UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Progetto Basi di Dati NoSQL

Domenico
Villari
Matricola 527221

Francesco Maria
Russo
Matricola 526864

ANNO ACCADEMICO 2022/2023

Indice

1 Introduzione

Nel contesto dell'evoluzione tecnologica e della gestione dei dati distribuiti, i database NoSQL (Not Only SQL) sono emersi come alternative ai tradizionali database SQL. Questo cambiamento è stato guidato dalla necessità di affrontare le sfide dei Big Data, caratterizzati da enormi volumi di dati, spesso non strutturati o semistrutturati, che superano le capacità dei database SQL convenzionali.
Nel complesso mondo dei database NoSQL, è essenziale sottolineare che ogni tecnologia è unica e offre prestazioni e comportamenti diversi. Non esiste un database NoSQL "migliore" in senso assoluto, ma piuttosto diverse categorie di database NoSQL che possono essere più o meno adeguati ad un data applicazione.
Infatti tutti i Database NoSQL rispettano il teorema CAP, il quale afferma che in un sistema distribuito è possibile garantire contemporaneamente solo due dei tre aspetti:

- Consistenza (Consistency)
- Disponibilità (Aviability)
- Tolleranza alle partizioni (Partition tolerance)

Tra le varie categorie di DB NoSQL troviamo:

- Database Key-Value (SimpleDB)
- Database Document-Oriented (MongoDB)
- Database Column-Oriented (Cassandra)
- Database Graph (Neo4j)

Ecco perché è fondamentale comprendere le diverse caratteristiche e capacità di ciascuna tipologia. Attraverso i benchmark, il progetto ci consentirà di valutare in modo oggettivo le prestazioni di due tipologie di database NoSQL, MongoDB e Neo4j, in uno scenario concreto, aiutandoci a trarre conclusioni informate sulla scelta migliore per l'applicazione presa in considerazione.

2 Problema Affrontato

La tematica centrale di questo studio riguarda l'utilizzo di analisi incrociate tra chiamate telefoniche, date e luoghi rilevanti al fine di identificare potenziali criminali o limitare il campo delle ricerche investigative. Un esempio concreto può essere il seguente: durante un'indagine su una rapina, si potrebbe procedere conducendo un'analisi delle chiamate effettuate nelle celle telefoniche che coprono le zone circostanti al luogo del crimine, in specifiche date. Questo approccio mira a individuare un gruppo di persone che potrebbero essere più frequentemente connesse a tali circostanze, sia come chiamanti che come chiamati, consentendo così di restringere il cerchio delle indagini a individui che potrebbero essere coinvolti nel crimine.

3 Database

I due database presi in considerazione nel seguente progetto, sono rispettivamente MongoDB e Neo4j, i quali sono molto diversi tra di loro, infatti il primo è Document Oriented, mentre il secondo è un Graph Database. Dunque nello studio saranno ben visibili le differenze tra i due.

4 MongoDB

MongoDB è un sistema di gestione di database NoSQL di tipo document-oriented e schema-free.
La sua struttura si basa su documenti in un formato simil JSON chiamto BSON. I vantaggi di MongoDB sono riscontrabili nella possibilità di fare query complesse, avere un installazione e configurazione molto semplice e per la sua scalabilità. Infatti la vera potenza di MongoDB è visibile nella sua configurazione distribuita, che per semplicità non è stata usata nel contesto del progetto. La tecnica dello sharding, ossia il partizionamento dei dati in cui il database distribuito divide orizzontalmente il carico di lavoro e la gestione dei dati su più server, chiamati nodi o shard (ognuno dei quali costituito da un replica set di almeno 3 nodi), permette una elaborazione parallela delle operazioni e fault-tolerance alla caduta di uno o più nodi. Nel caso specifico applicazione del progetto non si è utilizzata una configurazione distribuita per semplicità.

4.1 Container

Il database di MongoDB è stato utilizzato creando un container Docker utilizzando il comando sottostante:

```
$ docker run --name mongodb -d -p 27017:27017 mongo
```

È stato quindi creato un container usando l'immagine ufficiale di MongoDB ed esponendo la porta 27017, ovvero la porta usata di default da MongoDB per connessioni esterne.

5 Neo4j

Neo4j è un sistema di gestione di database NoSQL orientato ai grafi. Questo significa che organizza i dati seguendo la struttura di un grafo, in cui la più piccola unità di informazione è rappresentata da un nodo. Ogni nodo contiene le sue proprietà, e le relazioni tra i nodi sono rappresentate da archi orientati, ciascuno dei quali può avere anche delle proprietà.

Questa struttura di dati consente di sfruttare tutti i benefici di un grafo, come la possibilità di attraversare e cercare dati nell'intero grafo o in sottoinsiemi di esso. Inoltre, è possibile accedere direttamente alle proprietà dei nodi e delle relazioni, che sono implementate come coppie chiave-valore.

Un'altra caratteristica importante dei database a grafo, tra cui Neo4j, è la loro natura priva di schema. Ciò significa che ogni nodo può avere proprietà diverse

da altri nodi della stessa categoria, che vengono identificate attraverso etichette (labels) nel linguaggio di Neo4j chiamato Cypher.

I database a grafo sono estremamente versatili e solitamente offrono ottime prestazioni quando si tratta di gestire molte relazioni tra diverse entità. Tuttavia, presentano una limitazione nel fatto che il grafo non può essere suddiviso su più nodi, il che può rappresentare un problema in determinati scenari.

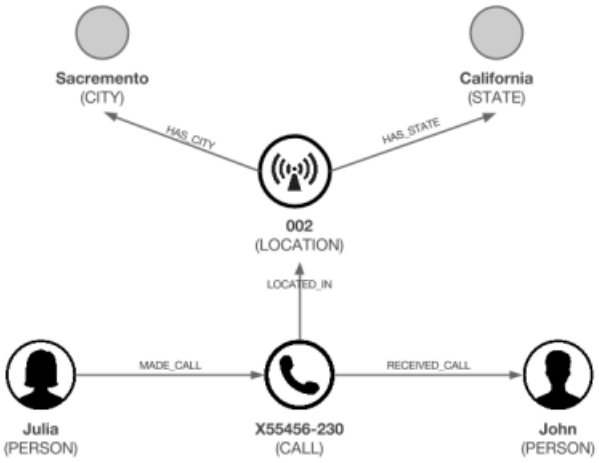
Il database di Neo4j è stato utilizzato creando un container Docker con il comando sottostante:

```
$ sudo docker run --name neo4jdb -p 7687:7687 -d
-p 7474:7474
-v $HOME/neo4j/data:/data -v $HOME/neo4j/logs:/logs
-v/home/domenico/Scrivania/ProgNoSQL
/Progetto-basi-NoSQL-2/csv:/var/lib/neo4j/import
-v $HOME/neo4j/plugins:/plugins  --env NEO4J_ACCEPT_LICENSE_AGREEMENT=yes
--env NEO4J_AUTH=none neo4j:5.11.0-enterprise
```

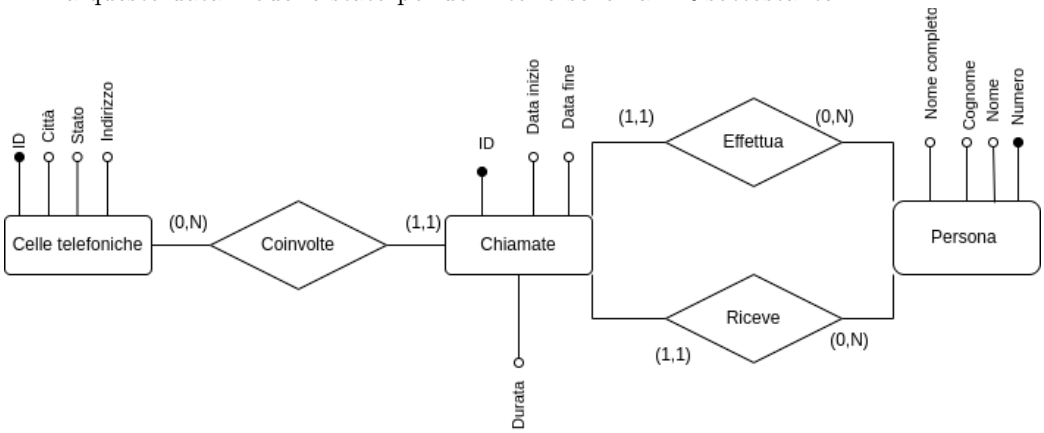
È stato creato un container usando l'immagine di Neo4j Enterprise in quanto da la possibilità di creare più database.
Inoltre si vanno ad esporre le porte 7474 e 7687. La prima da accesso a un'interfaccia browser per la gestione del database, mentre la seconda da accesso al database da connessioni esterne.

6 Datamodel

L'articolo fornitoci per la definizone del caso di studio va a definire un datamodel descritto dalla seguente figura:



Da questo data model è stato poi definito lo schema ER sottostante:



Il diagramma presenta 3 entità di base:

- *Persone*, caratterizzate da
  - Nome
  - Cognome
  - Nome Completo
  - Numero di telefono (primary key)
- *Chiamate*, caratterizzate da
  - Data inizio
  - Data fine
  - Durata (la differenza tra i primi 2)
  - ID (primary key)
- *Celle*, caratterizzate da
  - Città
  - Stato
  - Indirizzo
  - ID (primary key)

Le relazioni visibili sono le seguenti:

- *Effettua*, una persona effettua una chiamata (chiamante)
- *Riceve*, una persona riceve una chiamata (chiamato)
- *Coinvolte*, in una chiamata è coinvolta una cella telefonica

6.1 Datamodel in MongoDB

Su MongoDB le entità, diventate collezioni, sono le seguenti, con i relativi campi:

- *people*:
  - *full_name*, nome completo;
  - *first_name*, nome;
  - *last_name*, cognome;
  - *number*, numero della persona;
- *cells*:
  - *id*, id della cella;
  - *city*, città della cella;
  - *state*, provincia della cella;
  - *address*, indirizzo della cella;
- *calls*:
  - *cell_site*, cella su cui si è appoggiata la chiamata;
  - *calling*, persona che ha iniziato la chiamata;
  - *called*, persona che ha ricevuto la chiamata;
  - *startdate*, inizio della chiamata in timestamp;
  - *enddate*, fine della chiamata in timestamp;
  - *duration*, durata della chiamata;

L'ID della collezione *calls* è stato ottenuto utilizzando l'ID del documento assegnato automaticamente da MongoDB al momento dell'inserimento del documento stesso.

Per quanto riguarda le relazioni, MongoDB non consente di definirle in modo esplicito. Tuttavia, se c'è la necessità di gestire relazioni tra i dati, è compito del progettista sviluppare il database e le relative interfacce in modo da consentire l'esecuzione di query aggregate che possano tradurre in modo efficace le relazioni concettuali desiderate.

6.2 Datamodel in Neo4j

Su Neo4j le entità coinvolte sono 3, ossia:

- *person*:
  - *full_name*, nome completo;
  - *first_name*, nome;
  - *last_name*, cognome;
  - *number*, numero della persona;

- *cell*:
  - *id*, id della cella;
  - *city*, città della cella;
  - *state*, provincia della cella;
  - *address*, indirizzo della cella;
- *call*:
  - *cell_site*, cella su cui si è appoggiata la chiamata;
  - *calling*, persona che ha iniziato la chiamata;
  - *called*, persona che ha ricevuto la chiamata;
  - *startdate*, inizio della chiamata in timestamp;
  - *enddate*, fine della chiamata in timestamp;
  - *duration*, durata della chiamata;

A differenza di MongoDB, in Neo4j è possibile esplicitare le relazioni tra le entità e, come citato precedentemente, sono:

- *IS_CALLED*, da un nodo call a un nodo person;
- *IS_CALLING*, da un nodo person a un nodo call;
- *IS_DONE*, da un nodo call a un nodo cell;

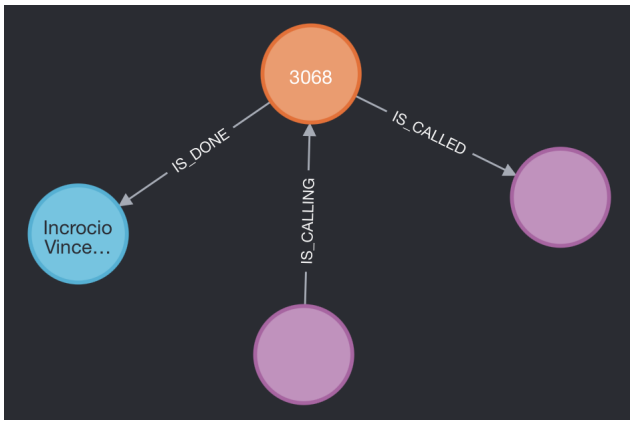


Figura 1: Esempio di relazione

7 Implementazione

Per l'ottenimento dei dati che popolano i database si è scritto uno script Python che generasse i le entità con i relativi attributi in modo casuale, tramite l'uso della libreria *Faker*.

Nello script vengono inizialmente definiti il numero massimo di entità e gli header dei rispettivi file csv.
Vengono generate le persone e le celle e, successivamente, vengono generate le chiamata in base alla dimensione del dataset. In questo modo le chiamate saranno generate sulla base delle persone e delle celle presenti in quella porzione di dataset.

```
'''Definizione delle variabili utili'''
percentage = [25, 50, 75, 100]
people= 4000
calls = 200000
cells = 2000

header_cells=["id","city","state","address"]
header_people=["full_name","first_name","last_name","number"]
header_calls=["cell_site",
    "calling","called","startdate","enddate","duration"]

cells_list=cells_generator(cells)
people_list=people_generator(people)


for p in percentage:
    num_people=people*p//100
    num_cells=cells*p//100
    num_calls=calls*p//100
    calls_list=calls_generator(calls,people_list[:num_people+1],
                cells_list[:num_cells+1])
    write_on_file("people",people_list[:num_people+1],
                header_people,p)
    write_on_file("cells",cells_list[:num_cells+1],
                header_cells,p)
    write_on_file("calls",calls_list[:num_calls+1],
                header_calls,p)
```

Di seguito la tabella dei dataset:

	25%	50%	75%	100%
Persone	1,000	2,000	3,000	4,000
Celle	500	1,000	1,500	2,000
Chiamate	50,000	100,000	150,000	200,000
Totale	**51,500**	**103,000**	**154,500**	**206,000**

Il codice per la generazione del dataset è possibile visionarlo nella repository di GitHub al seguente link.

7.1 Inserimento MongoDB

Per l'inserimento dei dati nel database di MongoDB si è usata la libreria *pymongo*. Per la connessione a MongoDB si usa l'oggetto *MongoClient*, si creano i database per le varie porzioni di dataset e le relative collezioni. Per l'inserimento dei dati nelle collezioni si è usata la funzione *insert_many*. Inoltre si è creato un indice sul campo *number* della collezione *people* per velocizzare il tempo di esecuzione delle query.

```
    client = pymongo.MongoClient("localhost", 27017)

percentage = [25, 50, 75, 100]

for p in percentage:
    db = client["progetto"+str(p)] #creazione db test tramite
        ogg client

    """
    Creazione 3 collezioni per distinte
    -1 Calls per identificare le chiamate definite da: id,
        numero chiamante, numero chiamato, durata
    -2 cells per identificare le celle telefoniche coinvolte
        nelle chiamate con: id, citta', stato, indirizzo
    -3 people per identificare le persone coinvolte
        caratterizzate da: numero di telefono, nome, cognome,
        nome completo
    """

    calls_coll = db["calls"]
    cells_coll= db["cells"]
    people_coll = db ["people"]

    calls_file = open("csv/calls"+str(p)+".csv")
    cells_file = open("csv/cells"+str(p)+".csv")
    people_file = open("csv/people"+str(p)+".csv")


    calls_reader = csv.DictReader(calls_file)
    cells_reader = csv.DictReader(cells_file)
    people_reader = csv.DictReader(people_file)

    calls_list = dictate(list(calls_reader))
    cells_list = dictate(list(cells_reader))
    people_list = dictate(list(people_reader))

    insert_cells = cells_coll.insert_many(cells_list)
    people_coll.create_index("number", unique=True,
        background=True)
    insert_people = people_coll.insert_many(people_list)
    insert_calls = calls_coll.insert_many(calls_list)
```

7.2 Inserimento Neo4j

Per l'inserimento su Neo4j, abbiamo usato la libreiria *py2neo*. Dapprima è stato generata la connessione tramite l'oggetto *Graph*, poi tramite la funzione *create_nodes_from_csv* andiamo a creare tutti i nodi nel seguente ordine: people,cells e calls. In particolare poi nella creazione dei nodi calls andiamo a recuperare i valori delle chiavi esterne nella tabella calls per poi, tramite 3 query, recuperare i nodi corrispondenti ed infine creare le corrette relazioni tra questi.

```
    for row in calls_csv:
        node = Node("call", cell_site=int(row["cell_site"]),
            calling=int(row["calling"]),
            called=int(row["called"]),
            startdate=int(row["startdate"]),
            enddate=int(row["enddate"]),
            duration=int(row["duration"]))
        graph.create(node)

        results=[]

        queries = [
                "MATCH (p1:person) WHERE \
                p1.number="+row["calling"]+" \
                RETURN p1",

                "MATCH (p2:person) WHERE \
                p2.number="+row["called"]+" \
                RETURN p2",

                "MATCH (c2:cell) WHERE \
                c2.id="+row["cell_site"]+" \
                RETURN c2"
        ]

        for i in range(len(queries)):
            results.append(graph.run(queries[i]))

        chiamante = results[0].evaluate()
        chiamato = results[1].evaluate()
        cella = results[2].evaluate()

        add_relationship(node, "IS_CALLED", chiamato)
        add_relationship(chiamante, "IS_CALLING", node)
        add_relationship(node, "IS_DONE", cella)
```

8 Test

I test delle query sono state effettuate su un computer con processore Intel i7 di undicesima generazione. In particolare sono state considerate 4 query di difficoltà crescente con le seguenti caratteristiche:

1. 1 WHERE;
2. 3 WHERE 1 JOIN;
3. 4 WHERE 2 JOIN;
4. 6 WHERE 3 JOIN;

La difficoltà crescente delle query serve a far evincere le divergenze tra i due DBMS presi in considerazione.

Le query sono state effettuate tramite script di Python che misurano anche i tempi di esecuzione delle query. In particolare gli script misurano:

- Tempo della prima esecuzione;
- Tempo delle 40 esecuzioni;
- Media delle 40 esecuzioni;
- Deviazione standard della media;

Dopo la prima esecuzione effettuata in un ambiente appena configurato, le 40 misurazioni mediate sono state effettuate per mettere a confronto le prestazioni dei meccanismi di caching dei due DBMS.

I dati vengono automaticamente salvati in un file csv per poi essere usati nella creazioni di istogrammi per il confronto.

8.1 Implementazione delle query

Per quanto concerne MongoDB, la prima query viene eseguita tramite un'operazione di tipo *find*. Questa operazione consente di cercare e ottenere i documenti che soddisfano un particolare criterio di ricerca. Le query successive, invece, fanno uso delle operazioni di *aggregate*. Queste ultime consentono di elaborare e analizzare in modo avanzato i dati, applicando operazioni di raggruppamento, trasformazione e altro, al fine di ottenere risultati più complessi rispetto alla semplice ricerca offerta dall'operazione *find*.

Di seguito la sezione di codice del file *mongo_ query.py* che consente la prima esecuzione delle query, il successivo calcolo del valore medio delle successive 40 esecuzioni, e dell'indice di confidenza. Infine viene esegueita la scrittura dei risultati su file.

```
    def query(coll, results, j):

    if (j == 1):
        start_time = time.time()
        coll.find(queries[j-1])
        end_time = (time.time() - start_time) * 1000
    else:
        start_time = time.time()
        coll.aggregate(queries[j-1])
        end_time = (time.time() - start_time) * 1000
    results.append(end_time)

    data = []

    for i in range(40):
        if (j == 1):
            start_time40 = time.time()
            coll.find(queries[j-1])
            end_time40 = (time.time() - start_time40) * 1000
        else:
            start_time40 = time.time()
            coll.aggregate(queries[j-1])
            end_time40 = (time.time() - start_time40) * 1000
        data.append(end_time40)
    results.append(sum(data))

    avg = results[3]/40
    results.append(avg)

    confidence_lvl = confidence(data)
    results.append(confidence_lvl)

    writer_result.writerow(results)
```

Per quanto riguarda Neo4j, la differenza risiede nel fatto che non vi sono operazioni di *find* o *aggregate* come in MongoDB. Le query vengono eseguite direttamente utilizzando il metodo *run*, rappresentando l'approccio primario per interrogare il database in Neo4j.
Di seguito il codice per l'esecuzione delle query in Neo4j, in cui come nell'esempio precedente, viene calcolato il primo tempo di esecuzione delle query, i successivi 40 tempi di cui si calcola il valore medio, ed il livello di confidenza per poi scrivere i risultati su file.

```
    for j in range(1, len(query)+1):
    for p in percentage:
        results = ["Query"+str(j), str(p)+"%"]
        graph = Graph("neo4j://127.0.0.1:7687",
            name="progetto"+str(p))

        start = time.time()
        graph.run(query[j-1])
        end = (time.time() - start) * 1000
        results.append(end)

        data = []
        for i in range(40):
            start40 = time.time()
            query_res = graph.run(query[j-1])
            end40 = (time.time() - start40) * 1000
            data.append(end40)
        results.append(sum(data))

        avg = results[3]/40
        results.append(avg)

        confidence_lvl = confidence(data)
        results.append(confidence_lvl)

        writer_result.writerow(results)
```

9 Query

9.1 Query n°1

La prima query essendo la più semplice si limita ad una semplice selezione di tutti quei nodi,nel caso di Neo4j, o documenti,nel caso di MongoDB, associati alle persone, dunque alla collezione *people*, in cui il campo *first_name* risulta essere "Laura". Va sottolineato, come già citato nella sezione 8.1, che questa query nel contesto di MongoDB utilizza il metodo *find* per avere performance migliori.

```
    {"first_name":"Laura"}
```

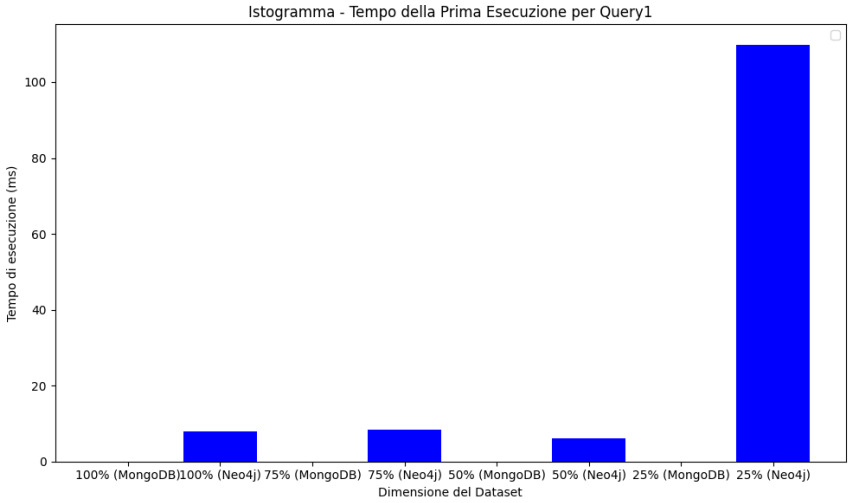
Listing 1: query MongoDB

```
    "MATCH (p:person)  \
        WHERE p.first_name='Laura'\
        RETURN p"
```

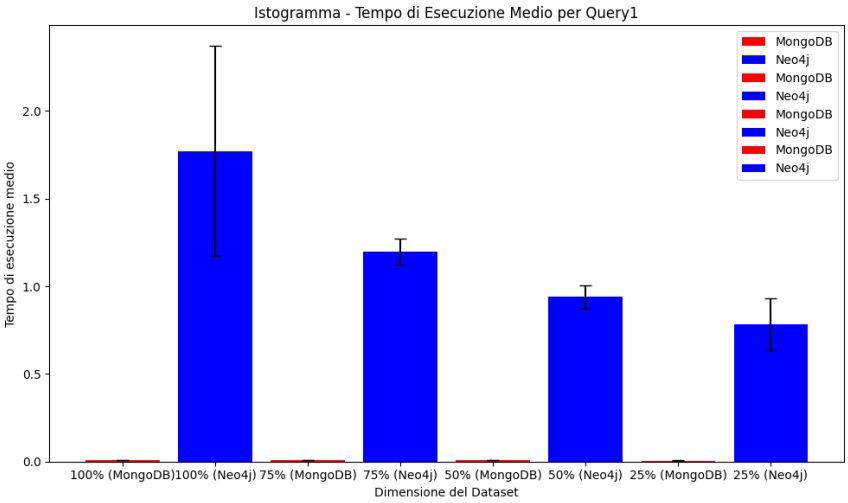
Listing 2: query Neo4j

Di seguito i tempi riscontrati:

Primi tempi		
	MongoDB	Neo4j
25%	0.0457	109.7426
50%	0.0197	6.1445
75%	0.0197	8.2762
100%	0.0283	7.9317



Tempi medi su 40 esecuzioni		
	MongoDB	Neo4j
25%	0.0058	0.7831
50%	0.0066	0.9396
75%	0.0066	1.1966
100%	0.0066	1.7711



9.2 Query n°2

Dalla query n°2 iniziano a esserci dei join tra più entità. Su MongoDB il join è stato gestito tramite l'operazione di *lookup*, mentre su Neo4j sono gestite nativamente.

```
    [{"$match": {"startdate": {
                        "$gte": start_search,
                        "$lt": end_search}}},
        { "$lookup": {
                        "from": "people",
                        "localField": "calling",
                        "foreignField": "number",
                        "as": "calling"}},
        {"$match":{"calling.first_name":"Laura"}}
        ]
```

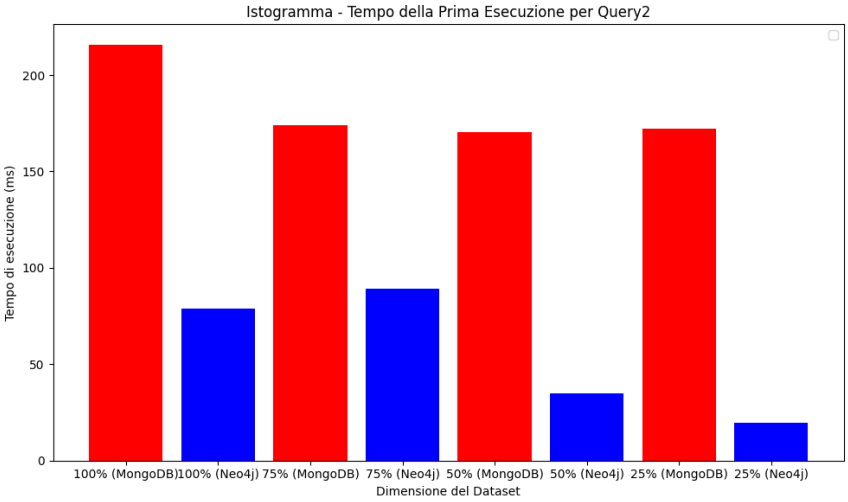
Listing 3: query MongoDB

```
    "MATCH (p:person)-[r1:IS_CALLING]->(c:call) \
        WHERE c.startdate>=1672617600 AND
            c.startdate<1672703999 \
        AND p.first_name='Laura'\
        RETURN p,c,r1"
```

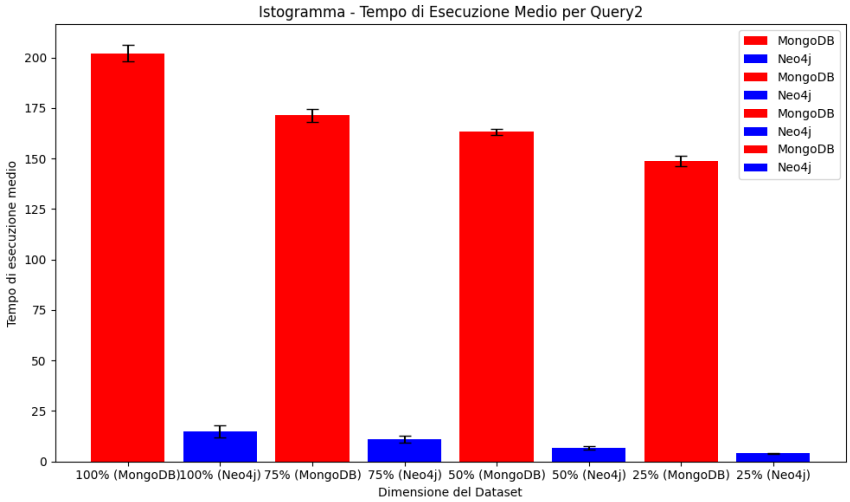
Listing 4: query Neo4j

Di seguito i tempi riscontrati:

Primi tempi		
	MongoDB	Neo4j
25%	172.1580	19.8302
50%	170.2880	35.0348
75%	174.1662	89.2856
100%	215.5680	78.7723



Tempi medi su 40 esecuzioni		
	MongoDB	Neo4j
25%	148.6856	3.9682
50%	163.2070	6.7623
75%	171.3641	10.9824
100%	202.0683	14.9304



9.3 Query n°3

Nella terza query sono presenti 4 WHERE e 2 JOIN.

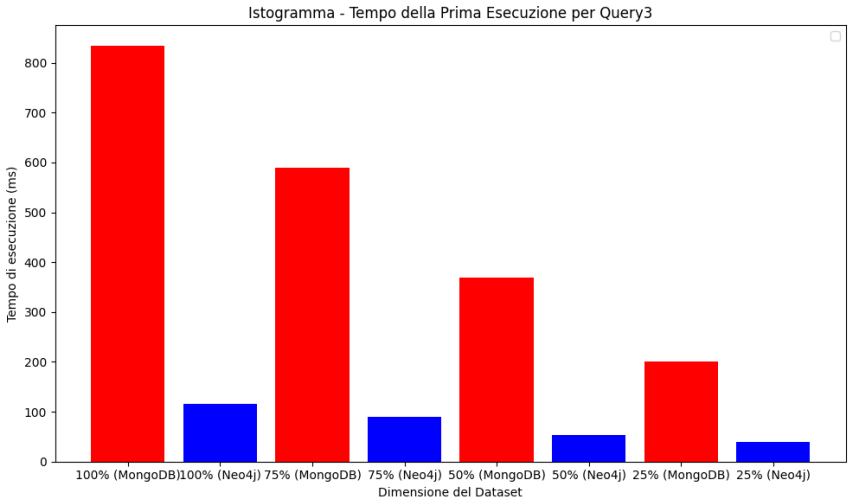
```
    [{"$match": {"startdate": {"$gte": start_search,
                                    "$lt": end_search}}},
            {"$lookup": {"from": "people",
                         "localField": "calling",
                         "foreignField": "number",
                         "as": "calling"}},
            {"$lookup": {"from": "cells",
                         "localField": "cell_site",
                         "foreignField": "id",
                         "as": "cell"}},
            {"$match": {"calling.first_name": "Laura" }},
            {"$match":{"cell.state":"Roma"}}
        ]
```

Listing 5: query MongoDB

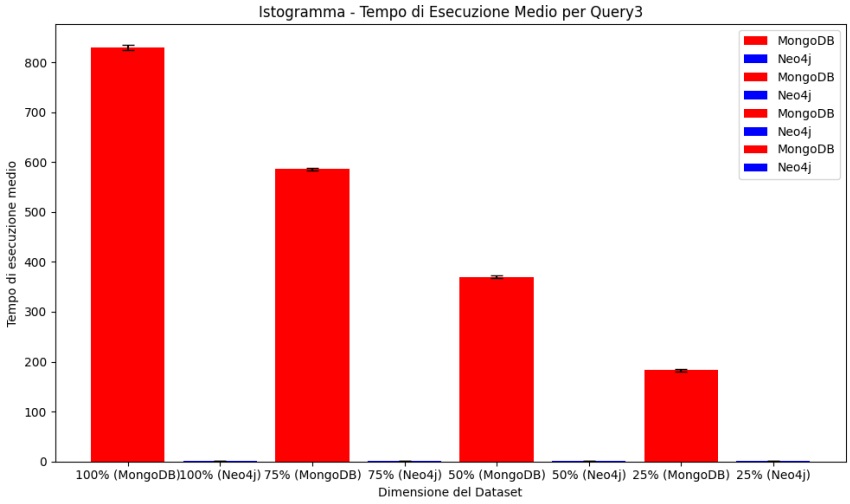
```
    "MATCH
    (p:person)-[r1:IS_CALLING]->(c1:call)-[r2:IS_DONE]->(c2:cell)
        WHERE c1.startdate>=1672617600 AND
            c1.startdate<1672703999 \
        AND p.first_name='Laura' \
        AND c2.state='Roma' \
        RETURN p,c1,r1,r2,c2"
```

Listing 6: query Neo4j

Primi tempi		
	MongoDB	Neo4j
25%	201.6532	39.9382
50%	368.8480	53.8678
75%	588.7808	90.3055
100%	833.6408	115.6044



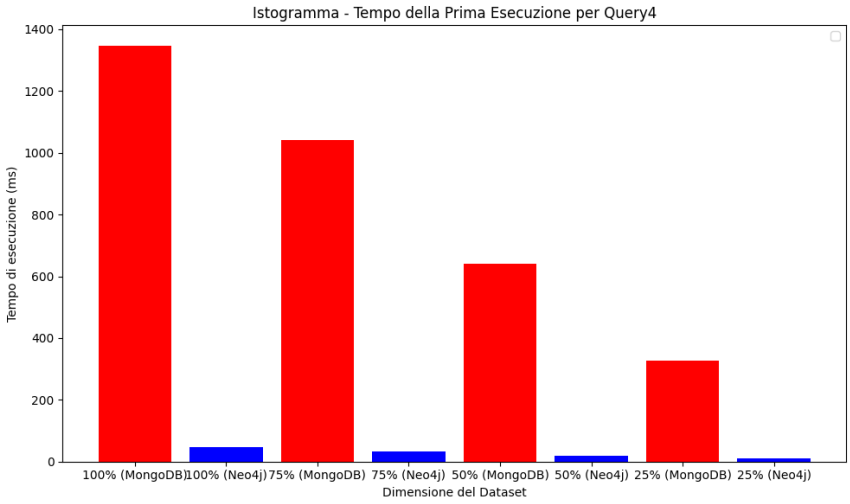
Tempi medi su 40 esecuzioni		
	MongoDB	Neo4j
25%	182.9889	0.8939
50%	370.2193	1.2012
75%	585.8394	1.6076
100%	829.4371	1.6807



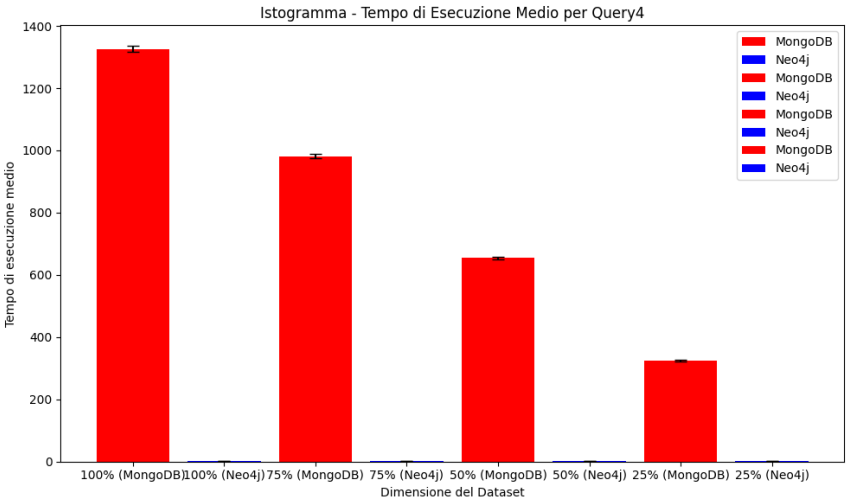
9.4 Query n°4

La quarta query, quella più complessa, presenta 6 WHERE e 3 JOIN.

Primi tempi		
	MongoDB	Neo4j
25%	326.4076	9.6461
50%	639.5936	18.2900
75%	1041.4028	32.9172
100%	1345.6213	48.0098



Tempi medi su 40 esecuzioni		
	MongoDB	Neo4j
25%	324.1247	0.9196
50%	654.6204	1.4808
75%	980.9311	1.6293
100%	1326.3194	2.1479



10 Conclusioni

In conclusione è possibile vedere come i due DBMS si comportano in modo diverso in base alla difficoltà ed alla natura della query .
Nei primi tempi della prima query entrambi i DBMS si comportano piuttosto bene, però MongoDB ha un tempo di esecuzione estremamente più piccolo rispetto a Neo4j. Ciò è dovuto dal fatto che MongoDB eccelle nell'indicizzazione e nell'accesso rapido ai singoli documenti.
Dalla seconda query in poi, ovvero da quando si fanno i JOIN tra le entità, MongoDB inizia ad avere tempi più elevati, dato che non supporta nativamente le relazioni, mentre Neo4j diventa estremamente veloce rispetto al primo essendo un database basato su grafi, quindi in grado di gestire relazioni anche molto complesse. Questa differenza si accentua sia ad aumentare della complessità della query che ad aumentare del dataset.