UNIVERTITY OF MESSINA

DEPARTMENT

MATHEMATICAL AND COMPUTER SCIENCE, PHYSICAL SCIENCES AND EARTH SCIENCES

DEGREE COURSE:

DATA ANALYSIS

DATABASE MOD B PROJECT

IDENTIFY CRIMINAL ACTIVITIES IN CALL RECORDS

By: Trinh Thi Lan Anh - 519161

Guided by: Professor Antonio Celesti

Academic year 2021-2022

I. Problem Addressed

Call records are a great source of information on real-life networks, but finding insights from intricate datasets can be hard. It is now well known that mobile phone data can be a valuable asset for analysts tasked with the investigation of criminal activity. Phone operators are authorized to collect information about whom their users call, for how long and from where. In certain circumstances, that data can be used by law enforcement. But how can investigators find insights quickly within intricate, sometimes very large, networks of phone records data? The first step for investigators wanting to leverage graph technology, is to model the data as a graph. The data, phone operators provide law enforcement with, is often tabular (a list) but inherently, phone record data constitute a graph, or a network, of devices linked together via calls. For years, investigators had to work with this data as tables and rows because the technology in use, relational databases, was built that way. Now we have so many types of databases to keep data in different way. Now we have so many types of database to keep the data. In this project I choose Neo4j and MongoDB database to make comparison between them to know which is better.

II. DBMS Solution Considered

1. Neo4j

Neo4j is the world's leading open source Graph Database which is developed using Java technology. It is highly scalable and schema free (NoSQL). A graph is composed of two elements - nodes (vertices) and relationships (edges). Graph database is a database used to model the data in the form of graph. In here, the nodes of a graph depict the entities while the relationships depict the association of these nodes. Neo4j is a popular Graph Database. Unlike other databases, graph databases store relationships and connections as first-class entities.

Advantages of Neo4j

- Flexible data model − Neo4j provides a flexible simple and yet powerful data model, which can be easily changed according to the applications and industries.
- Real-time insights − Neo4j provides results based on real-time data.
- High availability − Neo4j is highly available for large enterprise real-time applications with transactional guarantees.

- Connected and semi structures data − Using Neo4j, you can easily represent connected and semi-structured data.
- Easy retrieval − Using Neo4j, you can not only represent but also easily retrieve (traverse/navigate) connected data faster when compared to other databases.
- Cypher query language − Neo4j provides a declarative query language to represent the graph visually, using an ascii-art syntax. The commands of this language are in human readable format and very easy to learn.
- No joins − Using Neo4j, it does NOT require complex joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without joins or indexes.

2. MongoDB

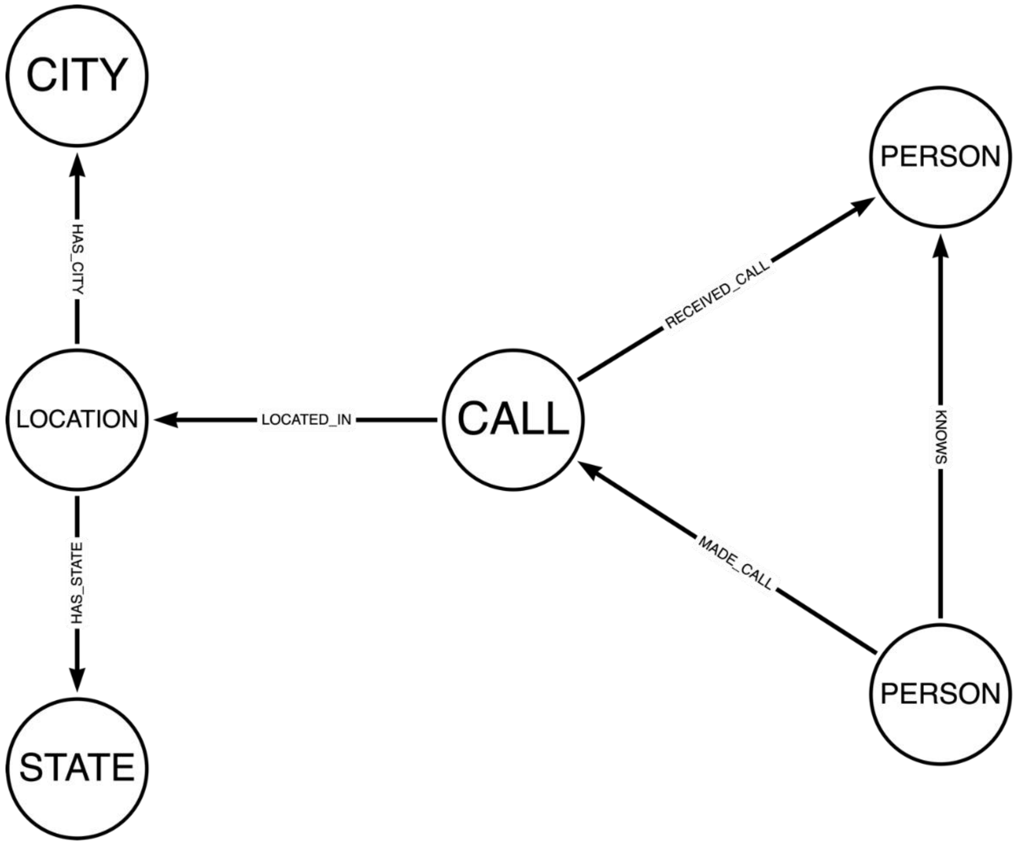
MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.
Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose. A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.
Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB, there is no concept of relationship.
Advantages of MongoDB over RDBMS

- Schema less − MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.

- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- Ease of scale-out − MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

III. Design

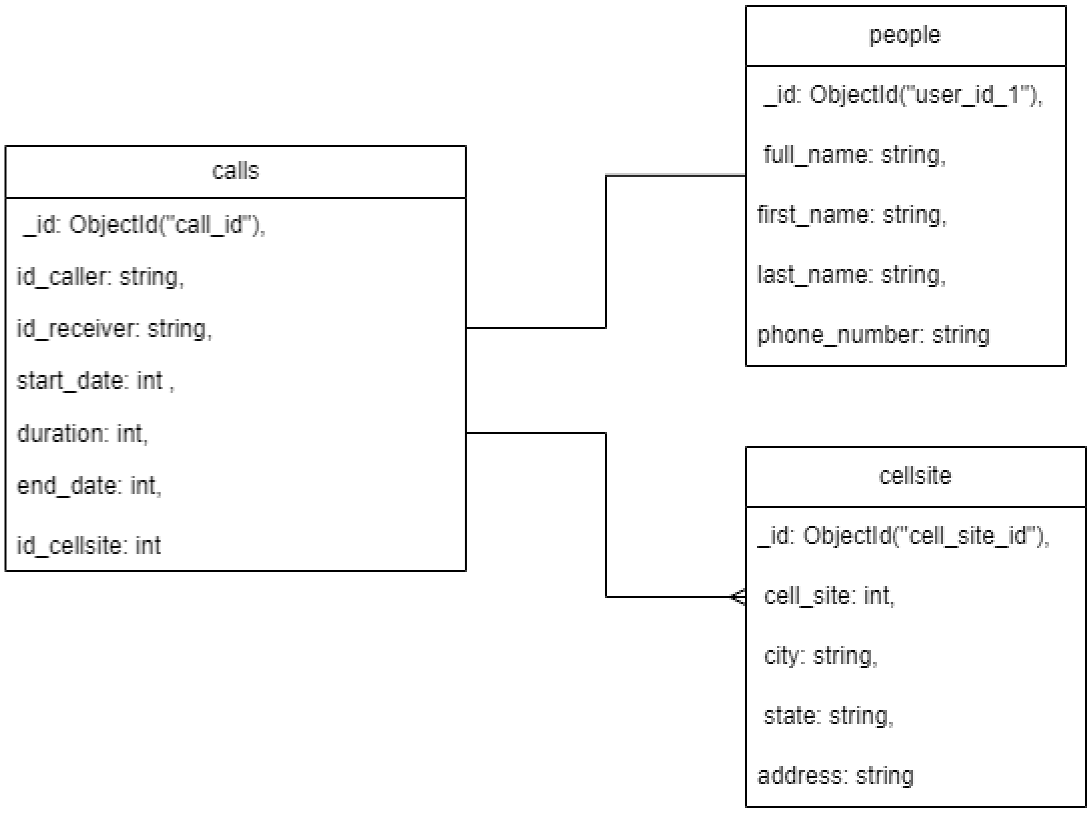
1. Neo4j Database Design



You can see above that our graph model for phone calls is centered around calls. A single phone call connects together 4 entities: 2 phone owners, a location (the cell site the caller was next to when he initiated the call), a state and a city.

It is important to note that in real life, most of the time we would not have access to the names of the phone numbers owners.

2. MongoDB database design



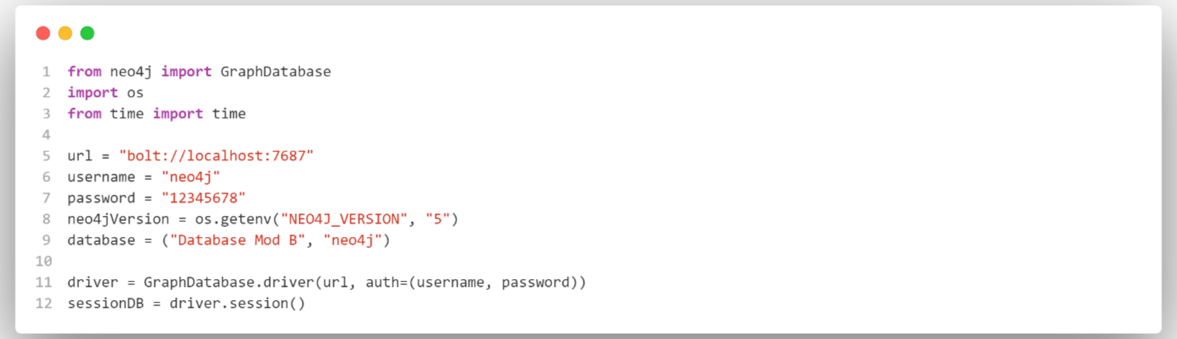
In this design, we have three collections: "calls", "cell_sites", and "people".

IV. Implementation

1. Connect to database

a. Neo4j

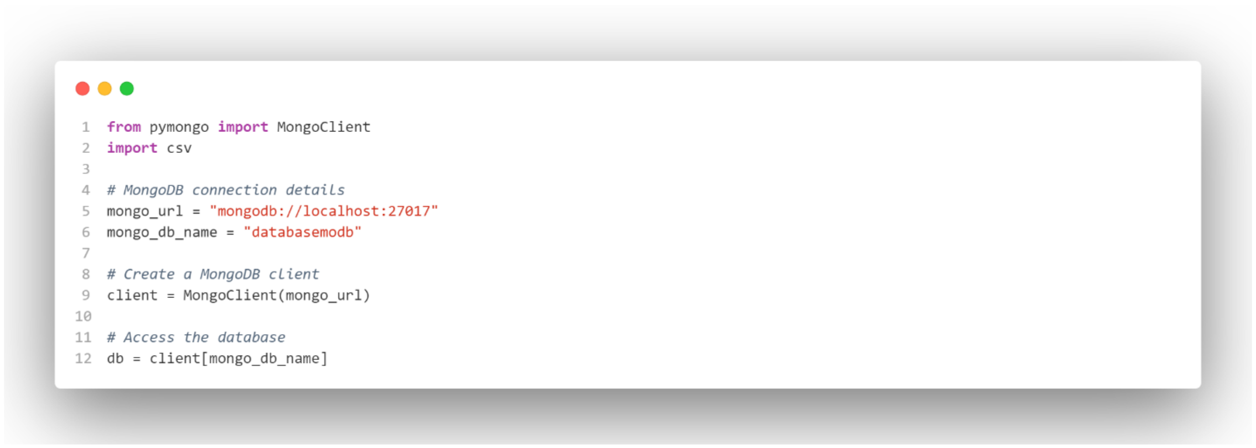
Import needed library for Neo4j connection.



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

url = "bolt://localhost:7687"
username = "neo4j"
password = "12345678"
neo4jVersion = os.getenv("NEO4J_VERSION", "5")
database = ("Database Mod B", "neo4j")

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

b. MongoDB

Import needed library for MongoDB

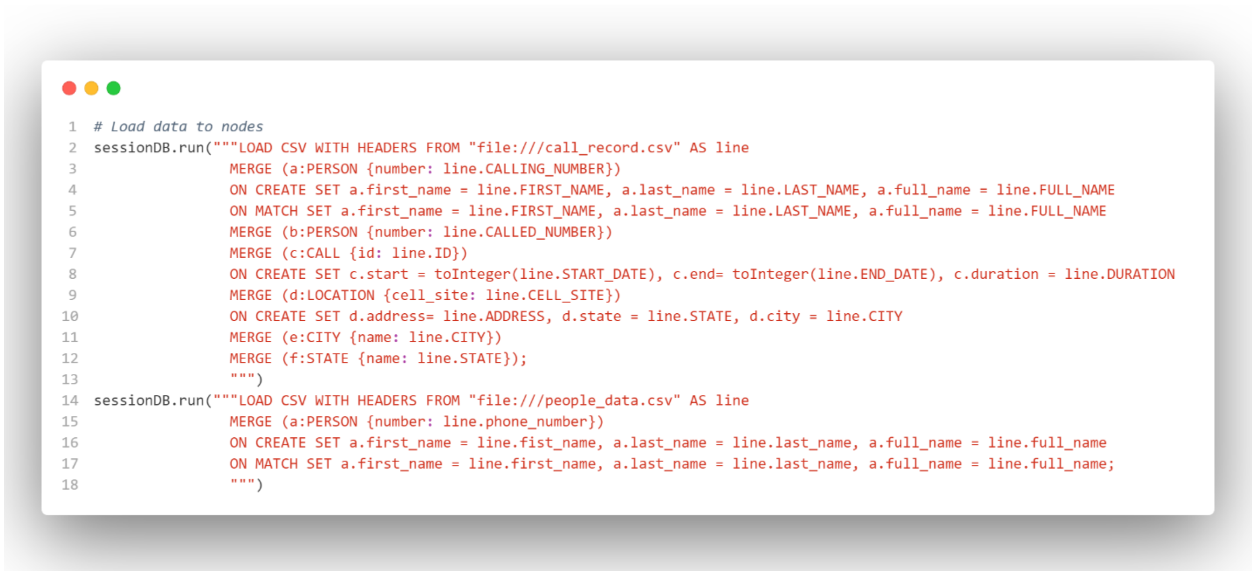


```python
from pymongo import MongoClient
import csv

# MongoDB connection details
mongo_url = "mongodb://localhost:27017"
mongo_db_name = "databasemodb"

# Create a MongoDB client
client = MongoClient(mongo_url)

# Access the database
db = client[mongo_db_name]
```

2. Import data

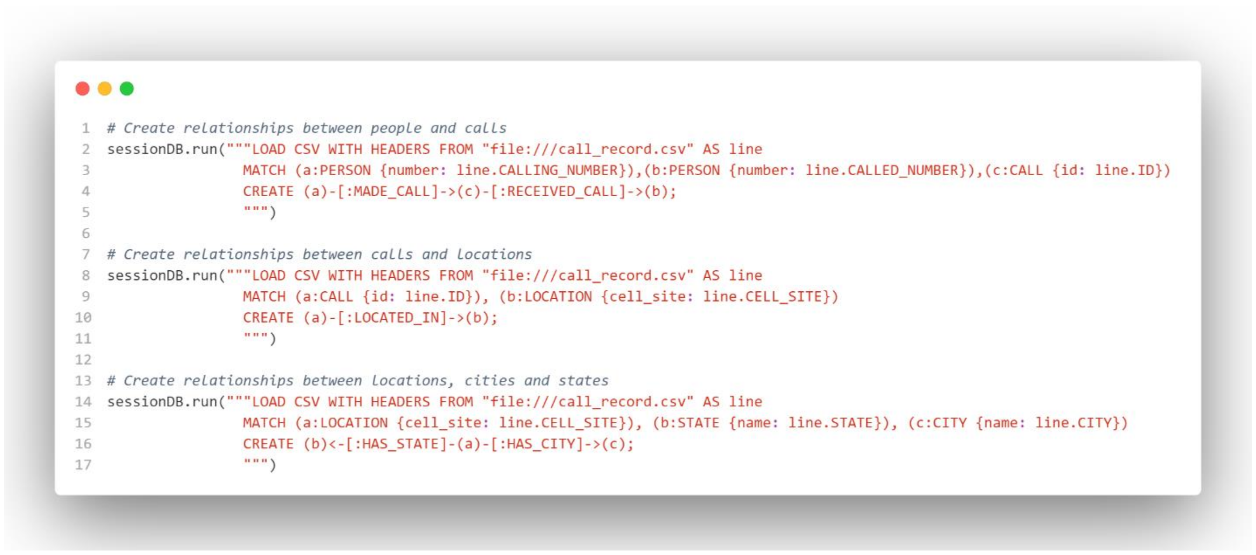
a. Neo4j

Load data to database

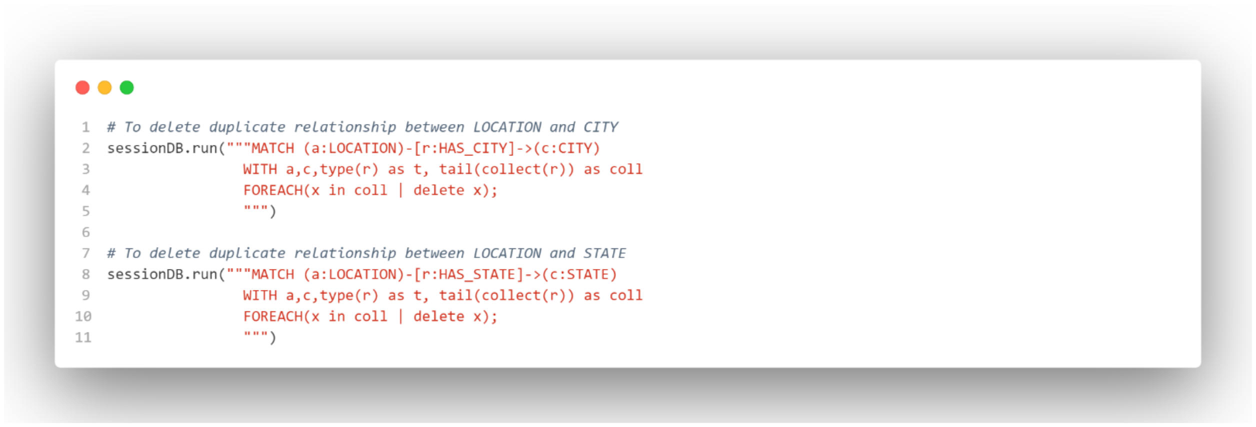


```python
# Load data to nodes
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///call_record.csv" AS line
                MERGE (a:PERSON {number: line.CALLING_NUMBER})
                ON CREATE SET a.first_name = line.FIRST_NAME, a.last_name = line.LAST_NAME, a.full_name = line.FULL_NAME
                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_NUMBER})
                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:///people_data.csv" AS line
                MERGE (a:PERSON {number: line.phone_number})
                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;
                """)
```

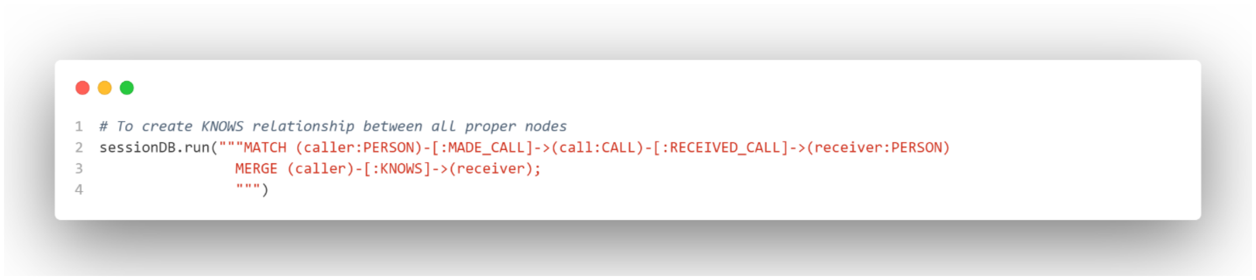
Create relationship between nodes


```
# Create relationships between people and calls
sessionDB.run("""LOAD CSV WITH HEADERS FROM "file:///call_record.csv" AS line
                MATCH (a:PERSON {number: line.CALLING_NUMBER}),(b:PERSON {number: line.CALLED_NUMBER}),(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:///call_record.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:///call_record.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);
                """)
```

Because in the previous step some relationships will be duplicate, we need to delete them.


```
# 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);
                """)
```

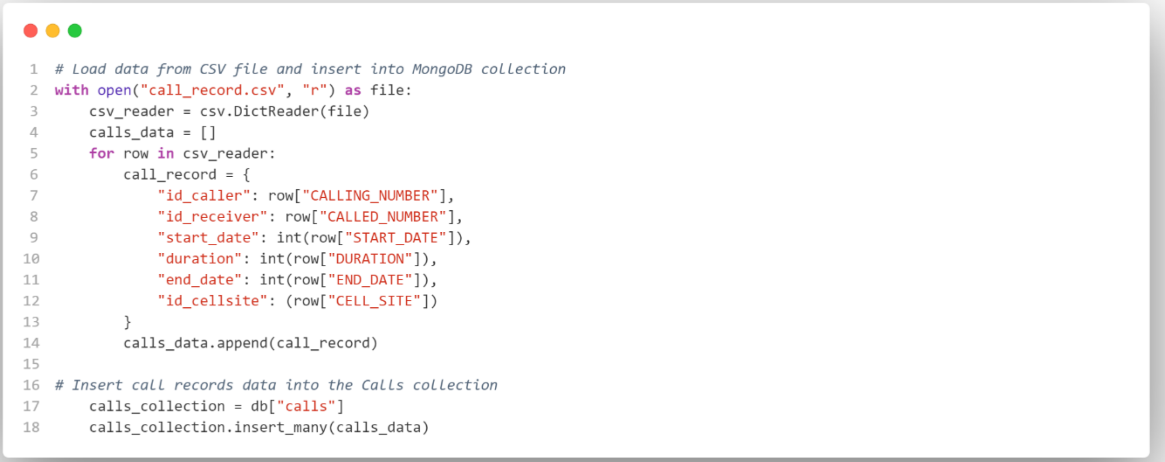
Two people calls each other we need to make the KNOWS relationship


```
# To create KNOWS relationship between all proper nodes
sessionDB.run("""MATCH (caller:PERSON)-[:MADE_CALL]->(call:CALL)-[:RECEIVED_CALL]->(receiver:PERSON)
                MERGE (caller)-[:KNOWS]->(receiver);
                """)
```

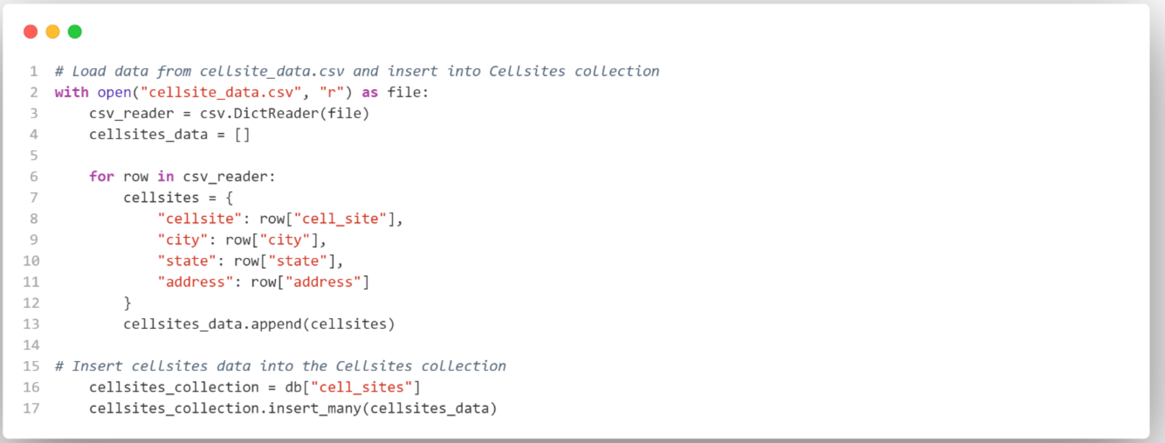
b. MongoDB

Like Neo4j, we need to import all collections into MongoDB

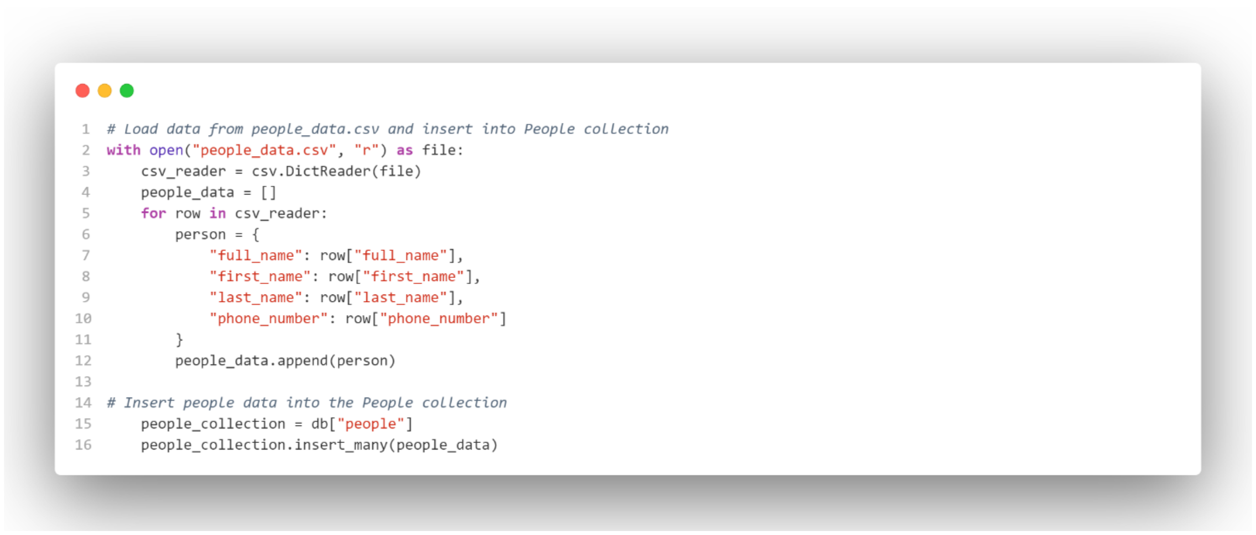
Below is to import Call collection.


```python
# Load data from CSV file and insert into MongoDB collection
with open("call_record.csv", "r") as file:
    csv_reader = csv.DictReader(file)
    calls_data = []
    for row in csv_reader:
        call_record = {
            "id_caller": row["CALLING_NUMBER"],
            "id_receiver": row["CALLED_NUMBER"],
            "start_date": int(row["START_DATE"]),
            "duration": int(row["DURATION"]),
            "end_date": int(row["END_DATE"]),
            "id_cellsite": (row["CELL_SITE"])
        }
        calls_data.append(call_record)

# Insert call records data into the Calls collection
    calls_collection = db["calls"]
    calls_collection.insert_many(calls_data)
```

And here is to import Cellsite Collection


```python
# Load data from cellsite_data.csv and insert into Cellsites collection
with open("cellsite_data.csv", "r") as file:
    csv_reader = csv.DictReader(file)
    cellsites_data = []

    for row in csv_reader:
        cellsites = {
            "cellsite": row["cell_site"],
            "city": row["city"],
            "state": row["state"],
            "address": row["address"]
        }
        cellsites_data.append(cellsites)

# Insert cellsites data into the Cellsites collection
    cellsites_collection = db["cell_sites"]
    cellsites_collection.insert_many(cellsites_data)
```

Last is the import of People Collection



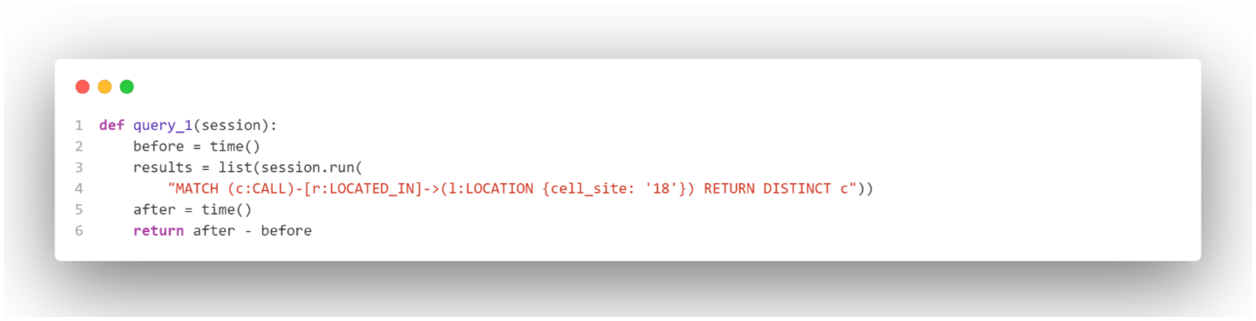
```python
# Load data from people_data.csv and insert into People collection
with open("people_data.csv", "r") as file:
    csv_reader = csv.DictReader(file)
    people_data = []
    for row in csv_reader:
        person = {
            "full_name": row["full_name"],
            "first_name": row["first_name"],
            "last_name": row["last_name"],
            "phone_number": row["phone_number"]
        }
        people_data.append(person)

# Insert people data into the People collection
    people_collection = db["people"]
    people_collection.insert_many(people_data)
```

3. Query and execution time

In this part, I define 4 queries with increasing degree of complexity from point of view of the number of entities involved and the selection filters.

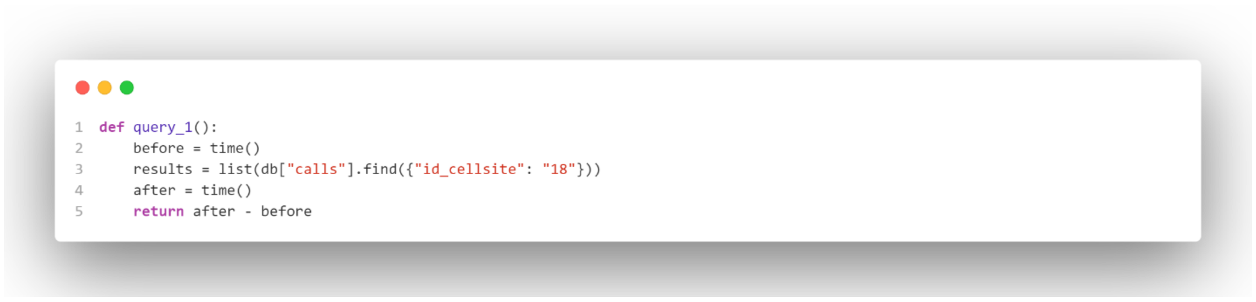
Query 1: Find all calls with their associated cell site information:

Neo4j



```python
def query_1(session):
    before = time()
    results = list(session.run(
        "MATCH (c:CALL)-[r:LOCATED_IN]->(l:LOCATION {cell_site: '18'}) RETURN DISTINCT c"))
    after = time()
    return after - before
```

MongoDB



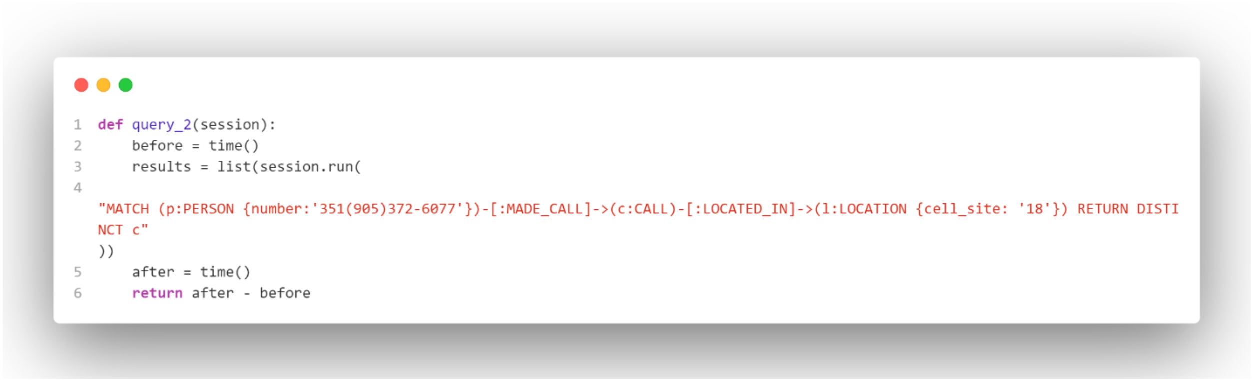
```python
def query_1():
    before = time()
    results = list(db["calls"].find({"id_cellsite": "18"}))
    after = time()
    return after - before
```

The MongoDB query uses a simple query to find all documents in the "calls" collection where the "id_cellsite" field has the value "18". The Neo4j query uses Cypher to find relationships between

CALL and LOCATION nodes based on the "cell_site" property being equal to "18". For simple queries like the one in the MongoDB function, the performance can be quite efficient.

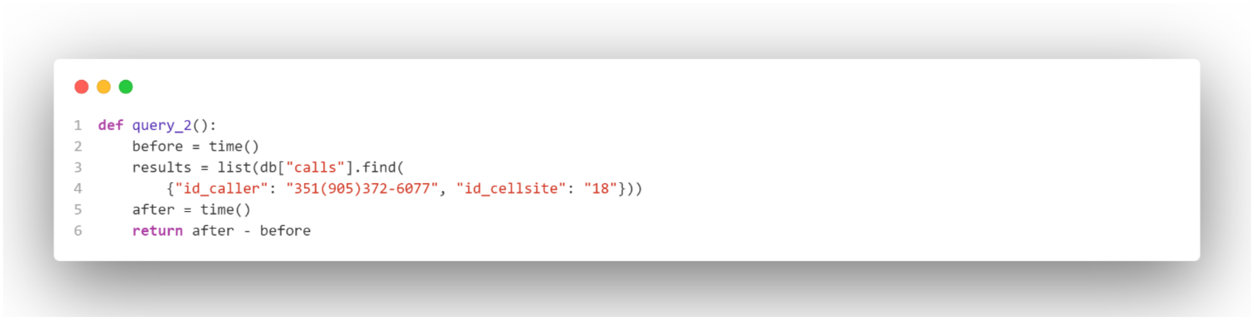
Query 2: Find call for a specific user with their associated cell site information:

Neo4j



```python
def query_2(session):
    before = time()
    results = list(session.run(
"MATCH (p:PERSON {number:'351(905)372-6077'})-[:MADE_CALL]->(c:CALL)-[:LOCATED_IN]->(l:LOCATION {cell_site: '18'}) RETURN DISTI
NCT c"
))
    after = time()
    return after - before
```

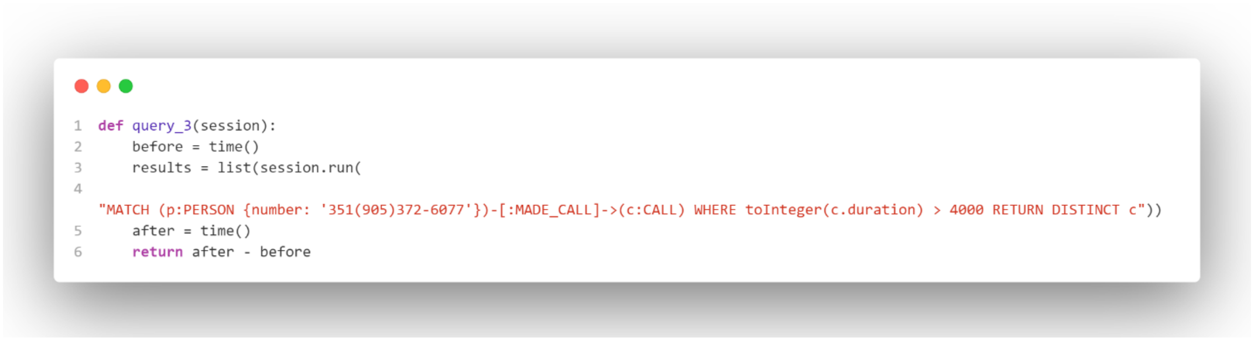
MongoDB



```python
def query_2():
    before = time()
    results = list(db["calls"].find(
        {"id_caller": "351(905)372-6077", "id_cellsite": "18"}))
    after = time()
    return after - before
```

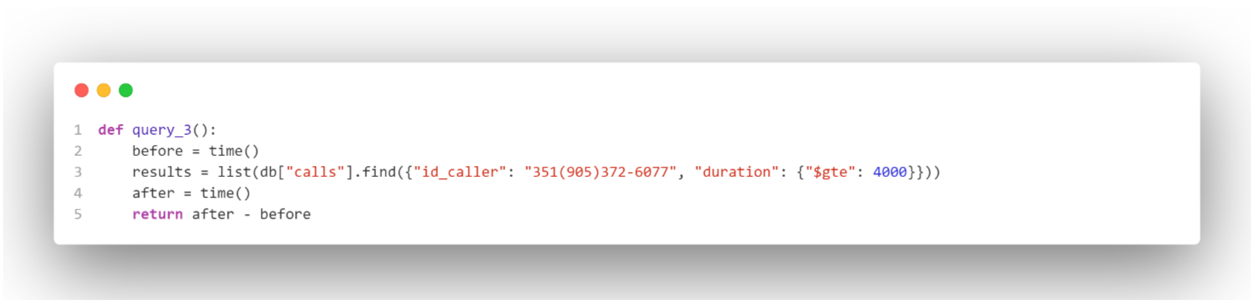
Query 3: Find call for a specific user within a specific time range:

Neo4j



```python
def query_3(session):
    before = time()
    results = list(session.run(
"MATCH (p:PERSON {number: '351(905)372-6077'})-[:MADE_CALL]->(c:CALL) WHERE toInteger(c.duration) > 4000 RETURN DISTINCT c"))
    after = time()
    return after - before
```

MongoDB

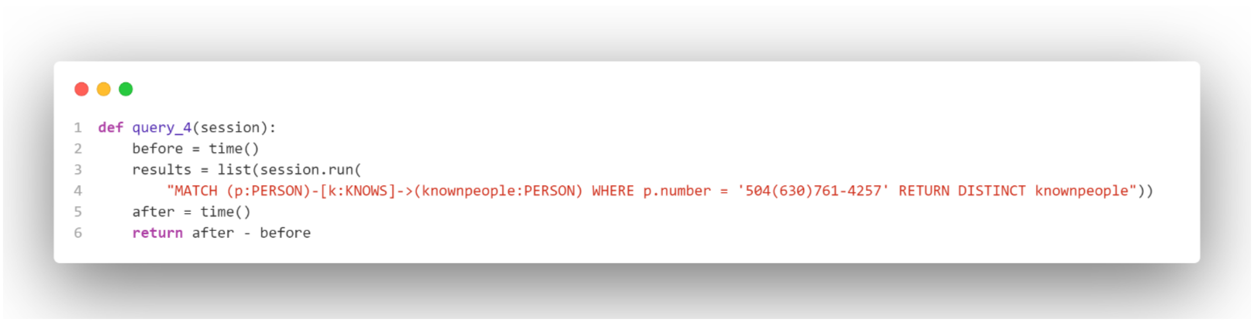


```python
def query_3():
    before = time()
    results = list(db["calls"].find({"id_caller": "351(905)372-6077", "duration": {"$gte": 4000}}))
    after = time()
    return after - before
```

Both queries aim to find calls associated with a specific caller number and with a duration greater than 4000. The Neo4j query returns a list of Neo4j nodes representing the CALL nodes matching the specified graph pattern. The MongoDB query returns a list of documents from the "calls" collection that match the query conditions

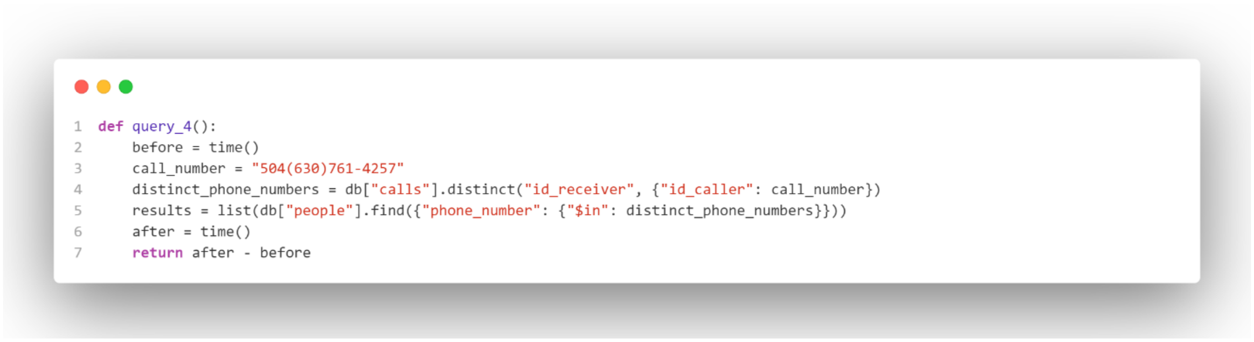
Query 4: Known people from phone number

Neo4j



```python
def query_4(session):
    before = time()
    results = list(session.run(
        "MATCH (p:PERSON)-[k:KNOWS]->(knownpeople:PERSON) WHERE p.number = '504(630)761-4257' RETURN DISTINCT knownpeople"))
    after = time()
    return after - before
```

MongoDB



```python
def query_4():
    before = time()
    call_number = "504(630)761-4257"
    distinct_phone_numbers = db["calls"].distinct("id_receiver", {"id_caller": call_number})
    results = list(db["people"].find({"phone_number": {"$in": distinct_phone_numbers}}))
    after = time()
    return after - before
```

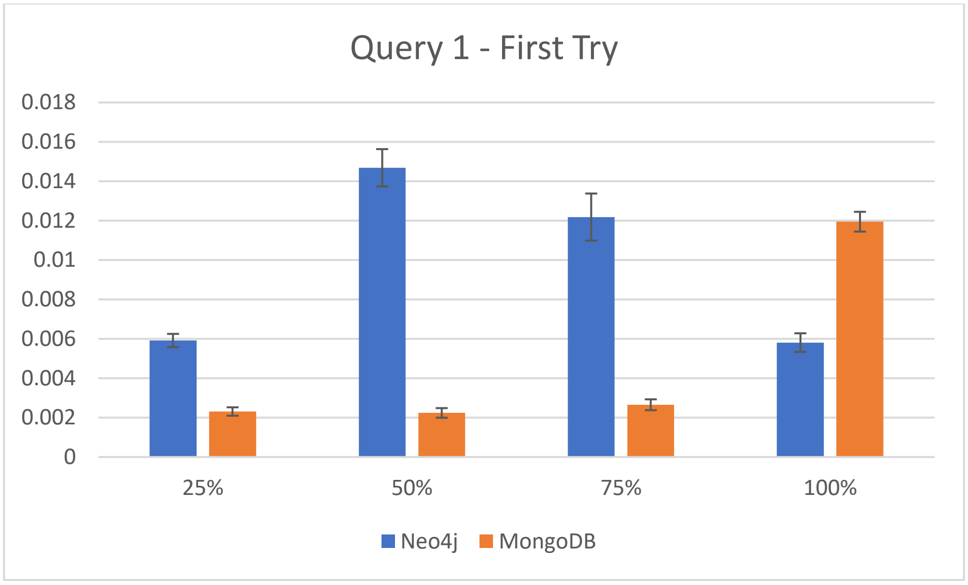
In the fourth query, the Neo4j has better performance due to its graph database nature and graph pattern matching capabilities.

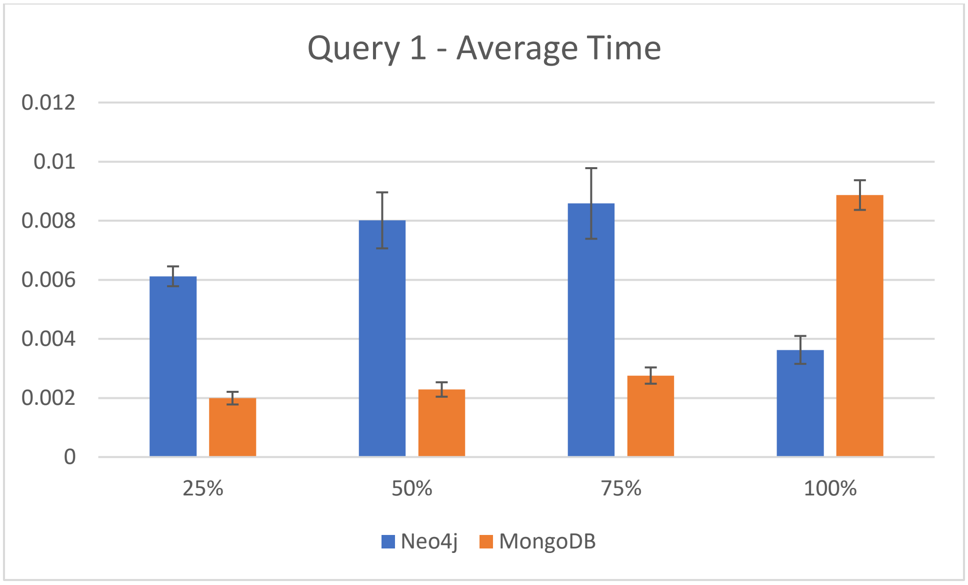
V. Experiments

1. Query 1

Neo4j				QUERY 1	MongoDB			
25%	50%	75%	100%		25%	50%	75%	100%
0.005915	0.014684	0.012183	0.00581		0.002311	0.002239	0.002653	0.011947
0.007289	0.010053	0.012862	0.006		0.001573	0.003532	0.00202	0.010747
0.006566	0.010913	0.012239	0.004689		0.001572	0.003016	0.001618	0.012326
0.008287	0.010047	0.012925	0.004393		0.002038	0.002052	0.003271	0.009543
0.005319	0.007521	0.011925	0.00645		0.001051	0.001573	0.004018	0.00996
0.007824	0.009224	0.01183	0.003929		0.003572	0.002546	0.002371	0.011202
0.004737	0.007799	0.008938	0.004526		0.00225	0.001095	0.00502	0.009452
0.00737	0.008295	0.006488	0.005013		0.002018	0.00401	0.003688	0.009988
0.006087	0.006798	0.005579	0.004		0.002041	0.003013	0.003041	0.011013
0.005272	0.018037	0.021787	0.002982		0.003065	0.002047	0.002333	0.008993
0.006538	0.008657	0.006307	0.007135		0.002837	0.001024	0.003575	0.008007
0.006092	0.006133	0.00745	0.00402		0.001038	0.002036	0.002046	0.007988
0.007535	0.009026	0.006341	0.003009		0.002055	0.003047	0.003084	0.008
0.005247	0.007421	0.006641	0.003026		0.001575	0.001555	0.002261	0.008086
0.006998	0.006999	0.006546	0.002993		0.002039	0.002721	0.002048	0.007015
0.006455	0.006378	0.006848	0.003015		0.001596	0.002036	0.002565	0.008069
0.00653	0.007008	0.00645	0.002985		0.001045	0.00158	0.004038	0.007774
0.005008	0.008972	0.010285	0.003002		0.003055	0.002588	0.003048	0.010136
0.006013	0.008243	0.007092	0.003333		0.002027	0.003059	0.002531	0.008882
0.006064	0.006087	0.006568	0.003029		0.002044	0.002026	0.002569	0.008309
0.005988	0.007025	0.007905	0.002006		0.002538	0.002031	0.003028	0.007694
0.005362	0.00683	0.006101	0.003013		0.002038	0.002534	0.002034	0.008169
0.005291	0.006693	0.007118	0.00403		0.002038	0.002035	0.00354	0.007313
0.00544	0.007586	0.005921	0.002996		0.002072	0.003021	0.002563	0.008414
0.006007	0.004982	0.00641	0.003007		0.002025	0.002017	0.002044	0.00754
0.00547	0.006767	0.007875	0.003004		0.002044	0.001574	0.002032	0.008986
0.005233	0.006581	0.006428	0.002009		0.001014	0.001574	0.002238	0.008663
0.007586	0.005985	0.006757	0.00299		0.002045	0.002052	0.003613	0.00701
0.005105	0.006982	0.007538	0.001997		0.002015	0.003016	0.002562	0.007744
0.006276	0.006051	0.010174	0.002001		0.002039	0.002206	0.002012	0.007965
0.004756	0.004576	0.006572	0.002029		0.001136	0.002031	0.002029	0.007952
0.006118	0.008011	0.008583	0.003626	Average Time	0.001994	0.002287	0.002758	0.008867
0.000954	0.002694	0.003396	0.00134	Standard Deviation	0.000607	0.000694	0.000783	0.001426
0.000336	0.000948	0.001195	0.000472	Confidence Interval 95%	0.000214	0.000244	0.000276	0.000502

MongoDB generally shows lower average execution times compared to Neo4j for most of the specific queries. However, in one of the measurements, MongoDB's execution time is slightly higher than Neo4j. The standard deviations for MongoDB are generally lower, indicating more consistent performance.

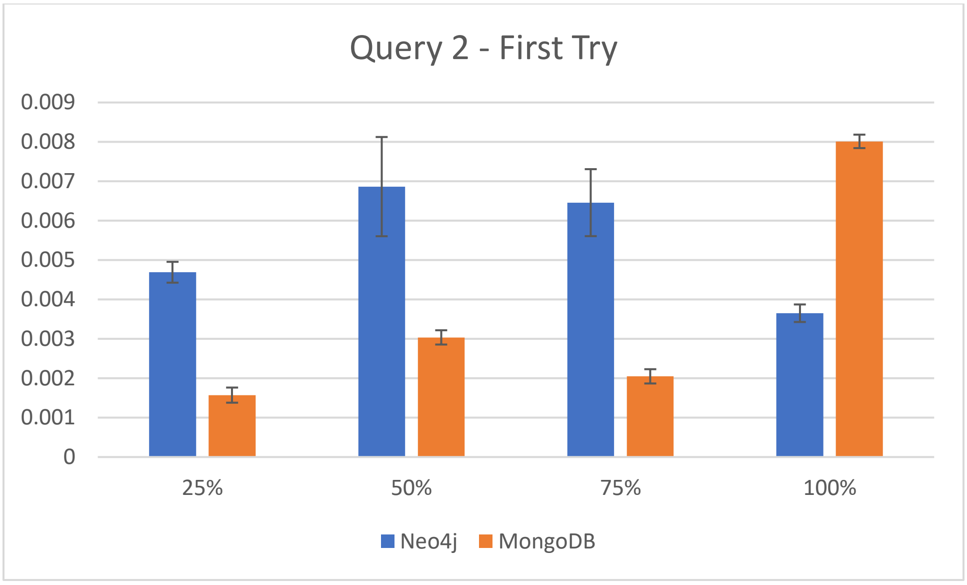

Query 1 - First Try
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
25%
50%
75%
100%
Neo4j
MongoDB

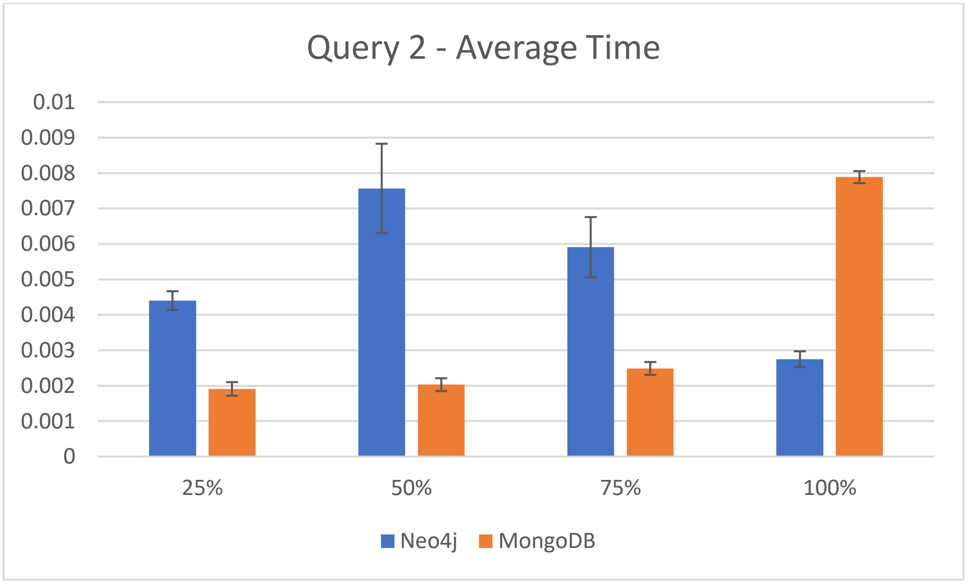

Query 1 - Average Time
0.012
0.01
0.008
0.006
0.004
0.002
0
25%
50%
75%
100%
Neo4j
MongoDB

2. Query 2

Neo4j				QUERY 2	MongoDB			
25%	50%	75%	100%		25%	50%	75%	100%
0.004691	0.006864	0.006458	0.00365		0.001571	0.003037	0.002048	0.008013
0.005065	0.005922	0.005307	0.00301		0.001575	0.002051	0.003088	0.008071
0.005379	0.005007	0.006194	0.002992		0.00155	0.002539	0.002538	0.008301
0.006395	0.005505	0.006123	0.002005		0.002037	0.002541	0.002047	0.007035
0.004162	0.005607	0.006839	0.001981		0.002024	0.002029	0.002747	0.008194
0.005035	0.00528	0.005379	0.004005		0.003031	0.002015	0.002034	0.00852
0.003996	0.005658	0.004681	0.003667		0.001554	0.000506	0.003025	0.007497
0.003998	0.006077	0.006155	0.004138		0.001044	0.001009	0.002547	0.007795
0.004997	0.005949	0.005362	0.002033		0.00202	0.001555	0.002559	0.007723
0.004051	0.024493	0.018285	0.002983		0.002181	0.002016	0.002562	0.007082
0.003701	0.014453	0.005361	0.001976		0.001617	0.001554	0.00205	0.00802
0.003762	0.008422	0.005484	0.00298		0.002018	0.002083	0.00157	0.008075
0.004199	0.007903	0.00631	0.002961		0.002055	0.00166	0.002558	0.007948
0.004001	0.008983	0.005368	0.003085		0.002036	0.002115	0.002033	0.008076
0.004	0.008448	0.007244	0.002016		0.00394	0.002036	0.002563	0.008024
0.004999	0.007591	0.005042	0.002972		0.002102	0.003119	0.002567	0.007977
0.004999	0.006674	0.005835	0.002958		0.001948	0.002029	0.003084	0.007785
0.005001	0.008126	0.005363	0.002599		0.002043	0.002573	0.002017	0.007429
0.003998	0.006715	0.005241	0.002825		0.001594	0.002671	0.002033	0.008805
0.004194	0.007121	0.004903	0.003083		0.002026	0.002035	0.004018	0.007275
0.004163	0.006463	0.006183	0.001971		0.002024	0.002041	0.002031	0.00813
0.005186	0.006804	0.00501	0.001951		0.001597	0.002374	0.003048	0.008329
0.005118	0.006815	0.005631	0.003013		0.002021	0.001554	0.002547	0.007349
0.005039	0.007746	0.00497	0.001946		0.001058	0.002034	0.00203	0.007547
0.003469	0.006035	0.005492	0.003029		0.002087	0.002049	0.002015	0.008509
0.003339	0.007002	0.005044	0.003048		0.002125	0.002015	0.003531	0.007929
0.003198	0.007747	0.005227	0.002007		0.001108	0.002039	0.002573	0.007639
0.005153	0.006	0.003991	0.002973		0.002125	0.002028	0.002543	0.007039
0.003211	0.007249	0.004954	0.001913		0.001543	0.001556	0.002038	0.008204
0.004399	0.005862	0.003709	0.002739		0.002018	0.002034	0.002543	0.007281
0.003578	0.006044	0.006018	0.002714		0.001561	0.002035	0.002539	0.00884
0.004402	0.007567	0.005909	0.002749	Average Time	0.001911	0.00203	0.002488	0.007885
0.000752	0.003578	0.002414	0.000633	Standard Deviation	0.000545	0.000515	0.000516	0.000484
0.000265	0.001259	0.00085	0.000223	Confidence Interval 95%	0.000192	0.000181	0.000182	0.00017

Based on the provided statistics, MongoDB generally shows lower average execution times compared to Neo4j for most of the specific queries

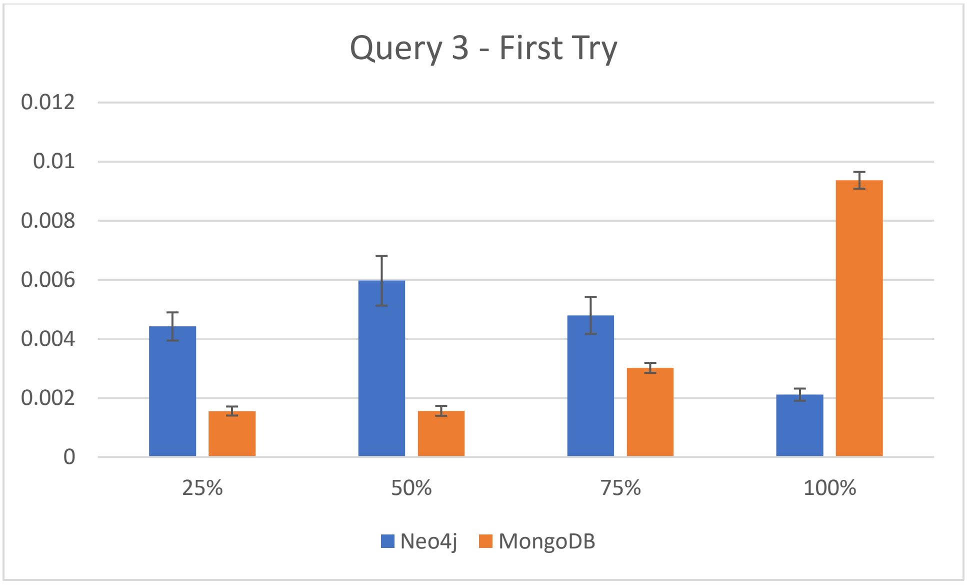

Query 2 - First Try
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
25%
50%
75%
100%
Neo4j
MongoDB

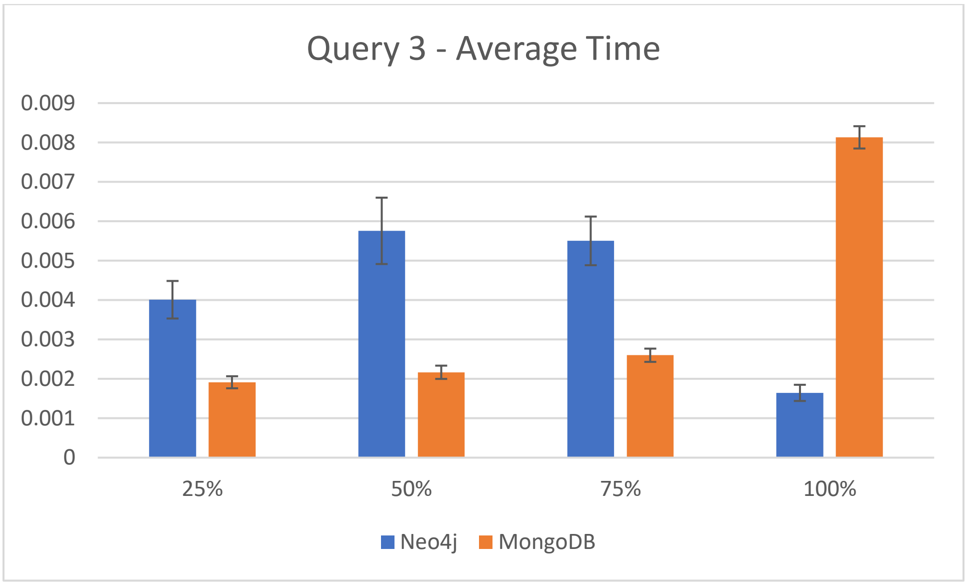

Query 2 - Average Time
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
25%
50%
75%
100%
Neo4j
MongoDB

3. Query 3

Neo4j				QUERY 3	MongoDB			
25%	50%	75%	100%		25%	50%	75%	100%
0.004419	0.005969	0.004791	0.002111		0.001556	0.001565	0.003019	0.009368
0.003001	0.005376	0.005289	0.000998		0.002024	0.002542	0.002565	0.008109
0.003991	0.005326	0.006223	0.001959		0.002042	0.002156	0.002546	0.007934
0.004298	0.006015	0.004735	0.002059		0.002038	0.002556	0.002024	0.008006
0.003552	0.005253	0.005333	0.00197		0.002015	0.00156	0.00205	0.007819
0.003096	0.007319	0.006038	0.000963		0.001559	0.002561	0.003029	0.008002
0.003997	0.008007	0.005187	0.002002		0.002041	0.002051	0.00235	0.008785
0.003982	0.006425	0.00471	0.001999		0.002038	0.003033	0.002594	0.007093
0.004311	0.006812	0.005262	0.001888		0.002023	0.002084	0.003016	0.007784
0.00314	0.017521	0.014536	0.00107		0.001556	0.002429	0.003037	0.007897
0.003011	0.004004	0.00493	0.001998		0.002037	0.002019	0.002045	0.008671
0.004259	0.005952	0.005308	0.001965		0.002032	0.002048	0.002563	0.007913
0.003328	0.005514	0.005942	0.001084		0.001557	0.002033	0.002604	0.008152
0.003596	0.004847	0.005002	0.001956		0.002019	0.002017	0.003029	0.007999
0.003125	0.005005	0.006067	0.000988		0.002039	0.001559	0.002067	0.010501
0.004225	0.004958	0.006029	0.00201		0.002025	0.001665	0.003191	0.01062
0.004028	0.004838	0.005207	0.001001		0.002025	0.00205	0.0026	0.007839
0.003525	0.004462	0.004626	0.003029		0.002032	0.002034	0.00205	0.007912
0.003686	0.004601	0.005791	0.002115		0.001559	0.001571	0.002044	0.008073
0.003873	0.004038	0.004789	0.001058		0.002018	0.003036	0.002034	0.007029
0.003627	0.005021	0.005795	0.002004		0.00103	0.002047	0.003156	0.007942
0.003572	0.004026	0.004934	0.001		0.001032	0.002074	0.002589	0.008351
0.004014	0.004406	0.005009	0.002019		0.002028	0.002054	0.00256	0.007678
0.004057	0.004372	0.004398	0.000994		0.001569	0.002036	0.002555	0.007875
0.010779	0.004333	0.004285	0.000996		0.002069	0.002036	0.004038	0.007679
0.003772	0.006001	0.005368	0.002		0.001575	0.001555	0.002019	0.008691
0.004102	0.004956	0.004999	0.001		0.002038	0.003037	0.002564	0.007568
0.00459	0.006121	0.005227	0.000988		0.002039	0.003031	0.002053	0.008008
0.003363	0.004968	0.004438	0.002008		0.003617	0.001586	0.003025	0.007623
0.002998	0.005403	0.005003	0.002668		0.002059	0.003021	0.003033	0.007113
0.004956	0.00664	0.005337	0.001074		0.002037	0.002145	0.002532	0.008025
0.004009	0.005758	0.005503	0.001644	Average Time	0.001914	0.002167	0.002599	0.008131
0.001353	0.002393	0.001754	0.000584	Standard Deviation	0.000433	0.000479	0.000479	0.000804
0.000476	0.000843	0.000618	0.000206	Confidence Interval 95%	0.000153	0.000169	0.000169	0.000283

MongoDB significantly outperforms Neo4j in terms of average execution times for all of the specific queries. MongoDB also demonstrates lower standard deviations, indicating more consistent performance. The confidence intervals for both databases are quite close, but MongoDB's confidence intervals are generally smaller, suggesting more precise estimates of the true average execution times.

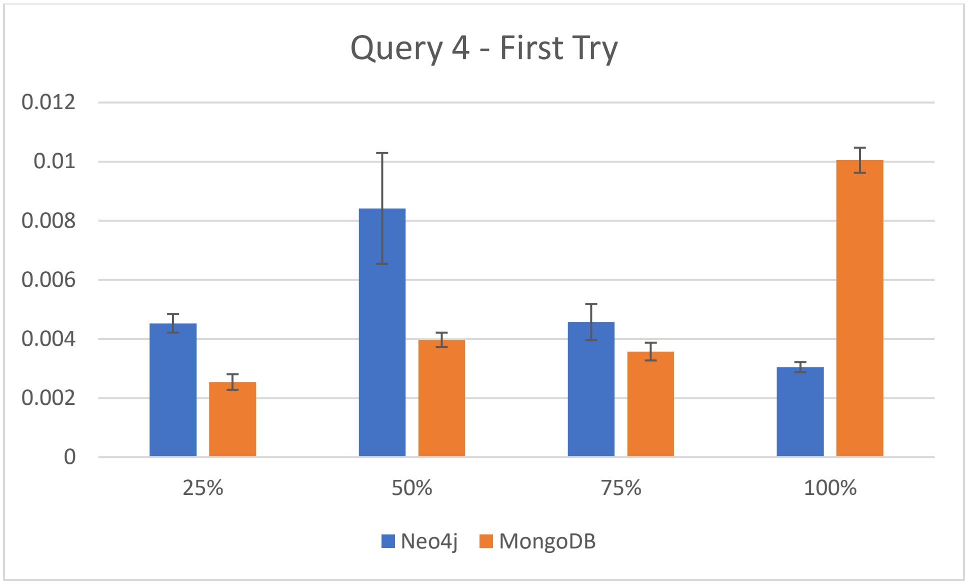

Query 3 - First Try
0.012
0.01
0.008
0.006
0.004
0.002
0
25%
50%
75%
100%
Neo4j
MongoDB

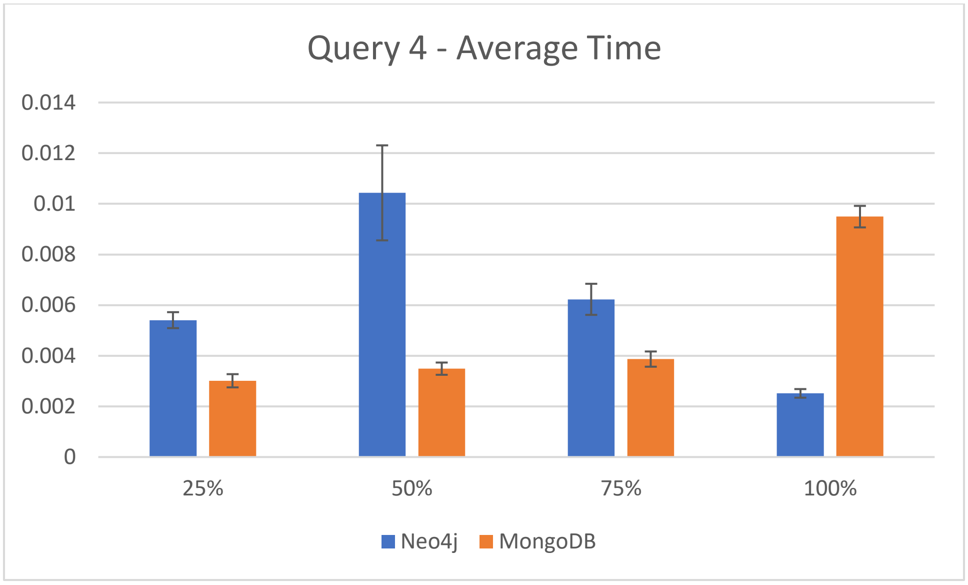

Query 3 - Average Time
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
25%
50%
75%
100%
Neo4j
MongoDB

4. Query 4

Neo4j				QUERY 4	MongoDB			
25%	50%	75%	100%		25%	50%	75%	100%
0.004525	0.008414	0.004572	0.003039		0.002538	0.003968	0.003569	0.010048
0.00623	0.007629	0.005577	0.002995		0.002558	0.004214	0.005305	0.009157
0.005683	0.006749	0.004802	0.002991		0.003036	0.003557	0.005037	0.008986
0.005285	0.011808	0.006027	0.002959		0.004037	0.003015	0.004556	0.009189
0.00619	0.013154	0.005005	0.001956		0.002468	0.003032	0.004047	0.008182
0.006159	0.018147	0.006307	0.002728		0.002563	0.004027	0.004291	0.009916
0.005084	0.017788	0.006243	0.002062		0.00304	0.003374	0.004023	0.011003
0.005195	0.014378	0.006423	0.001972		0.003562	0.00446	0.003025	0.014527
0.005425	0.012829	0.006782	0.003022		0.00254	0.003032	0.003038	0.010769
0.00473	0.031178	0.014592	0.001983		0.003028	0.003031	0.003927	0.008917
0.004495	0.014246	0.00578	0.002944		0.002024	0.004807	0.002784	0.009074
0.00524	0.013235	0.006522	0.002999		0.002038	0.003015	0.001988	0.009526
0.004016	0.005766	0.006224	0.00207		0.003541	0.00365	0.003017	0.008018
0.004001	0.01656	0.007277	0.001989		0.002562	0.004056	0.004023	0.010499
0.006318	0.008804	0.006287	0.002		0.002025	0.003035	0.003547	0.008001
0.005171	0.00501	0.008265	0.001998		0.003418	0.003046	0.003026	0.010006
0.005181	0.00783	0.006666	0.001994		0.004016	0.002558	0.00404	0.010077
0.005001	0.005995	0.005525	0.00301		0.003027	0.004038	0.004071	0.008804
0.004517	0.006159	0.006752	0.001989		0.004941	0.002949	0.003021	0.008181
0.005002	0.007715	0.00538	0.00259		0.002565	0.003026	0.006037	0.009454
0.005344	0.005623	0.004541	0.002126		0.004036	0.003018	0.005038	0.009952
0.004071	0.009365	0.005543	0.002008		0.003539	0.003396	0.004026	0.009187
0.005442	0.006861	0.006039	0.002963		0.002037	0.004035	0.004043	0.008996
0.006264	0.009589	0.00585	0.003005		0.002038	0.00305	0.00404	0.008845
0.007428	0.006739	0.004729	0.003028		0.004225	0.003017	0.004536	0.009795
0.004557	0.010101	0.006003	0.002881		0.003531	0.005017	0.003025	0.00908
0.005985	0.006999	0.005926	0.002099		0.003037	0.002538	0.004322	0.009261
0.006998	0.011263	0.00549	0.002021		0.002535	0.002558	0.004027	0.010211
0.005998	0.008032	0.006161	0.003016		0.003307	0.003048	0.003024	0.008967
0.007157	0.0086	0.006218	0.002281		0.002535	0.004725	0.003039	0.008572
0.004921	0.006851	0.005567	0.003229		0.003048	0.003909	0.004425	0.009111
0.005407	0.010433	0.006228	0.002515	Average Time	0.003013	0.00349	0.003868	0.009494
0.000895	0.005329	0.001743	0.000484	Standard Deviation	0.000742	0.000688	0.000855	0.001207
0.000315	0.001876	0.000614	0.00017	Confidence Interval 95%	0.000261	0.000242	0.000301	0.000425

MongoDB generally shows lower average execution times compared to Neo4j for these specific queries. The standard deviations and confidence intervals for both databases indicate relatively consistent performance with some slight variability





VI. Conclusions

Overall, when considering execution time, consistency, and precision, MongoDB demonstrates better performance characteristics in terms of average time, standard deviation, and confidence interval. However, it is important to note that the comparison is based on the specific queries provided, and the performance of each database can vary depending on the use case, data model, query complexity, indexing, and other factors.

Ultimately, the choice between Neo4j and MongoDB should consider the specific requirements of the application, the data structure, and the nature of the queries. It is recommended to conduct more extensive performance testing, evaluate the suitability of each database for the given use case, and consider other factors such as scalability, ease of development, and community support. For this project, the Neo4j is still a better choice because many relationships created through nodes is suitable for this type of data. Graph technology allows us to work with the data in its natural form