

Laboratory 3 - NoSQL

Prof. Dr. Laura E. Raileanu, Cédric Campos Carvalho, Elliot Ganty

5th November 2025

1 Introduction

In this laboratory, we will explore graph databases using [Neo4J](#).

Our goal is to model the main features of Switzerland's railway network (see Figure 1) in the form of a graph as follows:

- The **vertices** of the graph will represent the **cities**
- The **edges** will represent the **railway lines** connecting them



Figure 1: Simplified illustration of the Swiss railway network

1.1 Organisation

This laboratory must be carried out in groups of no more than **2** students.

1.2 Submission

Submit a zip file to [Moodle](#) containing:

- The source code of your implementation
- The HTML files corresponding to the results of the various tasks to be completed

Please write the names of the members of the group in the python files. No report is needed.

The deadline is **19th November, 2025 at 23:59:59**.

2 Setup

2.1 Provided files

- **cities.csv**: within the "data" folder. The cities are listed in this file. Each city has a name (`name`), GPS coordinates (`latitude`, `longitude`) and population size (`population`).
- **lines.csv**: within the "data" folder. The railway network lines are described in this file. For each line, we specify the two cities it connects (`city1`, `city2`), and three weight properties : the distance in kilometers (`km`), the average travel time in minutes (`time`) and the number of tracks on the line (`nbTracks`).
- **docker-compose.yml**: simplifies the installation and configuration of a Neo4J instance, contains the necessary plugins to apply algorithms on graphs.
- **requirements.txt**: contains the list of required Python dependencies for this project
- **index.py**: **file to be completed**. Generates vertices and relationships modeling our railway network in the database, as well as data for our graph algorithms. Currently, it only contains the code to generate the vertices (cities) from the `cities.csv` file.
- **display.py**: **file to be completed**. Retrieves information from the database as well as the results of the algorithms, and then displays it on a map as an HTML file. Currently, it only contains the code to fetch and show the cities (vertices) on the map.
- **"output" folder**: **folder to be completed**. Contains all the HTML files generated when running the `display.py` file for each step. Currently, it only contains the output you should obtain when running `display.py` as given.

2.2 Environment setup

To complete the required tasks within this laboratory, it is suggested to use **Python version 3.9** or higher. You may also find it useful to create a virtual environment. The required dependencies are described in the `requirements.txt` file.

- Use the Docker Compose file to launch the Neo4j instance
- Install the required Python dependencies
- Run `index.py` to create the nodes (cities) in the Neo4j database
- Run `display.py` to generate your first HTML map output and verify that it matches the existing `template.html` file

When the Neo4J instance is running, you will also have access to a web interface by opening <http://localhost:7474> in your browser (No authentication is needed here). This can be convenient for manually testing queries or completely clearing the database. The interface will be similar to the sandbox seen during lectures.

2.3 Useful queries

- `MATCH(n) RETURN n`: returns all nodes from the database
- `MATCH(n) DETACH DELETE n`: deletes all nodes and relationships

- `CALL gds.graph.drop('myGraph')`: deletes the in-memory graph called 'myGraph' (you will need this for tasks 3.3 and 3.4)

3 Tasks

You will have 4 different tasks to complete. For each task, we will first add data within our database by adding new functions and then running the **index.py** file again. Then, we will also need to add new functions and then run **display.py** file that will each time produce a new HTML file showing (or using) this newly added data on a map. Pay attention to what we want to display for each part, you can check the PowerPoint presentation for examples.

3.1 Creating railway lines connecting cities

We now want to add the railway lines.

First, in the **index.py** file, create a new function to create relationships in the database between the existing nodes (our cities). For each line described in the **cities.csv**, create **two directed relationships**, each going in the opposite direction. These relationships must have the properties **km**, **time**, and **nbTracks**.

Then, in the **display.py** file, create a new function to retrieve those newly created relationships and display them on the map as well as the cities. You can use the given display function `display_polyline_on_map()` to draw these new lines. Be careful with duplicate relationships. Choose one direction for display only.

3.1.1 Expected result

A new HTML file (`2_1.html`) showing cities and the railway lines between them.

3.1.2 Resources

[CREATE Command](#)

3.2 Query on cities

In the **display.py** file, write a new function that will fetch and show on the map the cities with more than 100,000 inhabitants that are located at four stops of fewer from Lucerne.

3.2.1 Expected result

A new HTML file (`2_2.html`) highlighting the cities returned by the query.

3.2.2 Resources

- [WHERE Command](#)
- [Variable Length Relationships](#)

3.3 Shortest path algorithm

We now want to find the shortest path between two cities according to a specific criterion. We will first use the distance in kilometers, then the travel time in minutes.

In order to do so, we must use the Dijkstra Source-Target algorithm from the Graph Data science plugin. (Directly available if the provided Docker image is used).

First, in the **index.py** file, we will need to create a graph in memory to efficiently compute paths without modifying our main database.

Then, in the **display.py** file, we can apply the algorithm to find the shortest path, then finally display it on the map.

All of this is explained in the **Resources** below.

3.3.1 Expected result

Two new HTML files:

- (**2_3_1.html**) highlighting the shortest path based on the distance in kilometers between Geneva and Chur
- (**2_3_2.html**) highlighting the shortest path based on the travel time in minutes between Geneva and Chur

3.3.2 Resources

[Dijkstra Source-Target Shortest Path](#)

3.4 Minimum spanning tree

We want to carry out maintenance work on the railway network. In order to reduce expenses, we want to minimize the total renovation cost. Our goal is for all cities to remain connected by at least one renovated line.

Each railway line has between 1 and 4 tracks. The cost of renovating 1 kilometer of railway depends on the number of tracks. It costs 1 million CHF per track per kilometer, that is:

- 4M CHF per km for a 4 track line
- 3M CHF per km for a 3 track line
- 2M CHF per km for a 2 track line
- 1M CHF per km for a single track line

A minimum spanning tree is a subset of edges in a connected, weighted graph that connects all vertices together without cycles and with the minimum possible total edge weight.

We will first add a new "cost" property to all our relationships indicating the renovation cost of each line.

This new property will be used to compute the minimum spanning tree. The lines included in this tree will be the ones selected for renovation. To achieve this we will use the Minimum Weight Spanning Tree algorithm also provided by the Graph Data Science plugin.

You can select "Chiasso" as source.

3.4.1 Expected result

A new HTML file (2_4.html) highlighting the selected lines that should be renovated on the map.

3.4.2 Resources

[Minimum Weight Spanning Tree](#)