added

Figure 10.26: Scalability measures

Chapter 11

Tips and tricks

11.1 For devs

You want to see how to add a new algorithm to the library ? Get a look at `https://github.com/CIRCL/douglas-quaid/commit/9942a004f87d79c0b7cecc177dd165de1f3514c4`. See also `https://github.com/CIRCL/douglas-quaid/commit/364898911d9171edaf6b8403427302ca528993a3`. These should act as nice commits that show you what to add and where.

Roughly, steps are :

- Create a *picture_YOUR_NEW_ALGO* file that computes the features from one picture
- Edit *feature worker* to call this new *picture_X* computation
- Create a *distance_YOUR_NEW_ALGO* file that computes the distance between two set of features
- Edit *distance engine* to call this new *distance_X* computation
- Edit *configuration files* to store what you need/can be modified. If an option is obviously better than an other one, hardcode it and do not put it in configuration file.

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.