# Network Analysis in ArangoDB

Alessandro d'Agostino, Mattia Ceccarelli, Riccardo Scheda

January 2020

**Abstract**

The project aim is to develop a simple python API that let the user interact with the open-source multi-model database ArangoDB to perform fast queries on multi-partite graphs, extract and visualize sub-network which can then be analized with powerful python libraries. The study also includes results on the computational efficiency of ArangoDB's algorithms, in order to have an estimate of the API timing behaviours with large networks.

## 1 Introduction

ArangoDB [1] is a multi-model, open-source database with flexible data models for documents, graphs and key-values. It is being used in different fields: from semantic analysis to genome studies.

In the present work we employed the library `python-arango` [2] to import and export multi-partite network-like databases, that are defined as collections of documents (nodes) and collections of relationships between them (edges).

In particular, a document is a python dictionary with a mandatory `_id` key assigned to a unique value. What we call an "edge" is also a python dictionary, but with an `_id` , `_source` and `_target` voices, to specify not only the linked nodes, but also the direction of said link in the case of directed graphs.

**Graph Object** A Graph is the combination of at least one node collection and one or multiple edge collections, called edge definitions. With a Graph two main operations are available: visualization and execution of queries.

The former is possible thanks to ArangoDB web interface, as shown in the figures below, which provide the user many options such as the starting node and the number of elements to load. By default, the starting node is random.

For the latter we relied on the poweful AQL, that is the query language implemented in arango. It is used to execute different kind of "search" through the graph, and select nodes and edges.

**Why ArangoDB?** ArangoDB resulted to be the NoSQL engine with the most favorable compromise between memory usage and computation speed among others available alternatives. A comparative table of the performance of different NoSQL engines is reported in figure 1 ArangoDB offers a valid web interface where all the commands can be given intuitively and graph and queries' results can be visualized in an organic way. Furthermore there are several ArangoDB Python interfaces available on line, which allowed us to taylor this poweful tool to

our needs.

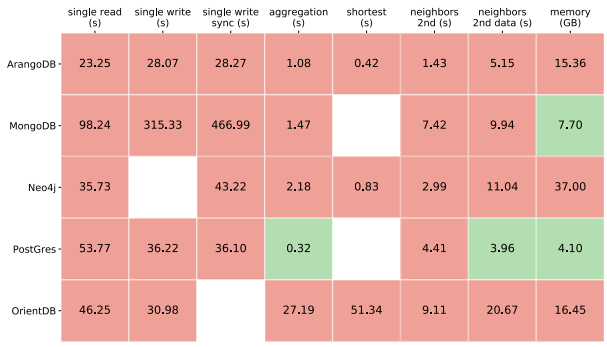


Figure 1: *Comparison between time performances of different NoSQL engines for some common operation. ArnagoDB shows the overall best results.*

**NetworkX** Even though ArangoDB allows fast queries it is necessary the right tool to study the characteristics of a resulting subnet. For this purpose we decided to exploit mainly the Python library NetworkX [3] due to its huge availability of function and tools.

## 2 Pipeline

Now we describe a typical usage case and all the operations needed to extract a subgraph in ArangoDB. The first thing to do is to read the file in the right input format (typically `.gexf`) through the function `read_gexf`:

```
ArDB_Net, Nx_Net =
  read_gexf(db,
            filename,
            nodes_collection_name='nodes',
            edges_collection_name='edges',
            graph_name='Net',
            multipartite=False)
```

The function takes the information about nodes and edges from the gexf file and creates the corrispondent collections on the ArangoDB engine. Then it returns two graphs: one python-arango type and one of networkx type.

Once the graph and the collections of nodes and edges are uploaded on Arango web interface we can proceed with futher analysis. For instance, we can perform a graph traversal aimed to extract the first neighbours of a certain node.

In this case we can make use of python-arango, calling the function:

```
first_neighbours = traverse(db,
        starting_node,
        nodes_collection_name,
        graph_name,
        direction='outbound',
        item_order='forward',
        min_depth=0,
        max_depth=1,
        vertex_uniqueness='global')
```

Tuning the given parameters, any traversal could be performed. The output of this type of function is meant to be the resulting sub net saved as Python dictionaries containing vertexes and paths crossed by the traverse.

Having the list of the first neighbours we can obtain the corresponding subnet just using the function `subgraph` given by Networkx:

```
Nx_Sub_Net =
        Nx_Net.subgraph([vertex['label']
        for vertex in
        first_neighbours['vertices']])
```

In this way we create the minimal subgraph containing only the nearest neighbours of the selected node with the labels of nodes and the corresponding edges. To recover every information

contained in the starting network, we add all the attributes back to each node:

```
for node in Nx_Sub_Net:
  attr = pa.get_vertex(db,
                       {'label':node},
                        'Sym_Deas')
  nx.set_node_attributes(Nx_Sub_Net,
                         {node : attr})
```

The last thing to do is to import the subnet in ArangoDB, using the two specific functions `nx_to_arango` and `export_to_arango`, for this passage is easier to convert the data in the node_link_data format:

```
sub_net =
  nx.readwrite.node_link_data(Nx_Sub_Net)
sub_net =
  pa.nx_to_arango(sub_net, 'Sub_Net')
Sub_Net = pa.export_to_arango(db,
                  msub_net,
                  'Sub_Net',
                  'Sub_Net_edges',
                  'Sub_Graph')
```

Now the resulting subnet is ready for every kind of analysis offered by NetworkX. This is a useful procedure to extract articulated subnets even on big non-relational base.

# 3 Timing

As already mentioned, one of the main focuses is to apply queries to very big networks. Therefore a study on how the time spent to perform a query scales with graph's dimension was necessary. Hence we chose two different queries to be performed on several graphs with similar characteristics but increasing dimensions.

The queries taken into consideration are the traversal that extract the second neighbours (`min_depth=0 max_depth=2`) and the first four shortest path between two random nodes .

We created random graphs with a number of nodes ranging from 100 to 10000, using the NetworkX function `fast_gnp_random_graph`. One important factor is the number of links: infact, a dense networks with such an high number of nodes is impossible to handle for a home computer. So, we chose to generate random graphs with $\rho = 0.075$, which comes from the analysis of the real case we tested the algorithms on.

We runned the queries under study onto 30 different starting nodes, in order to obtain an average value, its deviation and a idea of its trend.

# 4 Results

In the following section, we present the data obtained for timing, and an example on the visualization of a network and a sub-network through ArangoDB web interface.

**Performances** In figure 2 we present the trend of times with respect to number of nodes, for the algorithm that find the 4 shortest paths between two nodes. We performed a linear fitting on the sampled data and a linear relationship emerged in this case, with $R^2 = 0.957$.

In figure 3, we show the trend of times with respect to the dimensions of the network for the traversal that extract the second order neighbours of a chosen node. We performed a linear fitting on the sampled data and it seems to show a linear relathionship also in this case, with $R^2 = 0.94$.

From the analysis of both the two different Benchmark series we found that the time took

by the queries apparently scales linearly with the graph dimension. This is a useful information and it allows to make prediction based on the particular graph under analysis.

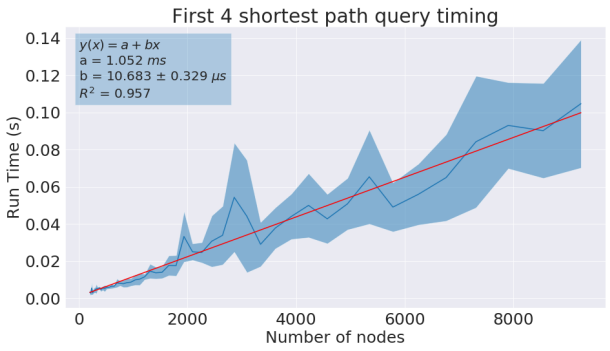


Figure 2: *Timing trend for the query that extracts the first 4 shortest paths beween two random nodes. The mean is computed for 30 different couples on the same graph.*

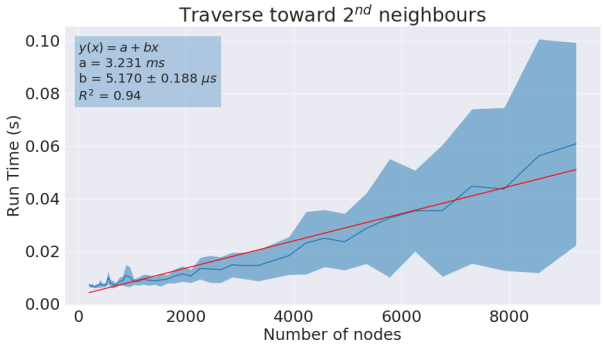


Figure 3: *Timing trend of the second neighbours extraction query. The mean is calculated for 30 different starting nodes on the same graph.*

**Visualization** In figure 4 we show an example of visualization of a random branch of the network used as testing ground for the API.

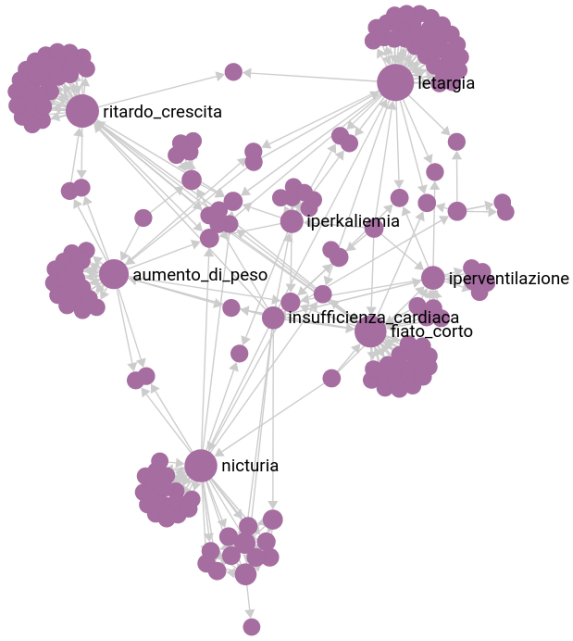


Figure 4: *ArangoDB visualization tool: in this figure we show a random branch of SymptomsNet, a graph of symptoms and diseases.*

In figure 5, instead, the subnetwork of the nearest neighbours of the nodes with label *emicrania* is shown, extracted using the pipeline descripted above.

In the case of a multipartite graph, objects from different classes can be visualized with different colours.

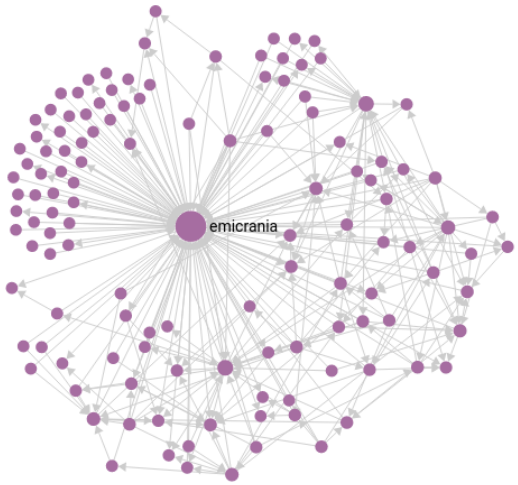


Figure 5: *ArangoDB visualization tool: in this figure we show the network of first neighbours of emicrania (cronic headache).*

## 5 Conclusions

We've produced an applicative to visualize and efficiently query a graph-like database with a simple yet resourceful programming language like python.

As shown in the results, the scaling of the ArangoDB query algorithm seems linear for the two cases taken into consideration, $k$ shortest paths and neighborhood search. This is promising for future application in large dataset.

**Open Problems** The main problems lay in the level of control we can gather through python. Indeed, it seems to be impossible to load and save default queries. Moreover, we couldn't manage to save graph visualization options, which could ease the accessibility.

Another issue, come from the slowness of the export to arango algorithm, but since it's a *una tantum* operation, it isn't a priority.

**Future Ideas** We will focus on preparing the applicative for the non-programmer user.

We think a Graphical User Interface (GUI) could greatly smoothen the user experience.

In the future we want to test the applicative with a much larger database than the one we had at disposal.

## References

[1] "Arangodb." `https://www.arangodb.com/`. open-source multi-model database.

[2] joowani, "python-arango." `https://github.com/Joowani/python-arango`. Python driver for ArangoDB.

[3] "Networkx." `https://networkx.github.io/`. Python library for Complex Networks.