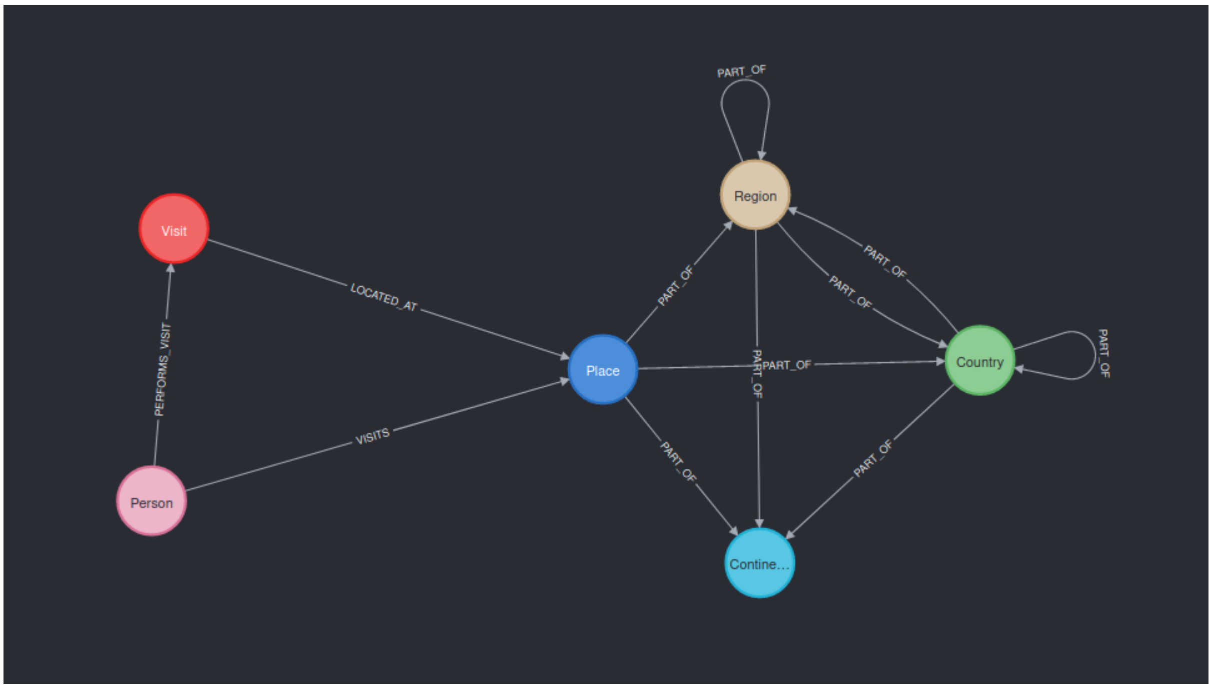
Graph Analytics activity on Neo4j contact tracing sandbox

The chosen sandbox for this graph analytics activity contains contact tracing data. Since it's quite a main topic nowadays, it immediately caught my interest.

The dataset is relatively simple, having as main nodes *Person*, *Visit* and *Place*. Then, connected to the Place node, we have *Region*, *Country*, and *Continent*.



1. *Schema of the database.*

Out of more than 5500 nodes, 500 represent people. These are almost equally divided between healthy and sick. *Visit* is by far the most common node, having 5000 of them, while there are around 100 different places, all of them part of the same region, country and continent.

Let's start writing some queries!

First of all, we can get an idea of how many healthy people are potentially infected by one single sick person (note that timestamps are not taken into consideration here)

```
MATCH (p1:Person)-[:VISITS]->(p:Place)<-[:VISITS]-(p2:Person)
WHERE p1.name = 'Erika Daniels' AND p2.healthstatus = 'Healthy'
RETURN p1,p,p2
```

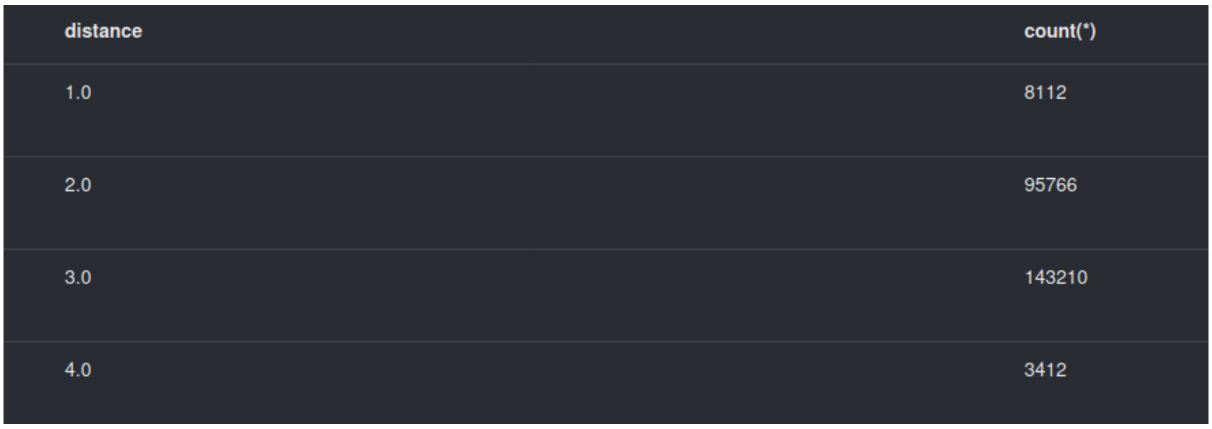
Path Finding

Now, since we'd like to apply some graph data science here, a new type of relationship is required. We can create the *[:MEETS]* relationship between different people (*id(p1)<id(p2)*), having *meettime* as the only attribute.

```
MATCH (p1:Person)-[v1:VISITS]->(pl:Place)<-[v2:VISITS]-(p2:Person)
WHERE id(p1)<id(p2)
WITH p1, p2, apoc.coll.max([v1.starttime.epochMillis,
v2.starttime.epochMillis]) AS maxStart,
     apoc.coll.min([v1.endtime.epochMillis,
v2.endtime.epochMillis]) AS minEnd
WHERE maxStart <= minEnd
WITH p1, p2, sum(minEnd-maxStart) AS meetTime
CREATE (p1)-[:MEETS {meettime: duration({seconds:
meetTime/1000})}]->(p2)
```

This relationship allows us to search for the diameter of relationships between people. This particular query aims to see how many *MEETS* relationships there are for each distance.

```
CALL gds.alpha.allShortestPaths.stream({
nodeProjection: 'Person',
relationshipProjection: {
MEETS: {
type: 'MEETS',
orientation: 'UNDIRECTED' }}
})
YIELD sourceNodeId, targetNodeId, distance
return distinct distance, count(*)
```

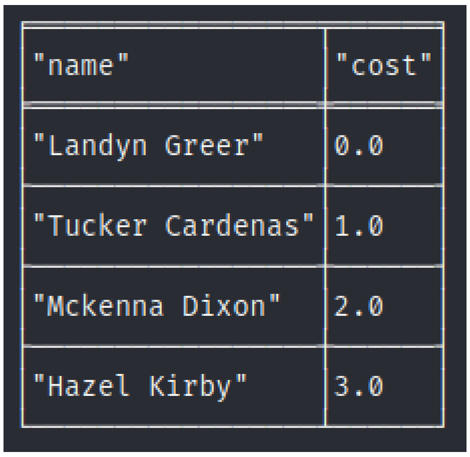


distance	count(*)
1.0	8112
2.0	95766
3.0	143210
4.0	3412

As we can see from the results, the graph is very dense, having the maximum distance (diameter) between two individuals at 4.0. Most of the couples in the graph are either 2 or 3 people away from each other.

Let’s take two random people, one healthy and one sick, and let’s compute the shortest path between them:

```
MATCH (p1:Person {name: 'Landyn Greer'}), (p2:Person {name: 'Hazel
Kirby'})
CALL gds.alpha.shortestPath.stream({
nodeProjection: 'Person',
relationshipProjection: {
MEETS: { type: 'MEETS',
orientation: 'UNDIRECTED'}},
startNode: p1,
endNode: p2})
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).name AS name, cost
```



```
"name"            "cost"
"Landyn Greer"    0.0
"Tucker Cardenas" 1.0
"Mckenna Dixon"   2.0
"Hazel Kirby"     3.0
```

Community Detection

Now, let’s create a graph containing all the possible infections due to interactions between people.

```
CALL gds.graph.create('possInfections', 'Person', 'MEETS')
```

Although we have already stated that the people in the current database are very interconnected, let’s search for weakly connected components, maybe there are a few. The next query returns only one component, that means (even if it could be misunderstood), that there are no *WCC* in the graph. Why?

```
CALL gds.wcc.stream('possInfections')
YIELD componentId
RETURN COUNT(DISTINCT componentId)
```

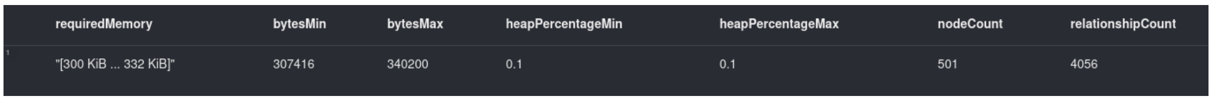
Using a different return condition of the algorithm helps us understand. ALL the people nodes are in the only *WCC* present. Long story short, it’s not actually a weakly connected one, but it’s the only one, containing the whole graph!

```
CALL gds.wcc.stream('possInfections')
YIELD nodeId, componentId
RETURN componentId, COUNT(nodeId)
```

Estimation for the previous named graph

```
CALL gds.graph.create.estimate('Person', 'MEETS')
YIELD requiredMemory, bytesMin, bytesMax, heapPercentageMin,
heapPercentageMax, nodeCount, relationshipCount
```

Reading the results is quite simple, the memory required is between 300 KiB and 322 KiB



	requiredMemory	bytesMin	bytesMax	heapPercentageMin	heapPercentageMax	nodeCount	relationshipCount
1	"[300 KiB ... 332 KiB]"	307416	340200	0.1	0.1	501	4056

Centrality

The same named-graph can be used for multiple purposes: we want to call the pagerank algorithm, but instead of streaming the results we wish to add a new property to the Person nodes.
First we will evaluate the memory required to add this property to all Person nodes.

```
CALL gds.pageRank.write.estimate('possInfections', {
  writeProperty: 'pageRank',
  maxIterations: 20,
  dampingFactor: 0.85
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax,
requiredMemory
```

Then we call the actual write execution mode

```
CALL gds.pageRank.write('possInfections', {
  maxIterations: 20,
  dampingFactor: 0.85,
  writeProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations
```

16 iterations were required in this case.

Once the property is written, we can use it in queries. For instance, let's show all the people who have a pagerank score bigger than 1:

```
MATCH (p:Person) WHERE p.pagerank > 1
RETURN p.name as name, p.pagerank AS pagerank
ORDER BY pagerank DESC
```

Similarity

First thing first, we should create a named graph containing people and visits.

Memory estimation

```
CALL gds.graph.create.cypher.estimate(
'MATCH (p1:Person) RETURN id(p1) AS id',
'MATCH (p1:Person)-[:VISITS]->(p:Place)<-[:VISITS]-(p2:Person)
RETURN id(p1) AS source, id(p2) AS target, p.name AS placeName,
p.type as placeType'
)
```

Creation of the named-graph called *person-visit-place* with cypher projection

```
CALL gds.graph.create.cypher(
'person-visit-place',
'MATCH (p1:Person) RETURN id(p1) AS id',
'MATCH (p1:Person)-[:VISITS]->(p:Place)<-[:VISITS]-(p2:Person)
RETURN id(p1) AS source, id(p2) AS target, p.name AS placeName,
p.type as placeType'
)
```

Then call the node similarity on the projection.

```
CALL gds.nodeSimilarity.stream('person-visit-place')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1,
gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESC, Person1, Person2
```

As we can see from the results, *Lacey Dixon* and *Zavier Galloway* have the highest similarity at almost 0.75, while the lowest is 0.63 with *Rose Tyler* and *Ashley Mcdaniel.*
This metric means that those with highest similarity have more common visited places than the others. We can interpret the result as a greater probability of these people to meet each other again.
Again, this gap in similarity is narrow because, as we've seen before, the people in the dataset are all interconnected (the diameter for the entire graph was 4).

Conclusion

Working on the contact tracing dataset has been very interesting, as some of the problems the world is facing right now could use some help from this graph analytics techniques.
The path finding section can give us a hint on how connected the graph is and how easy it would be for the virus to reach one specific person. It could be useful knowing how much time it takes for one person to infect another one (statistically).

The community detection algorithm confirmed what we saw before, the nodes are very interconnected and there are no WCC. In a situation like this, having small WCC could be useful, those would be easier to quarantine and separate from the rest of the world, ensuring that the virus doesn’t infect them.

With the centrality measure we realized who were the people with a higher level of contacts within the community, observing and tracking them could help in order to prevent a massive spread.

Finally, analyzing the similarity between couples of random people can be useful to prevent infection between people that visit the same places, but maybe don’t know each other.