UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTMENT:

Scienze matematiche e informatiche, scienze fisiche e scienze della terra

DEGREE COURSE:

INFORMATICA (L-31)

DATABASE II PROJECT

IDENTIFY CRIMINAL ACTIVITIES IN CALL RECORDS

By: Tien M. Nguyen
Guided by: Professor Antonio Celesti

Academic year 2020-2021

Table of Contents

I. Problematica affrontata

Call records are a good source of information on real-life networks but getting information from complex data sets can be difficult. It is now well-known that mobile phone data can be an asset to analysts tasked with investigating criminal activities. Phone operators are authorized to collect information about their users, for how long and where. In some cases, that data may be used by law enforcement.

To present our case of exploitation, let's use a common scenario: in a residential area, a store robbery was committed during the day by a gang of four criminals. Criminals are hidden, use a stolen car and do not leave fingerprints. In that case, finding the answer may take a lot of work. With a letter of authority, law enforcement can contact mobile operators to gather information on calls made and received around the robbery in the event of a robbery.

From there, the first step for researchers who want to use graph technology, is to model data as a graph. Data, telephone operators provide compliance with it, usually a table (list) but naturally, phone recording data forms a graph, or network, of devices connected by calls. For years, researchers had to work with such data as tables and lines because the technology used, the data, was structured that way. Trying to identify different phone numbers and their relationships from a spreadsheet for example is tedious. Instead, graph technology allows us to work with data in its natural way.

In this project, we will use two types of DBMS:

- Graph Database with Neo4j
- Traditional ORDBMS with Oracle Database

In conclusion part, I will compare these two types of databases to see advantages and disadvantages of both.

II. Soluzione database considerata

Following is my presentation of the characteristics of two types of Databases used in my project.

1. Neo4j overview

Neo4j stores data on nodes, building different data structures based on relationships.
In Neo4j, there are a few definitions that should be clarified:

- Node (node):
  - One of the two basic units that make up a graph.
  - Usually represent entities (depending on domain relationships).
  - Nodes and relationships can both contain attributes.
  - The simplest graph has only a single node.
- Relationships:
  - A relationship connects two nodes.
  - Relationships organize nodes into structures, allowing graphs to resemble trees, lists, maps, or composite entities.
  - Relationships can have attributes.
  - The relationship connecting two nodes is guaranteed to be valid from the start node to the end node.
  - The relationship is always directed, is determined in the direction of entering or leaving a node => is an important factor to be used when traversing the graph.
  - A node can be related to itself.
- Properties:
  - Attributes are key-value pairs where the primary key is a string.
  - Attribute does not contain null value, if an attribute has value = null then the attribute is considered as non-existent.
- Labels:
  - Is the name of a graph structure to group nodes into a set (set).
  - Labels define domains by grouping nodes into collections with certain names (label names).
  - Labels that can be added or removed at runtime are used to mark the temporary state of the nodes. A node can have 0 or more labels.
- Traversal: A way of querying the graph, navigating from a node to related nodes.

To query in Neo4j, they made Cypher Query Language. Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. Cypher makes querying graph data easy to learn, understand, and use for everyone, but also incorporate the power and functionality of other standard data access languages. Cypher's syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. Through Cypher, users can construct expressive and efficient queries to handle needed create, read, update, and delete functionality.

To interact with Neo4j Database, we can use Neo4j Browser and can directly get response as a beautiful graph visualization or a table. Besides that, we can interact with Neo4j through driver built by Neo4j from high levels program language. Neo4j officially supports the drivers for .Net, Java, JavaScript, Go, and Python.

2. Oracle Database overview

Oracle was born as a Relational DBMS with the SQL language used a lot, but with the progress of the DBMS studies, it has evolved into an Object-Relational DBMS, a hybrid structure between RDBMS and OODBMS (Object-Oriented DBMS).

Oracle uses relational theory for creating tables with attributes (columns) and records (rows) for storing data. Each record must be identified with a primary key, which can be an attribute or a set of attributes.
To implement the relationships between the tables, foreign keys are used, which link a set of attributes (X) from table A with a set of attributes (Y) from table B.

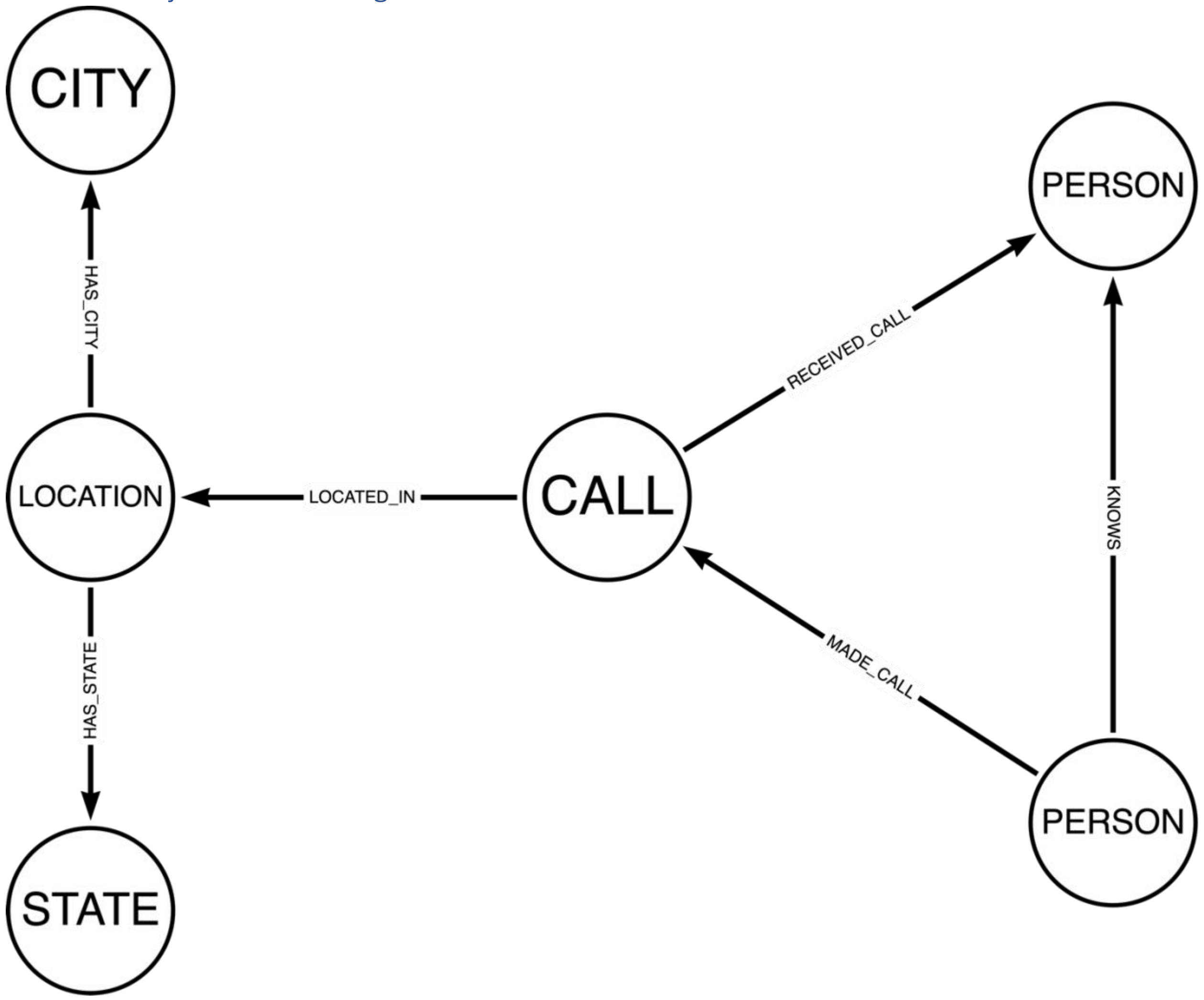
With the evolution to ORDBMS, Oracle has introduced the possibility of creating much more complex datatypes than the basic ones, necessary for many fields of study and work.
Also introducing the procedural language PL / SQL, which offers the ability to implement functions, procedures, triggers and cursors for extensive data manipulation and management.

To interact with Oracle, you can use the SQL developer development environment or connect via terminal to SQL *Plus. The SQL developer interface is simple and basic.
Also, for Oracle there are a series of drivers that support many programming languages.

Specifically, in this project we will use cx_Oracle in python and because the programming environment is MacOS, we need to install Oracle Instant Client as well.

III. Progettazione

1. Neo4j Database Design



The graphic model above, created with "Arrow", highlights the relationships between the “CITY”, “STATE”, “LOCATION”, “CALL”, “PERSON” as Caller and “PERSON” as Receiver in our data.

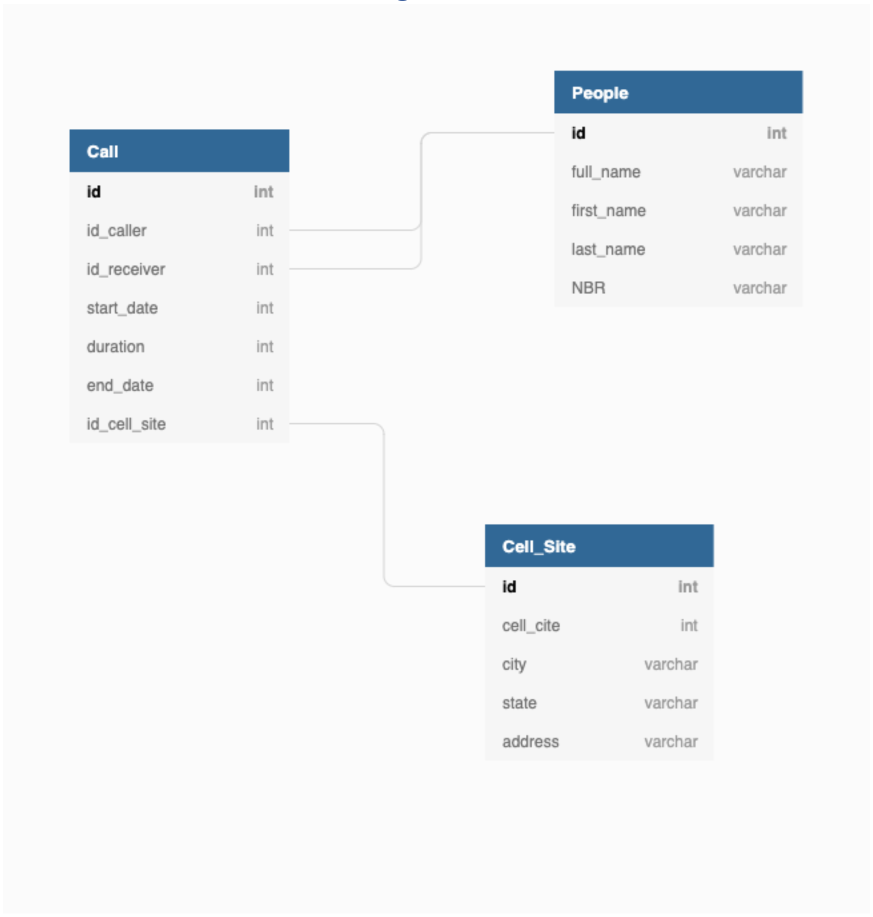
A CALL is a record in our database that will be made of much information. It has an ID in CSV import file, DURATION in seconds, START and END in epoch time format.

PERSON is distinguished by ID and has attributes such as NUMBER, FULL_NAME, FIRST_NAME and LAST_NAME. A Caller is a PERSON who MADE_CALL to a CALL and Receiver is a PERSON who RECEIVED_CALL by a CALL.

LOCATION is an address located in a selected City in a selected State. So, LOCATION will HAS_CITY a CITY and HAS_STATE a STATE.

Finally, a caller made a call to a receiver so we can consider that caller knows the person they made a call to, we create a relationship between two of them KNOWS.

2. Oracle Database Design



In Oracle Database, I design a simple schema that has 3 tables as above:

- Call
  - Contain id_caller and id_receiver which we can get info of that person in People table.
  - Contain id_cell_site so that we can find address where the call was made.
  - Contain other information about call like start_date, duration and end_date.
- People
  - Contain information of a person which are their name and phone number.
- Cell_Site
  - Contain cell site id, address, city and state where that cell site located.

With these 3 tables, we can join them to get results from query.

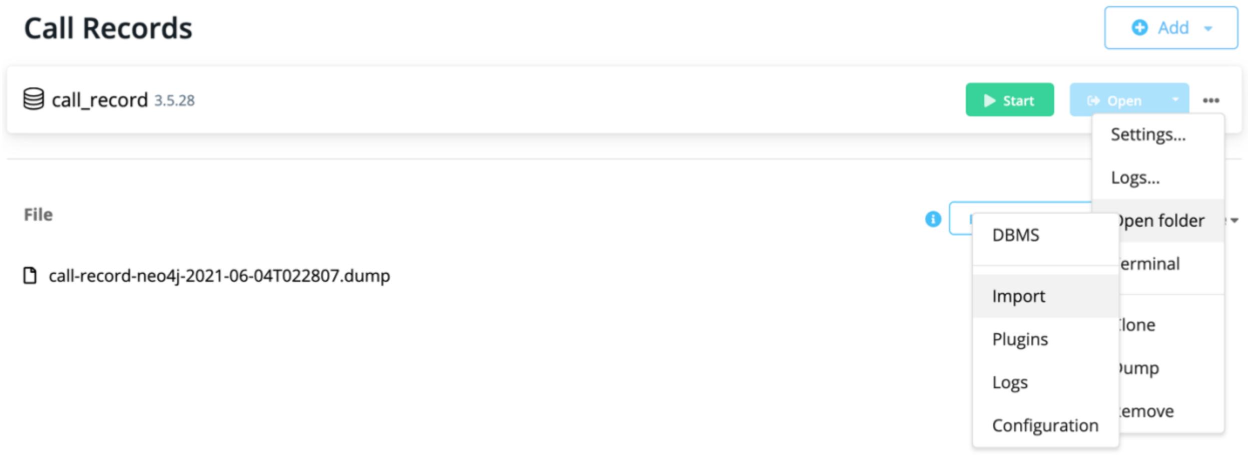
IV. Implementazione

In this section, I will write about how to use python to connect to both Databases using driver, GraphDatabase for Neo4j and cx_Oracle for Oracle Database.
Finally, a part for me explaining the query I build for compare those two Databases in Experiment section.

1. Import Data using CSV file

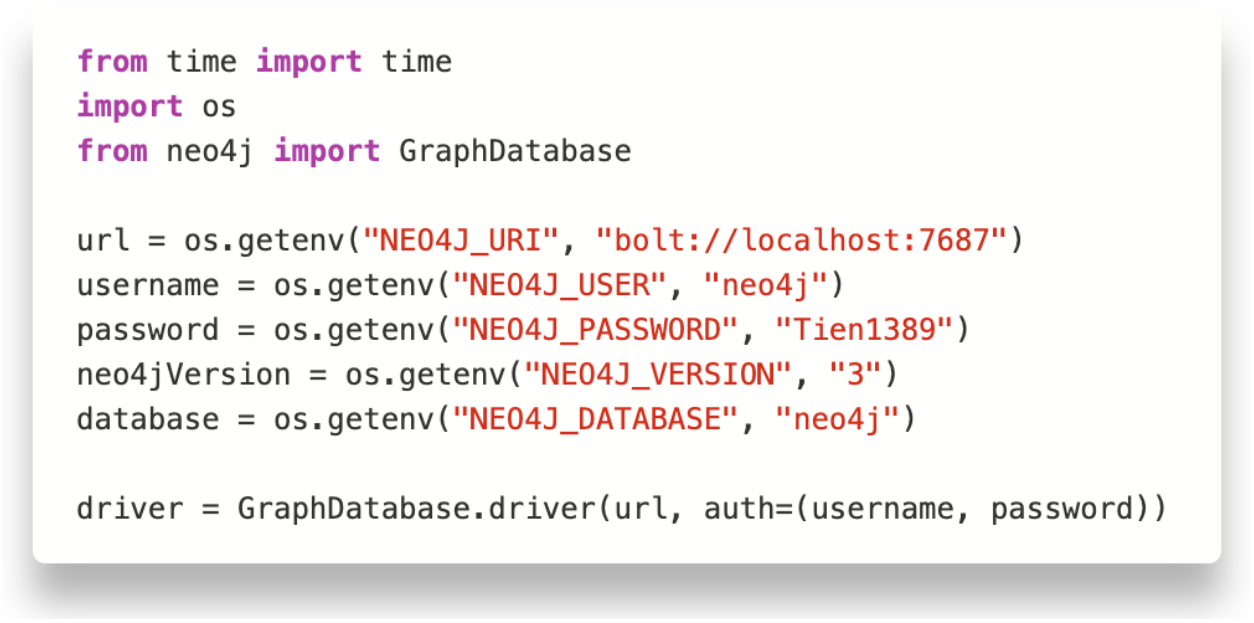
a. Import data to Neo4j Database

Firstly, we need copy **crime_ana_callscsv.csv** and **DB2_People.csv** to import folder of database.



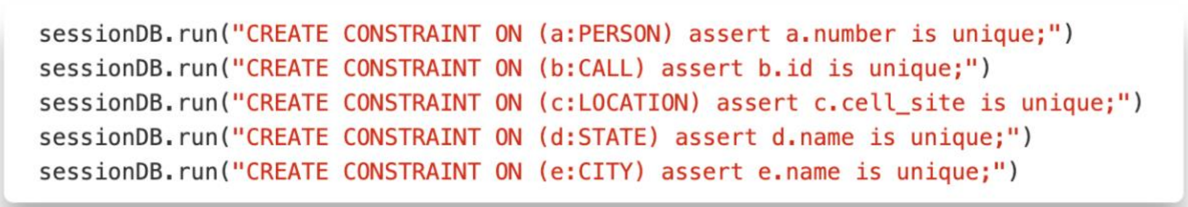
Then we can now import data from calling with python by following steps below:

1. Import needed library and config for neo4j connection



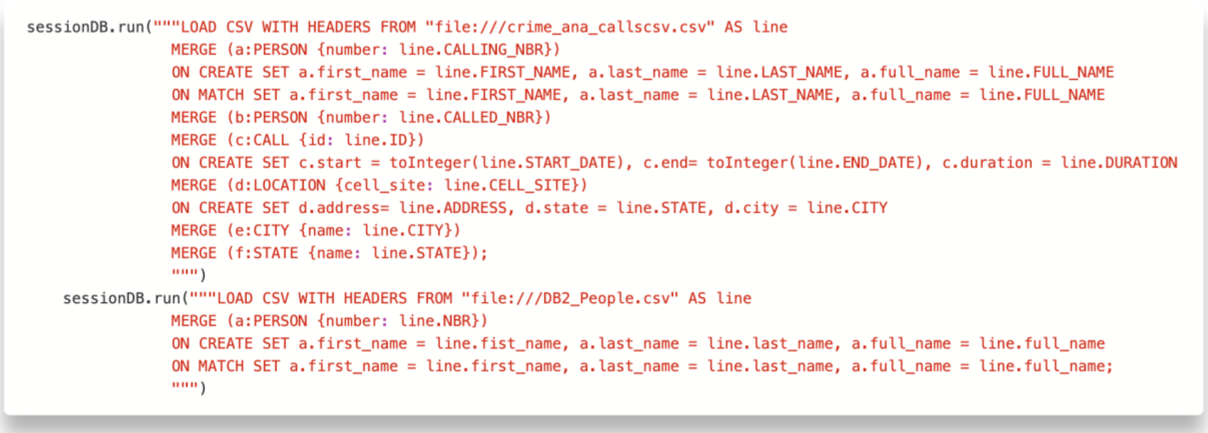
```
from time import time
import os
from neo4j import GraphDatabase

url = os.getenv("NEO4J_URI", "bolt://localhost:7687")
username = os.getenv("NEO4J_USER", "neo4j")
password = os.getenv("NEO4J_PASSWORD", "Tien1389")
neo4jVersion = os.getenv("NEO4J_VERSION", "3")
database = os.getenv("NEO4J_DATABASE", "neo4j")

driver = GraphDatabase.driver(url, auth=(username, password))
```

2. Setup initial constraints



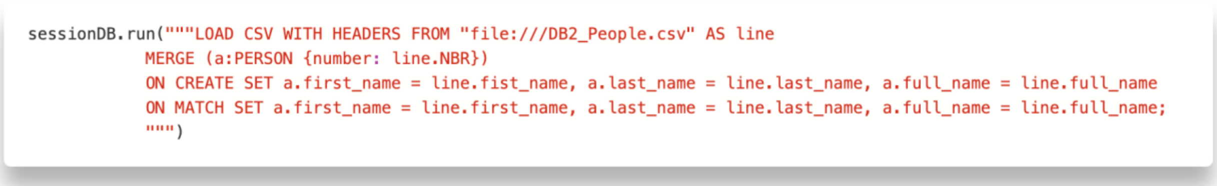
```
sessionDB.run("CREATE CONSTRAINT ON (a:PERSON) assert a.number is unique;")
sessionDB.run("CREATE CONSTRAINT ON (b:CALL) assert b.id is unique;")
sessionDB.run("CREATE CONSTRAINT ON (c:LOCATION) assert c.cell_site is unique;")
sessionDB.run("CREATE CONSTRAINT ON (d:STATE) assert d.name is unique;")
sessionDB.run("CREATE CONSTRAINT ON (e:CITY) assert e.name is unique;")
```

3. Load data to database from csv file



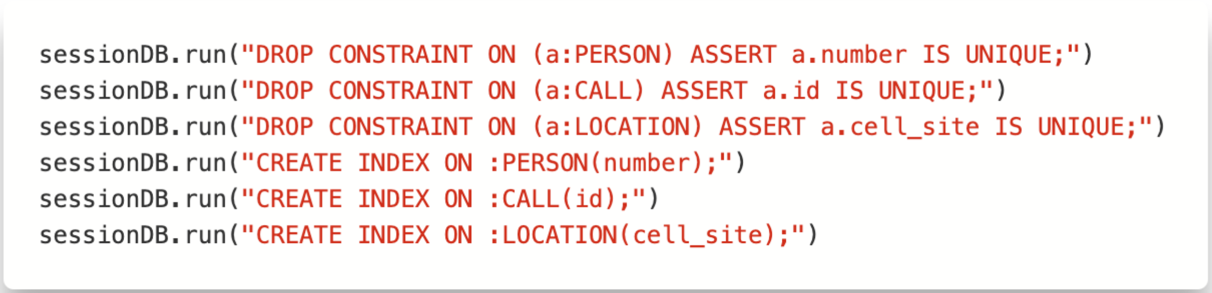
```
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///crime_ana_callscsv.csv" AS line
                MERGE (a:PERSON {number: line.CALLING_NBR})
                ON CREATE SET a.first_name = line.FIRST_NAME, a.last_name = line.LAST_NAME, a.full_name = line.FULL_NAME
                ON MATCH SET a.first_name = line.FIRST_NAME, a.last_name = line.LAST_NAME, a.full_name = line.FULL_NAME
                MERGE (b:PERSON {number: line.CALLED_NBR})
                MERGE (c:CALL {id: line.ID})
                ON CREATE SET c.start = toInteger(line.START_DATE), c.end= toInteger(line.END_DATE), c.duration = line.DURATION
                MERGE (d:LOCATION {cell_site: line.CELL_SITE})
                ON CREATE SET d.address= line.ADDRESS, d.state = line.STATE, d.city = line.CITY
                MERGE (e:CITY {name: line.CITY})
                MERGE (f:STATE {name: line.STATE});
                """)
    sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///DB2_People.csv" AS line
                MERGE (a:PERSON {number: line.NBR})
                ON CREATE SET a.first_name = line.fist_name, a.last_name = line.last_name, a.full_name = line.full_name
                ON MATCH SET a.first_name = line.first_name, a.last_name = line.last_name, a.full_name = line.full_name;
                """)
```

4. Because we don't have info of receiver in the spreadsheet, so we need to import DB2_People to take info for them.



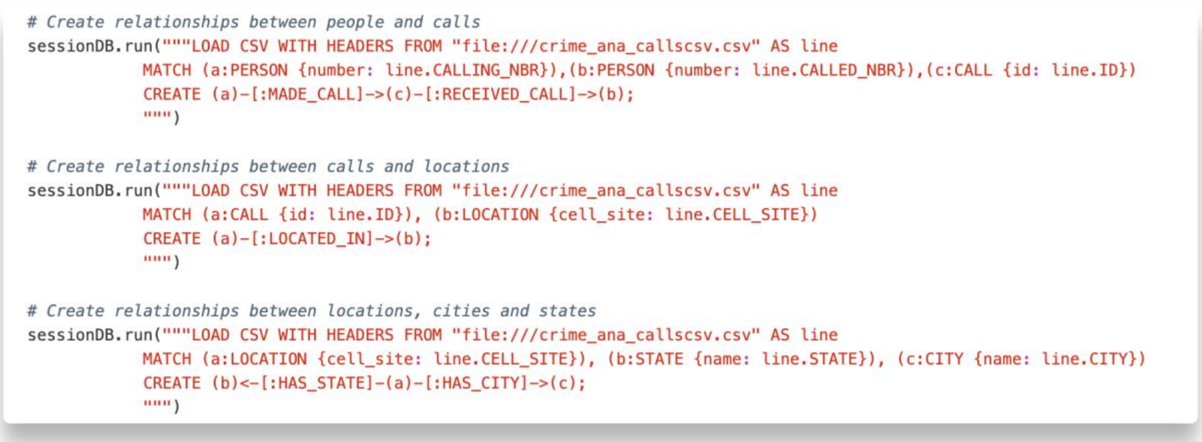
```
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///DB2_People.csv" AS line
            MERGE (a:PERSON {number: line.NBR})
            ON CREATE SET a.first_name = line.fist_name, a.last_name = line.last_name, a.full_name = line.full_name
            ON MATCH SET a.first_name = line.first_name, a.last_name = line.last_name, a.full_name = line.full_name;
            """)
```

5. Setup proper indexing



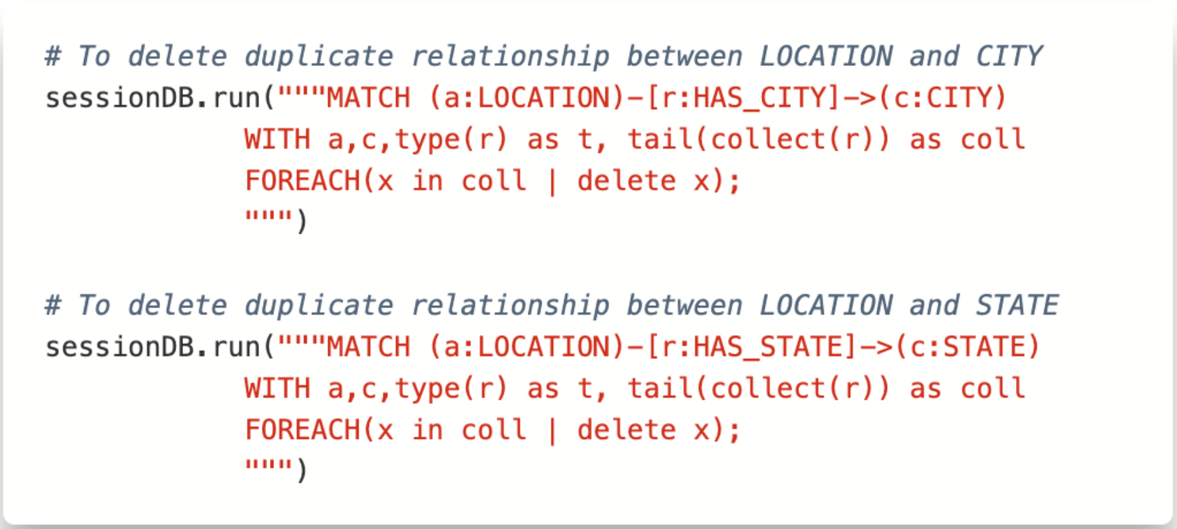
```
sessionDB.run("DROP CONSTRAINT ON (a:PERSON) ASSERT a.number IS UNIQUE;")
sessionDB.run("DROP CONSTRAINT ON (a:CALL) ASSERT a.id IS UNIQUE;")
sessionDB.run("DROP CONSTRAINT ON (a:LOCATION) ASSERT a.cell_site IS UNIQUE;")
sessionDB.run("CREATE INDEX ON :PERSON(number);")
sessionDB.run("CREATE INDEX ON :CALL(id);")
sessionDB.run("CREATE INDEX ON :LOCATION(cell_site);")
```

6. Create relationship between nodes



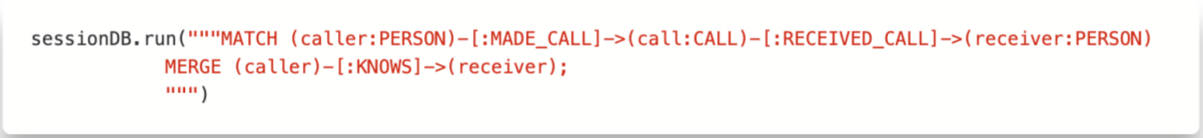
```
# Create relationships between people and calls
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///crime_ana_callscsv.csv" AS line
            MATCH (a:PERSON {number: line.CALLING_NBR}),(b:PERSON {number: line.CALLED_NBR}),(c:CALL {id: line.ID})
            CREATE (a)-[:MADE_CALL]->(c)-[:RECEIVED_CALL]->(b);
            """)

# Create relationships between calls and locations
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///crime_ana_callscsv.csv" AS line
            MATCH (a:CALL {id: line.ID}), (b:LOCATION {cell_site: line.CELL_SITE})
            CREATE (a)-[:LOCATED_IN]->(b);
            """)

# Create relationships between locations, cities and states
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///crime_ana_callscsv.csv" AS line
            MATCH (a:LOCATION {cell_site: line.CELL_SITE}), (b:STATE {name: line.STATE}), (c:CITY {name: line.CITY})
            CREATE (b)<-[:HAS_STATE]-(a)-[:HAS_CITY]->(c);
            """)
```

7. Because in the step create relationship, we created many same relationships within two nodes, so we need to delete those duplicated one.



```
# To delete duplicate relationship between LOCATION and CITY
sessionDB.run("""MATCH (a:LOCATION)-[r:HAS_CITY]->(c:CITY)
            WITH a,c,type(r) as t, tail(collect(r)) as coll
            FOREACH(x in coll | delete x);
            """)

# To delete duplicate relationship between LOCATION and STATE
sessionDB.run("""MATCH (a:LOCATION)-[r:HAS_STATE]->(c:STATE)
            WITH a,c,type(r) as t, tail(collect(r)) as coll
            FOREACH(x in coll | delete x);
            """)
```

8. Create KNOWS relationship between peoples as they are calling each other



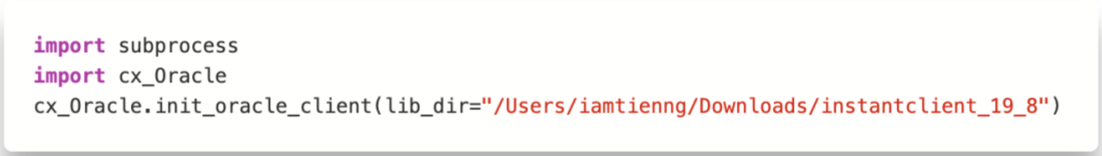
```
sessionDB.run("""MATCH (caller:PERSON)-[:MADE_CALL]->(call:CALL)-[:RECEIVED_CALL]->(receiver:PERSON)
          MERGE (caller)-[:KNOWS]->(receiver);
          """)
```

Here we done importing our data to Neo4j dataset.

b. Import data to Oracle Database

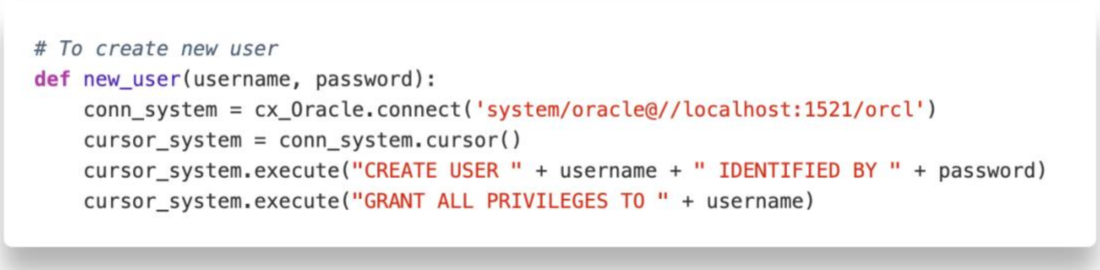
To import data to Oracle Database from Python, we use cx_oracle and Instant Client to create connect between Server side and Database side. And about importing data from CSV files, we use SQL *LOADER, a CLI program to import data from bash.

1. Make sure we import library to Python project



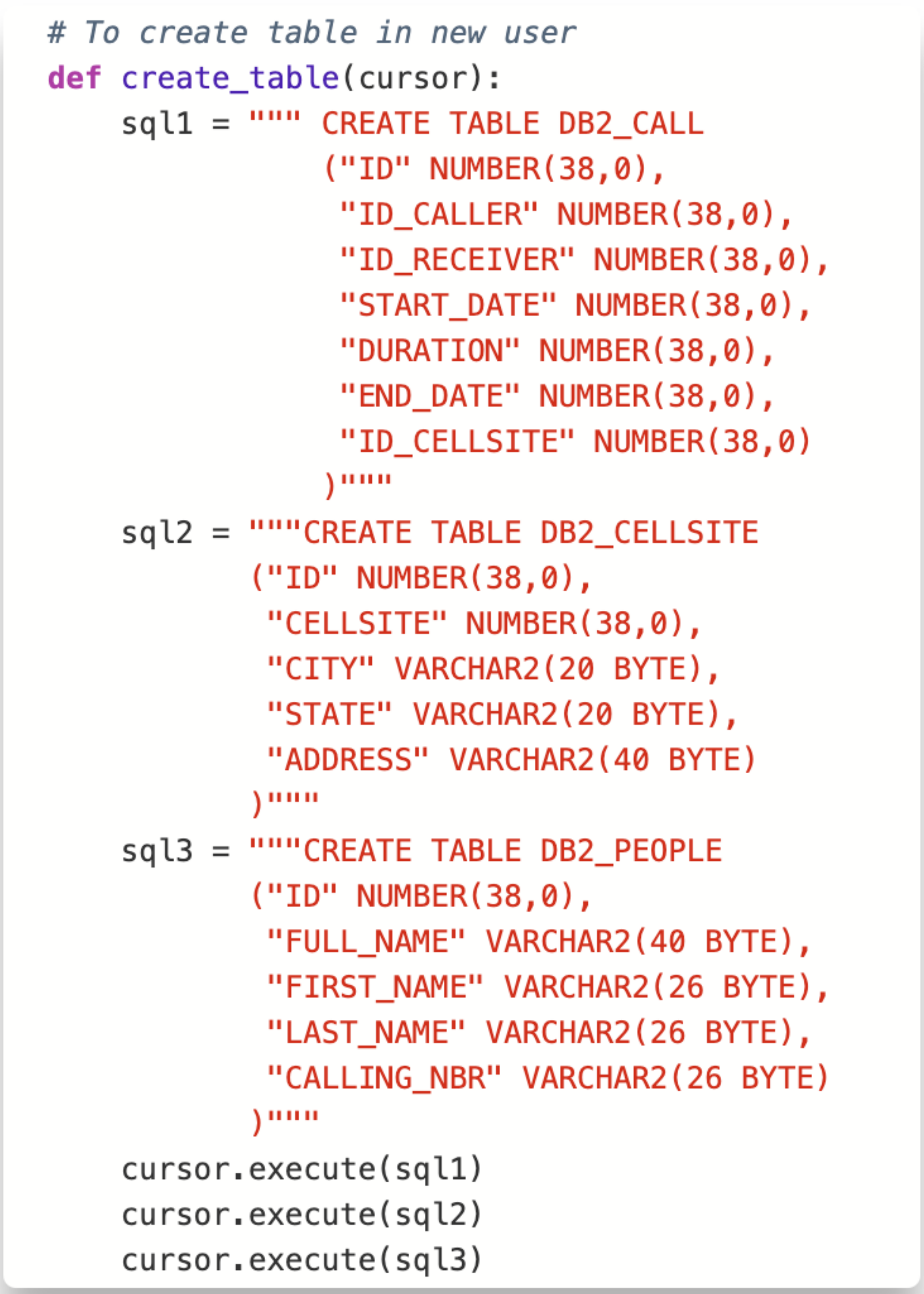
```python
import subprocess
import cx_Oracle
cx_Oracle.init_oracle_client(lib_dir="/Users/iamtienng/Downloads/instantclient_19_8")
```

2. First of all, we need to create new user for new Database, and it needed to be granted proper privileges. And to create new user, we need to connect to System user.



```python
# To create new user
def new_user(username, password):
    conn_system = cx_Oracle.connect('system/oracle@//localhost:1521/orcl')
    cursor_system = conn_system.cursor()
    cursor_system.execute("CREATE USER " + username + " IDENTIFIED BY " + password)
    cursor_system.execute("GRANT ALL PRIVILEGES TO " + username)
```

3. Before importing data from CSV files, it is necessary create tables.



```python
# To create table in new user
def create_table(cursor):
    sql1 = """ CREATE TABLE DB2_CALL
               ("ID" NUMBER(38,0),
                "ID_CALLER" NUMBER(38,0),
                "ID_RECEIVER" NUMBER(38,0),
                "START_DATE" NUMBER(38,0),
                "DURATION" NUMBER(38,0),
                "END_DATE" NUMBER(38,0),
                "ID_CELLSITE" NUMBER(38,0)
               )"""
    sql2 = """CREATE TABLE DB2_CELLSITE
           ("ID" NUMBER(38,0),
            "CELLSITE" NUMBER(38,0),
            "CITY" VARCHAR2(20 BYTE),
            "STATE" VARCHAR2(20 BYTE),
            "ADDRESS" VARCHAR2(40 BYTE)
           )"""
    sql3 = """CREATE TABLE DB2_PEOPLE
           ("ID" NUMBER(38,0),
            "FULL_NAME" VARCHAR2(40 BYTE),
            "FIRST_NAME" VARCHAR2(26 BYTE),
            "LAST_NAME" VARCHAR2(26 BYTE),
            "CALLING_NBR" VARCHAR2(26 BYTE)
           )"""
    cursor.execute(sql1)
    cursor.execute(sql2)
    cursor.execute(sql3)
```

4. For importing data, we use SQL *LOADER. To use SQL *LOADER, we need 3 components which are Empty TABLE, CTL (control file) and CSV file.

We have now Empty Tables we created before at 3rd step, CSV file and here is CTL files:

a. DB2_CALL CTL file

```
LOAD DATA
INTO TABLE DB2_CALL
FIELDS TERMINATED BY ','
(
   id, id_caller, id_receiver, start_date, duration, end_date, id_cellsite
)
```

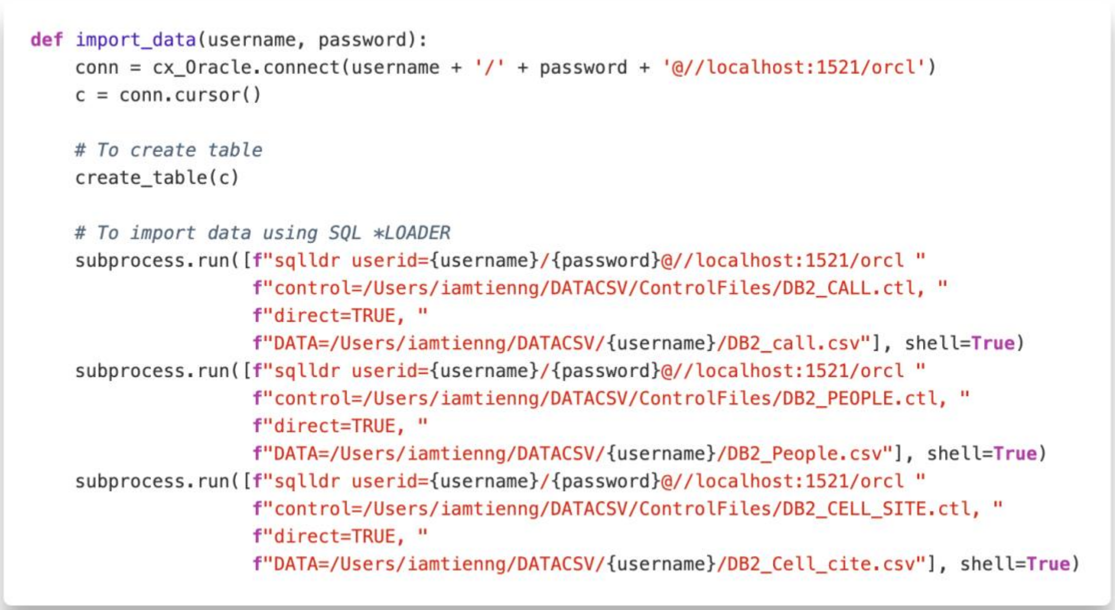
b. DB2_PEOPLE CTL file

```
LOAD DATA
INTO TABLE DB2_PEOPLE
FIELDS TERMINATED BY ','
(
   id, full_name, first_name, last_name, calling_nbr
)
```

c. DB2_CELLSITE CTL file

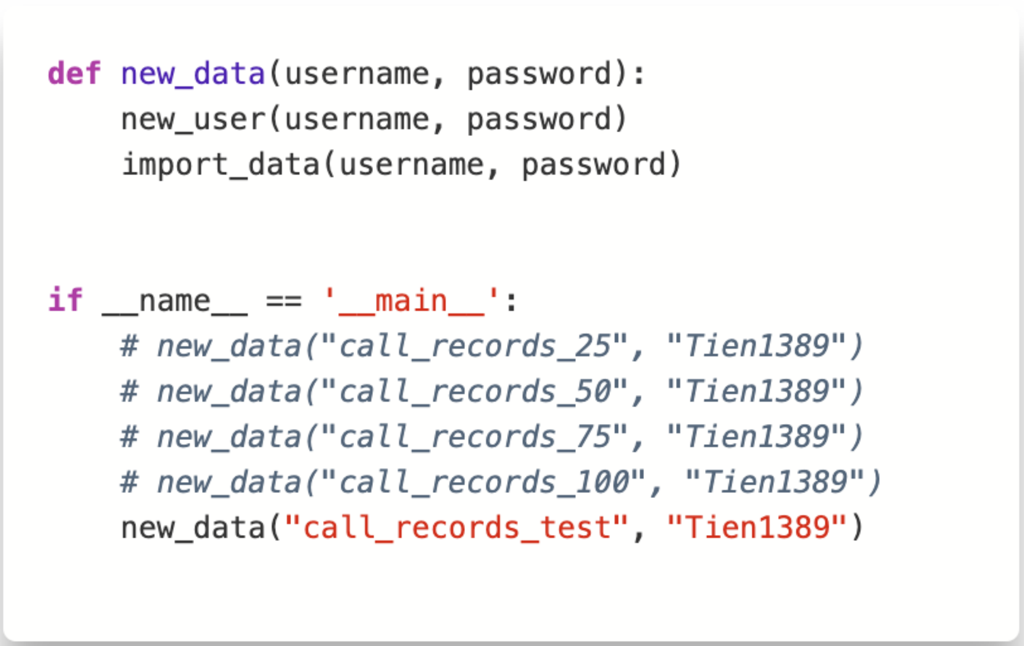
```
LOAD DATA
INTO TABLE DB2_CELLSITE
FIELDS TERMINATED BY ','
(
   id, cellsite, city, state, address
)
```

With all 3 components, we now can import data to table using SQL *LOADER:



```python
def import_data(username, password):
    conn = cx_Oracle.connect(username + '/' + password + '@//localhost:1521/orcl')
    c = conn.cursor()

    # To create table
    create_table(c)

    # To import data using SQL *LOADER
    subprocess.run([f"sqlldr userid={username}/{password}@//localhost:1521/orcl "
                    f"control=/Users/iamtienng/DATACSV/ControlFiles/DB2_CALL.ctl, "
                    f"direct=TRUE, "
                    f"DATA=/Users/iamtienng/DATACSV/{username}/DB2_call.csv"], shell=True)
    subprocess.run([f"sqlldr userid={username}/{password}@//localhost:1521/orcl "
                    f"control=/Users/iamtienng/DATACSV/ControlFiles/DB2_PEOPLE.ctl, "
                    f"direct=TRUE, "
                    f"DATA=/Users/iamtienng/DATACSV/{username}/DB2_People.csv"], shell=True)
    subprocess.run([f"sqlldr userid={username}/{password}@//localhost:1521/orcl "
                    f"control=/Users/iamtienng/DATACSV/ControlFiles/DB2_CELL_SITE.ctl, "
                    f"direct=TRUE, "
                    f"DATA=/Users/iamtienng/DATACSV/{username}/DB2_Cell_cite.csv"], shell=True)
```

We can trigger those functions by:

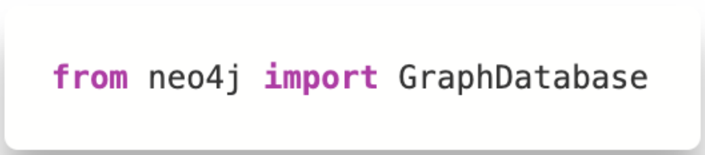


```python
def new_data(username, password):
    new_user(username, password)
    import_data(username, password)


if __name__ == '__main__':
    # new_data("call_records_25", "Tien1389")
    # new_data("call_records_50", "Tien1389")
    # new_data("call_records_75", "Tien1389")
    # new_data("call_records_100", "Tien1389")
    new_data("call_records_test", "Tien1389")
```

5. Connect to Database with Python

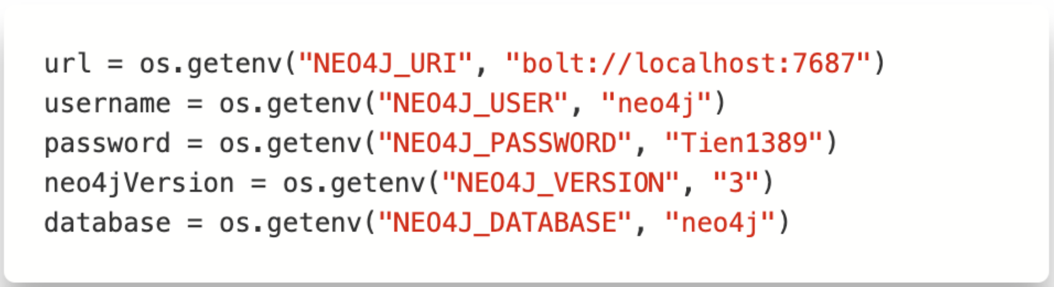
a. Neo4j Python Driver

The Neo4j Python driver is officially supported by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to Python.
To query in Python, firstly we need to import GraphDatabase from neo4j



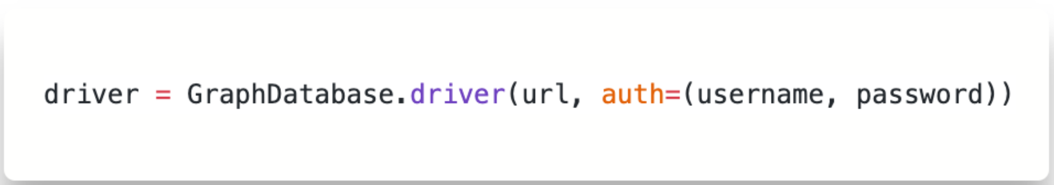
```
from neo4j import GraphDatabase
```

Set information of database we need to connect:



```
url = os.getenv("NEO4J_URI", "bolt://localhost:7687")
username = os.getenv("NEO4J_USER", "neo4j")
password = os.getenv("NEO4J_PASSWORD", "Tien1389")
neo4jVersion = os.getenv("NEO4J_VERSION", "3")
database = os.getenv("NEO4J_DATABASE", "neo4j")
```

And finally, we connect it by function driver of GraphDatabase:

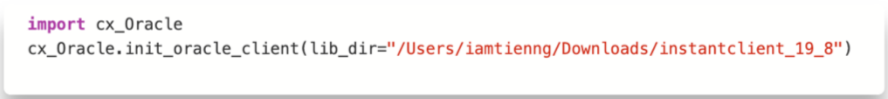


```
driver = GraphDatabase.driver(url, auth=(username, password))
```

b. Oracle cx_Oracle and Oracle Instant Client

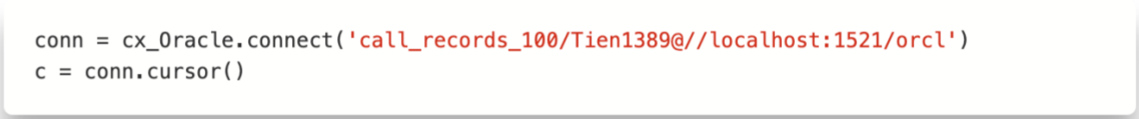
To connect to Oracle Database from Python, first we need to install cx_Oracle via pip for Python and Oracle Instant Client since I use MacOS as my developing environment. The user guide for installation can be found here: cx_Oracle

Then we import cx_Oracle as well as Oracle Instant Client:



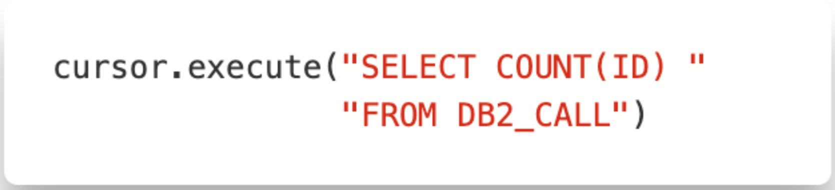
```python
import cx_Oracle
cx_Oracle.init_oracle_client(lib_dir="/Users/iamtienng/Downloads/instantclient_19_8")
```

We create connection with given information (user, password, URL port and service name) about database:



```python
conn = cx_Oracle.connect('call_records_100/Tien1389@//localhost:1521/orcl')
c = conn.cursor()
```

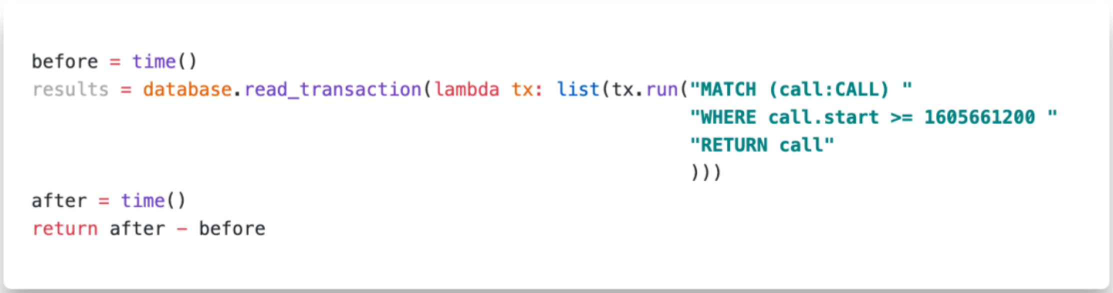
With cursor, we can get response with a runnable query:



```python
cursor.execute("SELECT COUNT(ID) "
               "FROM DB2_CALL")
```

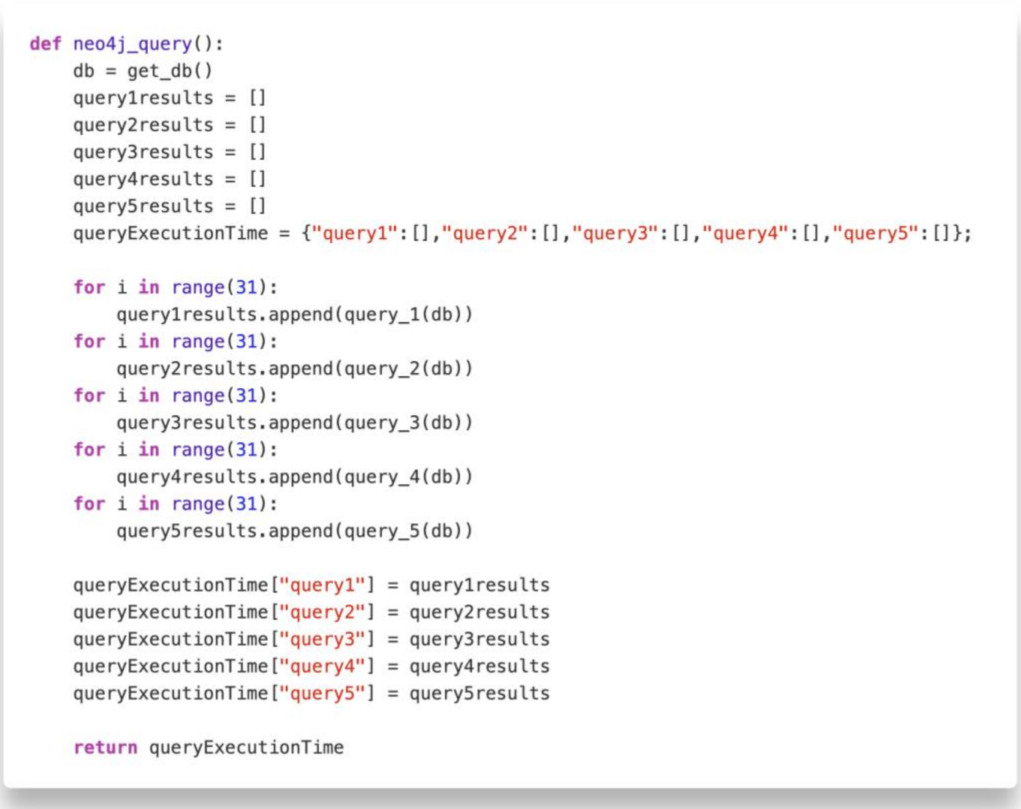
6. Query and execution time

In this section, the standard for me to evaluate the query as well as build the query is the same amount of data that the Database returns in both regardless of the number of operations or complexity included in the query. Details will be in each query I describe below.
To calculate execution time, I use library time in python by this way to return the execution time of the given query:



```
before = time()
results = database.read_transaction(lambda tx: list(tx.run("MATCH (call:CALL) "
                                                           "WHERE call.start >= 1605661200 "
                                                           "RETURN call"
                                                           )))
after = time()
return after - before
```

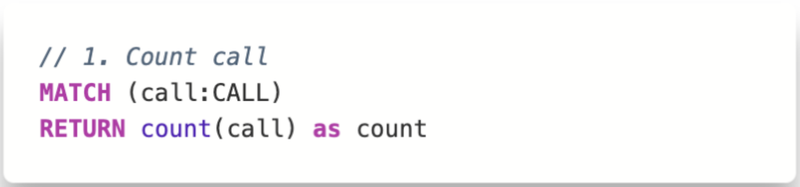
And then, I will use a function to catch them all:



```
def neo4j_query():
    db = get_db()
    query1results = []
    query2results = []
    query3results = []
    query4results = []
    query5results = []
    queryExecutionTime = {"query1":[],"query2":[],"query3":[],"query4":[],"query5":[]};

    for i in range(31):
        query1results.append(query_1(db))
    for i in range(31):
        query2results.append(query_2(db))
    for i in range(31):
        query3results.append(query_3(db))
    for i in range(31):
        query4results.append(query_4(db))
    for i in range(31):
        query5results.append(query_5(db))

    queryExecutionTime["query1"] = query1results
    queryExecutionTime["query2"] = query2results
    queryExecutionTime["query3"] = query3results
    queryExecutionTime["query4"] = query4results
    queryExecutionTime["query5"] = query5results

    return queryExecutionTime
```

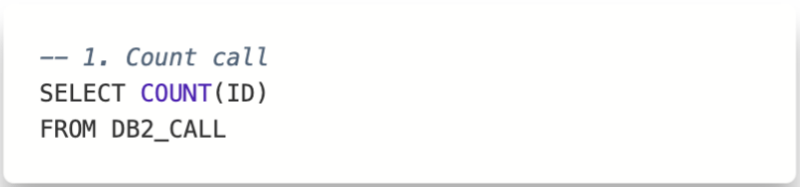
a. 1st Query

Returns the count number of calls contained in the database.
Neo4j:



```
// 1. Count call
MATCH (call:CALL)
RETURN count(call) as count
```

Oracle Database:

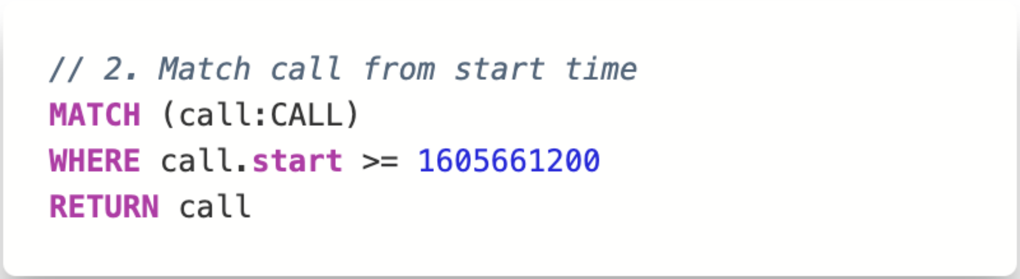


```
-- 1. Count call
SELECT COUNT(ID)
FROM DB2_CALL
```

The purpose of this query is to compare the characteristics of two types of Databases. Because Oracle Database in particular and RDBMS in general will usually store the number of records right in the table, the returned results can be said to be instant. In contrast, a NoSQL type, MongoDB, does not store the number of records in the collection, so the return results can be much longer. Thankfully Neo4j also supports Fast counts using the count store. About speed we will discuss in the Experiment section later.

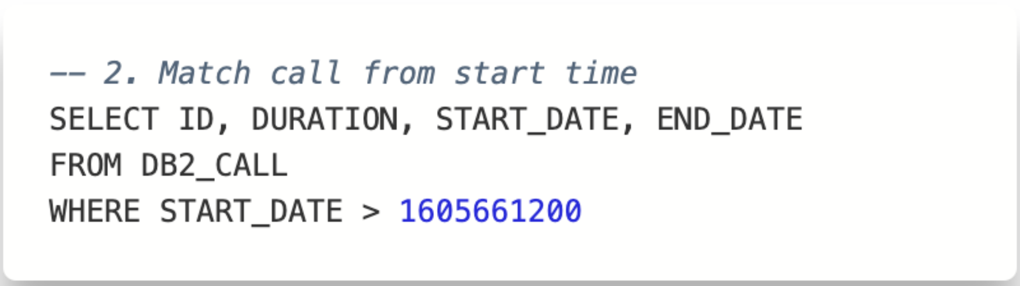
b. 2nd Query

Neo4j:



```
// 2. Match call from start time
MATCH (call:CALL)
WHERE call.start >= 1605661200
RETURN call
```

Oracle Database:

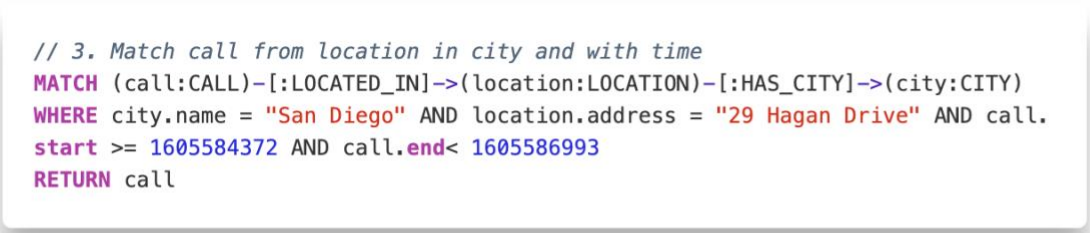


```
-- 2. Match call from start time
SELECT ID, DURATION, START_DATE, END_DATE
FROM DB2_CALL
WHERE START_DATE > 1605661200
```

The second query I just add an operator to query, in Oracle Database it doesn't have to join table yet.

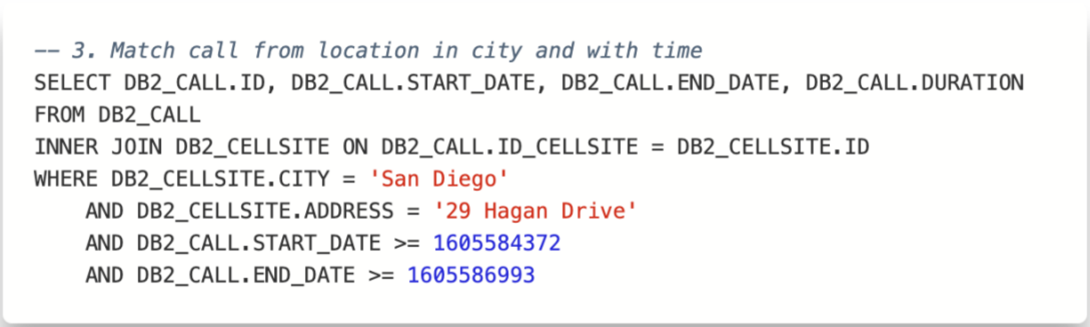
c. 3rd Query

Neo4j:



```
// 3. Match call from location in city and with time
MATCH (call:CALL)-[:LOCATED_IN]->(location:LOCATION)-[:HAS_CITY]->(city:CITY)
WHERE city.name = "San Diego" AND location.address = "29 Hagan Drive" AND call.
start >= 1605584372 AND call.end< 1605586993
RETURN call
```

Oracle Database:

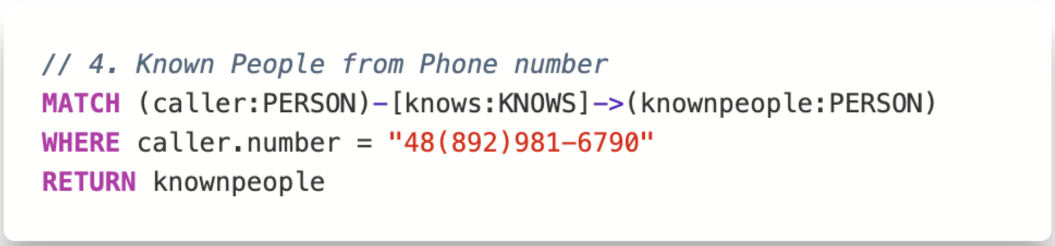


```
-- 3. Match call from location in city and with time
SELECT DB2_CALL.ID, DB2_CALL.START_DATE, DB2_CALL.END_DATE, DB2_CALL.DURATION
FROM DB2_CALL
INNER JOIN DB2_CELLSITE ON DB2_CALL.ID_CELLSITE = DB2_CELLSITE.ID
WHERE DB2_CELLSITE.CITY = 'San Diego'
    AND DB2_CELLSITE.ADDRESS = '29 Hagan Drive'
    AND DB2_CALL.START_DATE >= 1605584372
    AND DB2_CALL.END_DATE >= 1605586993
```

In the third query, I put a join to Oracle Database query and increase number of operators to query. This query is also the main function to find the call at a location in a chosen time to invest crimes through phone calls.

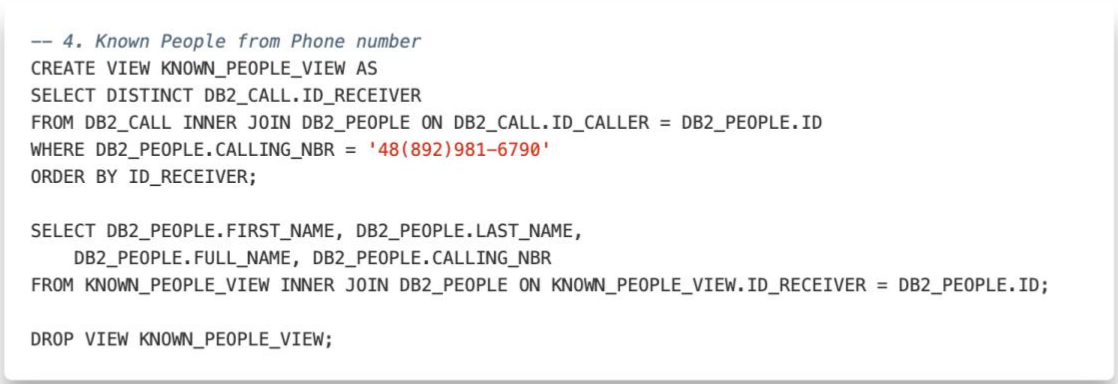
d. 4th Query

Neo4j:



```
// 4. Known People from Phone number
MATCH (caller:PERSON)-[knows:KNOWS]->(knownpeople:PERSON)
WHERE caller.number = "48(892)981-6790"
RETURN knownpeople
```

Oracle Database:



```
-- 4. Known People from Phone number
CREATE VIEW KNOWN_PEOPLE_VIEW AS
SELECT DISTINCT DB2_CALL.ID_RECEIVER
FROM DB2_CALL INNER JOIN DB2_PEOPLE ON DB2_CALL.ID_CALLER = DB2_PEOPLE.ID
WHERE DB2_PEOPLE.CALLING_NBR = '48(892)981-6790'
ORDER BY ID_RECEIVER;

SELECT DB2_PEOPLE.FIRST_NAME, DB2_PEOPLE.LAST_NAME,
    DB2_PEOPLE.FULL_NAME, DB2_PEOPLE.CALLING_NBR
FROM KNOWN_PEOPLE_VIEW INNER JOIN DB2_PEOPLE ON KNOWN_PEOPLE_VIEW.ID_RECEIVER = DB2_PEOPLE.ID;

DROP VIEW KNOWN_PEOPLE_VIEW;
```

In the fourth query, I must create a view in Oracle Database since I cannot join the third table directly. This is where Neo4j shines as the number of relationships has started to get more complicated.

e. 5th Query

Neo4j:

```
// 5. Find call with caller and receiver from ID
MATCH (caller:PERSON)-[:MADE_CALL]->(call:CALL)-[:RECEIVED_CALL]->(receiver:PERSON)
WHERE call.id = "20"
RETURN call, caller, receiver
```

Oracle Database:

```
-- 5. Find call with caller and receiver from ID
CREATE VIEW FIND_CALL_WITH_ID_VIEW AS
SELECT DB2_CALL.ID AS ID_CALL, DB2_CALL.START_DATE,
    DB2_CALL.END_DATE, DB2_CALL.DURATION, DB2_CALL.ID_CALLER,
    DB2_PEOPLE.FIRST_NAME AS CALLER_FIRST_NAME, DB2_PEOPLE.LAST_NAME AS CALLER_LAST_NAME,
    DB2_PEOPLE.FULL_NAME AS CALLER_FULL_NAME, DB2_PEOPLE.CALLING_NBR AS CALLER_NBR,
    DB2_CALL.ID_RECEIVER
FROM DB2_CALL INNER JOIN DB2_PEOPLE ON DB2_CALL.ID_CALLER = DB2_PEOPLE.ID
WHERE DB2_CALL.ID = 20;

SELECT FIND_CALL_WITH_ID_VIEW.*, DB2_PEOPLE.FIRST_NAME AS RECEIVER_FIRST_NAME,
DB2_PEOPLE.LAST_NAME AS RECEIVER_LAST_NAME, DB2_PEOPLE.FULL_NAME AS RECEIVER_FULL_NAME,
DB2_PEOPLE.CALLING_NBR AS RECEIVER_NBR
FROM FIND_CALL_WITH_ID_VIEW INNER JOIN DB2_PEOPLE ON FIND_CALL_WITH_ID_VIEW.ID_RECEIVER = DB2_PEOPLE.ID;

DROP VIEW FIND_CALL_WITH_ID_VIEW;
```

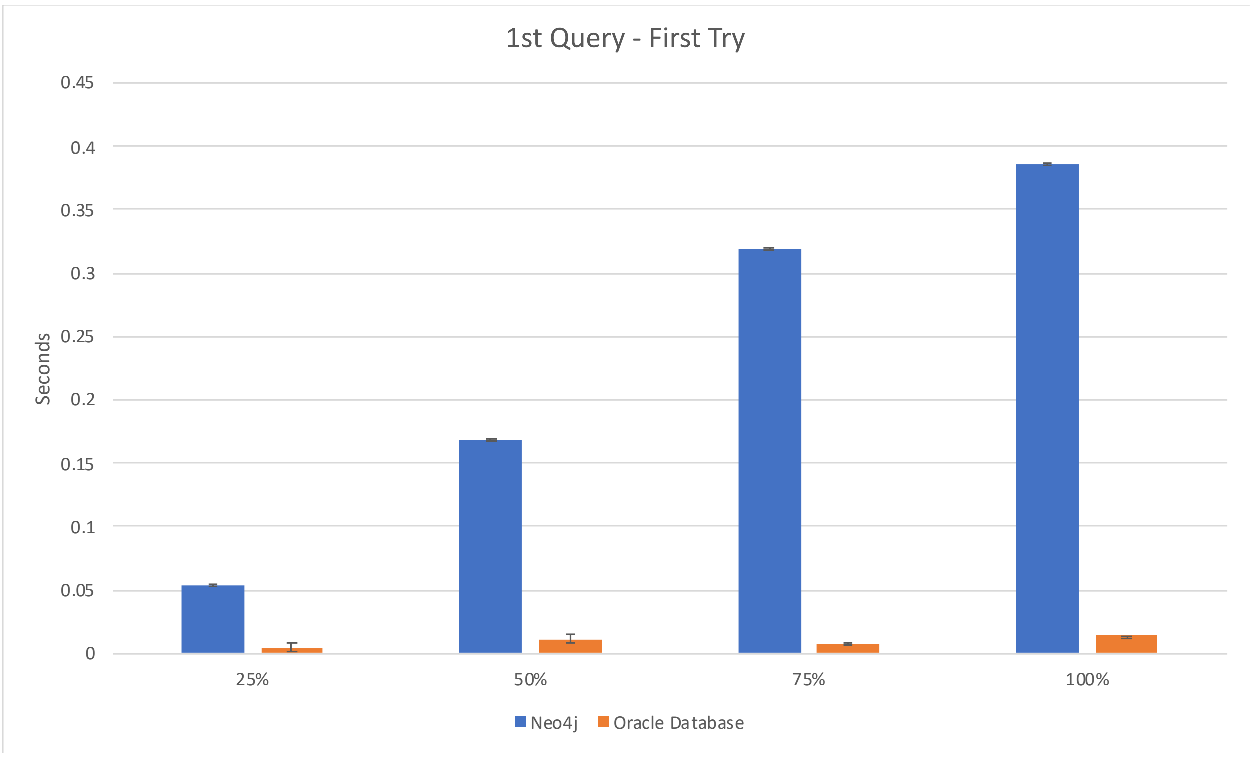
In the fifth query, the complexity is quite like the fourth query, but it requires one more relationship also I increase the attributes in the results.
As we can see, with Cypher, in Neo4j to get results we don’t have to write a complex query otherwise. On the other side, a more traditional DBMS like Oracle Database struggle with query require many relationships.

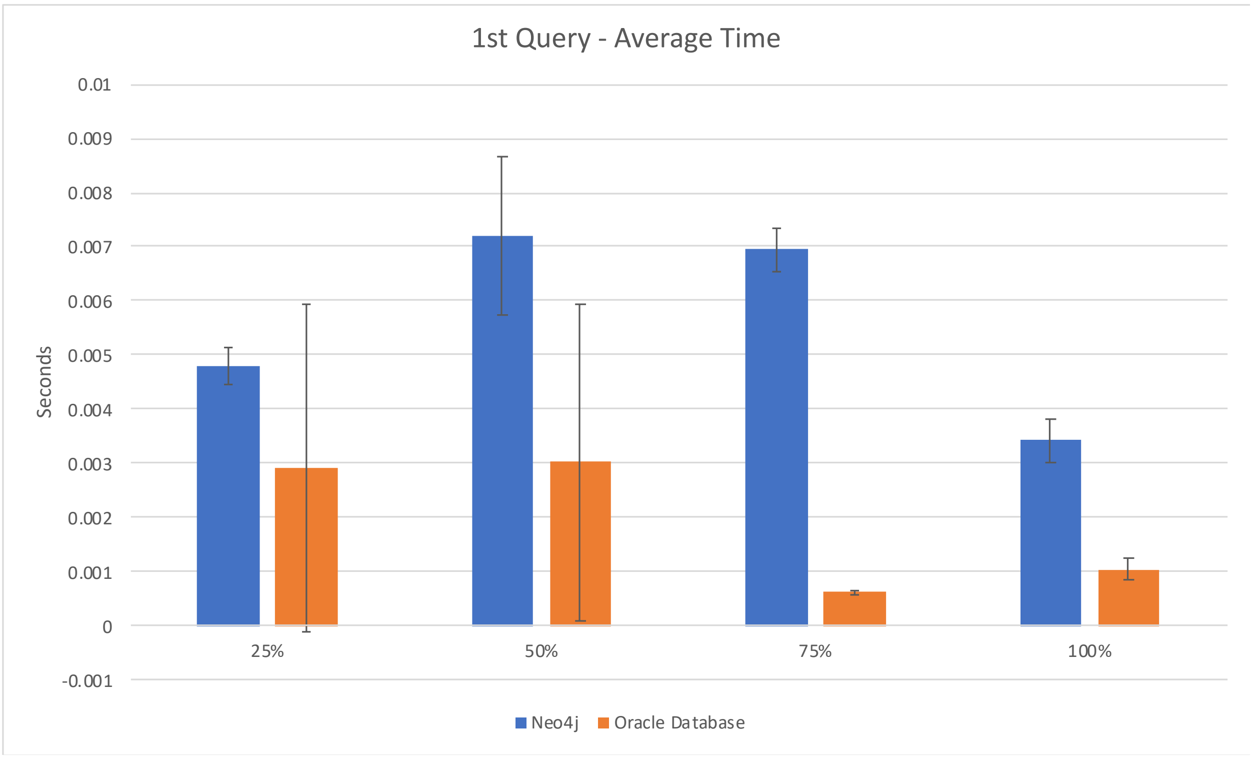
V. Esperimenti

1. 1st Query

Neo4j					Oracle			
25.00%	50.00%	75.00%	100.00%		25.00%	50.00%	75.00%	100.00%
0.05409598	0.16865087	0.31979537	0.38605785		0.00460792	0.01156688	0.00805783	0.01389313
0.00475693	0.00830007	0.00977206	0.00345111		0.03423905	0.03259587	0.00090909	0.00114989
0.0042429	0.01127005	0.00647402	0.0042789		0.00067115	0.00101805	0.00061417	0.00068498
0.00411391	0.005193	0.00650597	0.00786281		0.00071311	0.00060701	0.00057197	0.00089288
0.00402999	0.00611115	0.00669003	0.0056181		0.00061893	0.00073504	0.00066185	0.00107813
0.00473309	0.00828791	0.0072751	0.0027442		0.00064087	0.00069213	0.00063682	0.00103879
0.00444007	0.00991797	0.00801086	0.00298595		0.00071812	0.00509572	0.00063181	0.0008049
0.00539708	0.02600193	0.00786924	0.00333214		0.00059485	0.00065279	0.00058508	0.00107598
0.00454307	0.00461292	0.00694609	0.00345182		0.00054884	0.00071502	0.00061107	0.00070405
0.003865	0.0072279	0.00679684	0.0025959		0.00059605	0.00063992	0.00058794	0.00075698
0.00407004	0.00448585	0.00553584	0.00337195		0.0007019	0.00069594	0.00061226	0.0012548
0.004076	0.00499773	0.0061543	0.00356293		0.0006659	0.00065589	0.0005219	0.00126815
0.0066309	0.00535011	0.00642085	0.00394297		0.00055718	0.00055885	0.00057793	0.00092316
0.00430894	0.00632787	0.00584316	0.00297594		0.00053358	0.00077796	0.00079489	0.00085211
0.00435615	0.00484014	0.00548887	0.00307274	**1st Query**	0.00072098	0.00059295	0.00058913	0.00071216
0.00504017	0.0051949	0.00464892	0.00442123		0.0009768	0.00056314	0.00082207	0.00065899
0.00528693	0.01193595	0.00531483	0.00238299		0.00059319	0.00071597	0.0006218	0.00071311
0.00575423	0.01226282	0.00706792	0.00252509		0.00060296	0.00059009	0.00059199	0.00082684
0.00478482	0.00622296	0.00656676	0.00265503		0.00067186	0.03442097	0.00057697	0.00093913
0.0042057	0.00610709	0.00559998	0.00278401		0.00061226	0.00104785	0.00053	0.00106001
0.00409198	0.00622797	0.00803399	0.00240993		0.00061679	0.00082302	0.00048709	0.00058699
0.00501513	0.0054822	0.00731707	0.00262785		0.0007298	0.00068879	0.00056911	0.00060797
0.00394225	0.00598884	0.00726008	0.0024929		0.00096583	0.00065899	0.00058699	0.00088716
0.00509214	0.00617194	0.00744605	0.00250387		0.00076222	0.00056601	0.00057507	0.00064182
0.00411224	0.00485611	0.00857997	0.00276399		0.03469086	0.00060701	0.00058794	0.00201416
0.0040679	0.00474596	0.0072751	0.0039711		0.00104499	0.00059485	0.00052714	0.00357294
0.00385904	0.00473309	0.00649214	0.00387096		0.00067592	0.00050783	0.00066686	0.00093222
0.00834322	0.00451016	0.00846219	0.00374985		0.00054193	0.00078106	0.00047588	0.000772
0.00495291	0.00657296	0.00637507	0.00310278		0.00072002	0.00067687	0.00058317	0.00123882
0.00640702	0.00717115	0.0078392	0.00356698		0.00050092	0.00052118	0.00057387	0.00110626
0.00540185	0.00528383	0.00854206	0.0035069		0.00055099	0.00066495	0.00071907	0.00159597
0.00479739	0.00721308	0.00695349	0.00341943	Average Time	0.00292593	0.00301539	0.00061336	0.00104504
0.00098257	0.00416588	0.00113482	0.00110869	Standard Deviation	0.00857443	0.00833228	9.3391E-05	0.00056774
0.00034589	0.00146647	0.00039948	0.00039028	Confidence Interval 95%	0.00301837	0.00293313	3.2875E-05	0.00019986

As I expected, with Count, a more traditional DBMS like Oracle Database dominates in speed even though I have to run Oracle Database through a virtual machine.

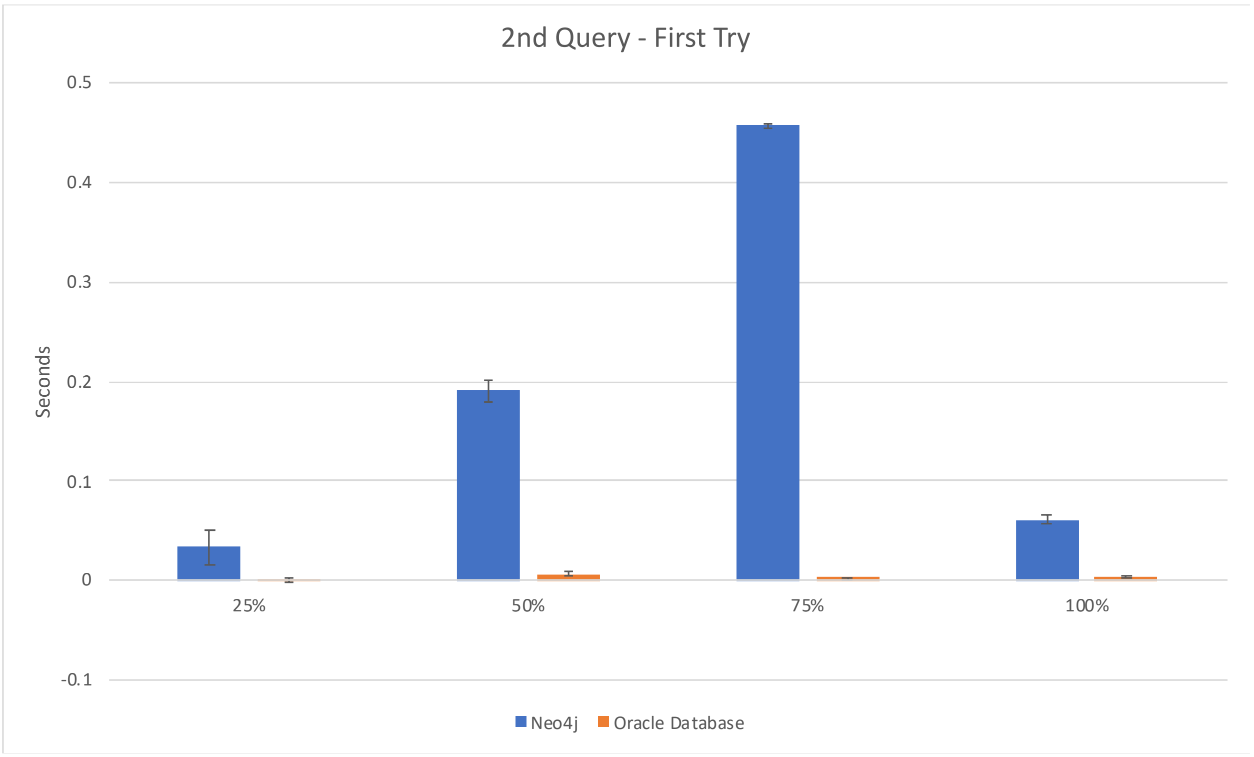

1st Query - First Try
Seconds
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
25%
50%
75%
100%
Neo4j
Oracle Database

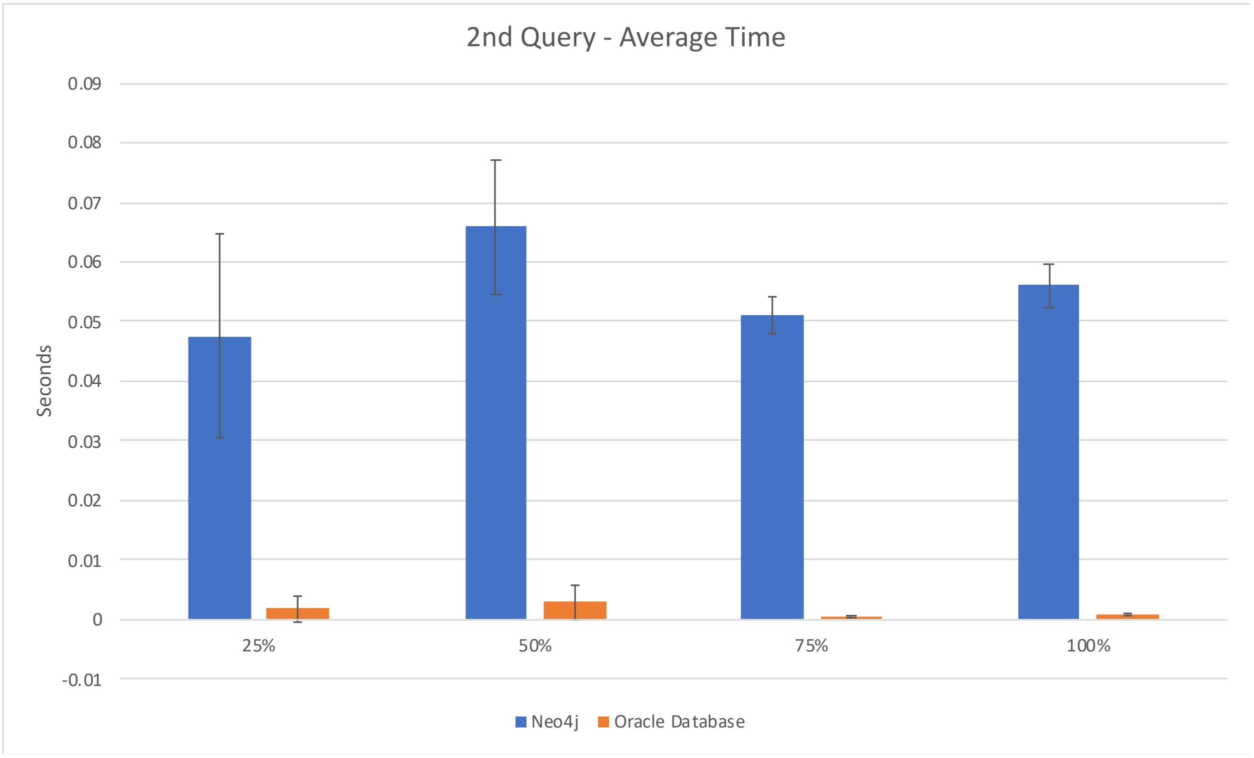

1st Query - Average Time
Seconds
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
-0.001
25%
50%
75%
100%
Neo4j
Oracle Database

2. 2nd Query

Neo4j					Oracle			
25.00%	50.00%	75.00%	100.00%		25.00%	50.00%	75.00%	100.00%
0.0332489	0.19039488	0.45667577	0.0612464		0.00142002	0.00683594	0.00305676	0.00381708
0.08063507	0.13656712	0.08101201	0.05444717		0.00068927	0.03475022	0.00061297	0.00088286
0.02914715	0.12534523	0.06513929	0.05480766		0.00076008	0.00062704	0.00058317	0.00089693
0.02700281	0.08795786	0.06462002	0.05403376		0.00078583	0.0007062	0.00062585	0.00261307
0.0310111	0.07832217	0.05532002	0.05239487		0.00059485	0.00058794	0.00051808	0.00175214
0.05298376	0.06130099	0.06237388	0.0535748		0.00054193	0.00047708	0.00050306	0.00087094
0.03277683	0.0726788	0.05285287	0.05041409		0.0004878	0.00065613	0.00054598	0.0008502
0.06076288	0.03844523	0.04961205	0.07610106		0.00071716	0.00051999	0.00055504	0.00084305
0.02564788	0.03417611	0.04895329	0.10269117		0.00052094	0.00063014	0.00046802	0.00070691
0.03264785	0.03328204	0.04848194	0.06799984		0.00051999	0.00052094	0.00063896	0.00081491
0.0342803	0.03590202	0.04723597	0.05562782		0.00044537	0.00044799	0.00049901	0.00089717
0.05193973	0.03493094	0.0444212	0.05585408		0.00069404	0.00063896	0.00064516	0.00071812
0.03155208	0.03129911	0.06051683	0.04928017		0.00045824	0.00051808	0.00059295	0.0006249
0.0867939	0.04386306	0.05057716	0.05432987		0.00078797	0.0005939	0.00082397	0.00075078
0.03000784	0.04564977	0.04607987	0.05086899	**2nd Query**	0.00046206	0.0005579	0.00065088	0.0009768
0.03872085	0.05209398	0.04835916	0.06357718		0.00055814	0.00054884	0.00063992	0.00069213
0.02630305	0.03814673	0.05346823	0.05089688		0.00078964	0.00050306	0.00053239	0.00051022
0.02900672	0.03030705	0.05062914	0.05055928		0.03460312	0.00058126	0.00049114	0.00054765
0.03840613	0.13117003	0.04622006	0.050174		0.0008328	0.00057578	0.00075793	0.00061393
0.05236602	0.11276317	0.04504108	0.05364799		0.00056696	0.0006187	0.00050998	0.00062609
0.02804494	0.09659719	0.04338098	0.05445576		0.00090194	0.00047302	0.00051308	0.00083899
0.02648211	0.09247589	0.04290605	0.05144882		0.0010879	0.00059485	0.00058722	0.0006361
0.02897	0.06789398	0.04300117	0.05080128		0.00059319	0.00046515	0.00052714	0.00068283
0.04838681	0.10770512	0.04670095	0.0497458		0.00062323	0.00059915	0.00051498	0.00089407
0.2595048	0.08361697	0.04535198	0.05847788		0.000458	0.00052309	0.00050592	0.00050688
0.15682602	0.05016708	0.04597497	0.05039907		0.00049305	0.00062108	0.0005672	0.00057888
0.02230978	0.04087019	0.04538822	0.0572269		0.00049877	0.00050497	0.00068879	0.00078297
0.015733	0.04563093	0.0578692	0.05237389		0.00060511	0.00058603	0.00076795	0.00061798
0.0159862	0.04488492	0.05390477	0.05425		0.000736	0.00052214	0.00062323	0.00076103
0.01587415	0.05726004	0.04235697	0.04950404		0.00058603	0.03443909	0.00051498	0.00067902
0.01723599	0.06528902	0.04345489	0.05101013		0.00058699	0.00050211	0.00056911	0.00088596
0.04757819	0.06588643	0.05104014	0.05603247	Average Time	0.00176621	0.00282969	0.0005858	0.00083512
0.04882641	0.03227639	0.00866205	0.01054447	Standard Deviation	0.00620376	0.008635	8.8308E-05	0.00040423
0.01718787	0.01136193	0.00304921	0.00371186	Confidence Interval 95%	0.00218385	0.00303969	3.1086E-05	0.0001423

With a simple query that does not require a join between tables and has an operator, Oracle Database still dominates with a speed many times faster than Neo4j.

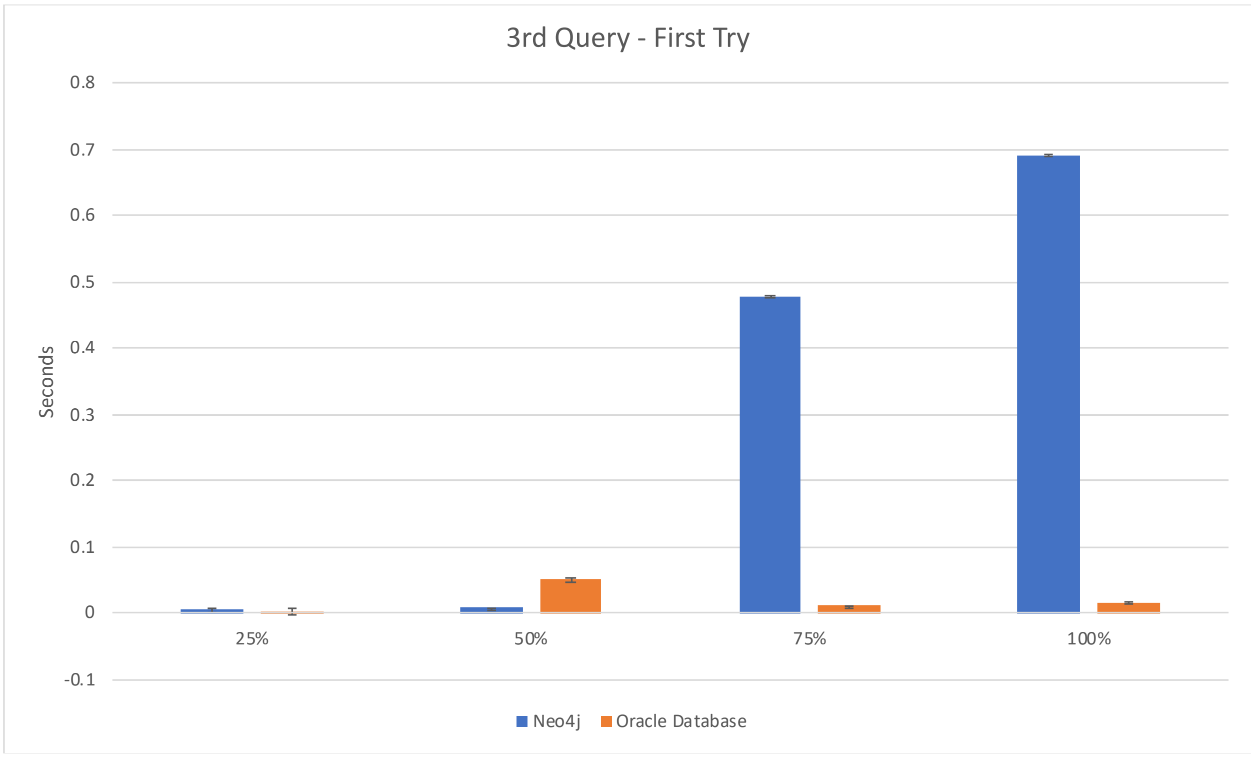

2nd Query - First Try
Seconds
0.5
0.4
0.3
0.2
0.1
0
-0.1
25%
50%
75%
100%
Neo4j
Oracle Database

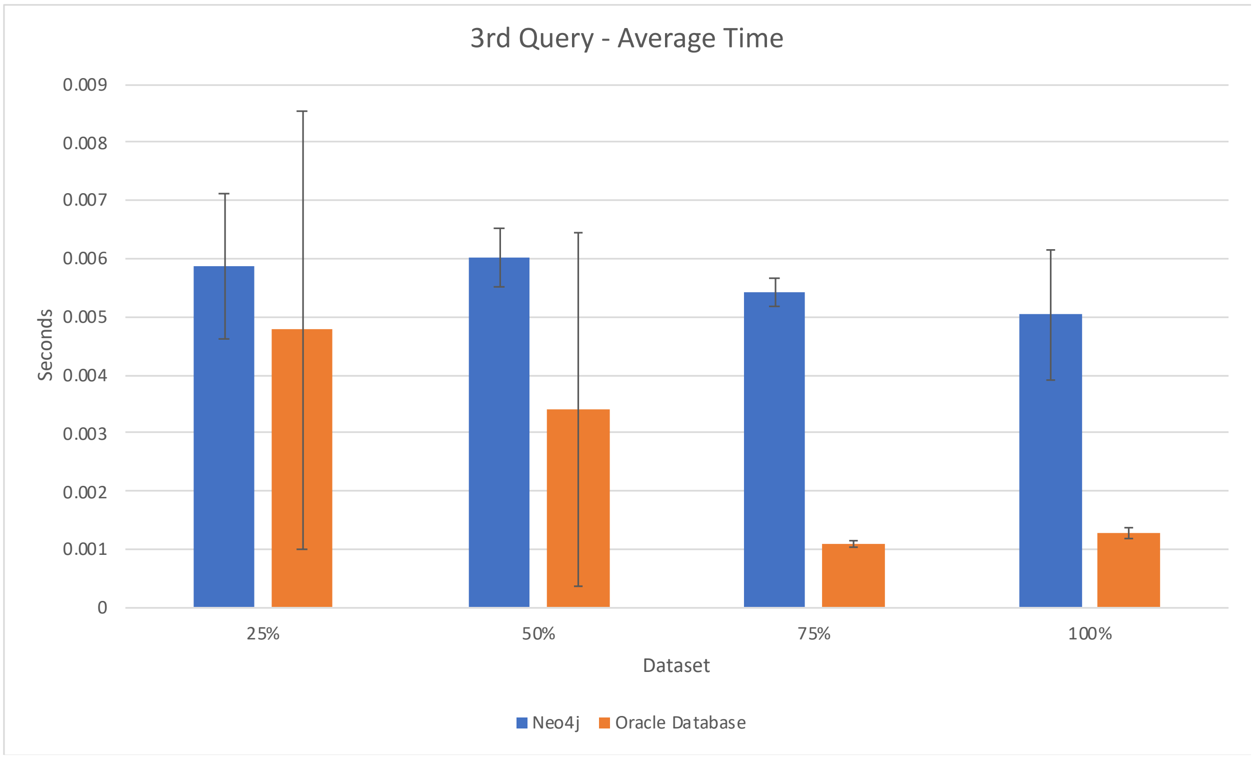

2nd Query - Average Time
Seconds
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
-0.01
25%
50%
75%
100%
Neo4j
Oracle Database

3. 3rd Query

Neo4j					Oracle			
25.00%	50.00%	75.00%	100.00%		25.00%	50.00%	75.00%	100.00%
0.00483489	0.00729084	0.47727299	0.69159913		0.00244212	0.05023122	0.01039219	0.015306
0.00475121	0.00703883	0.00732923	0.00869823		0.00133777	0.00154901	0.00147009	0.00165796
0.00426197	0.00498295	0.00595975	0.00482893		0.00129032	0.00127792	0.00105882	0.00134802
0.00452471	0.00603867	0.005548	0.00426817		0.03580427	0.00101709	0.00091624	0.00126982
0.00501299	0.00474596	0.00566602	0.00432897		0.00145197	0.00110912	0.00112391	0.00143409
0.00541091	0.00673366	0.00598288	0.00490975		0.001652	0.00131297	0.0010941	0.00149989
0.00430894	0.00465298	0.0058949	0.00426078		0.00104594	0.00108099	0.00142574	0.00119019
0.00522614	0.00730109	0.00537181	0.00412583		0.00142884	0.00139904	0.00103283	0.00105691
0.00790715	0.0045743	0.00545001	0.00420308		0.00134802	0.00088	0.00109911	0.00112796
0.00507212	0.00441122	0.00608826	0.00398421		0.00102091	0.00090599	0.00109005	0.001019
0.00547695	0.00401521	0.00529909	0.00455022		0.00120521	0.00090194	0.00115299	0.00116491
0.00723982	0.00529194	0.00515199	0.00661516		0.00111985	0.00082898	0.00090718	0.00115204
0.00570512	0.00514007	0.00615883	0.00388885		0.00106287	0.00102282	0.00113511	0.00128007
0.00722814	0.00561619	0.00575304	0.00438094		0.00103402	0.00123787	0.00111485	0.0012989
0.00468397	0.00856924	0.00566196	0.00458407	**3rd Query**	0.00110197	0.0349412	0.00110126	0.00166821
0.00624204	0.00768328	0.00521803	0.00510073		0.00104403	0.00161791	0.00107431	0.00124097
0.00582814	0.00590324	0.00497508	0.00520897		0.00120592	0.00106215	0.00124407	0.00230908
0.01889205	0.00558805	0.0059309	0.00368595		0.03680897	0.00105977	0.00100923	0.00124907
0.01760221	0.00495887	0.00458884	0.0034523		0.00114393	0.00105333	0.00091815	0.00159788
0.00402474	0.00516915	0.00447893	0.00386405		0.00114894	0.00113893	0.00093126	0.00135684
0.00474095	0.00475192	0.00522399	0.00417089		0.00081587	0.00134611	0.00101089	0.00119615
0.00605106	0.00731087	0.004884	0.00476503		0.00094318	0.00112104	0.00107408	0.00116777
0.00405598	0.0101099	0.00550604	0.00420094		0.00111389	0.0011363	0.00085902	0.00105
0.0040288	0.00776815	0.00473881	0.00454402		0.00106096	0.00094104	0.0011847	0.00116515
0.00362682	0.00595474	0.00552297	0.00346088		0.00126076	0.00111127	0.00123	0.00127006
0.00417709	0.00501299	0.00596094	0.00417495		0.0033648	0.00113797	0.00109792	0.00109577
0.00367117	0.00468206	0.00601602	0.00403094		0.03653407	0.00099993	0.00100398	0.00116396
0.00382161	0.00626707	0.00521016	0.00430703		0.00137019	0.00106502	0.00110412	0.00108027
0.0037868	0.0064199	0.00412798	0.00440216		0.00154495	0.03561401	0.00107598	0.00113702
0.00425076	0.00577402	0.00486708	0.02095509		0.00097609	0.00122118	0.00125313	0.00103664
0.00481606	0.00770402	0.00420189	0.00333905		0.00113201	0.00111389	0.00105596	0.00134969
0.00588088	0.00600568	0.00542558	0.00504301	Average Time	0.00477908	0.00340683	0.00109497	0.00128781
0.00353938	0.00141682	0.00066543	0.00317006	Standard Deviation	0.0107243	0.00866581	0.00013666	0.00026088
0.00124593	0.00049875	0.00023424	0.00111592	Confidence Interval 95%	0.00377517	0.00305054	4.8107E-05	9.1835E-05

When the query requires the appearance of a relationship, Oracle Database can still prevail with a simple table join. But as the chart below shows, we have seen Neo4j rising in speed.

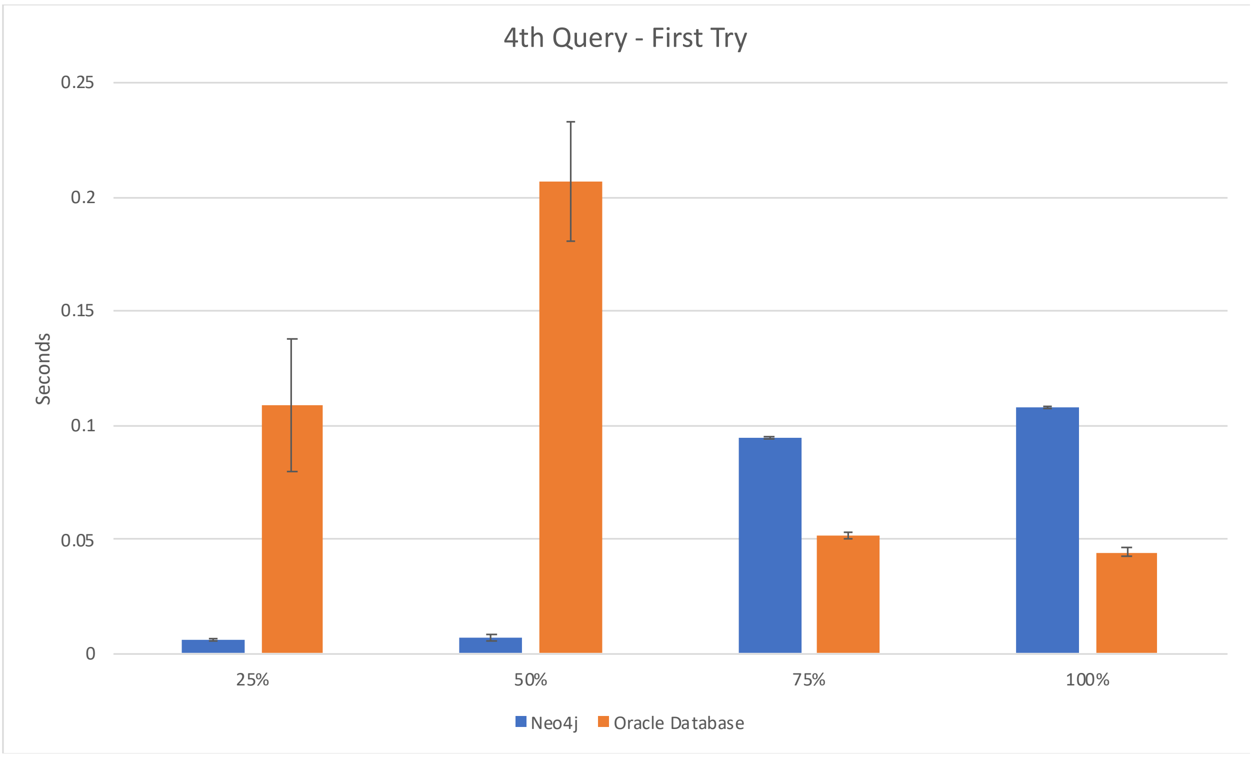

3rd Query - First Try
Seconds
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
25%
50%
75%
100%
Neo4j
Oracle Database

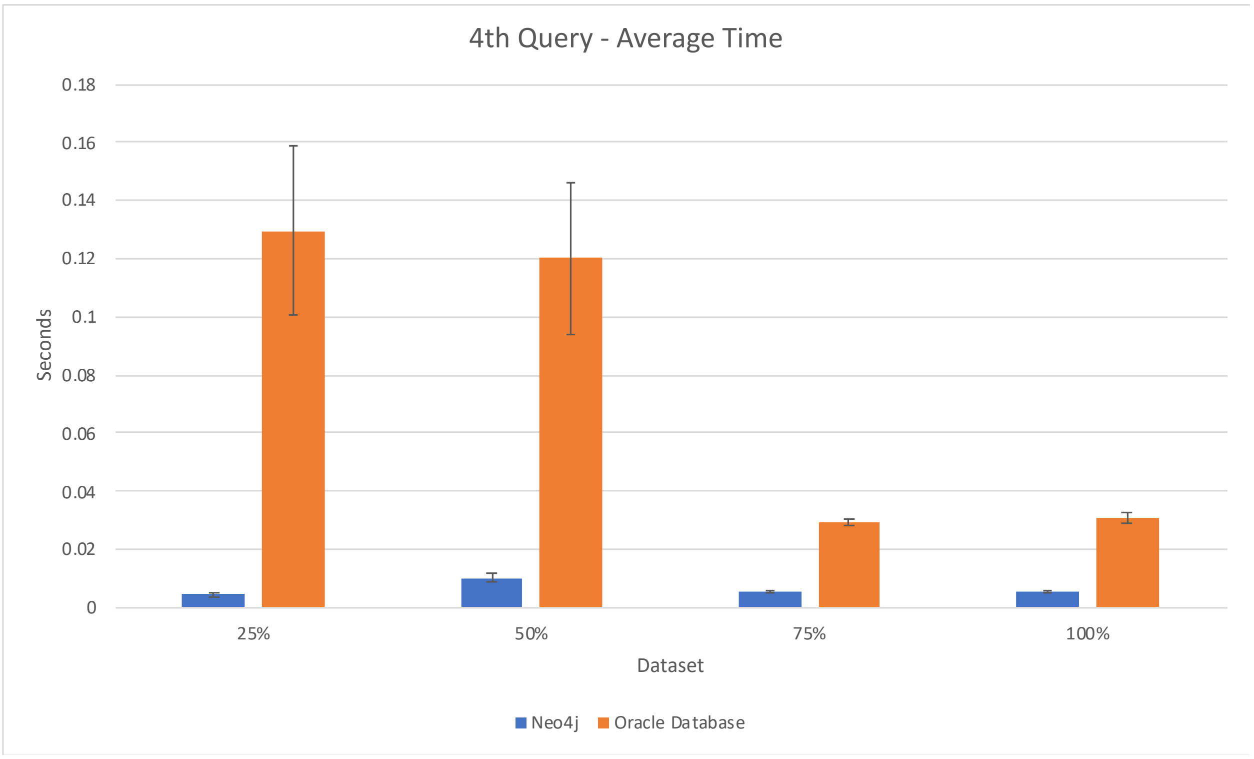

3rd Query - Average Time
Seconds
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
25%
50%
75%
100%
Dataset
Neo4j
Oracle Database

4. 4th Query

Neo4j					Oracle			
25.00%	50.00%	75.00%	100.00%		25.00%	50.00%	75.00%	100.00%
0.00582504	0.00721097	0.0949719	0.10778975		0.10856462	0.20650721	0.05160069	0.04446507
0.00935292	0.00704885	0.00610399	0.00518584		0.13628387	0.1040206	0.02943993	0.02931619
0.00523305	0.00762796	0.00550508	0.00475407		0.06498599	0.09998703	0.02926707	0.02761889
0.00886989	0.00536513	0.00549889	0.00545979		0.13443494	0.0974493	0.02859449	0.02988291
0.00475621	0.00920606	0.00605893	0.00468397		0.3097403	0.09709263	0.02418923	0.02763414
0.00562286	0.00632	0.00624919	0.004426		0.09800959	0.1068995	0.02578831	0.03506827
0.00552392	0.00640726	0.00584888	0.00463796		0.09723282	0.0987649	0.0298841	0.02698278
0.00537109	0.00817704	0.00563598	0.00643206		0.10400891	0.09978485	0.02920699	0.02944756
0.00481796	0.00806832	0.00473595	0.00616407		0.09472775	0.10041499	0.02934408	0.03088641
0.00474811	0.00811696	0.00498104	0.00723791		0.09936881	0.09693074	0.02686644	0.02633882
0.00414515	0.00631499	0.004354	0.00634599		0.14946699	0.13921785	0.03927779	0.03212571
0.00349212	0.00587964	0.00631785	0.00602126		0.49811292	0.10089397	0.02964377	0.02847648
0.00406122	0.00660992	0.00569987	0.00700521		0.11150956	0.10072589	0.02815437	0.02893686
0.00346112	0.0125711	0.00579715	0.00654888		0.09813881	0.09745574	0.02593589	0.04018378
0.00371003	0.00956798	0.00618291	0.00569129	**4th Query**	0.10513592	0.06584167	0.0322001	0.0307405
0.00352693	0.01402807	0.00414681	0.00413394		0.19645596	0.13734341	0.02629447	0.03353333
0.00310779	0.01692224	0.00501513	0.00448799		0.15046334	0.09758973	0.03035069	0.03028774
0.00427175	0.01338601	0.0072	0.00441003		0.09534073	0.10236311	0.02850795	0.029109
0.00384688	0.01254106	0.00665903	0.00473499		0.09724975	0.0611279	0.03319478	0.03139257
0.00355697	0.01521897	0.00593305	0.00472999		0.10488939	0.09791493	0.03021574	0.03194785
0.00404477	0.01860118	0.00499201	0.00509381		0.14391923	0.45299816	0.02602839	0.0484879
0.00342894	0.01674128	0.00551915	0.00498796		0.09903479	0.09996033	0.02831459	0.02965975
0.00357199	0.01570606	0.00484204	0.00488114		0.09874487	0.14551401	0.03080797	0.03100371
0.00323319	0.01305962	0.00515699	0.00466299		0.09989572	0.09913301	0.02458119	0.02908707
0.00340295	0.01196289	0.00461721	0.00423908		0.06161308	0.29704905	0.02977276	0.02941942
0.0038712	0.01084495	0.00517488	0.004421		0.099617	0.10012698	0.03062391	0.02934122
0.00365901	0.01142001	0.00462413	0.00461102		0.10053301	0.10214567	0.02866292	0.02671576
0.00345016	0.01253986	0.00451088	0.00504088		0.10224795	0.09928679	0.04182506	0.03082824
0.00416708	0.00764418	0.00547814	0.00528312		0.10003376	0.14161777	0.02994657	0.02736902
0.00369835	0.00517082	0.00467515	0.00511312		0.13929367	0.06514382	0.03009534	0.04059291
0.00352097	0.00398493	0.00500226	0.00546622		0.09987998	0.09929419	0.0291996	0.026649
0.00438415	0.01023511	0.00541722	0.00522972	Average Time	0.12967898	0.12013628	0.02954048	0.03096879
0.00146581	0.0040233	0.0007301	0.00084011	Standard Deviation	0.08290232	0.0747457	0.00365557	0.00473637
0.00051599	0.00141628	0.00025701	0.00029574	Confidence Interval 95%	0.02918327	0.02631197	0.00128683	0.0016673

When there are complex relationships that need to be queried. Neo4j shows its advantage when the query command looks very simply but brings much higher efficiency than Oracle Database. However, with large amounts of data like in Dataset 75% and 100%, we can see that on the first try Oracle still has a good enough speed.

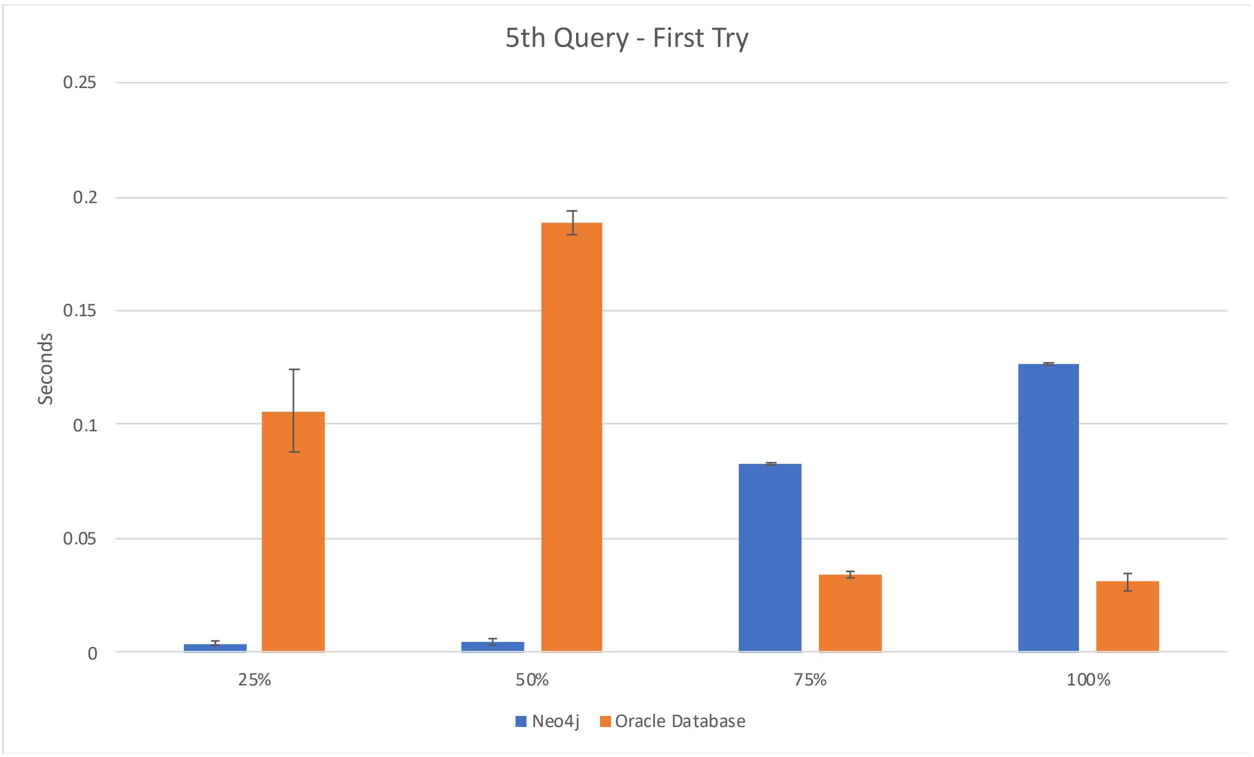

4th Query - First Try
Seconds
0.25
0.2
0.15
0.1
0.05
0
25%
50%
75%
100%
Neo4j
Oracle Database

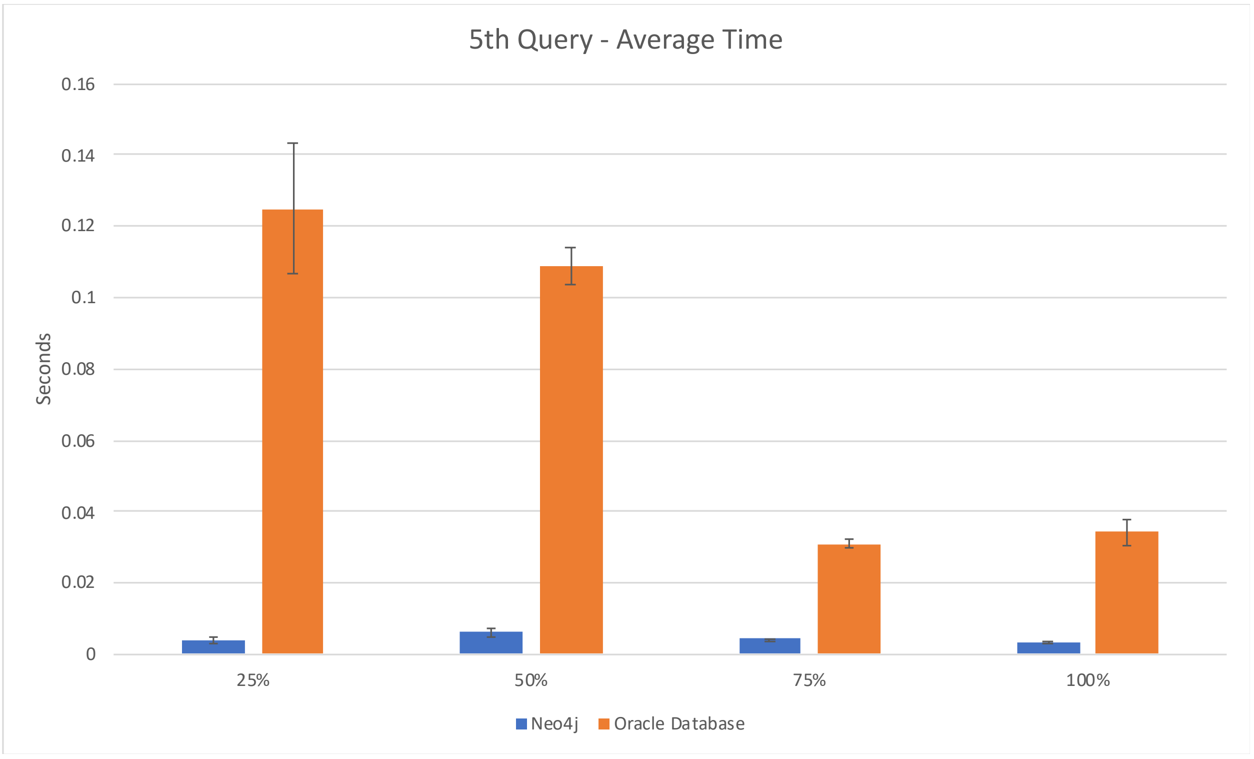

4th Query - Average Time
Seconds
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
25%
50%
75%
100%
Dataset
Neo4j
Oracle Database

5. 5th Query

Neo4j					Oracle			
25.00%	50.00%	75.00%	100.00%		25.00%	50.00%	75.00%	100.00%
0.00380111	0.00468922	0.0831461	0.12670207		0.10617232	0.18870282	0.03433633	0.03097391
0.00328302	0.00539494	0.00513387	0.00361824		0.10201573	0.09877586	0.02961516	0.02956986
0.00313807	0.00529695	0.00447297	0.00352883		0.10048771	0.10576105	0.03056622	0.0270884
0.0035851	0.00423789	0.00552487	0.00352597		0.14203906	0.10195303	0.03081942	0.05128479
0.00366497	0.00346899	0.00464702	0.00337386		0.10666442	0.10194016	0.03202534	0.02927685
0.003232	0.00328708	0.00451112	0.00330091		0.10178947	0.13872337	0.03011584	0.03039217
0.00338507	0.00380397	0.00526905	0.00332594		0.09887266	0.10022378	0.02685094	0.03115582
0.0032959	0.00363803	0.00402999	0.00332999		0.13864398	0.1000073	0.03051543	0.02841067
0.01507306	0.0033462	0.00563383	0.00301719		0.10822105	0.10665345	0.03063416	0.02969003
0.01083517	0.00420189	0.00483394	0.00343704		0.10194635	0.10541821	0.0371151	0.03108382
0.00340176	0.00393581	0.00526786	0.00338364		0.10314488	0.09880209	0.02946997	0.0311749
0.00318694	0.0038321	0.00427985	0.0054121		0.13914037	0.13940072	0.03372192	0.03350711
0.00320888	0.00530696	0.004318	0.00423789		0.10321665	0.09981298	0.03866291	0.0309279
0.00331903	0.00787401	0.00369096	0.00440001		0.10450745	0.10233235	0.03191686	0.02651834
0.00296187	0.00482917	0.00310612	0.00374889	**5th Query**	0.10076046	0.10083318	0.02580619	0.041713
0.00341606	0.00464106	0.0038631	0.00332808		0.23774123	0.14669085	0.02719593	0.03244019
0.00287318	0.0046699	0.00489116	0.00303912		0.15527081	0.11147285	0.02581096	0.02953362
0.00308013	0.01189685	0.00393796	0.0033989		0.10109115	0.1003201	0.0208559	0.02757597
0.00327516	0.00759506	0.00394607	0.00324702		0.09915304	0.10113335	0.02535319	0.02805281
0.00321984	0.00887513	0.00412297	0.00342488		0.09963679	0.09848976	0.02759719	0.02742529
0.00369263	0.01349783	0.00370193	0.00313401		0.10633588	0.14060879	0.02886653	0.03175187
0.00341487	0.01410794	0.00427318	0.00304103		0.34857702	0.10318089	0.03580737	0.03270483
0.00317597	0.01732302	0.00388908	0.00274396		0.15031004	0.10012388	0.02951574	0.03454137
0.00306511	0.004884	0.00367427	0.00330329		0.14324856	0.0974288	0.03569221	0.03451943
0.00347185	0.00457096	0.00382996	0.00294709		0.10208368	0.10297894	0.0366075	0.03808689
0.00346398	0.00510383	0.00342393	0.00309777		0.10883403	0.1021862	0.03252888	0.03898907
0.00315809	0.00604105	0.00439215	0.00284505		0.13870525	0.10377073	0.03264594	0.03925204
0.00322509	0.00570393	0.00484991	0.00314498		0.10024643	0.14091396	0.0323782	0.03005195
0.00321603	0.00510907	0.003968	0.00298405		0.09711313	0.10592389	0.03665566	0.03049231
0.00672293	0.00568509	0.0038588	0.00281096		0.10273385	0.10230875	0.03324389	0.08469534
0.00283599	0.00570226	0.00393128	0.00299788		0.10605597	0.1043489	0.03140712	0.03494287
0.00402926	0.00626203	0.00430911	0.00337095	Average Time	0.1249529	0.10875061	0.03099992	0.03422832
0.00257533	0.00349976	0.00063477	0.00053325	Standard Deviation	0.05125113	0.01510946	0.00402775	0.0108548
0.00090657	0.00123198	0.00022345	0.00018771	Confidence Interval 95%	0.01804142	0.00531883	0.00141785	0.00382111

In the 5th query that I built; I want to return more data with the same amount of relations as with the 4th query. In the chart below we can see that Neo4j is still struggling with the larger amount of data in the Dataset. 75% and 100%. But once cached, we can see that the average time Neo4j takes to return results over the next 30 attempts is shorter than Oracle Database.


5th Query - First Try
Seconds
0.25
0.2
0.15
0.1
0.05
0
25%
50%
75%
100%
Neo4j
Oracle Database


5th Query - Average Time
Seconds
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
25%
50%
75%
100%
Neo4j
Oracle Database

VI. Conclusioni

After carrying out the various tests on queries of different complexity and databases of increasing size (250, 500, 750, 1000) I can draw a series of conclusions.

With simple queries I get comparable performance with both Neo4j and Oracle Database. In some case, such as count, since RDBMS such as MySQL or Oracle Database usually store number of elements in table, they get the result very fast.

With more complex queries but still require joining one or two table, Oracle Database still proves its capacity when the time is significantly shorter than Neo4j even when increasing the size of the database.

As for the most two complex queries, we can see how Neo4j is more performing than Oracle Database. To give the same result, the SQL query for Oracle Database is obviously more complex than Neo4j's Cypher.

Queries in Neo4j are considerably less complex to write because the Cypher language, created specifically for queries, includes numerous clauses capable of providing a lot of functionality and therefore managing nodes and relationships in a remarkably efficient way.

The relationships created through nodes are the real strength of Neo4j, so it is the best choice for databases with many relationships that require complex queries. Oracle Database on the other hand is considered the best solution in case there are simple queries and data with few relationships.