UNIVERSITÀ DEGLI STUDI DI MESSINA

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

Corso di Laurea Triennale in Informatica

MongoDB vs Neo4j

Docente:
Prof. Massimo Villari

Studenti:
Claudio Anchesi, 513605
Danny De Novi, 517611

ANNO ACCADEMICO 2021/2022

Indice

1 Abstract **3**

2 Introduzione **4**

3 Database **4**
3.1 MongoDB 4
3.2 Neo4j 5

4 Progettazione **6**
4.1 Caso di studio 6
4.1.1 MongoDB 7
4.1.2 Neo4j 8

5 Implementazione **9**
5.1 Generazione 9
5.2 Inserimento 9

6 Test **11**
6.1 Query n°1 13
6.2 Query n°2 15
6.3 Query n°3 17
6.4 Query n°4 19

7 Osservazioni e conclusioni **21**

1 Abstract

Il fine ultimo di questo scritto è quello di riportare i risultati prestazionali di due tra i più noti *DataBase Management Systems* (DBMS) NoSQL degli ultimi tempi; la scelta è ricaduta sul *document-oriented* MongoDB e sul *graph database* Neo4j. A tal scopo ci si è avvalsi di un caso di studio di sfondo per la definizione di un *data model* per popolare i suddetti DBMS, con le dovute differenze, e per poterci così eseguire delle *query* note di complessità computazionale crescente ed effettuare una stima delle prestazioni tramite la misurazione del tempo di esecuzione, o più propriamente di risposta.

Per l'utilizzo dei DBMS sono stati istanziati dei *containers* tramite *Docker* su una macchina iMac, con processore *Apple M1*, 4 cores e 16 GB di RAM dedicati. L'utilizzo di una macchina Apple comporta la presenza di un macchina virtuale nativa di Docker per l'istanziazione di containers; ciononostante, le prestazioni non verranno inficiate in maniera significativa.

Generazione e inserimento dei dati sono stati automatizzati tramite degli script in *Python*, con l'utilizzo di driver appositi; lo stesso dicasi per l'interrogazione dei database. I tempi misurati in millisecondi sono stati poi scritti su file csv, importati su una cartella di lavoro Excel e opportunamente organizzati in tabelle e grafici.

Le conclusioni, portano ad un esito non inattesto, ossia alla risultante miglior ottimizzazione di uno rispetto all'altro in casi differenti: in particolare, è evidente la miglior prestazione di Neo4j quando si effettuano query coinvolgenti più entità e relazioni rispetto alla più efficiente responsività di MongoDB sulla selezione condizionata su singola entità.

I codici e i risultati sono visionabili su GitHub[1].

[1] Link alla repository

2 Introduzione

I database di tipo NoSQL nacquero per la necessità di immagazzinare, organizzare e interagire con grandi quantità dati senza che vi sia uno schema rigido, com'è invece per i database relazionali (RDBMS, *Relational DBMS*), e in tempi accettabili per le varie necessità dell'utenza. Ciò portò allo sviluppo di nuovi paradigmi, come il *key-value*, *a grafo* o *column-based*, ognuno con caratteristiche e implementazioni differenti, prestazioni migliori o peggiori dipendentemente da casi e circostanze. Esempi per ognuno delle categorie sopracitate sono rispettivamente MongoDB, etichettato più precisamente come document-oriented, Neo4j e Cassandra.

Tutti i DBMS rispettano il *teorema di Brewer*, o *Teorema CAP*, per cui un sistema informatico distribuito non può soddisfare tutte e tre le seguenti garanzie:

- Consistenza (*Consistency*);
- Disponibilitá (*Availability*);
- Partizionamento (*Partitioning*).

Difatti, esempi come i database key-value rispettano la sigla CP, consistenza e partizionamento, mentre i database a grafo rispettano la sigla CA, consistenza e disponibilità (le stesse caratteristiche *ACID* dei database relazionali). Queste sono proprio le caratteristiche dei DBMS trattati in questo scritto; ciononostante, qualsiasi prestazione è misurata su una distribuzione su singolo nodo, senza perciò aver a che fare in alcun modo con il partizionamento e agevolando la disponibilità.

3 Database

Come già presentati, i due DBMS scelti per questa comparazione sono MongoDB e Neo4j, sia per via dei loro distanti paradigmi sia per la loro semplicità di implementazione, *deployment* tramite containers e interfacciamento con l'utente via riga di comando e no.

3.1 MongoDB

MongoDB è un DBMS di tipo NoSQL *document-oriented* e *schema-free*: la prima etichetta indica che la più piccola porzione di informazione ricavabile dai dati è chiamato *documento*; la seconda evidenzia come questi ultimi non necessitino di alcun tipo di schema rigido o semi-rigido.

Difatti, un documento di MongoDB è una porzione di dati archiviati come una coppia *key-value*, così da permettere un accesso al dato da estrapolare diretto, con complessità computazionale `O(1)`, e senza alcun tipo di vincolo di coerenza di forma o schema tra documenti della stessa collezione (raccolta di documenti, analogo alle tabelle dei RDBMS). Inoltre, il formato dei documenti, detto *BSON* (*Binary Serialized dOcument Notation*), è derivato dal formato testuale *JSON* (*JavaScript Object Notation*), e ne rappresenta il corrispettivo serializzato come dato binario. Per questa sua tipologia, un documento in formato BSON gode di una più veloce elaborazione.

Una caratteristica rilevante di MongoDB, ma non sfruttata in questo studio, è la possibilità di dividere fisicamente i dati su più nodi distinti (*sharding*), sui quali

eseguono dei *replica-set*, ossia più repliche del demone di MongoDB; ciò permette una elaborazione parallela delle operazioni e *fault-tolerance* alla cadura di uno dei demoni. Tutto ciò è possibile per via della gestione ad un più alto livello dei dati secondo i documenti e le collezioni di cui sopra.

Un database document-oriented come MongoDB eccelle nella ricerca dei dati per via delle coppie key-value e dei calcoli paralleli sui più nodi del database distribuito, peccando però dell'impossibilità di *JOIN* ottimizzati tra più documenti o collezioni, ad oggi possibile grazie alla funzionalità *aggregate* aggiunta solo in versioni più recenti del DBMS.

MongoDB è stato containerizzato tramite la funzione *compose* di Docker, assieme a Neo4j, con un file *yaml*; di seguito la sezione di codice rilevante:

```
mongodb:
    container_name: "mongodb"
    image: "mongo"
    ports:
        - "27017:27017"
```

Con ciò si è istanziato un container partendo dall'immagine *mongo*[2] ufficiale presente nell'Hub pubblico di Docker; si è anche pubblicata la porta *27017*, di default quella attraverso la quale il DBMS resta in attesa di connessione, cosicché sia possibile permettere un'interazione con il database, nel nostro caso da parte degli script Python sopracitati. Non è stato utilizzato alcun volume per mantenere i dati persistenti in quanto non necessario ai fini dello studio.

3.2 Neo4j

Neo4j è un DBMS NoSQL a grafo, il che indica che i dati sono da esso organizzati secondo l'omonima struttura dati. Difatti, la minima porzione di informazione ricavabile è un nodo del grafo, i cui dati sono le proprietà del nodo; analogamente le relazioni sono effettivamente gli archi del grafo, sempre orientati, e anch'esse con proprietà, ove necessario.

Questa struttura dati permette di sfruttare tutti i vantaggi di un grafo, come gli attraversamenti e le ricerche, effettuabili sull'intero grafo o su sottografi, combinati agli accessi alle proprietà di nodi e relazioni, implementate come coppie *key-value*, con accesso diretto.

Inoltre, anche i database a grafi sono *schema-free*, in quanto ogni nodo può possedere proprietà differenti da altri nodi della stessa categoria (*label* secondo il linguaggio di Neo4j).

I database a grafi sono i più versatili e godono di essere, generalmente, i più prestazionali quando vi è l'esigenza di tener conto di molte relazioni tra varie entità, inciampando però nell'impossibilità di partizionamento del grafo su più nodi.

Neo4j è stato containerizzato tramite il medesimo file *yaml* del precedente DBMS, con la sezione di codice che segue:

[2] Docker Hub - Mongo

```
neo4j:
    container_name: "neo4j"
    image: "neo4j"
    ports:
        - "7474:7474"
        - "7687:7687"
    environment:
        - "NEO4J_AUTH=none"
    volumes:
        - "<path_locale>:/var/lib/neo4j/import"
```

Il container è istanziato a partire dall'immagine ufficiale *neo4j*[3] e sono state pubblicate due porte: la *7474*, attraverso la quale viene messa a disposizione un'interfaccia web per l'interazione con il database, e la *7687*, attraverso la quale il DBMS resta in ascolto in attesa di connessione esterna per l'interazione con il database. La variabile d'ambiente `NEO4J_AUTH=none` è settata per evitare che Neo4j, come di default al primo accesso, richieda una modifica obbligatoria della password. Inoltre, in questo caso è stato necessario l'utilizzo di un volume, o più precisamente il *binding* del filesystem dell'host con quello del container, per permettere la popolazione del database tramite file esterni; come sopra, ciò non è necessario ai fini dello studio.

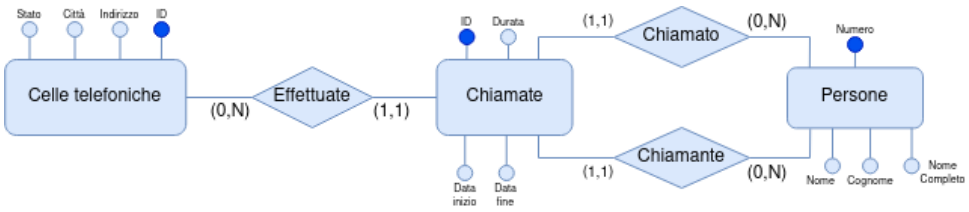
4 Progettazione

4.1 Caso di studio

La tematica di sfondo affrontata in questo studio tratta della ricerca di criminali tramite delle operazioni di incrociamento di chiamate, date e luoghi di interesse, al fine di trovare contatti che questi possono avere o aver avuto, o quantomeno di restringere il campo di ricerca.

Un esempio potrebbe essere la ricerca di un gruppo di criminali che opera in un furto: si potrebbe immaginare una ricerca tra le chiamate effettuate nelle celle telefoniche interessate alle zone limitrofe al luogo del reato, in determinate date, per risalire così possibilmente ad una cerchia ristretta di persone più spesso presenti in quelle circostanze come chiamanti e/o chiamati.

Si può supporre che, dati i permessi legali dei gestori telefonici di memorizzare alcuni dati inerenti le chiamate, si possa costruire un database con uno schema concettuale corrispondentemente a quello che segue:



[3]Docker Hub - Neo4j

Si evincono le tre entità del database:

- *Persone*, caratterizzate da un *numero* di telefono, univoco, un nome, un cognome, e il nome completo;
- *Chiamate*, caratterizzate da un ID univoco, due date, di inizio e fine chiamata, dove per data si intende la codifica *UNIX Epoch* dell'istante corrente, una durata, le due persone comunicanti e la cella telefonica nella quale è effettuata;
- *Celle telefoniche*, caratterizzate da un ID univoco, uno Stato, banalmente l'Italia nel nostro esempio, una città e una via dove queste sono localizzate.

Le relazioni che si osservano sono le seguenti:

- *Chiamante/Chiamato*, che legano le persone alle chiamate da loro effettuate/ ricevute;
- *Effettuate*, che lega le chiamate alla cella telefonica nella quale sono effettuate.

4.1.1 MongoDB

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

- *people*:
  - *Number*, numero di telefono;
  - *FullName*, nome completo;
  - *FirstName*, nome;
  - *LastName*, cognome;
- *calls*:
  - *CallingNbr*, numero del chiamante;
  - *CalledNbr*, numero del chiamato;
  - *StartDate*, momento di inizio chiamata;
  - *EndDate*, momento di fine chiamata;
  - *Duration*, durata;
  - *CellSite*, ID della cella;
- *cells*:
  - *CellSite*, ID della cella;
  - *City*, Città;
  - *State*, Stato (Italia);
  - *Address*, indirizzo stradale;

Come ID della collezione *calls* ci si è avvalsi dell'*Document ID* che MongoDB assegna automaticamente al momento dell'inserimento del documento.

Le relazioni, invece, non sono esplicitabili con questo DBMS, ma, qualora ve ne sia il bisogno, è necessario che sia il progettista a ideare il database e le interfacce con esso in maniera tale che vi sia modo di effettuare delle query *aggregate* per concretizzare le relazioni concettuali.

4.1.2 Neo4j

In Neo4j, le entità, diventate *labels*, sono le seguenti tre, con relative proprietà dei nodi:

- *person*:
  - *Number*, numero di telefono;
  - *FullName*, nome completo;
  - *FirstName*, nome;
  - *LastName*, cognome;
- *call*:
  - *CallingNbr*, numero del chiamante;
  - *CalledNbr*, numero del chiamato;
  - *StartDate*, momento di inizio chiamata;
  - *EndDate*, momento di fine chiamata;
  - *Duration*, durata;
  - *CellSite*, ID della cella;
- *cell*:
  - *CellSite*, ID della cella;
  - *City*, Città;
  - *State*, Stato (Italia);
  - *Address*, indirizzo stradale.

Analogamente a MongoDB, gli ID dei nodi *call* sono rappresentati dagli *id* che Neo4j assegna alla creazione del nodo.

Le relazioni, a differenza del precedente DBMS, sono esplicitabili come archi del grafo, e sono le seguenti tre:

- *is_ calling*, diretto da un nodo *person* verso un nodo *call*;
- *is_ called*, diretto da un nodo *call* verso un nodo *person*;
- *is_ done*, diretto da un nodo *call* verso un nodo *cell*.

5 Implementazione

5.1 Generazione

Per l'ottenimento dei dati popolanti i database, si è scritto uno script in Python che generasse quanto necessario pseudo-casualmente tramite le API del modulo open-source *Faker*.

Lo script presenta tre funzioni separate per la generazione dei dati delle tre entità previste, alla cui fine di ciascuno si effettua una scrittura su file csv (*Comma Separated Values*), creato, o sovrascritto se esistente, runtime.

Le tre funzioni vengono invocate su due *threads*, in maniera tale da eseguire concorrentemente la generazione delle persone e delle celle telefoniche, per poi attenderne la fine e richiamare la generazione delle chiamate. Tale implementazione della generazione dei dati è stata scelta per ottenere un maggior *throughput* e minor tempo di esecuzione, nonostante la necessità di eseguire separatamente l'ultima delle tre funzioni per via della dipendenza dell'entità *chiamate* dalle altre due, come si evince dal modello concettuale di cui sopra.

La quantità di dati generata, come da traccia, è pari al 25%, 50%, 75% e 100% di un massimo deciso, esprimibile al codice come parametro al momento di esecuzione da riga di comando; di seguito la tabella dei dataset:

	25%	50%	75%	100%
Persone	5,000	10,000	15,000	20,000
Celle Tel.	4,000	8,000	12,000	16,000
Chiamate	250,000	500,000	750,000	1,000,000
Totale	**259,000**	**518,000**	**777,000**	**1,036,000**

Di seguito una vista della sezione dedicata al multithread, tramite modulo *threding*:

```
thread = Thread(target=gen_cells, args=(num_cells,))

thread.start()
gen_people(num_people)
thread.join()

gen_calls(num_calls, num_people, start_date, end_date,
                                          range_call)
```

Il codice è visionabile nella repository di GitHub referenziata nella sezione 1.

5.2 Inserimento

Anche per la popolazione dei database è stato utilizzato uno script Python, con implementazione *multithread*.

Lo script è tale da accettare come parametro dei flag per indicare su quale database inserire i csv generati, con possibilità di popolazione di entrambi i database

concorrentemente tramite due threads. Le funzioni utilizzate sono fornite dai driver open-source *pymongo* e *neo4j* per Python.

La connessione ai due database è effettuata tramite valori hardcoded di indirizzo IP e porta (*127.0.0.1*, *27017* per MongoDB e *7687* per Neo4j) ad hoc per la macchina di sviluppo e testing, considerando i due DBMS distribuiti tramite *Docker images*, come descritto nella sezione 3.

Di seguito una vista della sezione dedicata al multithread, tramite modulo *threding*:

```
if mongo:
    if not neo:
        insert_mongo(debug=debug)
    else:
        thread_mongo = Thread(target=insert_mongo,
                    kwargs={'debug': debug})
        thread_mongo.start()

if neo:
    insert_neo(debug=debug)

if 'thread_mongo' in locals():
    thread_mongo.join()
```

Il codice è visionabile nella repository di GitHub referenziata nella sezione 1.

6 Test

Le queries che sono state effettuate sui due database sono 4 e seguono il seguente schema, per semplicità descritto con terminologia SQL, caratterizzato da una difficoltà computazionale crescente:

1) 2 WHERE;

2) 2 WHERE 1 JOIN;

3) 2 WHERE 2 JOIN;

4) 3 WHERE 2 JOIN.

Ciò è scelto per mostrare come i due DBMS si comportino al crescere della complessità delle interrogazioni e delle dimensioni dei dataset (sezione 5.1); l'obiettivo è quello di stabilire quali siano i punti di forza e debolezza dei due software all'aumentare delle due variabili che caratterizzano un database: quantità di dati da gestire e difficoltà computazionale delle interrogazioni.

Le queries non sono effettuate tramite le interfacce CLI built-in dei due DBMS (*mongosh* e *cypher-shell* per rispettivamente MongoDB e Neo4j), bensì tramite degli script Python che implementano gli stessi moduli citati nella sezione 5.2; gli script inoltre prevedono anche una misurazione dei tempi d'esecuzione automatica tramite *flag* all'esecuzione del codice da terminale. I risultati delle misurazioni sono poi scritti su file csv nella seguente maniera:

- Tempo alla prima esecuzione;
- Tempi delle 30 esecuzioni successive;
- Media delle 30 esecuzioni misurate;
- Deviazione standard del campione misurato.

Dopo la prima esecuzione effettuata in un ambiente appena configurato, le 30 misurazioni mediate sono state effettuate per mettere a confronto le prestazioni dei meccanismi di *caching* dei due DBMS.

Questi dati sono successivamente stati importati manualmente su Microsoft Excel al fine di elaborare due istogrammi per query che meglio mostrino quanto evincentesi dai risultati: uno che confronti il tempo della prima esecuzione della query in esame, e un secondo che metta a paragone le medie delle 30 misurazioni successive di cui sopra, con annesso intervallo di confidenza al 95% dello studio statistico inferenziale effettuato. Tutti i grafici mostrano i dati della query corrente con tutti e 4 i dataset considerati.

Le funzioni scritte per l'esecuzione delle query sui due database sono le seguenti (si consiglia la visione completa sulla repository indicata nella sezione 1):

```python
# MongoDB
def exec_query(query, client=connect_mongo(), t=False):
    start = timestamp()
    client.test.calls.aggregate(query)

    return timestamp(start) if t else 0

# Neo4j
def exec_query(query, client=connect_neo(), t=False):
    start = timestamp()
    with client.session() as db:
        db.run(query)

    return timestamp(start) if t else 0
```

La prima funzione, che si occupa dell'esecuzione delle query su MongoDB, opera tramite il metodo *aggregate()* fornito dai moduli sopracitati, il quale corrisponde all'omonima funzione built-in del DBMS; questa necessita di una lista di dizionari (strutture dati native di Python), i quali corrispondono alle varie operazioni che devono essere effettuate sui dati, secondo un meccanismo di *pipeline*. Si è utilizzata questa funzione in tutte le query, nonostante non fosse sempre necessaria e fosse sostituibile con la più consueta *find()*, per mantenere una coerenza fra tutte; inoltre, questa è l'unica modalità di interrogazione ove presente un'operazione JOIN (secondo un'analogia con il linguaggio SQL), introdotta nella versione 3.2 del DBMS.

La seconda funzione, che si occupa dell'esecuzione delle query su Neo4j, utilizza il metodo *run()* fornito dai moduli di cui sopra, il quale opera semplicemente tramite una stringa contenente l'interrogazione in esame. Questo è valido per una qualsiasi esecuzione di query su Neo4j, nonostante vi siano altri metodi come *write_transaction()*, che però operano secondo altre politiche rispetto alla mera esecuzione.

Entrambe le funzioni ritornano, se opportunamente indicato tramite parametro, il tempo di esecuzione della query, altrimenti 0 per indicare una corretta esecuzione.

6.1 Query n°1

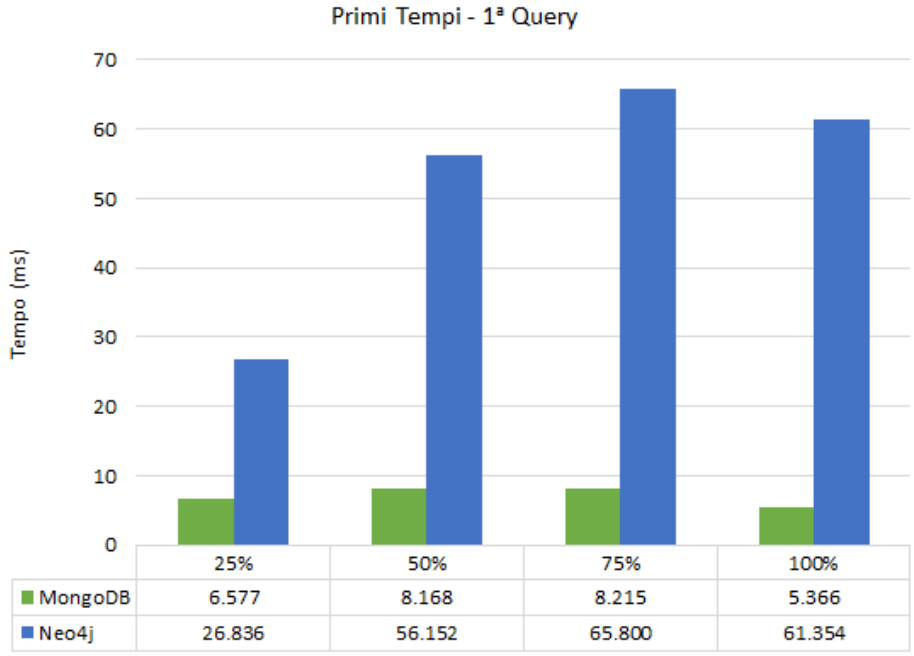
La prima query effettuata presenta una selezione condizionata con 2 WHERE su singola entità; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
# WHERE StartDate >= 1580083200 AND EndDate < 1580256000

# MongoDB
start_search = int(mktime(datetime(2020, 1, 27).timetuple()))
end_search = int(mktime(datetime(2020, 1, 29).timetuple()))
query = [
            {"$match": {"StartDate": {"$gte": start_search,
                                      "$lt": end_search}}}
        ]

# Neo4j
query = "MATCH (c:call) \
         WHERE c.StartDate >= 1580083200 \
             AND c.StartDate < 1580256000 \
         RETURN c"
```

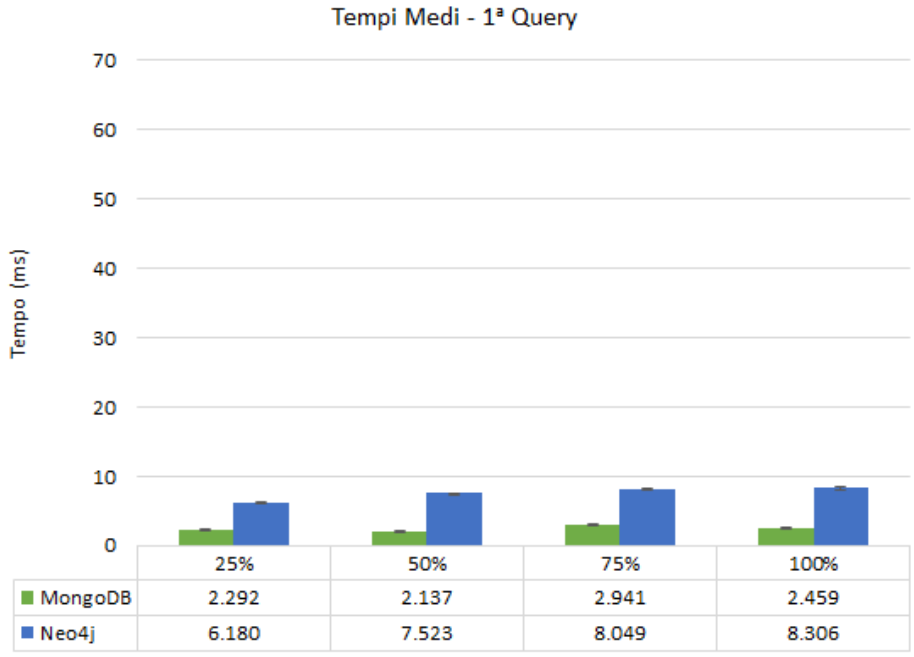
I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	Primi Tempi	
	MongoDB	Neo4j
25%	6.577	26.836
50%	8.168	56.152
75%	8.215	65.800
100%	5.366	61.354

	Tempi Medi - 30 rip.	
	MongoDB	Neo4j
25%	2.292	6.180
50%	2.137	7.523
75%	2.941	8.049
100%	2.459	8.306



	25%	50%	75%	100%
MongoDB	6.577	8.168	8.215	5.366
Neo4j	26.836	56.152	65.800	61.354



	25%	50%	75%	100%
MongoDB	2.292	2.137	2.941	2.459
Neo4j	6.180	7.523	8.049	8.306

6.2 Query n°2

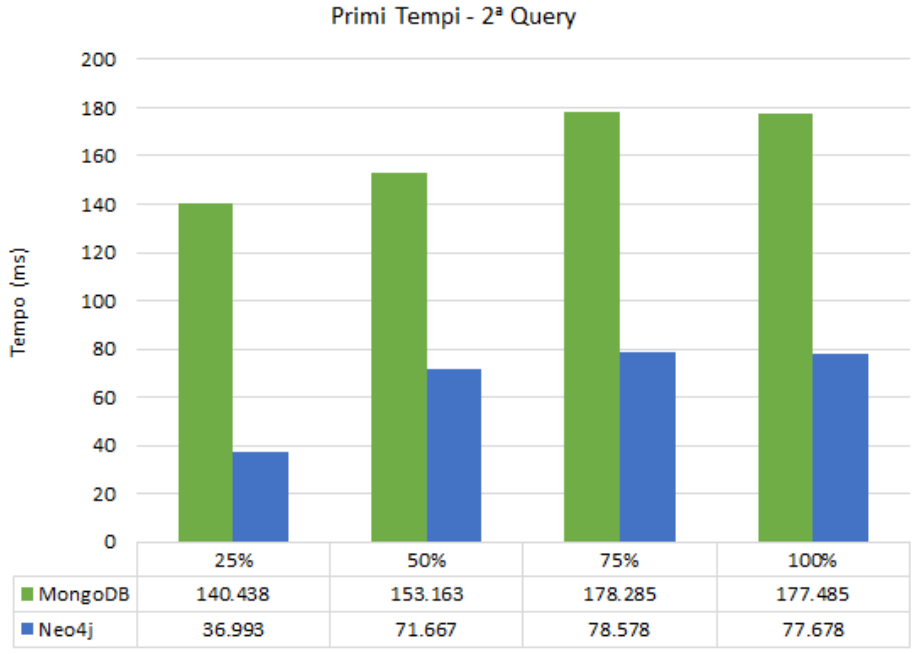
La seconda query effettuata presenta una selezione condizionata con 2 WHERE su due entità, unite tramite 1 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
#     JOIN people
#         ON calls.CallingNbr = people.Number
# WHERE StartDate >= 1580083200 AND EndDate < 1580256000

# MongoDB
query =[
            {"$match": {"StartDate": {"$gte": start_search,
                                      "$lt": end_search}}},
            {"$lookup": {"from": "people",
                         "localField": "CallingNbr",
                         "foreignField": "Number",
                         "as": "Calling"}}
        ]

# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
         WHERE c.StartDate >= 1580083200 \
             AND c.StartDate < 1580256000 \
         RETURN c"
```

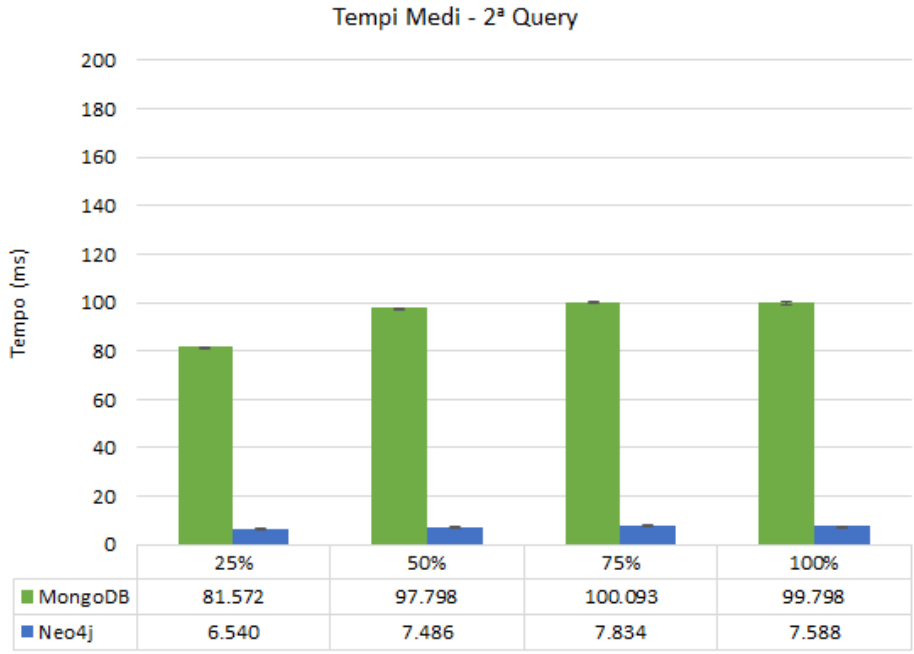
I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	Primi Tempi	
	MongoDB	Neo4j
25%	140.438	36.993
50%	153.163	71.667
75%	178.285	78.578
100%	177.485	77.678

	Tempi Medi - 30 rip.	
	MongoDB	Neo4j
25%	81.572	6.540
50%	97.798	7.486
75%	100.093	7.834
100%	99.798	7.588



	25%	50%	75%	100%
MongoDB	140.438	153.163	178.285	177.485
Neo4j	36.993	71.667	78.578	77.678



	25%	50%	75%	100%
MongoDB	81.572	97.798	100.093	99.798
Neo4j	6.540	7.486	7.834	7.588

6.3 Query n°3

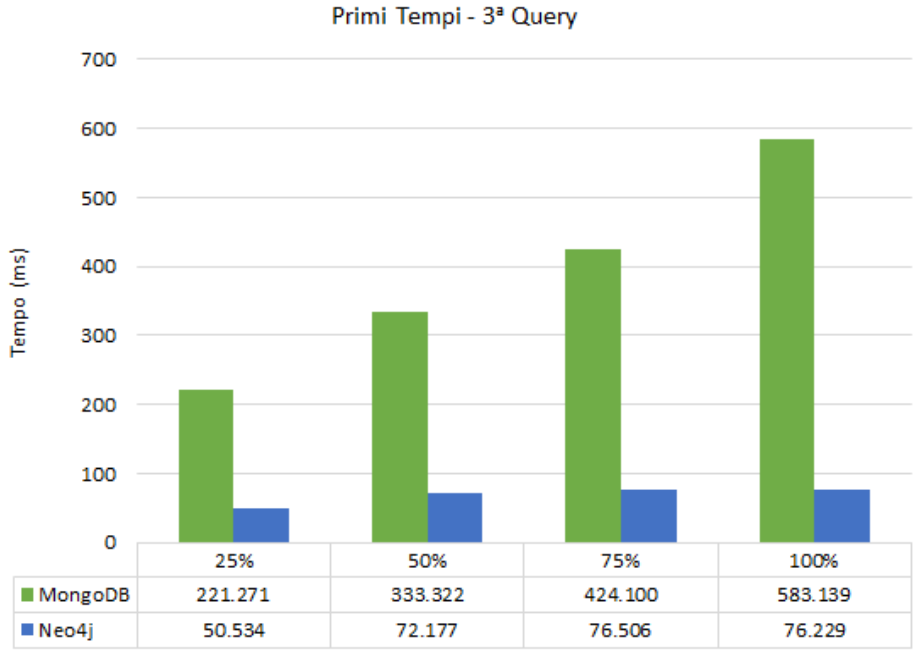
La terza query effettuata presenta una selezione condizionata con 2 WHERE su tre entità, unite tramite 2 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
#     JOIN people
#         ON calls.CallingNbr = people.Number
#     JOIN cells
#         ON calls.CellSite = cells.CellSite
# WHERE StartDate >= 1580083200 AND EndDate < 1580256000

# MongoDB
query = [
            {"$match": {"StartDate": {"$gte": start_search,
                                      "$lt": end_search}}},
            {"$lookup": {"from": "people",
                         "localField": "CallingNbr",
                         "foreignField": "Number",
                         "as": "Calling"}},
            {"$lookup": {"from": "cells",
                         "localField": "CellSite",
                         "foreignField": "CellSite",
                         "as": "Cell"}}
        ]

# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
                          -[r2:is_done]->(ce:cell) \
         WHERE c.StartDate >= 1580083200 \
             AND c.StartDate < 1580256000 \
         RETURN p1, r1, c, r2, ce"
```

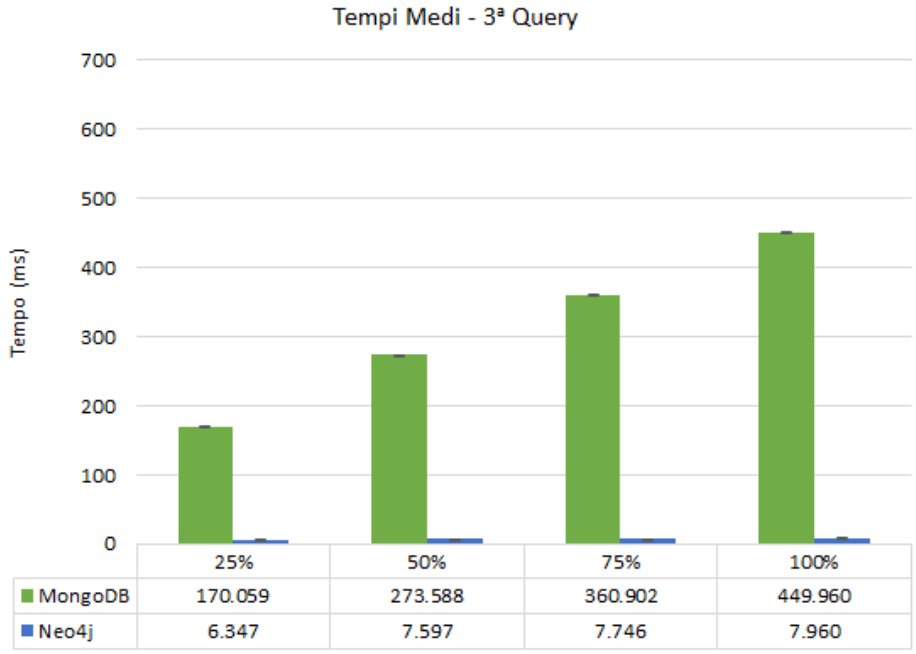
I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	Primi Tempi	
	MongoDB	Neo4j
25%	221.271	50.534
50%	333.322	72.177
75%	424.100	76.506
100%	583.139	76.229

	Tempi Medi - 30 rip.	
	MongoDB	Neo4j
25%	170.059	6.347
50%	273.588	7.597
75%	360.902	7.746
100%	449.960	7.960



	25%	50%	75%	100%
MongoDB	221.271	333.322	424.100	583.139
Neo4j	50.534	72.177	76.506	76.229



	25%	50%	75%	100%
MongoDB	170.059	273.588	360.902	449.960
Neo4j	6.347	7.597	7.746	7.960

6.4 Query n°4

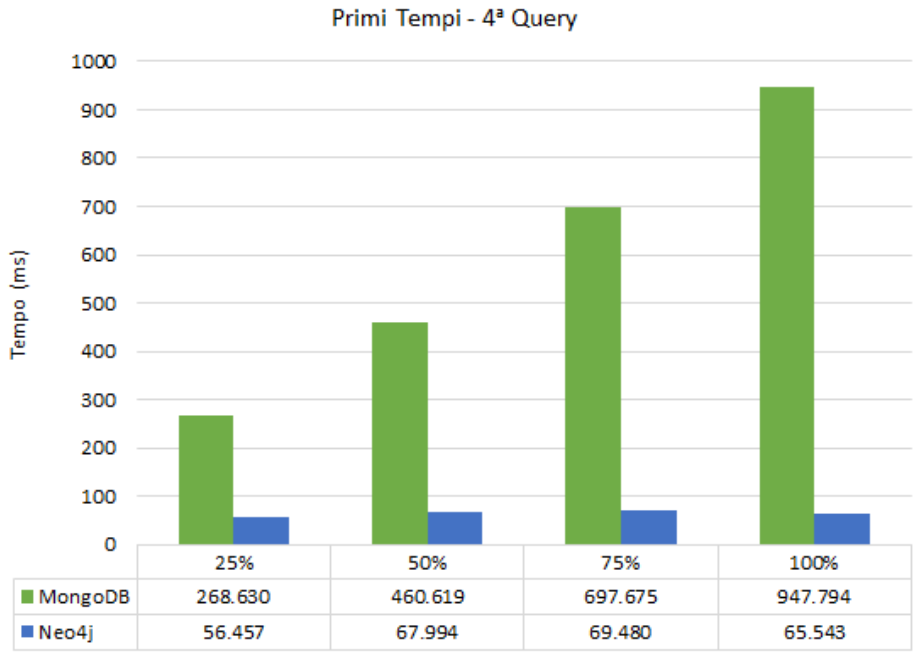
La quarta query effettuata presenta una selezione condizionata con 3 WHERE su tre entità, unite tramite 2 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
#     JOIN people
#         ON calls.CallingNbr = people.Number
#     JOIN cells
#         ON calls.CellSite = cells.CellSite
# WHERE StartDate >= 1580083200 AND EndDate < 1580256000
#       AND Duration > 900

# MongoDB
query = [
            {"$match": {"StartDate": {"$gte": start_search,
                                      "$lt": end_search},
                        "Duration": {"$gte": dur_search}}},
            {"$lookup": {"from": "people",
                         "localField": "CallingNbr",
                         "foreignField": "Number",
                         "as": "Calling"}},
            {"$lookup": {"from": "cells",
                         "localField": "CellSite",
                         "foreignField": "CellSite",
                         "as": "Cell"}}
        ]

# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
                          -[r2:is_done]->(ce:cell) \
         WHERE c.StartDate >= 1580083200 \
             AND c.StartDate < 1580256000 \
             AND c.Duration > 900 \
         RETURN p1, r1, c, r2, ce"
```

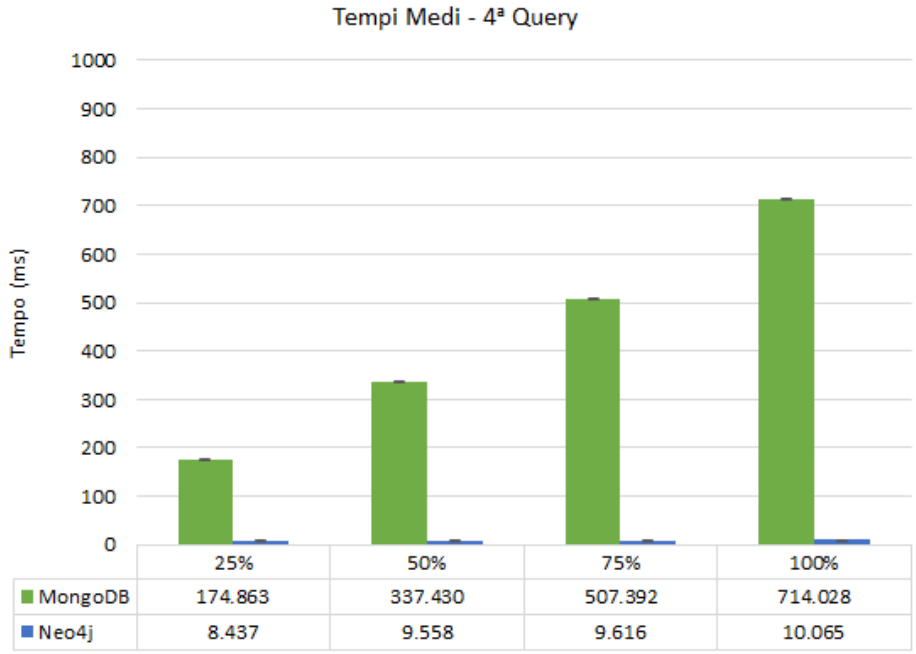
I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	Primi Tempi	
	MongoDB	Neo4j
25%	268.630	56.457
50%	460.619	67.994
75%	697.675	69.480
100%	947.794	65.543

	Tempi Medi - 30 rip.	
	MongoDB	Neo4j
25%	174.863	8.437
50%	337.430	9.558
75%	507.392	9.616
100%	714.028	10.065



	25%	50%	75%	100%
MongoDB	268.630	460.619	697.675	947.794
Neo4j	56.457	67.994	69.480	65.543



	25%	50%	75%	100%
MongoDB	174.863	337.430	507.392	714.028
Neo4j	8.437	9.558	9.616	10.065

7 Osservazioni e conclusioni

Una prima evidenza è la sostanziale differenza tra i tempi di esecuzione dapprima in ambiente privo di alcun dato presente nella memoria cache dei database e successivamente sfruttando la suddetta. Facendo due conti, si osserva come MongoDB mantenga un'efficienza nell'uso della cache nettamente inferiore e non costante rispetto a Neo4j: il primo manifesta una riduzione del tempo d'esecuzione delle query sostanzialmente decrescente, con un valore massimo del 64.3% nella prima fino ad un minimo del 19.7% nella terza, con un anomalo aumento nell'ultima, comunque posizionata al terzo posto in scala decrescente; il secondo mantiene una riduzione del tempo d'esecuzione abbastanza costante, anzi lievemente crescente, con un massimo del 89.1% nella terza query e un minimo del 83.4% nella quarta, anch'esso anomalo in quanto successivamente all'ultima, in scala decrescente, v'è una prosecuzione ordinata delle query presentate.

Osservando i risultati della prima query in relazione alle successive è possibile estrapolare i punti di forza dei due DBMS: ove presente un'operazione su singola entità, limitando l'affermazione alle selezioni, MongoDB risulta più efficiente, rimanendo pienamente sotto la soglia dei 10 ms con tutti i dataset utilizzati, risultato raggiungo da Neo4j solo con l'uso della cache; successivamente, quest'ultimo si trova sempre in netto vantaggio, mantenendosi, al massimo, sotto la metà del tempo d'esecuzione di MongoDB per i medesimi query e dataset, permettendo di affermare che, ove presente un'operazione di corrispondenza tra diverse entità per mezzo delle relazioni, Neo4j non risente in maniera considerevole di alcun aumento dei dati e della complessità di operazioni da effettuare su di essi. Per la precisione, Neo4j si mantiene in ogni situazione proposta sotto la soglia degli 80 ms, situazione inesistente per MongoDB al di fuori della sua favorevole, ossia in assenza di aggregazioni (JOIN in SQL).

Tirando le somme, al netto delle differenze di cui sopra, entrambi i database rispettano tempi di risposta al più sotto l'1 s, con un dataset di più di un milione di istanze di entità, permettendo di affermare che rispettano appieno l'obiettivo dei database NoSQL, ossia una migliore, seppur diversa, gestione di un'ampia quantità di dati in tempi ragionevoli, relativamente alle necessità. Ciò detto, si potrebbe asserire che MongoDB sia più adatto, in relazione alla sua gestione document-oriented, a situazioni in cui v'è la necessità di mantenere grandi quantità di dati ma che non presentino frequenti relazioni tra di loro, così da poter sfruttare pienamente il paradigma *key-value* dei documenti per la ricerca. Di contro, Neo4j è più adatto in situazione ove è necessaria una maggiore responsività nella ricerca di dati strettamente correlati, fortemente ottimizzata grazie all'organizzazione del database a grafo e dell'utilizzo delle relative e note operazioni su tale struttura dati.