SMBUD 2021 - Project work 1

Aman Gabba - 10793117
Andrea Cerasani - 10680486
Giovanni Demasi - 10656704
Pasquale Dazzeo - 10562130
Vlad Marian Cimpeanu - 10606922

POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Problem Specification

The aim of this project was to design a 'query graph data structure' in Neo4j for supporting a contact tracing application for COVID-19. The database must register all the necessary information about the users including vaccines and Covid swabs in order to have a pandemic trend overview for a given country. The application using this database will be able to exploit all the data coming from tracking applications and from all the public facilities.

1.2 Hypothesis

The assumptions taken into account are the following:

- People belonging to the same family live in the same house if not explicitly specified. Using the concept of "house" instead of "family", it is offered the possibility to differentiate domicile from residence.
- All the personal data are verified by an authoritative figure, for instance the government.
- The domicile declaration is assumed to be truthful.
- All the data coming from public spaces are always considered true and complete.
- People always provide all the necessary information to the staff when they visit a certain public space.
- Every MEETS relationship is automatically added to the database by a tracing app when two mobile phones stay in the same range for more than 15 minutes. We assume that to every device corresponds only one person and vice versa.
- Relationships use the Data type instead of Timestamp (with the exception of Test relationship because could happen that a person due to a positive resulting rapid test does in the same day another test which results negative - the use of Timestamp is important to recognise false positive) to register the relations because of safety and simplicity reasons. The former reason allows people using the Database to trace contacts during all the day and not only during a range of time (for more control). The latter reason is required due to avoid mistakes by the staff of public facilities during the time registration (an error in time registration could lead to a wrong tracing).
- The designed model doesn't take into consideration the deaths, as the main focus of the project was the tracking of Covid infections. However in a more complete model that considers this aspect, a death node could

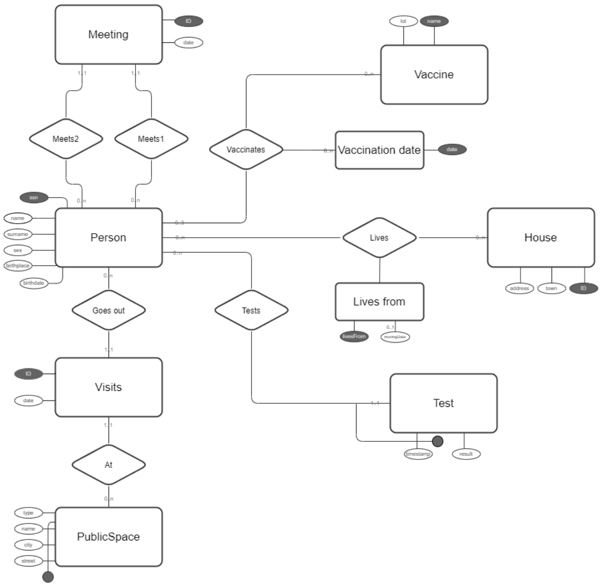
be used to represent it through a dead relationship that connect the dead person to the death node with a date attribute. This choice could be useful to analyze the death/infection ratio related to the Covid pandemic.

2 ER diagram

The designed ER diagram contains the following entities: Person, Public Space, Vaccine, House and Covid Test. The other classes have been introduced due to ER diagram correctness reasons, but they haven't been taken into account during the design of the Graph Database because the last one allows a more flexible design.

As said before the concept of 'Family' has been replaced with the concept of 'House' since it is more realistic and more useful for contact tracing.

The Person entity has SSN ('Codice Fiscale' in Italy) as primary key and all the relevant data. Every Vaccine has its name and lot. Houses are described by a unique ID, address and town. Covid tests are identified by the timestamp and they also have the result as attribute. Every Public Space has name, city and streets as primary key and also a type attribute to identify its category. Visits and Meeting entities, as said before, have been introduced due to correctness aspect but they are represented only as relationship in the graph diagram. Lives and Vaccines relationship are identified by date.



3 Graph diagram

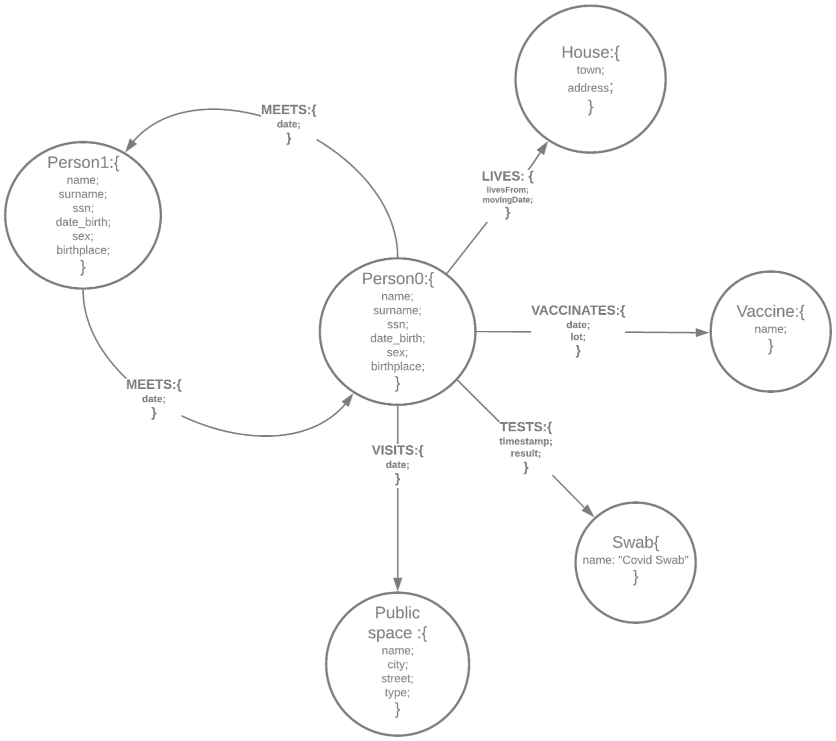
The designed Graph diagram is shown below. Person, House, Vaccine, Public Space and Covid Swab are represented as nodes and they all have the same attributes shown in the ER above.
The relationships of the database are MEETS, LIVES, TESTS, VACCINATES and VISITS, they all have date as attribute (excepted tests that has timestamp). VACCINATES also has the lot of the vaccine and TESTS also has the result of the test.
More specifically LIVES has as parameters livesFrom, that identifies the date from which a specific person started to live in a specific house and movingDate, that identifies the date from which a specific person decided to move from a specific house.

This design decision may seem memory expensive, but it is meant to face some problems: first of all, it was assumed that the database can handle only people who live in the same country of the government (in this case Italy), so if someone moves abroad, this database should not consider anymore that given person.
For instance, if a member of a given family moves abroad, the database will be not able to link this person to a new house, so it will be not possible to know the given person has moved away and in case of positive test of one of his/her last roommates, he/she will be tracked even though he/she is not living anymore in the last house.
Another more naive reason is that it is much easier to find roommates of a given person at a given time, when writing queries.

The implementation of LIVES relationship was preferred over a LIVES_WITH relationship, in order to reduce the redundancy of data, indeed for each person added to a certain house through a LIVES relationship, it will take only one arc, whereas adding a person to a certain family through LIVES_WITH relationships, it will take a number of arcs equal to the amount of members of the family.

With this design choice the time and space complexity of adding people to a certain family has been reduced from $O(n^2)$ to $O(n)$.

The decision of making vaccines as nodes, comes from the idea of speeding up queries involving vaccines exploiting pattern matching of the database.


Person1:{
name;
surname;
ssn;
date_birth;
sex;
birthplace;
}
MEETS:{
date;
}
MEETS:{
date;
}
House:{
town;
address;
}
LIVES: {
livesFrom;
movingDate;
}
Person0:{
name;
surname;
ssn;
date_birth;
sex;
birthplace;
}
VACCINATES:{
date;
lot;
}
Vaccine:{
name;
}
TESTS:{
timestamp;
result;
}
VISITS:{
date;
}
Swab{
name: "Covid Swab"
}
Public
space :{
name;
city;
street;
type;
}

4 Dataset description

The Dataset has been built by making a script in Python, using some useful packages like: random-italian-person that automatically generates people with random attributes (this package has been modified due to our necessity); the official binary Neo4j driver for Python for the communication with the Database since it allows to make queries directly from Python.

We have also used this piece of code to generate amenities data. This query has been used in overpass turbo (a web-based tool for extracting OpenStreetMap data) selecting the map area of Rome, Naples and Milan.

```
[out:csv("amenity","name", "addr:street", "addr:housenumber", "addr:city
    "; true; ",")];
// gather results
(
  //query part for: amenity= post_box
  node"amenity"="restaurant";
  node"amenity"="bar";
  node"amenity"="cinema";
);
// print results
out body;
>;
out skel qt;
```

5 Queries and Commands

5.1 Queries

5.1.1 Vaccination percentage for a given age range

The following query can be used to show the percentage of vaccinated people in an age range.
$age_min and $age_max must be two positive integers with $age_min ≤ $age_max.

```
MATCH (sample: Person)
WHERE duration.between(date(sample.birthdate), date()).years >= $age_min
      AND
      duration.between(date(sample.birthdate), date()).years <= $age_max
WITH sample AS sizeSample
MATCH (vaccinated:Person)-[:VACCINATES]->(vaccine: Vaccine)
WHERE duration.between(date(vaccinated.birthdate), date()).years >=
      $age_min
      AND
      duration.between(date(vaccinated.birthdate), date()).years <=
      $age_max
RETURN
CASE
WHEN COUNT(DISTINCT vaccinated) = 0 THEN 0.0
WHEN COUNT(DISTINCT sizeSample) = 0 THEN 0.0
ELSE (COUNT(DISTINCT vaccinated) * 1.0 / COUNT(DISTINCT sizeSample) *
    1.0) * 100.0
END AS Percentage
```

5.1.2 Infection ratio per month

Query that returns the infection ratio among all the tested people for each month.

```
MATCH ()-[t:TESTS]->()
WITH date(apoc.date.format(apoc.date.parse(t.timestamp, 'ms', 'yyyy-MM-dd
    '), 'ms', 'yyyy-MM-dd')).month AS month,
     date(apoc.date.format(apoc.date.parse(t.timestamp, 'ms', 'yyyy-MM-dd
    '), 'ms', 'yyyy-MM-dd')).year AS year, count(t) as all_tests
OPTIONAL MATCH ()-[t:TESTS]->()
WHERE date(apoc.date.format(apoc.date.parse(t.timestamp, 'ms', 'yyyy-MM-
    dd'), 'ms', 'yyyy-MM-dd')).month = month AND
    date(apoc.date.format(apoc.date.parse(t.timestamp, 'ms', 'yyyy-MM-dd
    '), 'ms', 'yyyy-MM-dd')).year = year AND
    t.res = 'Positive'
RETURN round((COUNT(t) * 1.0 / all_tests * 1.0) * 100.0 * 100.0) / 100.0
    AS ratio, month, year
ORDER BY year DESC, month DESC
```

5.1.3 False positives

The following query finds false positives (for false negatives it is necessary to switch pre with post in the last two lines before the Return statement).

```
MATCH (p: Person)-[pre: TESTS]->()<-[post:TESTS]-(p)
WHERE duration.between(datetime({ epochMillis: apoc.date.parse(pre.
    timestamp, 'ms', 'yyyy-MM-dd HH:mm:ss')}), datetime({ epochMillis:
    apoc.date.parse(post.timestamp, 'ms', 'yyyy-MM-dd HH:mm:ss')})).
    hours < 24
AND duration.between(datetime({ epochMillis: apoc.date.parse(pre.
    timestamp, 'ms', 'yyyy-MM-dd HH:mm:ss')}), datetime({ epochMillis:
    apoc.date.parse(post.timestamp, 'ms', 'yyyy-MM-dd HH:mm:ss')})).
    hours > 0
AND post.res="Negative"
AND pre.res="Positive"
RETURN p, pre, post
```

5.1.4 Most unsafe public spaces for a given city

This query returns the infection ratio of every place of a desired city, in a selected year and month, as sum of people who became positive after being in a place over the total amount of visits of that place in the same period. $month and $year are, respectively, the desired month and year and $city is the city taken into consideration.

```
MATCH (p1:Person)-[v:VISITS]->(ps:PublicSpace)
WHERE date(v.date).month = $month AND date(v.date).year = $year
      AND ps.city = $city
WITH ps.name AS space_name, v AS total_visits, p1 AS total_people
OPTIONAL MATCH (p:Person)-[t:TESTS]->()
WHERE p.ssn IN total_people.ssn AND t.res = 'Positive' AND duration.
    inDays(date(total_visits.date), date(apoc.date.format(apoc.date.
    parse(t.timestamp, 'ms', 'yyyy-MM-dd'), 'ms', 'yyyy-MM-dd'))).days
    >= 0 AND duration.inDays(date(total_visits.date), date(apoc.date.
    format(apoc.date.parse(t.timestamp, 'ms', 'yyyy-MM-dd'), 'ms', 'yyyy
    -MM-dd'))).days <= 15
RETURN
CASE
WHEN COUNT(DISTINCT p) = 0 THEN 0.0
ELSE (COUNT(DISTINCT p)*1.0 / COUNT(DISTINCT total_people)*1.0)*100.0
END AS percentage, space_name ORDER BY percentage DESC
```

5.1.5 Vaccine efficacy

This query can be used to analyze the Vaccine efficacy computed as the ratio of the vaccinated who became positive over the total number of vaccinated people.

```
MATCH (p1:Person)-[:VACCINATES]->(v1:Vaccine {name: 'AstraZeneca'})
MATCH (p2:Person)-[:VACCINATES]->(v2:Vaccine {name: 'Moderna'})
MATCH (p3:Person)-[:VACCINATES]->(v3:Vaccine {name: 'Pfizer'})
MATCH (p4:Person)-[:VACCINATES]->(v4:Vaccine {name: 'Jensen'})
WITH COUNT(DISTINCT p1) AS vaccinated_astrazeneca, COUNT(DISTINCT p2) AS
    vaccinated_moderna, COUNT(DISTINCT p3) AS vaccinated_pfizer, COUNT(
    DISTINCT p4) AS vaccinated_jensen
OPTIONAL MATCH ()<-[t1:TESTS]-(p1:Person)-[v1:VACCINATES]->(vacc1:Vaccine
     {name: 'AstraZeneca'})
WHERE t1.res = 'Positive'
    AND date(apoc.date.format(apoc.date.parse(t1.timestamp, 'ms', 'yyyy-
    MM-dd'), 'ms', 'yyyy-MM-dd')) > date(v1.date)
OPTIONAL MATCH ()<-[t2:TESTS]-(p2:Person)-[v2:VACCINATES]->(vacc2:Vaccine
     {name: 'Moderna'})
WHERE t2.res = 'Positive'
    AND date(apoc.date.format(apoc.date.parse(t2.timestamp, 'ms', 'yyyy-
    MM-dd'), 'ms', 'yyyy-MM-dd')) > date(v2.date)
OPTIONAL MATCH ()<-[t3:TESTS]-(p3:Person)-[v3:VACCINATES]->(vacc3:Vaccine
     {name: 'Pfizer'})
WHERE t3.res = 'Positive'
    AND date(apoc.date.format(apoc.date.parse(t3.timestamp, 'ms', 'yyyy-
    MM-dd'), 'ms', 'yyyy-MM-dd')) > date(v3.date)
OPTIONAL MATCH ()<-[t4:TESTS]-(p4:Person)-[v4:VACCINATES]->(vacc4:Vaccine
     {name: 'Jensen'})
WHERE t4.res = 'Positive'
    AND date(apoc.date.format(apoc.date.parse(t4.timestamp, 'ms', 'yyyy-
    MM-dd'), 'ms', 'yyyy-MM-dd')) > date(v4.date)
WITH vaccinated_astrazeneca,vaccinated_moderna,vaccinated_pfizer,
    vaccinated_jensen, COUNT(DISTINCT p1) AS vaccinated_infected_astra,
    COUNT(DISTINCT p2) AS vaccinated_infected_moderna, COUNT(DISTINCT p3
    ) AS vaccinated_infected_pfizer, COUNT(DISTINCT p4) AS
    vaccinated_infected_jensen
RETURN (1- (toFloat(vaccinated_infected_astra)/toFloat(
    vaccinated_astrazeneca)))*100 AS AstraZenecaEfficacy,
    (1- (toFloat(vaccinated_infected_moderna)/toFloat(vaccinated_moderna)
    ))*100 AS ModernaEfficacy,(1- (toFloat(vaccinated_infected_pfizer)/
    toFloat(vaccinated_pfizer)))*100 AS PfizerEfficacy,(1- (toFloat(
    vaccinated_infected_jensen)/toFloat(vaccinated_jensen)))*100 AS
    JensenEfficacy
```

5.1.6 People to quarantine

This query returns all the people a given person has been in contact with in the last 15 days. In detail it returns all the people met by a given person in the last 15 days (incubation period) from a given date with the tracking app, his/her family and all the people that have been in the same place during the same day as him/her. The following query can be used to find all the people who need to be put in quarantine due to a positive test of the given person. $ssn is the fiscal code of the person receiving the vaccine (16 digits alphanumeric code), $end_date is the last day of the incubation period ($incubation).

```
MATCH (infected:Person)-[l1:LIVES]->(h)<-[l2:LIVES]-(person:Person)
WHERE infected.ssn = $ssn
  AND date(l1.livesFrom) <= $end_date AND (date(l1.movingDate) >=
  $end_date - $incubation OR l1.movingDate is  NULL)
  AND date(l2.livesFrom) <= $end_date AND (date(l2.movingDate) >=
  $end_date - $incubation OR l2.movingDate is NULL)
  AND (date(l2.movingDate) >= date(l1.livesFrom) OR date(l2.livesFrom) <=
  date(l1.movingDate))
RETURN person
UNION
MATCH (infected: Person)-[v1:VISITS]->()<-[v2:VISITS]-(person: Person)
WHERE infected.ssn = $ssn AND v1.date = v2.date AND duration.inDays(date(
    v1.date), date($end_date)).days <= 15 AND duration.inDays(date(v1.
    date), date($end_date)).days >= 0
RETURN person
UNION
MATCH (infected: Person)-[m:MEETS]->(person: Person)
WHERE infected.ssn = $ssn AND duration.inDays(date(m.date), date(
    $end_date)).days <= 15 AND duration.inDays(date(m.date), date(
    $end_date)).days >= 0
RETURN person
```

5.2 Commands

5.2.1 New vaccination

The following command can be used to add a vaccination. $ssn is the fiscal code of the person receiving the vaccine (16 digits alphanumeric code), $vax_name is the name of the vaccine ('AstraZeneca', 'Jensen', 'Pfizer', 'Moderna'), $lot is the lot of the vaccine (for simplicity reasons, the generated lots are in the following format: AZ##, P##,J##,M##). $vax_date is the date of the vaccination and must be in the following format YYYY-MM-DD.

```
MATCH (person_to_vaccinate: Person),
      (vax: Vaccine)
WHERE person_to_vaccinate.ssn = $ssn
      AND
      vax.name = $vax_name
CREATE (person_to_vaccinate)-[:VACCINATES {lot: $lot,
        date: $vax_date}]->(vax)
```

5.2.2 Moving a family

The following query changes all the LIVES relations of the people who live in the same house by adding them a movingDate attribute and makes new relations with the new house (this command could be useful in case of moving family). $ssn is one of the people who lives in the old house. $id is the id of the new house (Integer number) and $moving_date is the date of the moving and it must be in the following format YYYY-MM-DD.

```
MATCH (p1:Person)-[l1:LIVES]->(h:House)
WHERE p1.ssn = $ssn AND l1.movingDate IS NULL
WITH h AS old_house
MATCH (p2:Person)-[l2:LIVES]->(old_house)
WHERE l2.movingDate IS NULL
WITH p2 as people_moving, l2
MATCH (h:House) WHERE ID(h) = $id
CREATE (people_moving)-[:LIVES{livesFrom: $moving_date}]->(h)
SET l2.movingDate = $moving_date
```

5.2.3 New Covid Test

The following command can be used to add a Covid Test. $ssn is the fiscal code of the person tested (16 digits alphanumeric code), $result is the result of the test ('Positive' or 'Negative'). $timestamp is the timestamp of the test and must be in the following format 'YYYY-MM-DD hh:mm:ss'.

```
MATCH (person_tested: Person),
      (test: Swab)
WHERE person_tested.ssn = $ssn
      AND
      test.name = 'Covid Swab'
CREATE (person_tested)-[:TESTS {res: $result, timestamp: $timestamp}]->
       (test)
```

6 Generator database

The "neo4jDB-populator/main.py" file is responsible of generating a random database. In order to correctly run the generator, it is mandatory to store the Neo4j password in "neo4jDB-populator/password.txt", in this way the generator will be able to connect with the database.
This script exploits the following packages: pandas, numpy, neo4j and python-codicefiscale.

6.1 Cities generation

This generator will create data for the following cities: Rome, Milan and Naples. For each city it will create nodes representing public spaces available in that specific town. For safety sake, the parameter that sets the number of public spaces to generate should be lower than 46.

6.2 People generation

In order to consistently generate people, it was decided to assign a number of houses per city, by default the number of houses per city is set to 15. Every time an house is built, the generator will also create a family of 'Person' to link to that house. Each family randomly consists of 1 to 6 members.

6.3 Relationships generation

For improving the quality information extracted from the data, people living in a certain city must "VISITS" only public spaces of the same city, and for the same reason, people can "MEETS" people of the same city with 90% of probability. In order to get significant results from queries, the number of "VISITS" relations was set to 1 visit per person every 3-5 days. This setting may impact on the generation data-set time, indeed it could take up to 1-2 minutes to generate all the data.

7 Application description

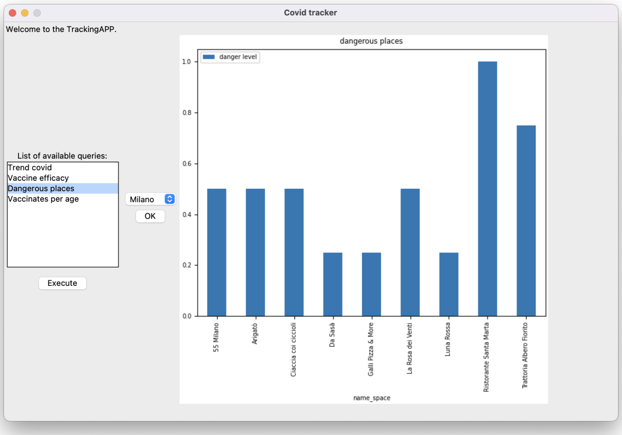
Goal of the application "GUI/App.py" is to give some information about the pandemic. The user can use the application in order to know about:

- Trend Covid: the application will display the ratio of positive people among all the tests carried out per each month.

- Vaccine efficacy: the application will display how many people vaccinated with a specific vaccine has been negative over all the people vaccinated with the same vaccine.

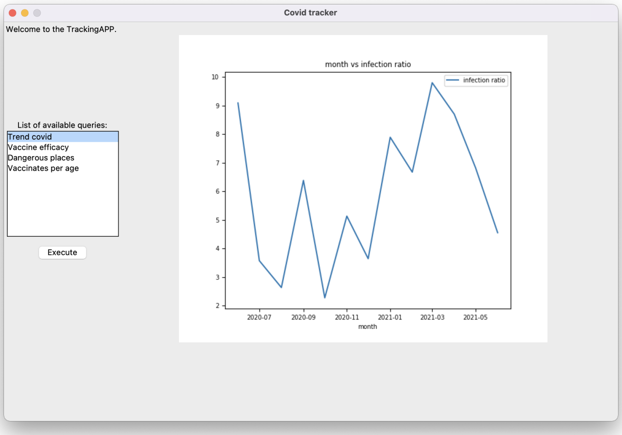
- Dangerous places: selected a specific city, the user can see which are the most dangerous public spaces in that city. The danger level of a given public space is calculated as the percentage of people that tested positive after 15 or less days, and then is normalized considering all the other public places: in this way the most dangerous public space will have a danger level of 1, whereas the less unsafe place will have a danger level equal to 0.

- Vaccinates per age: the application will display the percentage of people vaccinated per each age range. The age range considered are [20,29], [30,39], [40, 49], ..., [90, 100].


Covid tracker

Welcome to the TrackingAPP.

Figure 1: Application displaying danger levels in Milan.


Covid tracker

Welcome to the TrackingAPP.

Figure 2: Application displaying how Covid infection is spreading.

8 User guide

The application is meant to work on pc or macOS. As the application has been entirely developed in Python, it is not operative system dependent.
In order to correctly run the application it is mandatory to store the Neo4j password in `"neo4jDB-populator/password.txt"`, in this way the application will be able to connect with the database.
The user must check to have installed correctly in the virtual environment the following pacakges: `tkinter, matplotlib, pandas, neo4j, numpy and python-codicefiscale`.

9 Conclusion

Some interesting conclusions can be drawn from the development of this project:
Graph databases are, if well designed, easy to use and really scalable.
The creation of credible databases using Python could be really convenient to simulate a real situation and understand in advance which behaviours could be useful to manage the situation in exam.
Graph databases are more flexible than the classical relational databases since they allow to create same situations in an easier and more realistic way.

10 References and Sources

- Random-italian-person package: https://pypi.org/project/random-italian-person
- Neo4j package: https://neo4j.com/docs/python-manual/current
- Tkinter package: https://docs.python.org/3/library/tkinter.html
- Overpass turbo API: https://wiki.openstreetmap.org/wiki/Overpass_API