UNIVERSITÀ
DI TORINO

Corso di Laurea Magistrale in Informatica

Modelli e Architetture Avanzati di Basi di Dati

Progetto NoSQL

Camoia Andrea

2025

di.unito.it

DIPARTIMENTO
DI INFORMATICA

Indice

1 Introduzione

1.1 Generazione dei Dati

Il dataset è stato generato utilizzando l'**LDBC Social Network Benchmark (SNB) Datagen**, sfruttando l'implementazione basata su **Spark** in esecuzione all'interno di un container **Docker**. Il generatore è stato configurato in modalità *Interactive*, questa configurazione produce la versione una versione del dataset denominata "**Composite Merged FK**", ovvero uno schema relazionale normalizzato in cui:

- per le relazioni 1 : *N*, le chiavi esterne sono conservate all'interno delle tabelle dei nodi;
- le relazioni molti-a-molti (come `Person_knows_Person`) sono proiettate in tabelle di join dedicate.

Tale struttura rappresenta un compromesso ideale per le scelte progettuali intraprese. Rispetto alla versione "Composite Projected FK", dove anche gli attributi multivalore (es. `Person.email`) vengono proiettati in tabelle separate (es. `Person_email_EmailAddress`), questa configurazione riduce la frammentazione dei dati, limitando la complessità di gestione delle join che risulterebbe eccessivamente onerosa per il database *document-based*.

C	File	Content
N	static/Organisation/part-*.csv	id | type | name | url | LocationPlaceId
N	static/Place/part-*.csv	id | name | url | type | PartOfPlaceId
N	static/Tag/part-*.csv	id | name | url | TypeTagClassId
N	static/TagClass/part-*.csv	id | name | url | SubclassOfTagClassId
N	dynamic/Comment/part-*.csv	creationDate | id | locationIP | browserUsed | content | length | CreatorPersonId | LocationCountryId | ParentPostId | ParentCommentId
E	dynamic/Comment_hasTag_Tag/part-*.csv	creationDate | CommentId | TagId
N	dynamic/Forum/part-*.csv	creationDate | id | title | ModeratorPersonId
E	dynamic/Forum_hasMember_Person/part-*.csv	creationDate | ForumId | PersonId
E	dynamic/Forum_hasTag_Tag/part-*.csv	creationDate | ForumId | TagId
N	dynamic/Person/part-*.csv	creationDate | id | firstName | lastName | gender | birthday | locationIP | browserUsed | LocationCityId | language | email
E	dynamic/Person_hasInterest_Tag/part-*.csv	creationDate | personId | interestId
E	dynamic/Person_knows_Person/part-*.csv	creationDate | Person1Id | Person2Id
E	dynamic/Person_likes_Comment/part-*.csv	creationDate | PersonId | CommentId
E	dynamic/Person_likes_Post/part-*.csv	creationDate | PersonId | PostId
E	dynamic/Person_studyAt_University/part-*.csv	creationDate | PersonId | UniversityId | classYear
E	dynamic/Person_workAt_Company/part-*.csv	creationDate | PersonId | CompanyId | workFrom
N	dynamic/Post/part-*.csv	creationDate | id | imageFile | locationIP | browserUsed | language | content | length | CreatorPersonId | ContainerForumId | LocationCountryId
E	dynamic/Post_hasTag_Tag/part-*.csv	creationDate | PostId | TagId

Figure 1.1: Lista dei File CSV in output dalla generazione `csv-composite-merged-fk` con la relativa struttura. [1]

Il comando Docker utilizzato per la generazione dei dati è il seguente:

```
docker run -it --rm \
      --name ldbc_datagen \
      --memory=8g \
      -v "$(pwd)/ldbc_output":/out \
      ldbc/datagen-standalone:0.5.0-2.12_spark3.2 \
      -- \
      --mode interactive \
      --scale-factor 1 \
      --format csv \
      --output-dir /out
```

1.2 Query

Le query implementate in questo progetto sono le seguenti:

1.2.1 Query Lookup

- Individuare la posizione dell'università dove ha studiato una certa persona e quella dell'azienda in cui lavora.
- Data una persona, individuare tutte le altre persone che conosce ("knows") all'interno dell'azienda in cui lavora / dell'università in cui studia. (Cross-Database).

1.2.2 Query Lookup

- Individuare la persona più influente in termini di Likes totali a tutti i suoi post.
- Individuare i 5 Tag più utilizzati in un dato range temporale
- Individuare l'utente più popolare in termini di persone conosciute all'interno di un'università (Cross-Database)

2 Progettazione

2.1 Document-Based Database

Come Document-Based Database è stato utilizzato **MongoDB**: al suo interno sono state **gestite tutte le entità con molti attributi** ma che partecipano a **poche relazioni**, in modo da ottimizzare e sfruttare al meglio la natura di questa tipologia di database. In particolare sono state previste le seguenti collezioni:

- `Person`
- `Organisation`
- `Place`
- `Person_workAt_Company`
- `Person_studyAt_University`

in cui la struttura dei documenti corrisponde esattamente a quella dei CSV mostrati nella tabella precedente. [1.1]. La scelta di introdurre **anche le due relazioni** `Person_workAt_Company` e `Person_studyAt_University`, pur non essendo perfettamente in linea con la filosofia di questa tipologia di database, è stata tuttavia obbligata dalla struttura del dataset LDBC, che separa queste informazioni in file distinti e non prevede una rappresentazione incorporata nativamente all'interno delle entità principali. In questo caso specifico, però, la soluzione **non rappresenta un problema significativo** poichè le entità `Person` partecipano a pochissime relazioni di tipo "studyAt" (generalmente una sola per persona) e a un numero ridotto di relazioni "workAt", che in media si limita a 4/5 valori.

In questo modo è possibile gestire tutte le informazioni e query necessarie riguardo le entità `Person` e `Organisation` senza incidere sulle prestazioni o sulla scalabilità del sistema.

2.2 Graph Database

Per quanto riguarda il database a grafo, la scelta è ricaduta su **Neo4j**. Coerentemente alla natura di questa tipologia di database, al suo interno sono state **gestite tutte le relazioni tra le entità del dataset**, modellando principalmente quelle sociali. In particolare il grafo prevede i seguenti nodi e archi:

- `(Person)-[KNOWS]->(Person)`
- `(Person)-[CREATED]->(Post)`
- `(Person)-[LIKES]->(Post)`
- `(Post)-[HASTAG]->(Tag)`

In questo caso, **i nodi** corrispondenti alle entità hanno come unico attributo il proprio **ID** *(tranne nel caso di* `Person` *in cui sono presenti anche firstName e lastName)*, grazie al quale è possibile effettuare **il mapping con le relative entità nel database document-based** contenente le informazioni complete.

3 Implementazione

Entrambi i database sono stati implementati e gestiti tramite python, sfruttando le rispettive librerie `pymongo` e `neo4j`. In particolare sono state create due **classi manager**, che gestiscono le istanze e le conseguenti interrogazioni verso i database, i quali girano il locale.

3.1 MongoDB Manager

MongoDB viene gestito tramite la classe `MongoDBManager`,
implementata nel file `mongo_db_manager.py`. Questa classe consente innanzitutto di instanziare un oggetto responsabile dell'avvio e della gestione della connessione con il database. Inoltre, al suo interno è stato implementato il metodo *load_data()*, che si occupa della creazione delle collezioni e del caricamento dei documenti a partire dai file CSV forniti dal dataset.

La classe include infine tutti i metodi necessari per l'esecuzione delle query, basati su interrogazioni dirette alla libreria `pymongo`. I risultati ottenuti vengono restituiti come oggetti Python facilmente manipolabili, favorendo così l'integrazione con le altre componenti del progetto

3.2 Neo4j Manager

In maniera analoga alla precedente, per la gestione del database Neo4j è stata implementata la classe `Neo4jManager` nel file `neo4j_manager.py`.

In questo caso, tutte le operazioni vengono effettuate **dichiarando in maniera esplicita le query in formato Cypher** che poi vengono sottoposte al database connesso tramite la libreria. La particolare struttura del database ha richiesto anche l'implementazione di appositi metodi per la creazione dei nodi ed archi del grafo che contengono le informazioni.

Ad esempio tramite il seguente metodo vengono creati i nodi relativi ai `Post` e le rispettive relazioni CREATED che li collegano con il creatore `Person`.

```
    def load_posts(self, csv_file):
        """
        Loads Post nodes and creates [:CREATED] relation to Person.
        """

        query = """
        LOAD CSV WITH HEADERS FROM 'file:///' + $file AS row
           FIELDTERMINATOR '|'
        MERGE (post:Post {id: row.id})
        WITH post, row
        MATCH (creator:Person {id: row.CreatorPersonId})
        MERGE (creator)-[:CREATED {creationDate : datetime(row.
           creationDate)}]->(post)
        """
        with self.driver.session() as session:
            session.run(query, file=csv_file)
            print(f"Posts Nodes and CREATED relation loaded")
```

3.3 Query

3.3.1 Query 1: Location Finder

> Individuare la posizione dell'università dove ha studiato una certa persona e quella dell'azienda in cui lavora.

Questa query è stata implementata interamente in MongoDB all'interno del metodo `get_person_locations(person_id)`.

La procedura risulta abbastanza semplice: dato l'id della persona in input, vengono sfruttate le relazioni `Person_workAt_Company` e `Person_studyAt_University` per ricavare gli ID dell'università in cui la persona ha studiato e quali sono le aziende in cui ha ha lavorato. `university_ids` e `company_ids` vengono usati a loro volta per ricercare le informazioni della relativa organizzazione all'interno della collezione `Organisations`, ed in particolare l'attributo `LocationPlaceId` che fa riferimento ad un luogo geografico salvato all'interno della collezione `Places`. Da qui finalmente vengono estrapolate e restituite la Città e la Nazione di ogni Organizzazione ricercata.

Sebbene lo stesso risultato possa essere ottenuto tramite operazioni di `$lookup` (equivalenti alle join), si è preferito **suddividere l'elaborazione in sotto-query sequenziali**, evitando l'utilizzo di join interne. Questo perchè in MongoDB **le join sono generalmente poco efficienti** e non adatte a scenari distribuiti, in quanto richiedono l'aggregazione di dati provenienti da più collezioni.

Figure 3.1: Esempio di risultato ottenuto dalla Query 1.

3.3.2 Query 2: Known Colleagues

> Data una persona, individuare tutte le altre persone che conosce ("knows") all'interno dell'azienda in cui lavora / dell'università in cui studia.

Questa seconda query è di tipo Cross-Database, e richiede quindi l'interazione tra MongoDB e Neo4j: in una prima fase viene utilizzato MongoDB per ricavare l'elenco dei colleghi (studenti e lavoratori) associati alla persona identificata dal `person_id` in input:

```
university_colleagues = mongo_manager
                        .get_university_colleagues(person_id)

work_colleagues = mongo_manager.get_work_colleagues(person_id)
```

Le due liste ottenute, contenenti gli ID delle persone appartenenti alla stessa università e/o alla stessa azienda, vengono poi passate al metodo `get_known_from_list(person_id, id_list)`, implementato in Neo4j. Tale metodo sfrutta la struttura a grafo per verificare efficientemente quali tra questi individui sono direttamente connessi alla persona tramite la relazione `[:KNOWS]`:

```python
    def get_known_from_list(self, person_id, id_list):
        """Returns the people that the given person knows from the
           given list."""
        query = """
            MATCH
            (person:Person {id: $person_id})-[:KNOWS]->(known:Person)
            WHERE known.id IN $id_list
            RETURN known.id AS KnownPersonId
        """
        with self.driver.session() as session:
            result = session.run(query, person_id=person_id, id_list=
               id_list)
            return [record.data()["KnownPersonId"] for record in
               result]
```

La combinazione dei due database consente di sfruttare i punti di forza di entrambi i modelli: MongoDB per filtrare rapidamente i colleghi a partire dalla struttura tabellare del dataset, e Neo4j per valutare in modo naturale le relazioni sociali.

University Information

Total Colleagues: 102
Total Known Colleagues: 7
Known Colleagues: Hao Chen (24189255812529),Jun Liu (24189255815385),Lei Chen (26388279069262),Lei Chen (26388279069970),Yang Chen (26388279074216),Jun Zhu (32985348834925),Chen Liu (32985348841918)

Company Information

Total Colleagues: 86
Total Known Colleagues: 2
Known Colleagues: Hao Li (28587302324694),Peng Zhao (28587302326703)

Figure 3.2: Esempio di risultato ottenuto dalla Query 2.

3.3.3 Query 3: Most Likes

Individuare la persona più influente in termini di Likes totali a tutti i suoi post.

Questa query viene eseguita in Neo4j, sfruttando direttamente la struttura a grafo del social network. L'obiettivo è individuare l'autore che ha ottenuto il maggior numero di `LIKE` sui

contenuti che ha creato.

La query Cypher sfrutta un **interessante match tra due relazioni**: la relazione `[:CREATED]` tra i nodi `Person` e `Post`, e poi le relazioni `[:LIKES]` in ingresso per ciascun post precedentemente individuato. Una volta individuate tutte le relazioni le conteggia e ordina i risultati in base al numero totale di like, restituendo la persona con il valore più alto:

```
    def get_most_liked_person(self):
        """Returns the person with the most likes (across all posts)."""
        query = """
        MATCH (author:Person)-[:CREATED]->(post:Post)<-[like:LIKES]-(:Person)
        RETURN author.firstName AS Name,
               author.lastName AS Surname,
               count(like) AS TotalLikes
        ORDER BY TotalLikes DESC
        LIMIT 1
        """
        with self.driver.session() as session:
            result = session.run(query)
            record = result.single()
            if record:
                return record.data()

            return None
```

Most Influent Person

Name: Shweta
Surname: Kapoor
TotalLikes: 7515

Figure 3.3: Esempio di risultato ottenuto dalla Query 3.

3.3.4 Query 4: Top Tag

Individuare i 5 Tag più utilizzati in un dato range temporale.

Anche questa query viene eseguita interamente in Neo4j, **filtrando i post in base alla data**

di creazione e conteggiando le associazioni con i tag. Notiamo come in questo caso la data di creazione sia un parametro della relazione stessa. La query Cypher seleziona tutti i post creati nel range temporale, risale ai tag collegati tramite la relazione `[:HASTAG]` e calcola il numero di occorrenze per ciascun tag. I risultati vengono poi ordinati in ordine decrescente e limitati ai primi cinque:

```
    def get_most_used_tag(self, begin_date, end_date):
        """Returns the tag with the most usages (posts) during a given
            time period."""
        query = """
        MATCH (:Person)-[r:CREATED]->(post:Post)-[:HASTAG]->(tag:Tag)
        WHERE r.creationDate >= $begin_date AND r.creationDate <=
           $end_date
        RETURN tag.name AS TagName,
               count(post) AS TotalUsages
        ORDER BY TotalUsages DESC
        LIMIT 5
        """
        with self.driver.session() as session:
            # Pass the datetime objects directly; the driver converts
               them to Neo4j DateTime
            result = session.run(query, begin_date=begin_date,
               end_date=end_date)
            record = result.single()
            if record:
                return record.data()
            return None
```

Figure 3.4: Esempio di risultato ottenuto dalla Query 4.

3.3.5 Query 5: Most Influent Person

Individuare l'utente più popolare (in termini di persone che lo conoscono) all'interno di un'università.

Quest'ultima query è di tipo Cross-Database. MongoDB in questo caso viene utilizzato per ricavare tutti gli iscritti a una determinata università tramite il metodo `get_university_students(university_id)`, ottenendo così la lista degli ID degli studenti. Questi ID vengono poi passati a Neo4j, che esegue una query Cypher per individuare la persona più popolare, ovvero quella che è conosciuta dal maggior numero persone.

```
    def get_most_popular_in_list(self, person_ids):
        """ get the most known person from a list of people """

        query = """
            MATCH (person:Person)-[:KNOWS]->(known:Person)
            WHERE known.id IN $person_ids
            RETURN  known.id AS KnownPersonId,
                    count(person) AS KnownCount
            ORDER BY KnownCount DESC
            LIMIT 1
        """
```

```
        with self.driver.session() as session:
            result = session.run(query, person_ids=person_ids)
            record = result.single()
            if record:
                return record.data()

            return None
```

Infine, individuato l'ID della persona più conosciuta, il controllo passa nuovamente a mongoDB che estrae maggiori informazioni riguardo ad essa *(Location, Birithday, etc.)*.

Figure 3.5: Esempio di risultato ottenuto dalla Query 5.

3.4 WebApp

Infine è stata realizzata un'applicazione web di supporto che permette di eseguire le query implementate. L'interfaccia è stata sviluppata utilizzando **la libreria Python Eel**, una tecnologia che permette di collegare in modo diretto una grafica web intuitiva con la logica applicativa del backend. Grazie a questo strumento, l'utente può selezionare ed eseguire in modo dinamico le diverse tipologie di interrogazioni personalizzandone i parametri; l'applicazione si occupa di ricevere i comandi, coordinare le richieste verso i database MongoDB e Neo4j fino a restituire a video i risultati ottenuti in tempo reale.

4 Conclusioni

Il codice ed altre informazioni sono disponibili nella Repository GitHub del progetto: `https://github.com/AndCamo/NoSQL_Project`

References

[1] Renzo Angles et al. The LDBC Social Network Benchmark. *CoRR*, abs/2001.02299, 2020.