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

30 novembre 2022

# Indice

# 1 Abstract

## 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-**c**onsistency)
- Disponibilità (**a**vailability)
- Tolleranza di partizione (**p**artitioning)

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

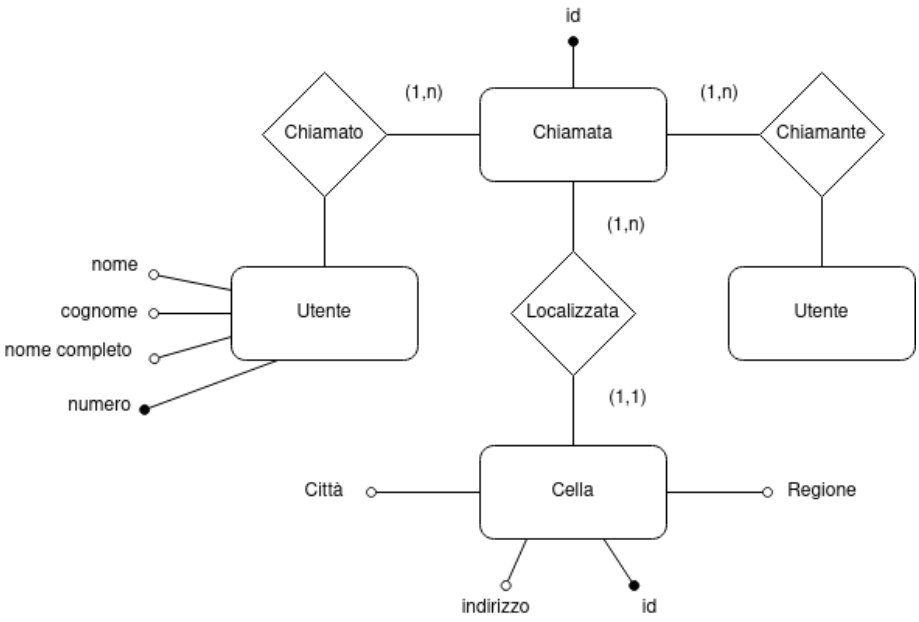
In questo trattato sono illustrate le misurazioni di performance (tempo di esecuzione e di risposta) dei DBMS NoSQL presi in considerazione in riferimento a quattro query scelte con complessità computazionale incrementale. Per questa analisi è stato scelto il caso di studio *How to use graph technology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

# 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 textitreplica 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, implementanto 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. Dispone inoltre di `matplotlib`, una libreria per la generazione di grafici. Si utilizza oltre ciò 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 → call.calling_number
- *received_call*: call.called_number → person.number
- *located_in*: call.cell_site → 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 → call.calling_number
- *received_call*: call.called_number → person.number
- *located_in*: call.cell_site → 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 50000 persone, 10000 celle e 200000 chiamate.

```
[load]
MAX_PEOPLE=50000
MAX_CELLS=10000
MAX_CALLS=200000

```

Listing 1: config.ini

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

```
def gen_people(size: int):
    with open(data_path("people.csv"), 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['first_name', 'last_name', 'full_name', 'number'])

        for i in range(size):
            name = fake.name().split()

            #genera un numero univoco (simulazione di un do-while)
            while True:
                number = gen_fake_phone_number()
                if number not in people:
                    break

            writer.writerow([name[0], name[1], " ".join(name), number])
            people.append(number)

        file.close()

```

Listing 2: gen_people()

```
def gen_cells(size: int):
    global ncells

    with open(data_path("cells.csv"), 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['cell_site', 'city', 'address', 'state'])

        for i in range(size):
            city = fake.city()
            address = fake.street_name()
            state = fake.current_country_code()

            writer.writerow([i, city, address, state])

        file.close()


```

Listing 3: gen_cells()

```
def gen_calls(size: int):
    npeople = len(people)

    with open(data_path("calls.csv"), 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['calling_number', 'called_number', '
    start_date', 'end_date', 'duration', 'cell_site'])

        for i in range(size):
            caller = randint(0, npeople-1)

            #prende ripetutamente un numero dalla lista delle
    persone se il numero del chiamante e' uguale a quello del
    chiamato
            while True:
                called = randint(0, npeople-1)
                if called != caller:
                    break

            duration = randint(0, 1000)
            cell_site = randint(0, ncells-1)

            #data di inizio [solo anno attuale]
            start_date = fake.date_time_this_year()

            #la data di fine della chiamata equivale alla data
    di inizio della chiamata + la durata(in secondi)
            end_date = start_date + timedelta(seconds=duration)

            writer.writerow([people[caller], people[called],
    int(round(datetime.timestamp(start_date))), int(round(
    datetime.timestamp(end_date))), duration, cell_site])
        file.close()


```

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.

```
def load_data(handle: MongoClient):
db = handle.progettodb2

with open(data_path('cells.csv'), "r") as cfile:
    db.cells.insert_many(to_dict(list(DictReader(cfile))))

    cfile.close()

with open(data_path('people.csv'), "r") as pfile:
    db.people.create_index("number", unique=True)
    db.people.insert_many(to_dict(list(DictReader(pfile))))

    pfile.close()

with open(data_path('calls.csv'), "r") as cafile:
    db.calls.insert_many(to_dict(list(DictReader(cafile))))

    cafile.close()


```

Listing 5: load_data()

# 6 Risultati degli esperimenti

## 6.1 2 WHERE

Viene effettuata una selezione condizionata con due clausole `WHERE`

```
"MATCH (c:call) WHERE c.start_date >"
+ str(gt_start)
+ " AND c.start_date < "
+ str(lt_start)
+ " RETURN c"

```

Listing 6: Neo4j

```
{
    "$match": {
        "start_date": {
            "$gte": gt_start,
            "$lt": lt_start
        }
    }
}

```

Listing 7: MongoDB

## 6.2 3 WHERE

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

```
"MATCH (c:call) WHERE c.start_date > "
+ str(gt_start)
+ " AND c.start_date < "
+ str(lt_start)
+ " AND c.duration >= "
+ str(duration)
+ " RETURN c"


```

Listing 8: Neo4j

```
{
    "$match": {
        "start_date": {
            "$gte": gt_start,
            "$lt": lt_start
        },
```

```
        "duration": {
            "$gte": duration
        }
    }
}

```

Listing 9: MongoDB

### 6.3 3 WHERE 1 JOIN

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

```
"MATCH (p:person)-[r:made_call]->(c:call) WHERE c.start_date >"
+ str(gt_start)
+ " AND c.start_date < "
+ str(lt_start)
+ " AND c.duration >= "
+ str(duration)
+ " RETURN c, r, p"

```

Listing 10: Neo4j

```
{
    "$match": {
        "start_date": {
            "$gte": gt_start,
            "$lt": lt_start
        },
        "duration": {
            "$gte": duration
        }
    }
},
{
    "$lookup": {
        "from": "people",
        "localField": "calling_number",
        "foreignField": "number",
        "as": "caller"
    }
}

```

Listing 11: MongoDB

### 6.4 3 WHERE 2 JOIN

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

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

```

Listing 12: Neo4j

```
{
    "$match": {
        "start_date": {
            "$gte": gt_start,
            "$lt": lt_start
        },
        "duration": {
            "$gte": duration
        }
    }
},
{
    "$lookup": {
        "from": "people",
        "localField": "calling_number",
        "foreignField": "number",
        "as": "caller"
    }
},
{
    "$lookup": {
        "from": "cells",
        "localField": "cell_site",
        "foreignField": "cell_site",
        "as": "cell"
    }
}

```

Listing 13: MongoDB

# 7 Conclusioni

# Riferimenti bibliografici

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