

UNIVERSITÀ DI MESSINA

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

CORSO DI LAUREA TRIENNALE IN INFORMATICA

---

# Analisi del raffronto tra Neo4j e MongoDB nel caso di studio dell'identificazione di attività criminali

---

*Autori:*

Gabriele ALOISIO (503264)  
Samuel Giacomo RAFFA  
(518206)

*Docenti:*

prof. Antonio CELESTI  
prof. Massimo VILLARI

15 gennaio 2023



# Indice

<b>1</b>	<b>Abstract</b>	<b>2</b>
<b>2</b>	<b>Introduzione</b>	<b>2</b>
<b>3</b>	<b>Caso di studio</b>	<b>4</b>
<b>4</b>	<b>Architettura</b>	<b>5</b>
4.1	Neo4j . . . . .	5
4.2	MongoDB . . . . .	5
4.3	Python . . . . .	6
<b>5</b>	<b>Implementazione</b>	<b>6</b>
5.1	Neo4j . . . . .	6
5.2	MongoDB . . . . .	7
5.3	Generazione dei dati . . . . .	8
5.4	Importazione dei dati . . . . .	11
<b>6</b>	<b>Risultati degli esperimenti</b>	<b>12</b>
6.1	2 WHERE . . . . .	12
6.2	3 WHERE . . . . .	14
6.3	3 WHERE 1 JOIN . . . . .	16
6.4	3 WHERE 2 JOIN . . . . .	18
<b>7</b>	<b>Conclusioni</b>	<b>20</b>

## 1 Abstract

Questa tesi presenta un'analisi delle performance dei DBMS NoSQL MongoDB e neo4j in relazione a quattro query scelte con complessità computazionale incrementale. Il caso di studio preso in considerazione è quello di utilizzare la tecnologia a grafo per identificare attività criminali attraverso l'analisi delle chiamate telefoniche. I test sono stati eseguiti su una macchina virtuale con specifiche hardware e utilizzando il linguaggio Python. I risultati dimostrano che MongoDB ha tempi di esecuzione più brevi rispetto a neo4j per tutte le query, con tempi di esecuzione più stabili e prevedibili. Inoltre, MongoDB mostra una maggiore efficienza nella gestione dei carichi più leggeri rispetto a neo4j. Questi risultati indicano che MongoDB è più efficiente per le query considerate rispetto a neo4j.

## 2 Introduzione

In questa trattazione scientifica si descrivono le differenze del rendimento di calcolo tra due popolari *Database Management Systems* (DBMS) di tipo NoSQL. I DBMS in questione sono il graph-oriented Neo4j e il document-oriented MongoDB. I DBMS NoSQL concedono di gestire il dato in modo più flessibile, rispetto al tradizionale e rigido modello tabulare dei database relazionali. I NoSQL infatti permettono di modellare la struttura in base alla necessità, introducendo paradigmi strutturali come *grafo*, *column-based* e *key-value*, ognuno aventi i suoi punti forti e casi obiettivo. Con essi comparirono anche il concetto di scalabilità delle basi di dati come sistemi distribuiti, dando vita al *Teorema CAP*, di Eric Brewer. Questo teorema afferma che **per un sistema informatico distribuito è impossibile fornire simultaneamente tutte e tre le seguenti garanzie:**

- Coerenza (copy-consistency)
- Disponibilità (availability)
- Tolleranza di partizione (partitioning)

Di fatto, le uniche coppie formabili con sistemi reali sono CP, AP e CA.

In questo trattato sono illustrate le misurazioni di performance (tempo di esecuzione e di risposta) dei DBMS NoSQL presi in considerazione in riferimento a quattro query scelte con complessità computazionale incrementale. Per questa analisi è stato scelto il caso di studio *How to use graph technology*

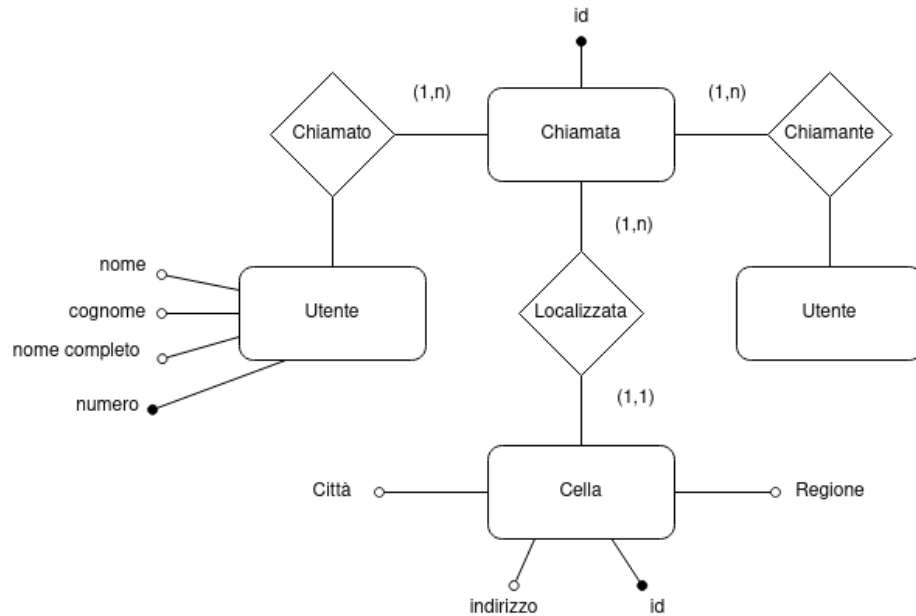
*to identify criminal activities in call records?*[1] presentato da Linkurious. I test sono stati eseguiti su una macchina virtuale *Debian Bullseye 11.5.0* con 6GB di memoria RAM e 4 core di un *AMD Ryzen 7 3700X* dedicati. Viene utilizzato il linguaggio *Python* per la generazione del dataset fittizio e il plotting delle misurazioni tramite fogli di calcolo (LibreOffice Calc).

### 3 Caso di studio

Si pone il problema dell'identificazione dell'attività criminale tramite l'analisi delle chiamate telefoniche effettuate nell'arco di tempo e nel luogo più inerenti al caso. Vengono utilizzati i DBMS NoSQL per facilitare il processo di rintracciamento di soggetti o per lo meno il restringimento del campo di ricerca. Questo viene fatto avvantaggiandosi delle rappresentazioni di strutture dati dei DBMS NoSQL rispetto alla tradizionale rappresentazione tabulare dei DBMS classici.

Prendiamo come esempio un gruppo di criminali che compie un furto. Sappiamo che il furto è avvenuto in un luogo  $L$  ad un tempo  $t$ . Tramite i strumenti a nostra disposizione, possiamo effettuare una ricerca sul database delle chiamate effettuate nel luogo  $L$  ed in un intorno definito di  $t$  e grazie alla modellazione a grafo è possibile visualizzare le chiamate incrociate, permettendoci di identificare terzi che sono inclusi indirettamente nel crimine.

Supposto che i gestori telefonici diano accesso al database e alla sua architettura, possiamo ammettere una struttura del tipo:



Da qui desumiamo le seguenti entità:

- Utente - identificato da un *numero* di telefono univoco, *nome*, *cognome* e *nome completo*
- Chiamata - identificata da un *id*
- Cella - identificata da un *id*, caratterizzata da una *città*, *indirizzo* e una *regione*

e le relazioni *localizzata*, *Chiamante* e *Chiamato*.

## 4 Architettura

### 4.1 Neo4j

Neo4j è un DBMS NoSQL *graph-oriented* open source sviluppato in Java. Permette un utilizzo in modalità server ed una modalità embedded. Lanciato in modalità server il database è un processo indipendente a cui si accede tramite lo stile architetturale REST. Sono supportati i plugin. In modalità embedded, il processo incorpora il database nell'applicazione Java e viene eseguito all'interno della JVM. Per l'importazione massiva di dati è supportata l'esecuzione in modalità non concorrente. Al dato importato è possibile crearne un nodo ed assegnarvi il tipo di dato (tipi elementari di Java) ed un nome. Come risultato abbiamo un grafo *schema-less*, che Permette un'eterogeneità del dato con il minimo sforzo. Neo4j si mostra molto efficiente nell'estrazione e rappresentazione di strutture ad albero come nei file XML, filesystem e reti. Da questo derivano implementazioni pronte all'uso di operazioni più comuni sui grafi come l'algoritmo di Dijkstra e ricerca dei cicli. Risulta invece scomodo nelle ricerche complesse, per esempio basate su confronti matematici e non supporta lo *sharding*.

### 4.2 MongoDB

MongoDB è un DBMS NoSQL *document-oriented*. Si basa sulla memorizzazione in base di documenti in formato *BSON*, ispirato al JSON ma con l'aggiunta della dinamicità. Supporta query *ad hoc* per la ricerca su base *regex*, intervalli o campi. Qualsiasi campo è indicizzabile, anche con indici secondari, unici, sparsi, geospaziali e full-text. MongoDB dispone dei *text-replica set*, cioè due o più copie di dati. Ogni copia è identificata come primaria o secondaria. Quando una replica primaria fallisce viene avviato un processo per la determinazione della copia più adatta a sostituire quella

primaria. Lo sharding permette a MongoDB di scalare orizzontalmente, implementando inoltre un meccanismo di load-balancing. La funzione GridFS permette di utilizzare il DBMS come file system, esponendo agli sviluppatori delle funzioni per la manipolazione del dato. GridFS divide il file in chunks e memorizza ognuno di questi in un documento separato.

### 4.3 Python

Python è un linguaggio di programmazione orientato agli oggetti. Si è scelto di utilizzare questo linguaggio per la semplicità di utilizzo e la facile disponibilità dei driver per accedere e manipolare i database MongoDB e Neo4j. In questo caso si utilizza la libreria *Faker* per generare i dati fittizi.

## 5 Implementazione

Come visto nella sezione 3, abbiamo esumato delle entità e relazioni. Queste sono state implementate nei due DMBS nel loro rispettivo modo.

### 5.1 Neo4j

Sono definiti i seguenti nodi

- *cell*:
  - cell\_site, ID
  - state
  - city
  - address
- *person*:
  - first\_name
  - last\_name
  - full\_name
  - number
- *call*:
  - calling\_number
  - called\_number

- start\_date
- end\_date
- duration
- cell\_cite, si riferisce al cell\_site di *cell*

E i seguenti archi (*label*)

- *made\_call*: person.number  $\rightarrow$  call.calling\_number
- *received\_call*: call.called\_number  $\rightarrow$  person.number
- *located\_in*: call.cell\_site  $\rightarrow$  cell.cell\_site

## 5.2 MongoDB

Sono definite le seguenti collezioni

- *cells*:
  - cell\_site, ID
  - state
  - city
  - address
- *people*:
  - first\_name
  - last\_name
  - full\_name
  - number
- *calls*:
  - calling\_number
  - called\_number
  - start\_date
  - end\_date
  - duration
  - cell\_cite, si riferisce al cell\_site di *cell*



E le relazioni

- *made\_call*: `person.number`  $\rightarrow$  `call.calling_number`
- *received\_call*: `call.called_number`  $\rightarrow$  `person.number`
- *located\_in*: `call.cell_site`  $\rightarrow$  `cell.cell_site`

### 5.3 Generazione dei dati

Si generano quattro dataset di dimensioni diverse da inserire successivamente sui database MongoDB e Neo4j. Le dimensioni stabilite sono 100%, 75%, 50%, 25%, che indicano rispettivamente le dimensioni dei dataset in base al dataset con il 100% del contenuto di informazioni. Per popolare il database con dati fittizi, si è scritto un codice Python che utilizza Faker. Il file `config.ini` contiene i parametri fissi che indicano il numero di record da generare. Al momento dell'avvio dello script viene passato un parametro *L* che indica la percentuale del carico in base a questi valori. Come valori predefiniti si è scelto 75000 persone, 25000 celle e 1000000 chiamate.

```
1      [load]
2      MAX_PEOPLE=75000
3      MAX_CELLS=25000
4      MAX_CALLS=1000000
5
```

Listing 1: `config.ini`

Il file `datasetgen.py` si occupa della generazione dei dati. È definita una funzione per ogni entità:

```
1 def gen_people(size: int):
2     with open(data_path("people.csv"), 'w', newline='') as file:
3         :
4             writer = csv.writer(file)
5             writer.writerow(['first_name', 'last_name', 'full_name',
6                             , 'number'])
7
8         for i in range(size):
9             name = fake.name().split()
10
11             #genera un numero univoco (simulazione di un do-
12             while)
13             while True:
14                 number = gen_fake_phone_number()
15                 if number not in people:
16                     break
17
18             writer.writerow([name[0], name[1], " ".join(name),
19                             number])
20             people.append(number)
21
22     file.close()
```

Listing 2: `gen_people()`

```
1 def gen_cells(size: int):
2     global ncells
3
4     with open(data_path("cells.csv"), 'w', newline='') as file:
5         writer = csv.writer(file)
6         writer.writerow(['cell_site', 'city', 'address', 'state',
7                             ,])
8
9         for i in range(size):
10             city = fake.city()
11             address = fake.street_name()
12             state = fake.current_country_code()
13
14             writer.writerow([i, city, address, state])
15
16     file.close()
```

Listing 3: `gen_cells()`

```

1 def gen_calls(size: int):
2     npeople = len(people)
3
4     with open(data_path("calls.csv"), 'w', newline='') as file:
5         writer = csv.writer(file)
6         writer.writerow(['calling_number', 'called_number', '
start_date', 'end_date', 'duration', 'cell_site'])
7
8         for i in range(size):
9             caller = randint(0, npeople-1)
10
11             #prende ripetutamente un numero dalla lista delle
persone se il numero del chiamante e' uguale a quello del
chiamato
12             while True:
13                 called = randint(0, npeople-1)
14                 if called != caller:
15                     break
16
17             duration = randint(0, 1000)
18             cell_site = randint(0, ncells-1)
19
20             #data di inizio [solo anno attuale]
21             start_date = fake.date_time_this_year()
22
23             #la data di fine della chiamata equivale alla data
di inizio della chiamata + la durata(in secondi)
24             end_date = start_date + timedelta(seconds=duration)
25
26             writer.writerow([people[caller], people[called],
int(round(datetime.timestamp(start_date))), int(round(
datetime.timestamp(end_date))), duration, cell_site])
27             file.close()
28
29

```

Listing 4: gen\_calls()

## 5.4 Importazione dei dati

I record vengono letti dai file csv esportati in precedenza. Tramite la funzione `load_data()` in `mongo_manager.py` è possibile caricare i dati su MongoDB, mentre Neo4j importa automaticamente i file csv che sono presenti nella directory `/relate-data/dbmss/<database id>/import` di Neo4j.

```
1 def load_data(handle: MongoClient):
2     db = handle.progettodb2
3
4     with open(data_path('cells.csv'), "r") as cfile:
5         db.cells.insert_many(to_dict(list(DictReader(cfile))))
6
7         cfile.close()
8
9     with open(data_path('people.csv'), "r") as pfile:
10        db.people.create_index("number", unique=True)
11        db.people.insert_many(to_dict(list(DictReader(pfile))))
12
13        pfile.close()
14
15    with open(data_path('calls.csv'), "r") as cfile:
16        db.calls.insert_many(to_dict(list(DictReader(cfile))))
17
18        cfile.close()
19
20
```

Listing 5: `load_data()`

## 6 Risultati degli esperimenti

Le query presentate di seguito sono definite nei file sorgenti `mongo_queries.py` e `neo_queries.py`. I valori `gt_start` e `lt_start` definiscono rispettivamente la data massima e la data minima (in formato *epoch*) delle chiamate da prendere in considerazione. Il valore `duration` definisce la durata massima delle chiamate.

È stato utilizzato LibreOffice Calc per la manipolazione dei dati e la generazione dei grafici a seguire.

### 6.1 2 WHERE

Viene effettuata una selezione condizionata con due clausole `WHERE`

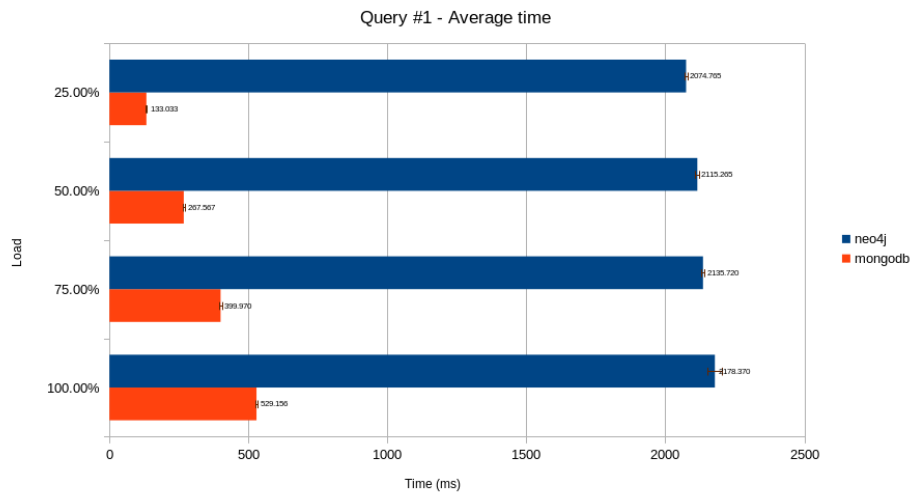
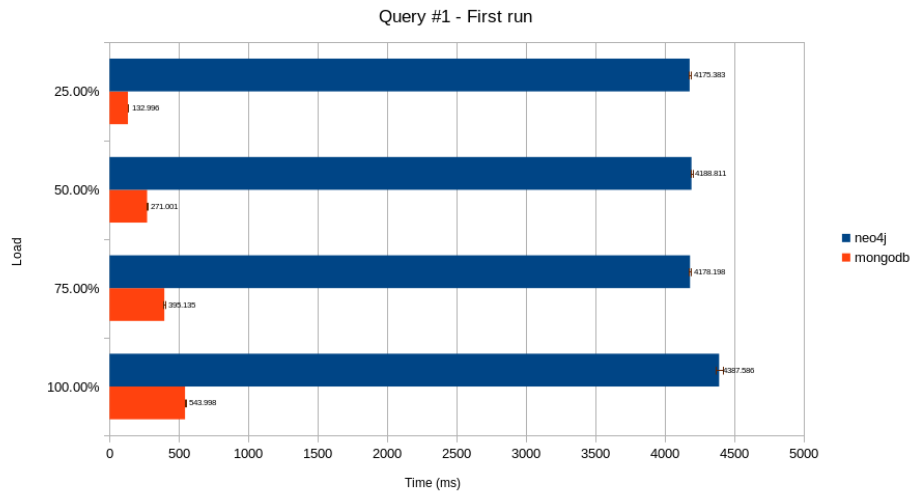
```
1 "MATCH (c:call) WHERE c.start_date >"
2 + str(gt_start)
3 + " AND c.start_date < "
4 + str(lt_start)
5 + " RETURN c"
6
```

Listing 6: Neo4j

```
1 {
2   "$match": {
3     "start_date": {
4       "$gte": gt_start,
5       "$lt": lt_start
6     }
7   }
8 }
9
```

Listing 7: MongoDB

Query #1	Neo4j				MongoDB			
	100%	75%	50%	25%	100%	75%	50%	25%
First run	4387.586	4178.198	4188.811	4175.383	543.998	395.135	271.001	132.996
Media	2178.370	2135.720	2115.265	2074.765	529.156	399.970	267.567	133.033
Dev. Std.	26.720	5.894	7.725	7.222	3.942	5.207	2.956	1.497
95% Conf	26.271	5.795	7.595	7.100	3.876	5.120	2.906	1.472



## 6.2 3 WHERE

Viene effettuata una selezione condizionata con tre clausole **WHERE** e un **JOIN**

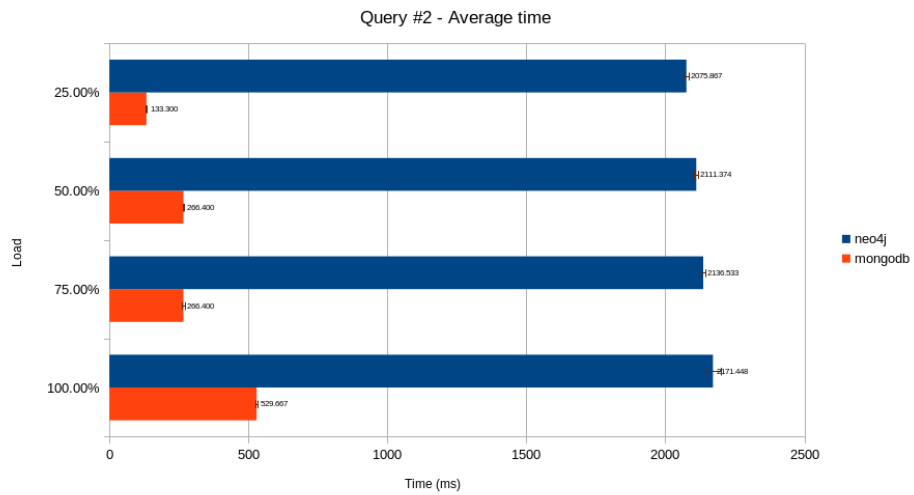
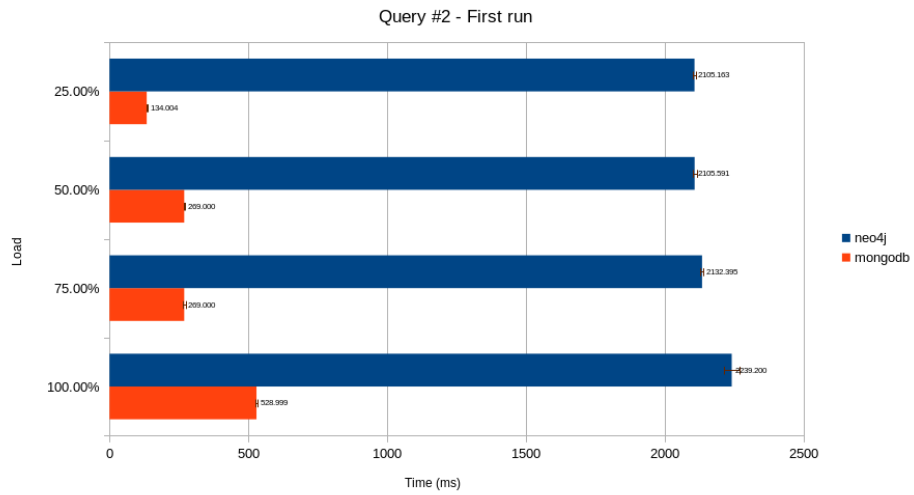
```
1 "MATCH (c:call) WHERE c.start_date > "  
2 + str(gt_start)  
3 + " AND c.start_date < "  
4 + str(lt_start)  
5 + " AND c.duration >= "  
6 + str(duration)  
7 + " RETURN c"  
8  
9
```

Listing 8: Neo4j

```
1 {  
2   "$match": {  
3     "start_date": {  
4       "$gte": gt_start,  
5       "$lt": lt_start  
6     },  
7     "duration": {  
8       "$gte": duration  
9     }  
10  }  
11 }  
12
```

Listing 9: MongoDB

Query #2	Neo4j				MongoDB			
	100%	75%	50%	25%	100%	75%	50%	25%
First run	2239.200	2132.395	2105.591	2105.163	528.999	269.000	269.000	134.004
Media	2171.448	2136.533	2111.374	2075.867	529.667	266.400	266.400	133.300
Dev. Std.	12.528	7.978	7.654	7.226	4.045	3.125	3.125	0.916
95% Conf	12.317	7.844	7.525	7.105	3.977	3.073	3.073	0.900





### 6.3 3 WHERE 1 JOIN

Viene effettuata una selezione condizionata con tre clausole WHERE e un JOIN

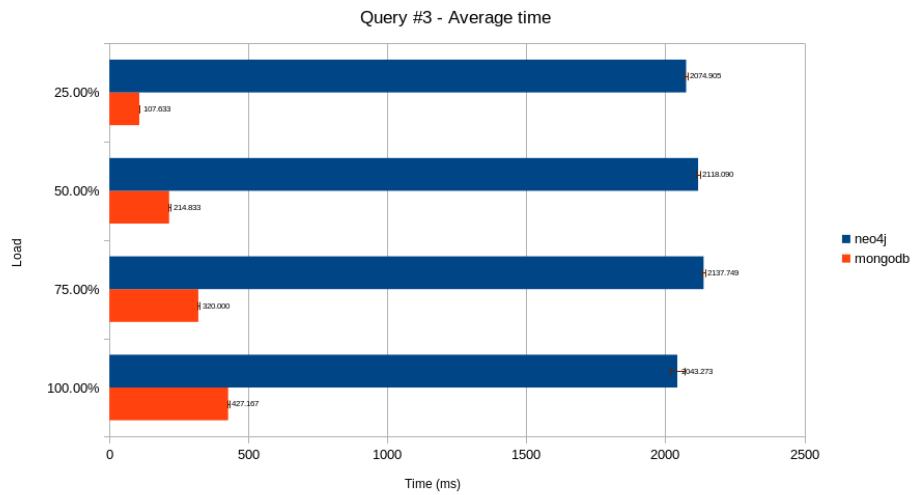
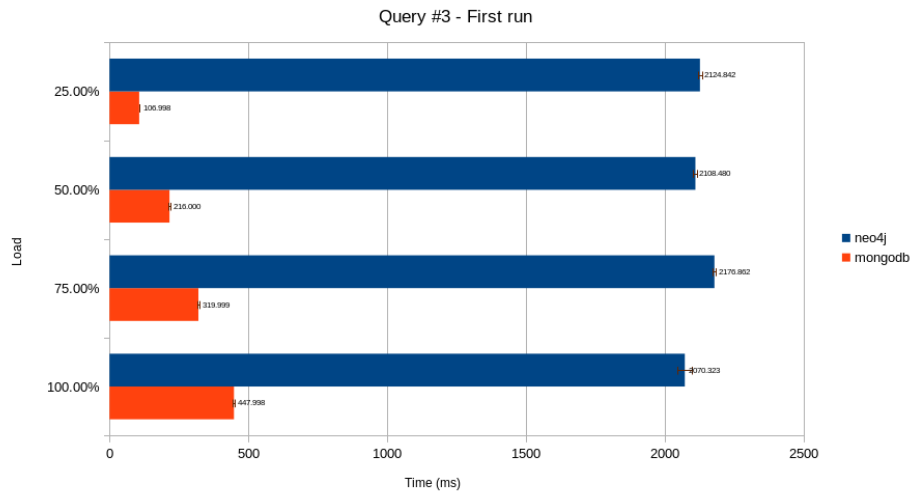
```
1 "MATCH (p:person)-[r:made_call]->(c:call) WHERE c.start_date >"
2 + str(gt_start)
3 + " AND c.start_date < "
4 + str(lt_start)
5 + " AND c.duration >= "
6 + str(duration)
7 + " RETURN c, r, p"
8
```

Listing 10: Neo4j

```
1 {
2   "$match": {
3     "start_date": {
4       "$gte": gt_start,
5       "$lt": lt_start
6     },
7     "duration": {
8       "$gte": duration
9     }
10  },
11 },
12 {
13   "$lookup": {
14     "from": "people",
15     "localField": "calling_number",
16     "foreignField": "number",
17     "as": "caller"
18   }
19 }
20
```

Listing 11: MongoDB

Query #3	Neo4j				MongoDB			
	100%	75%	50%	25%	100%	75%	50%	25%
First run	2070.323	2176.862	2108.480	2124.842	447.998	319.999	216.000	106.998
Media	2043.273	2137.749	2118.090	2074.905	427.167	320.000	214.833	107.633
Dev. Std.	5.924	14.014	7.839	8.907	1.511	1.598	2.069	0.928
95% Conf	5.824	13.778	7.707	8.758	1.486	1.571	2.035	0.912



## 6.4 3 WHERE 2 JOIN

Viene effettuata una selezione condizionata con tre clausole `WHERE` e due `JOIN`

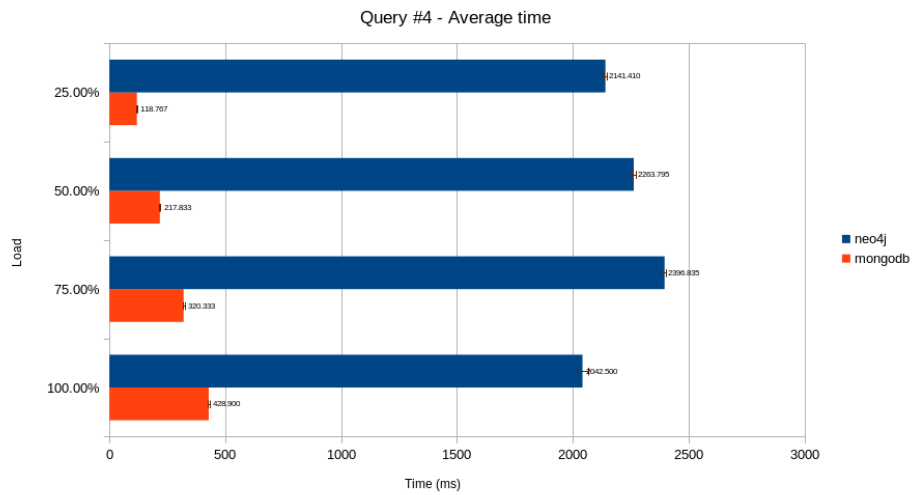
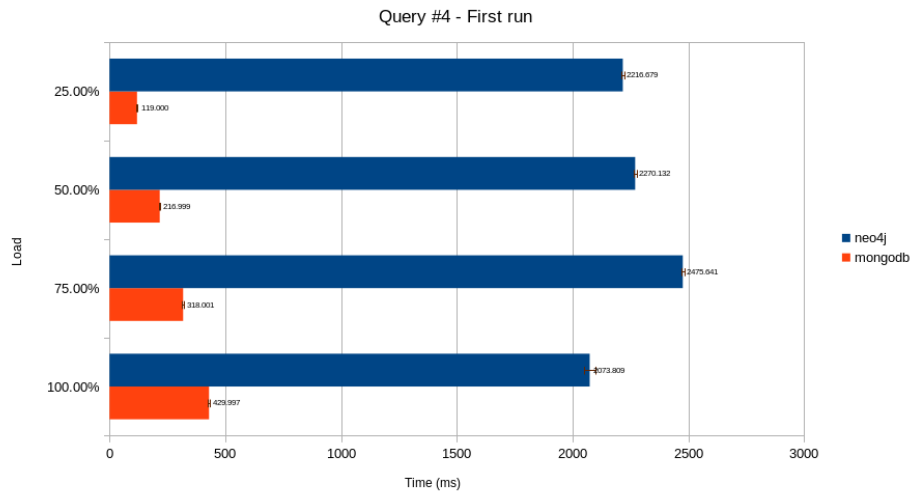
```
1 "MATCH(p:person)-[r1:made_call]->(c:call)-[r2:located_in]->(ce:
   cell) WHERE c.start_date >"
2 + str(gt_start)
3 + " AND c.start_date < "
4 + str(lt_start)
5 + " AND c.duration >= "
6 + str(duration)
7 + " RETURN c,p,r1,r2,ce"
8
```

Listing 12: Neo4j

```
1 {
2   "$match": {
3     "start_date": {
4       "$gte": gt_start,
5       "$lt": lt_start
6     },
7     "duration": {
8       "$gte": duration
9     }
10  }
11 },
12 {
13   "$lookup": {
14     "from": "people",
15     "localField": "calling_number",
16     "foreignField": "number",
17     "as": "caller"
18   }
19 },
20 {
21   "$lookup": {
22     "from": "cells",
23     "localField": "cell_site",
24     "foreignField": "cell_site",
25     "as": "cell"
26   }
27 }
28
```

Listing 13: MongoDB

Query #4	Neo4j				MongoDB			
	100%	75%	50%	25%	100%	75%	50%	25%
First run	2073.809	2475.641	2270.132	2216.679	429.997	318.001	216.999	119.000
Media	2042.500	2396.835	2263.795	2141.410	428.900	320.333	217.833	118.767
Dev. Std.	7.759	14.194	10.531	10.238	3.166	2.106	1.488	0.858
95% Conf	7.628	13.955	10.354	10.066	3.113	2.071	1.463	0.844



## 7 Conclusioni

I risultati dimostrano che MongoDB presenta tempi di esecuzione più brevi rispetto a neo4j per tutte e quattro le query. Man mano che la complessità computazionale delle query aumenta, i tempi di esecuzione di neo4j aumentano in modo significativo, a differenza dei tempi di MongoDB che rimangono relativamente costanti. I dati mostrano che, aumentando il carico, i tempi di esecuzione di entrambi i DBMS aumentano, ma in misura maggiore per neo4j. Pertanto, MongoDB sembra essere più efficiente per queste query rispetto a neo4j. Inoltre possiamo esumere dai risultati che la deviazione standard per i tempi di esecuzione di MongoDB è generalmente più bassa rispetto a quella di neo4j, indicando che i tempi di esecuzione di MongoDB sono più prevedibili e stabili. La differenza tra i tempi di esecuzione della prima esecuzione e la media è più piccola per MongoDB rispetto a neo4j, il che indica che MongoDB è più rapido nel raggiungere una performance stabile rispetto a neo4j. Oltre ciò, si può dedurre in base ai dati che MongoDB ha una maggiore efficienza nella gestione dei carichi più leggeri rispetto a neo4j. In conclusione, questi risultati indicano che MongoDB è più efficiente per le query considerate rispetto a neo4j, sia in termini di tempi di esecuzione sia di stabilità e prevedibilità dei tempi di esecuzione.

## Riferimenti bibliografici

- [1] <https://linkurious.com/blog/how-to-use-phone-calls-and-network-analysis-to-identify-criminals/>