Project Work - Neo4J Information System

Systems and Methods for Big and Unstructured Data

POLITECNICO
MILANO 1863

Curti Gabriele [10624502]
Cutrupi Lorenzo [10629494]
Samuele Mariani [10622653]
Alessandro Molteni [10623928]
Matteo Monti [10622780]

November 13, 2021

Contents

1 Introduction

The project is about designing, storing and querying a database using technologies shown during the lessons. In particular we were asked to build a noSQL database to support a contact tracing application for Covid-19.

1.1 Problem specifications

We need to store information about:

- Personal data for each person:
  - Name and surname
  - Birthdate
  - Vaccination
  - Covid tests
- Connection between people, with time and place if possible, from 3 different sources:
  - Family/household relations
  - Contact tracing app results
  - Explicit data collection (visited restaurant, cinema, etc...)

Other personal information can be added in the creation of the database, but in the scope of this project we decided to have the strictly necessary information.

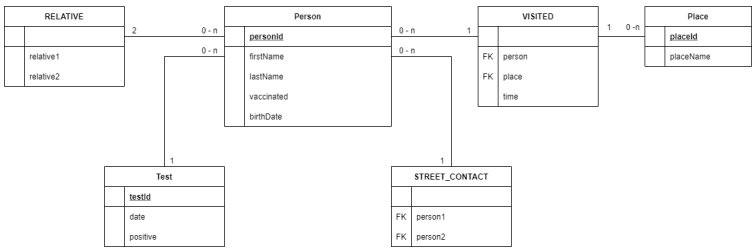
1.2 Hypothesis

We made some hypotheses to simplify the construction of the database without restricting its capabilities.

1. The relations `VISITED` and `STREET_CONTACT` are referred to a single day (2021 November 3rd) in order to have a meaningful database without needing too many nodes.
2. We represented just 5 places, to have a higher chance for people to meet.
3. The entire dataset was built from random sources, so it does not really represent real world correlation, but in our context is sufficient to demonstrate its capabilities.
4. Places and persons nodes have just the strictly necessary attributes, but in a real world application they can be extended to have more information.

2 ER diagram

The diagram represents the internal structure of the database. There are three different classes for persons, places and COVID tests. These will model the nodes in the Neo4J database. Other classes model the relations between these nodes: `RELATIVE`, `VISITED`, `STREET_CONTACT` and `TEST`.

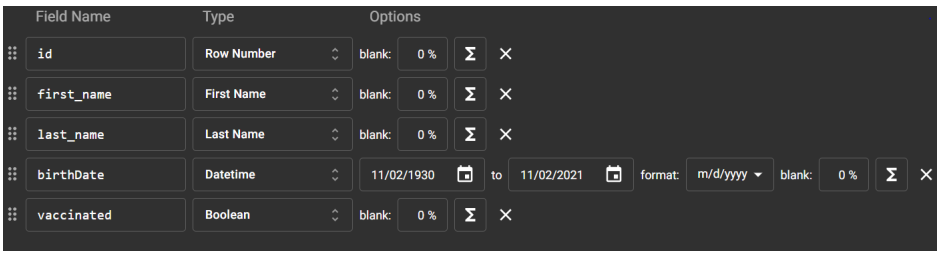


3 Dataset description and creation

In order to generate data for our database, we used some techniques and tools that allowed us to save a lot of time while creating a rich and correct database. For simple data, we used an online tool at the website mockaroo.com, that provides a random generator of a lot of different types of information (such as name, surname, date, time) and converts the file into `.csv`, a format that is very easy to convert into nodes in Neo4J.

3.1 `Person` Nodes generation

This was one of the simplest to create since it can be totally random generated. We used a row number (that auto-increases starting from 1) for `id`, a random generator of `first_name` and `last_name`, a Datetime for the `birthDate` and a Boolean to represent the `vaccinated` attribute.



The resulting `.csv` is included in the file named *Person_Data.csv*. To give a rough idea of what it looks like there is a short example showing seven nodes:

id	first_name	last_name	birthDate	vaccinated
1	Hilde	Larvent	12/20/1987	false
2	Griffin	Dullard	3/20/1956	true
3	Saxon	McGrudder	10/4/2004	false
4	Trever	Stendall	4/9/1992	true
5	Lavinie	Blumfield	1/26/1940	true
6	Mirilla	Alvar	9/20/1938	false
7	Dierdre	Meigh	11/21/1984	false

3.2 `Place` Nodes generation

Similarly to the *Person_Data* file, this table is composed of an incremental number and a name, but to keep things ordered and easy for the queries to have meaningful results, we decided to create only 5 places manually, and this was the result:

id	place name
1	McDonald di Assago
2	McDonald di Lorenteggio
3	Boschetto di Rogoredo
4	Carcere minorile Cesare Beccaria
5	Casa di Sergio

3.3 `Test` Nodes generation

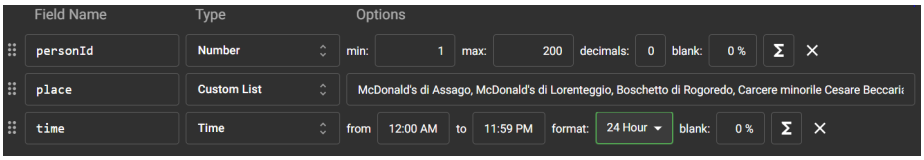
The Test table is also randomly generated, using a random number from 1 to 200, a datetime for the `date` of the test, and a Boolean `positive` to represent the result of the test (true if positive, false otherwise). This is an example of the result:

person	date	positive
188	4/25/2021	false
41	4/29/2021	false
149	10/3/2021	false
52	9/13/2021	true
178	1/24/2021	false

In the final file we also added a `TestId` attribute to better model some queries.

3.4 `VISITED` Relation generation

This table represents all the visits of each person, also created in mockaroo.com with random numbers ranging from 1 to 200 (the amount of people present in the database), a random string between the five places, and the time of the visit. We did not use a date because the amount of rows required to generate a meaningful database would have been very high.



An example of the result is below:

personId	place	time
132	McDonald's di Assago	1:45
16	McDonald's di Lorenteggio	18:30
130	Boschetto di Rogoredo	12:46
5	McDonald's di Assago	10:49
117	Carcere minorile Cesare Beccaria	18:00
119	Boschetto di Rogoredo	1:22

3.5 `STREET_CONTACT` Relation generation

Similarly to the `VISITED` relation table, this one is also generated in mockaroo.com using two random `personId` ranging between 1 and 200, representing the two persons, and the `time` the meeting has happened. Like the previous table, there's no date for the meeting in order to make the database thinner. This is an example of the result:

person1	person2	time
87	139	17:06
112	170	20:31
178	55	9:02
166	60	0:19
26	3	3:42

3.6 `RELATIVE` Relation generation

The relation `RELATIVE` is an example of a complex structure, because it requires small groups of people to be all related to the other members of the group, and

with nobody outside of it (if A is related to B, then all the people related to A must be related to B too). To accomplish this requirement we created a simple C++ program as below:

```
//30 families of two people
for(i=1;i<61;i=i+2){
    cout<<i<<","<<i+1<<endl;
}
//30 families of three people
for(i=61;i<151;i=i+3){
    cout<<i<<","<<i+1<<endl;
    cout<<i<<","<<i+2<<endl;
    cout<<i+1<<","<<i+2<<endl;
}
//10 families of four people
for(i=151;i<191;i=i+4){
    cout<<i<<","<<i+1<<endl;
    cout<<i<<","<<i+2<<endl;
    cout<<i<<","<<i+3<<endl;
    cout<<i+1<<","<<i+2<<endl;
    cout<<i+1<<","<<i+3<<endl;
    cout<<i+2<<","<<i+3<<endl;
}
//2 families of five people
for(i=191;i<201;i=i+5){
    cout<<i<<","<<i+1<<endl;
    cout<<i<<","<<i+2<<endl;
    cout<<i<<","<<i+3<<endl;
    cout<<i<<","<<i+4<<endl;
    cout<<i+1<<","<<i+2<<endl;
    cout<<i+1<<","<<i+3<<endl;
    cout<<i+1<<","<<i+4<<endl;
    cout<<i+2<<","<<i+3<<endl;
    cout<<i+2<<","<<i+4<<endl;
    cout<<i+3<<","<<i+4<<endl;
}
```

The following is an example of the result:

Person1Id	Person2Id
1	2
3	4
5	6
7	8

Four families of two people each.

Person1Id	Person2Id
31	32
31	33
32	33
34	35
34	36
35	36

Two families of three people each.

Person1Id	Person2Id
121	122
121	123
121	124
122	123
122	124
123	124

One family of four people.

Person1Id	Person2Id
161	162
161	163
161	164
161	165
162	163
162	164
162	165
163	164
163	165
164	165

One family of five people.

4 Queries and commands

We firstly designed commands to load the dataset in Neo4j from the `.csv` files in order to build the Database structure. Then we designed some queries with the intent to retrieve some useful data from the dataset, from both user perspective and big data analysis perspective.

4.1 Database creation commands

These commands simply load the csv file placed in the import folder and create all the necessary nodes and relations.

```
MATCH (n) DETACH DELETE n;
```

Command 1: Delete all, to start from a blank dataset

```
LOAD CSV WITH HEADERS FROM 'file:///Person_Data.csv' AS csvLine
CREATE (p:Person {personId: toInteger(csvLine.id), firstName: csvLine.first_name,
    lastName: csvLine.last_name, birthdate: csvLine.birthdate, vaccinated:
    toBoolean(csvLine.vaccinated)});
```

Command 2: Load person nodes

```
LOAD CSV WITH HEADERS FROM 'file:///Place_Data.csv' AS csvLine
CREATE (p:Place {placeId: toInteger(csvLine.id), placeName: csvLine.place_name});
```

Command 3: Load place nodes

```
LOAD CSV WITH HEADERS FROM 'file:///Tests_Data.csv' AS row
CREATE (t: Test {testId: toInteger(row.testId), personId: toInteger(row.personId), date:
    row.date, positive: toBoolean(row.result)})
MERGE (person:Person {personId: toInteger(row.personId)})
MERGE (person)-[:TEST]->(t);
```

Command 4: Load test node and attach them to the person through the `personId` attribute

```
LOAD CSV WITH HEADERS FROM 'file:///Relatives_Data.csv' AS row
MERGE (person1:Person {personId: toInteger(row.person1Id)})
MERGE (person2:Person {personId: toInteger(row.person2Id)})
MERGE (person1)-[:RELATIVE]->(person2);
```

Command 5: Create the relation `RELATIVE` from the file

```
LOAD CSV WITH HEADERS FROM 'file:///Street_Contact.csv' AS csvLine
MATCH
  (a:Person),
  (b:Person)
WHERE NOT(a.personId = b.personId) AND a.personId = toInteger(csvLine.person1) AND
    b.personId = toInteger(csvLine.person2)
CREATE (a)-[r:STREET_CONTACT{time:csvLine.time, date: '2021-11-03'}]->(b);
```

Command 6: Create the relation `STREET_CONTACT` from the file, loading the time and date of the contact

```
LOAD CSV WITH HEADERS FROM 'file:///Visited_Data.csv' AS csvLine
MATCH
  (a:Person),
  (b:Place)
WHERE a.personId = toInteger(csvLine.personId) AND b.placeName = csvLine.place
CREATE (a)-[r:VISITED{time:csvLine.time, date: '2021-11-03'}]->(b);
```

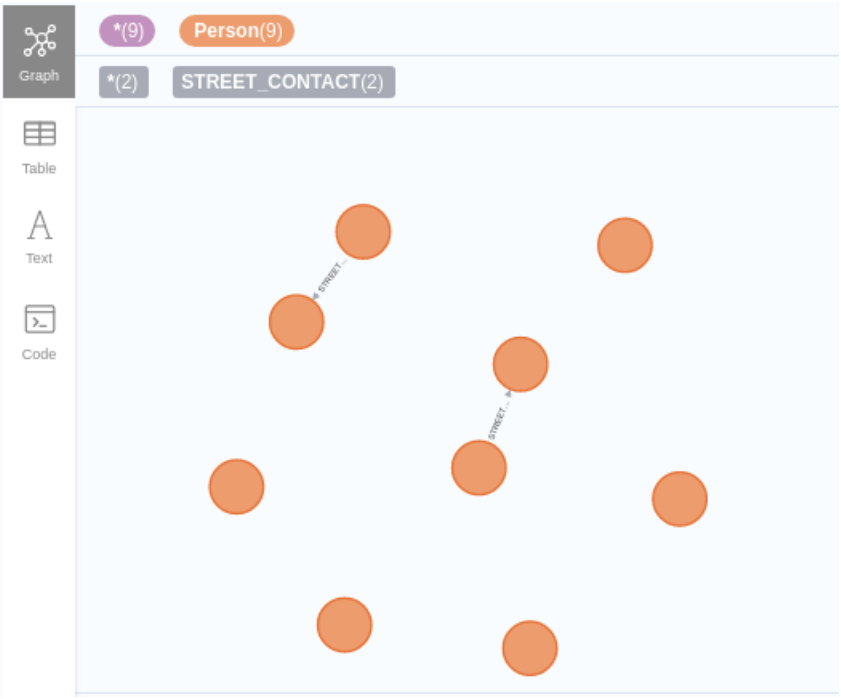
Command 7: Create the relation `VISITED` from the file, loading also time and date of the visit

4.2 Queries

4.2.1 Find all the contacts of a given person

```
MATCH (person: Person{personId: 2}) - [v: VISITED] -> (place: Place) <- [i: VISITED]
      - (placeContact: Person)
WHERE (time(i.time) - duration({hours: 2})) <= time(v.time) AND
time(v.time) <= (time(i.time) + duration({hours: 2}))
RETURN placeContact AS contact
UNION
MATCH (person: Person{personId: 2}) - [r: STREET_CONTACT] - (streetContact:
      Person)
RETURN streetContact AS contact
UNION
MATCH (person: Person{personId: 2}) - [a: RELATIVE] - (relativeContact: Person)
RETURN relativeContact AS contact
```

This is a useful command to trace all the contacts of a person, to both check if he is in danger or to find which persons are in danger, in the case the first one was found positive. The result on our dataset for the person with Id 2 is:



4.2.2 Find the number of new positive on a given day

```
MATCH(test:Test{date:"2021-11-01"})
WHERE test.positive = TRUE
RETURN COUNT(test) AS newContagion
```

Simple but very useful query to check the evolution of the pandemic. The result on our dataset in the given date is:

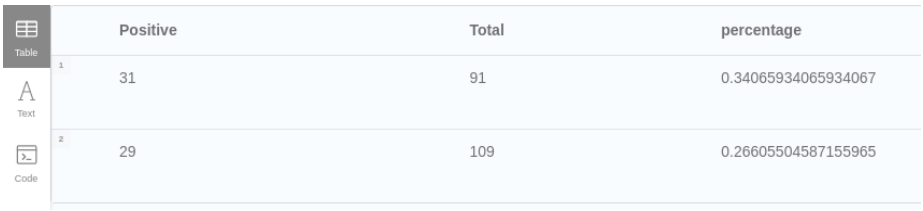
4.2.3 Find the percentage of positive over the vaccinated population and against the non vaccinated population

```
MATCH(vaccinatedPositive: Person{vaccinated: TRUE}) -[r1:TEST]-
    (test1:Test{positive:TRUE})
OPTIONAL MATCH (totalPositive: Person{vaccinated: TRUE})
RETURN COUNT(DISTINCT vaccinatedPositive) AS Positive, COUNT(DISTINCT totalPositive) AS
    Total, tofloat ((COUNT(DISTINCT vaccinatedPositive)*1.0)/(COUNT(DISTINCT
    totalPositive))) AS percentage

UNION

MATCH(notVaccinatedPositive: Person{vaccinated: FALSE}) -[r2:TEST]-
    (test2:Test{positive:TRUE})
OPTIONAL MATCH (totalPositive: Person{vaccinated: FALSE})
RETURN COUNT(DISTINCT notVaccinatedPositive) AS Positive, COUNT(DISTINCT totalPositive)
    AS Total, tofloat ((COUNT(DISTINCT notVaccinatedPositive)*1.0)/(COUNT(DISTINCT
    totalPositive))) AS percentage
```

This query aims to check whether the vaccine actually works or not. In our dataset, given that the population is completely random, the data found is not optimistic:



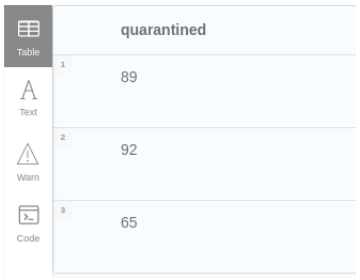
	Positive	Total	percentage
1	31	91	0.34065934065934067
2	29	109	0.26605504587155965

4.2.4 Find the number of contact between a positive and a negative

```
MATCH (p:Person)−[s: RELATIVE]−(positive1: Person)−[t: TEST]− (positiveTest1:
    Test{positive:TRUE})
MATCH (positiveTest2:Test{positive:FALSE})−[t2: TEST]−(person1: Person)−[s1:
    RELATIVE]−(positive3:Person)−[t1:TEST]−(positiveTest3:Test{positive:TRUE})
WHERE NOT (p)−[:TEST]−>(:Test)
RETURN (COUNT (DISTINCT person1)+ COUNT (DISTINCT p)) AS quarantined
UNION
MATCH (p:Person)−[s: STREET_CONTACT]−(positive1: Person)−[t: TEST]−
    (positiveTest1: Test{positive:TRUE})
MATCH (positiveTest2:Test{positive:FALSE})−[t2: TEST]−(person1: Person)−[s1:
    STREET_CONTACT]−(positive3:Person)−[t1:TEST]−(positiveTest3:Test{positive:TRUE})
WHERE NOT (p)−[:TEST]−>(:Test)
RETURN (COUNT (DISTINCT person1)+ COUNT (DISTINCT p)) AS quarantined
UNION
MATCH (p:Person)−[s: VISITED]−> (place:Place) <−[v:VISITED]−(positive1: Person)−[t:
    TEST]− (positiveTest1: Test{positive:TRUE})
MATCH (positiveTest2:Test{positive:FALSE})−[t2: TEST]−(person1: Person)−[s1:
    VISITED]−> (place:Place) <−[v1:VISITED]−(positive3:Person) −[t1:TEST]−
    (positiveTest3:Test{positive:TRUE})
WHERE NOT (p)−[:TEST]−>(:Test)
RETURN (COUNT (DISTINCT person1)+ COUNT (DISTINCT p)) as quarantined
```

This query finds the number of people that should stay self isolated, to measure the impact of the virus on society. The three values come from the different types of contact, and they are not intended to be summed as they have repeated persons.

The result is the following:



4.2.5 Find the top 10 people with the most number of contacts

```
MATCH (person: Person) − [r: STREET_CONTACT] − (streetContact: Person)
OPTIONAL MATCH (person: Person) − [a: RELATIVE] − (relativeContact: Person)
OPTIONAL MATCH (person: Person) − [v: VISITED] −> (place: Place) <− [i: VISITED] −
     (placeContact: Person)
WHERE (time(i.time) − duration({hours: 2})) <= time(v.time) AND
time(v.time) <= (time(i.time) + duration({hours: 2}))

WITH person, toFloat(COUNT(DISTINCT relativeContact) + COUNT(DISTINCT streetContact) +
     COUNT(DISTINCT placeContact)) AS totalContact ORDER BY totalContact DESC LIMIT 10

RETURN COLLECT ( person), totalContact
```

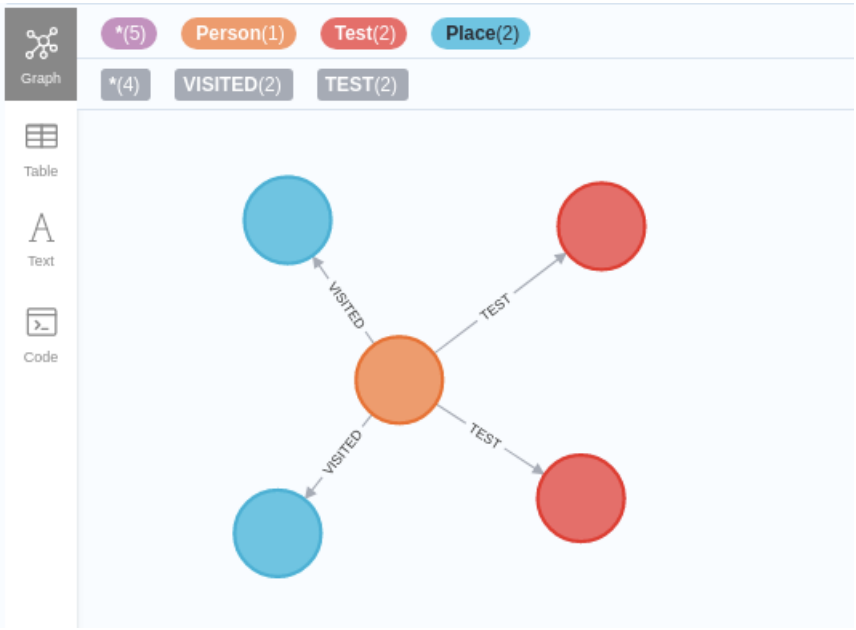
This query finds the ranking of the person with most contacts, the one that may be undervaluing the social distancing. In our dataset the result will be:

followed by the rest of the ranking.

4.2.6 Find all the people that become positive after visiting a place

```
MATCH (test1: Test{positive: FALSE})<-[:TEST]-(person: Person)-[:TEST]->(test2:
      Test{positive: TRUE}),
(person)-[v: VISITED]->(place:Place)
WHERE NOT(test1.testId = test2.testId) AND date(test1.date) < date(test2.date)
AND date(test1.date) <= date(v.date) AND date(v.date) <= date(test2.date)
RETURN person, place
```

This query retrieves the people who got infected after visiting a place and the place itself. It could be useful to make a list of the most dangerous places.



4.3 Commands

4.3.1 Delete a person from the database

```
MATCH (n:Person {personId:15}) DETACH DELETE n;
MATCH (n:Test {personId:15}) DETACH DELETE n;
```

When a user uninstalls the application, all his personal information will be cancelled in order to respect his privacy.

4.3.2 Set a person as vaccinated

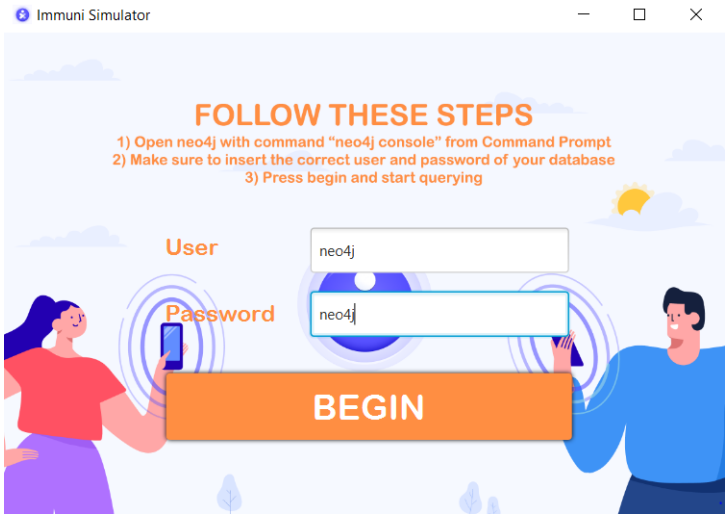
```
MATCH (p:Person{personId: 1})
SET p.vaccinated = TRUE
RETURN p
```

It is useful to know whether a user is vaccinated or not, so the `vaccinated` field should be updated when a person gets the vaccine.

5 UI application

The application is a graphic implementation of the queries described above. It is made in Java with the use of a couple of external libraries (JavaFX for the UI, Neo4J for the queries to the database). It is composed of two pages, one for the login in the database, and one for making the queries. The second one is the most complex, since it handles all the queries: you can select the query or the action you want to perform, and for each of them there is a brief explanation of what will be asked to the database, optional text fields if the user has to insert some information, and the results of the query.

Login interface:



Query interface:

Connecting to Neo4j is very easy thanks to the library offered by the developers, and the necessary code lines are:

Driver to connect to the database

```
Driver driver = GraphDatabase.driver( "bolt://localhost:7687", AuthTokens.basic(
    Global.dbUser, Global.dbPassword ) );
```

Session to make queries

```
Session session = driver.session();
```

The query itself, the return of the query is stored in Result

```
Result result = session.run( QUERY );
```

6 Sources

- Slides from the lessons and exercise session.
- www.mockaroo.com to generate simple data.
- neo4j.com documentation for cypher.
- stackoverflow.com for some help in constructing the queries.