PROGETTO 2 – CORSO BIG DATA 2021/2022

UNIVERSITÀ ROMA TRE

Polyglot Data Management - Creazione di uno o più scenari poliglotti

MARIO CUOMO

Sommario

PROGETTO

Tra le 4 V che caratterizzano i big data – Variety, Velocity, Veracity, Volume – il progetto approfondisce il tema della varietà dei dati. In particolare ci si concentra sul polyglot data management attraverso studi e creazione di scenari poliglotti.

Durante il progetto sono stati effettuati degli esperimenti e studi che traggono ispirazione dalle seguenti fonti.

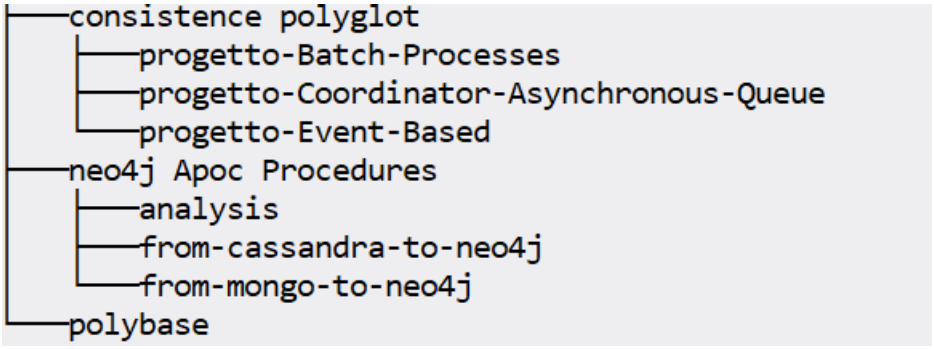
1. Managing Global Data in Microservice Polyglot Persistence Scenarios
   https://dzone.com/articles/manage-global-data-microservice-polyglot-persistence
   Questo post descrive 3 modi per sincronizzare un dato globale all'interno di un sistema poliglotta. Si immagini una applicazione realizzata a micro-servizi che per il suo funzionamento utilizza degli storage eterogenei. A questo punto si immagini l'arrivo di una richiesta di aggiornamento di un dato a un database x e per consistenza tale dato deve essere sincronizzato sul database y. Per avviare ed effettuare la sincronizzazione del dato in x e y esistono diversi pattern che possono essere seguiti.

2. Transform MongoDB collections automagically into Graphs
   https://medium.com/neo4j/transform-mongodb-collections-automagically-into-graphs-9ea085d6e3ef
   Questo post descrive diverse procedure messe a disposizione dal sistema neo4j per convertire dati dal formato json in dati che hanno una struttura a grafo. I dati sono estratti dalle collezioni di mongodb e caricati in neo4j.
   Le procedure in questione sono le APOC, *Awesome Procedures on Cypher* – un insieme di procedure che permettono all'utente di creare delle user defined functions.
   Esulando dal contesto descritto nel post, è stato effettuato un test anche per convertire dati da un column-store (cassandra).

3. Towards Polyglot Data Stores
   https://arxiv.org/abs/2204.05779
   Questo articolo identifica i requisiti fondamentali dei polyglot data stores, dalle tecniche del processamento di query all'architettura dell'intero sistema.
   Tra i vari sistemi proposti si è approfondito PolyBase – una soluzione in casa Microsoft per virtualizzare i dati in formato relazionale attraverso una mappatura tramite tabelle.
   Si noti come se nel punto 2 i dati vengono effettivamente trasferiti da una base di dati a un'altra, con PolyBase si crea una tabella utilizzata per virtualizzare i dati senza modificarne la natura.

STRUTTURA DEL PROGETTO

Il progetto è disponibile all'indirizzo *https://github.com/mariocuomo/polyglot-systems* e ha la seguente struttura.



La cartella `consistence polyglot` contiene gli esperimenti svolti sulla gestione di un dato globale in una architettura a micro-servizi. I 3 progetti sono realizzati come applicazione springboot composte con docker compose. Ogni progetto contiene una cartella `sito` – una applicazione web sviluppata in python.

La cartella `neo4j Apoc Procedure` contiene gli esperimenti effettuati con le APOC di Neoj4 – estrazione di dati da cassandra e mongodb e analisi su struttura a grafo.

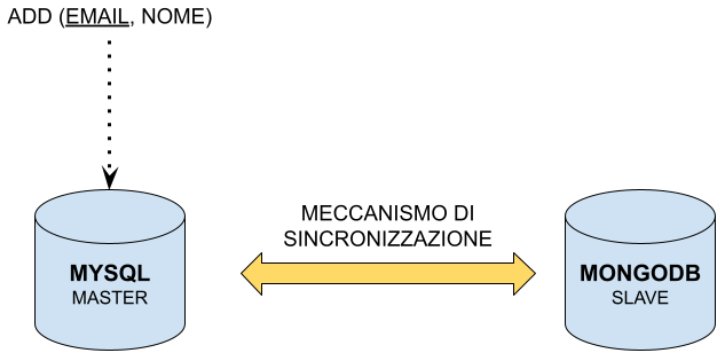
La cartella `polybase` contiene uno script per la realizzazione di una tabella virtualizzata in SQL Server.

MANAGING GLOBAL DATA IN MICROSERVICE POLYGLOT PERSISTENCE SCENARIOS

Questo capitolo descrive alcuni pattern per sincronizzare un dato globale tra diverse basi di dati eterogenee in una architettura a micro-servizi. Si considera quindi il requisito della consistenza. Si considerano applicazioni springboot containerizzate con docker e composte con docker compose.

RIGURARDO LO SCENARIO

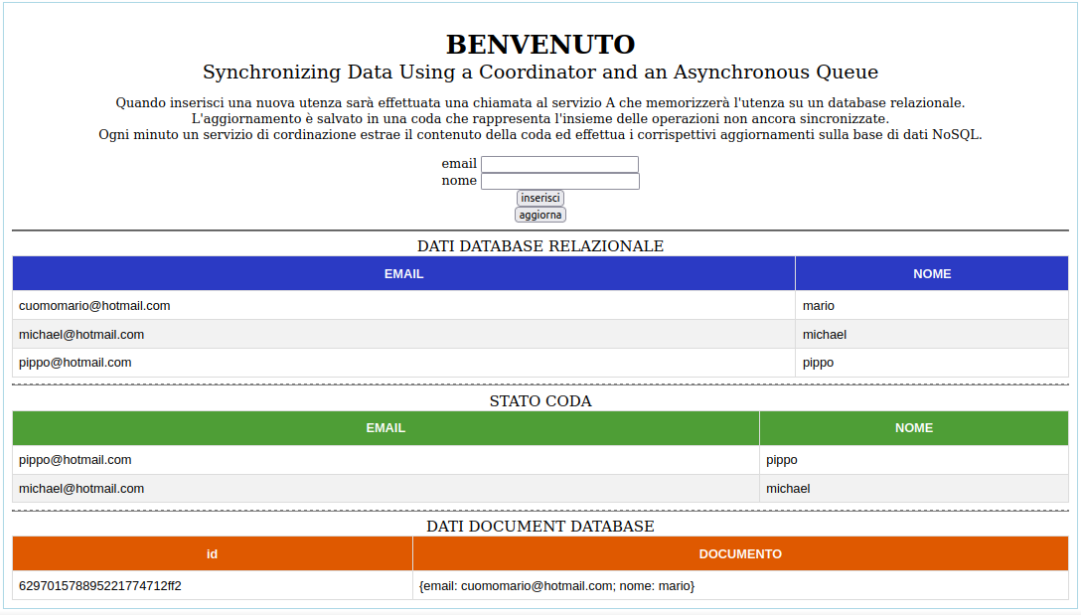
Per semplicità si consideri una applicazione che utilizza 2 data store: uno di tipo relazionale – per esempio mysql – e l'altro di tipo NoSQL – per esempio mongoDB.
Tra i due, l'unico a ricevere richieste di inserimento è il database mysql.
Si può immaginare di avere una relazione master-slave tra il database che riceve aggiornamenti dall'esterno e i database che devono mantenere i dati consistenti.
Ancora per semplicità, si consideri una relazione utenti(email, nome) e una collezione di documenti – caratterizzati anch'essi dai campi email e nome.



Per eseguire i progetti sono necessari i seguenti strumenti

- Gradle versione 6.4.1 per la compilazione dei progetti springboot e la gestione delle dipendenze
- Docker versione 20.10.16 per conteinerizzazione dei microservizi, delle basi di dati, dei sisemi di messaggistica e di service discovery
- docker-compose versione 1.25.0 per composizione dei micro-servizi

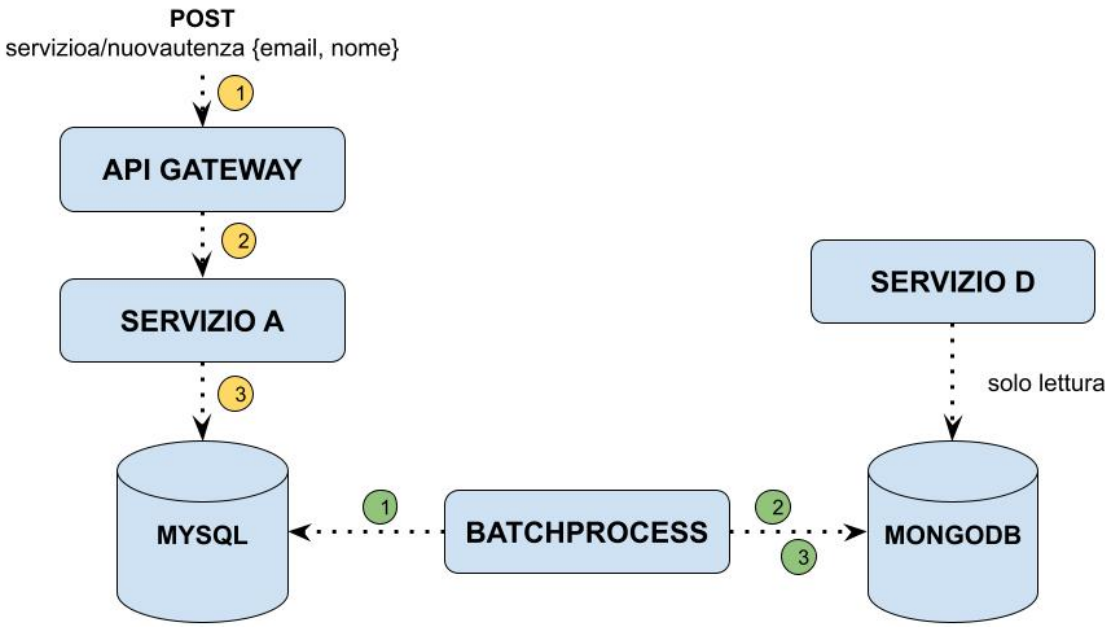
Le varie cartelle sito contengono delle semplici applicazioni web sviluppate in Python col framework Flask e pensate per essere eseguite in locale.
Sono delle interfacce in cui l'utente può interagire con il sistema eseguendo una richiesta di inserimento e visualizzare il contenuto delle basi di dati in tempo reale.



BATCH PROCESSES

Questo approccio si basa sui processi batch – quei processi che in genere sono utilizzati dagli etl per estrarre, trasformare e caricare i dati da una sorgente a un'altra.
Per il motivo sopracitato sono dei processi poco flessibili: sono strettamente legati alla sorgente ma anche alla destinazione.

L'idea è quella di effettuare sempre e solo le scritture sul database master e a intervalli di tempo regolari attivare i batch processes che hanno il compito di verificare se i database slaves sono allineati al master e, in caso negativo, occuparsi della sincronizzazione.
Si può immaginare che questo comporta delle operazioni arbitrariamente complesse: si potrebbe effettuare un hash(db) e verificare se questo coincide con l'ultimo hash calcolato e salvato in locale.
Nel caso specifico e semplicistico presentato si verifica se per ogni ennupla della relazione utenti è presente un documento con gli stessi attributi del record.



Schematizzando, si può riassumere il meccanismo come segue

1. il sistema riceve una chiamata POST per inserire una nuova utenza
2. l'api gateway cattura la richiesta e la inoltra al servizioA
3. il servizioA salva la nuova utenza sul database MySQL

Ogni minuto si attiva la funzione di sincronizzazione del micro-servizio batchprocess

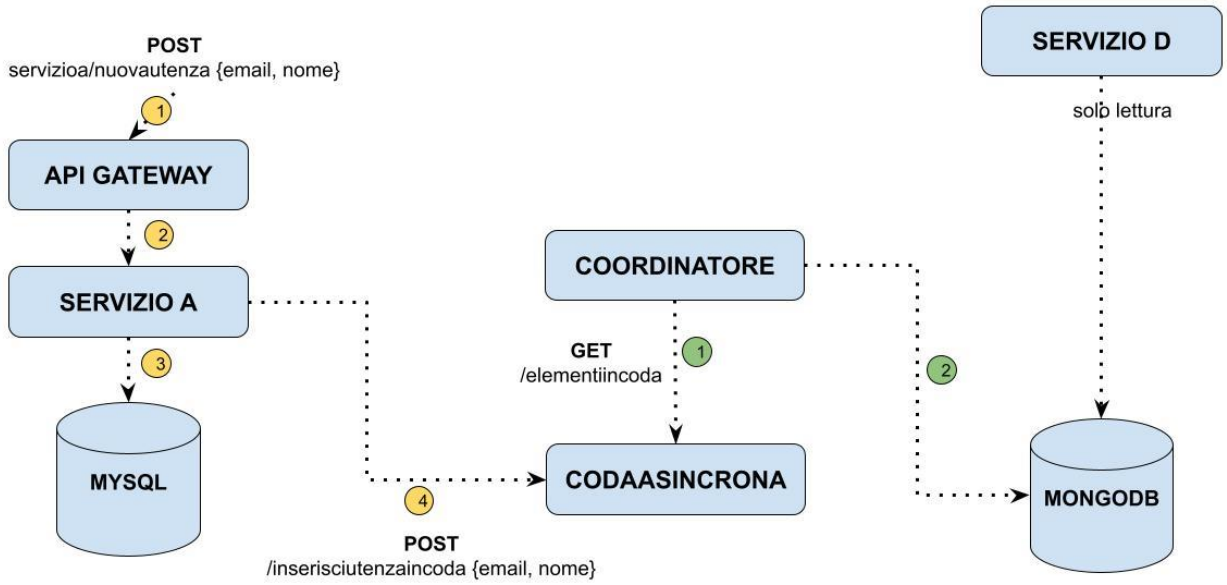
1. si estrae il contenuto dal database MySQL
2. si estrae il contenuto dal database Mongodb
3. si aggiorna il contenuto dal database Mongodb

NOTA
il servizioD è di supporto alla webapp python per leggere il contenuto di mongodb.

COORDINATOR ASYNCHRONOUS QUEUE

Questo approccio si basa sull'utilizzo di una coda asincrona che mantiene in memoria gli aggiornamenti effettuati sulla base di dati master. L'utilizzo della coda permette di mantenere l'ordine temporale delle operazioni eseguite e quindi garantire la coerenza di esse.
Si utilizza un coordinatore che ha il compito di estrarre di volta in volta il contenuto dalla coda.

In questo modo si è indipendenti dalle varie tecnologie utilizzate per la persistenza e i vari micro-servizi slave sono disaccoppiati dalla particolare tecnologia di sincronizzazione usata.
È una soluzione utilizzata quando si hanno pochi dati globali da sincronizzare: un vincolo da rispettare infatti è quello della transazionalità in quanto tutti le sincronizzazioni sugli storage slave devono essere effettuate in un'unica sessione del coordinatore.



Schematizzando, si può riassumere il meccanismo come segue

1. il sistema riceve una chiamata POST per inserire una nuova utenza
2. l'api gateway cattura la richiesta e la inoltra al servizioA
3. il servizioA salva la nuova utenza sul database MySQL
4. il servizioA effettua una chiamata POST al servizio codaasincrona con le informazioni dell'utenza inserita*

Ogni minuto si attiva la funzione di sincronizzazione del micro-servizio coordinatore

1. il servizio coordinatore effettua una chiamata GET al servizio codaasincrona per ottenere gli ultimi aggiornamenti
2. il coordinatore salva le nuove utenze sul database Mongodb

*si può pesare di avere un meccanismo di triggering attaccato alla comunicazione tra il servizioA e il database MySQL che genera la chiamata POST autonomamente senza l'intervento attivo del servizioA.

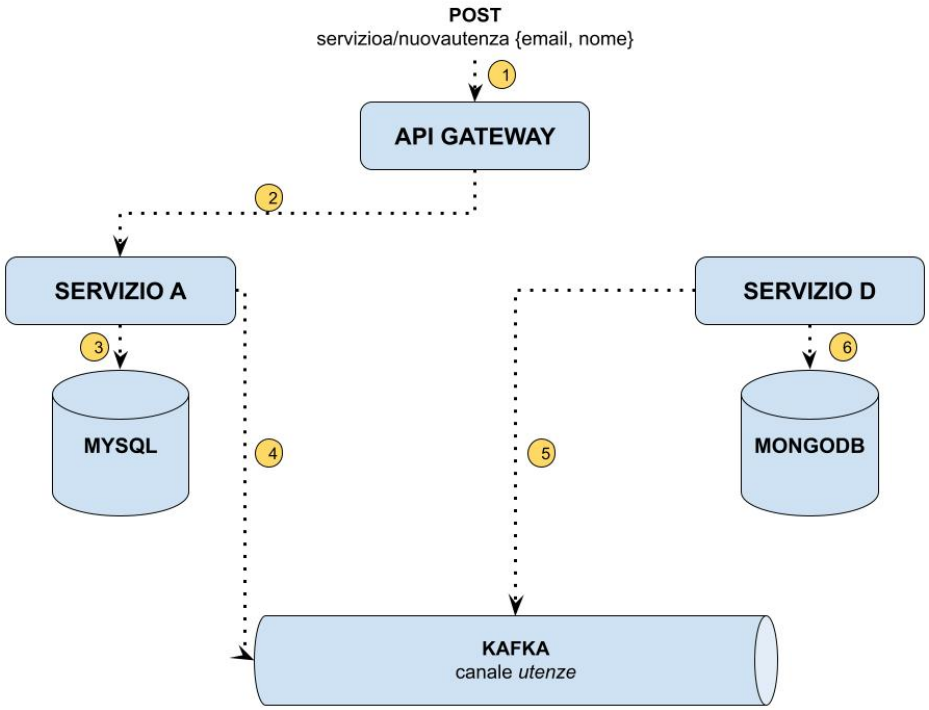
NOTA
il servizioD è di supporto alla webapp python per leggere il contenuto di mongodb.

EVENT BASED

Questo approccio si basa sulla realizzazione di una backbone di messaggi.
Tra i vari servizi di messaggistica kafka risulta essere lo standard de facto per lo scambio di messaggi.
L'idea è quella di avere un topic per ogni dato globale che si vuole mantenere: il database master è un publisher e gli slave sono dei subscriber per i topic di interesse.

Il meccanismo di funzionamento è semplice: il master riceve una richiesta di aggiornamento, la completa e pubblica un messaggio sul canale a cui è iscritto come publisher. Gli slave leggono dal canale tale aggiornamento.

L'effetto positivo è sicuramente la scalabilità del sistema e il meccanismo di sincronizzazione indipendente per ogni slave. Il prezzo da pagare è il costo iniziale di design dei vari topic kafka.



Schematizzando, si può riassumere il meccanismo come segue

1. il sistema riceve una chiamata POST per inserire una nuova utenza
2. l'api gateway cattura la richiesta e la inoltra al servizioA
3. il servizioA salva la nuova utenza sul database MySQL
4. il servizioA produce un messaggio con le informazioni dell'utenza inserita*
5. il servizioD è un consumatore dei messaggi prodotti dal servizioA
6. il servizioD salva la nuova utenza sul database Mongodb

*si può pesare di avere un meccanismo di triggering attaccato alla comunicazione tra il servizioA e il database MySQL che genera il messaggio autonomamente senza l'intervento attivo del servizioA.

AWESOME PROCEDURES ON CYPHER

Questo capitolo mostra come utilizzare le APOC messe a disposizione in neo4j per estrarre il contenuto da una base di dati NoSQL (in particolare mongodb e cassandra), trasformarlo e crearne un grafo rappresentativo.
Si utilizza docker per containerizzare le basi di dati e docker compose per comporle.

FROM MONGODB TO NEO4J

Con le APOC è possibile convertire documenti dal formato json a una struttura a grafo.
In questo scenario si considera una collezione di documenti in mongodb.

Si ipotezzi di avere un caso d'uso in cui una biblioteca ha necessità di conoscere per ogni libro quanti e quali sono gli utenti che lo hanno letto. Si è realizzato il tutto con una collezione di documenti che modellano un libro. Ogni documento ha un campo $nome\_libro$ e una lista $utenti$ che rappresenta gli utenti che lo hanno letto.

Per prima cosa si recupera l'intera collezione dall'istanza di mongodb e la si salva nella variabile value.

```
CALL apoc.mongodb.get('mongodb://mongo:neo4j@mongo:27017', 'test',
'utenti_libri', {}, true) YIELD value
```

A questo punto con la primitiva fromDocument si effettua un mapping dalla collezione di documenti alla strutta a grafo. In particolare la semantica dell'operazione è principalmente racchiusa nella sezione mappings. Quello che si fa è trasformare ogni documento in un nodo di tipo Libro con identificatore il campo $nome\_libro$ e per ogni utente nella lista $utenti$ si crea un nodo di tipo Person caratterizzato dalle proprietà $nome$ e $rating$.

```
CALL apoc.graph.fromDocument (value,
        { write: true,
          skipValidation: true,
          mappings: {
                    `$`: 'Libro{!nome_libro}',
                    `$.utenti`: 'Person{nome,rating}'
                    }
         } ) YIELD graph AS g1
return g1
```

In realtà la proprietà $rating$ è una caratteristica della relazione che collega un libro a un utente.
Si imposta quindi la proprietà alla relazione.

```
MATCH (a)-[r]->(b)
SET r.rating = b.rating
```

Purtroppo con l'operazione precedente si sono creati dei duplicati dei nodi di tipo Person.
Si pensi alla situazione in cui l'utente x ha letto sia il libro y che il libro z. In questo caso stati creati 4 nodi: due relativi al libro y e z e due relativi allo stesso utente x.
È possibile far collassare in un unico nodo le due copie.

```
MATCH (n:Person)
WITH toLower(n.nome) as nome,
collect(n) as nodes

CALL apoc.refactor.mergeNodes(nodes) yield node RETURN *
```

Per coerenza si elimina la proprietà $rating$ dai nodi di tipo Person – perché spostata sulla relazione tra utenti e libri.

```
MATCH (n:Person) REMOVE n.rating
```

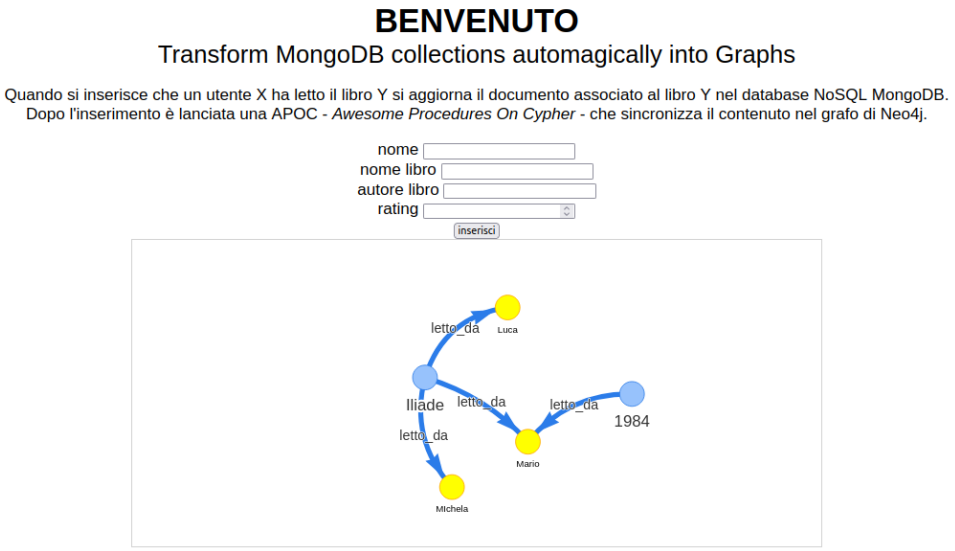
A questo punto è si effettua il rename delle relazioni che collegano Person e Libro.

```
MATCH ()-[rel]->() WITH collect(rel) AS rels

CALL apoc.refactor.rename.type("UTENTI", "letto_da", rels) YIELD committedOperations

RETURN committedOperations
```

Per eseguire i progetti sono necessari i seguenti strumenti

- Docker versione 20.10.16 per conteinerizzazione neo4j e mongodb
- docker-compose versione 1.25.0

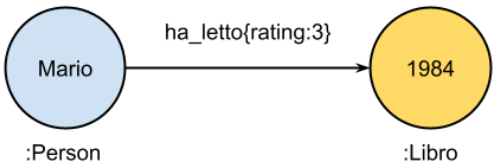
La cartella sito contiene una semplice applicazione web sviluppata in Python col framework Flask e pensata per essere eseguita in locale.
È una interfaccia in cui l'utente può interagire con il sistema eseguendo una richiesta di inserimento e visualizzare il contenuto delle basi di dati in tempo reale.



FROM CASSANDRA TO NEO4J

Con le APOC è possibile convertire dati in formato tabellare a dati con una struttura a grafo.
In questo scenario si considera come data store di partenza cassandra.

Si ipotezzi di avere un caso d'uso in cui una biblioteca ha necessità di conoscere per ogni libro quanti e quali sono gli utenti che lo hanno letto. Si è realizzato il tutto con una unica tabella in cassandra con le colonne $nome\_utente$, $nome\_libro$ e $rating$.



Per prima cosa si definisce uno schema in neo4j per i dati recuperati da un sistema semi-tabellare, come potrebbe essere cassandra o mysql.

```
CALL apoc.schema.assert(
  {User:['nome_utente']},
  {Libro:['nome_libro']});
```

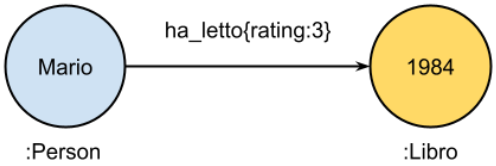
A questo punto si recupera la collezione da cassandra e la si legge riga per riga.
Una riga è definita dalla variabile temporanea row.
Quello che si fa è creare per ogni riga al più 2 nodi e una relazione in neo4j: un nodo è di tipo User e ha una unica proprietà che è $nome\_utente$; l'altro è di tipo Libro ed è caratterizzato dalla proprietà $nome\_libro$; la relazione invece è di tipo ha_letto ed è caratterizzata dalla proprietà $rating$.

```
CALL apoc.load.jdbc('jdbc:cassandra://172.18.1.2:9042/utenze_libri','utenze_libri') yield row
MERGE (u:User {nome:row.nome_utente})
MERGE (l:Libro {nome_libro:row.nome_libro})
CREATE (u)-[:ha_letto{rating:row.rating}]->(l);
```

Sul repository github è spiegato come inizializzare neo4j e cassandra.

ANALYSIS

Questa sperimentazione contiene alcune query di analisi per un graph database in neo4j
Il grafo è composto da due tipi di nodi - : Person e : Libro.
I nodi sono connessi tra loro tramite relazioni direzionali di tipo ha_letto da nodi : Person a nodi : Libro. La relazione ha un attributo rating.



QUERY 1

Per ogni persona restituire il numero di libri letti - ordinato in modo decrescente

```
MATCH (n:Person)
WITH n, size((n)-->()) AS count
ORDER BY count DESC
RETURN n, count;
```

QUERY 2

Restituire i 3 libri più letti e il numero di utenti che li hanno letti

```
MATCH (n:Libro)
WITH n, size((n)<--()) AS count
ORDER BY count DESC
RETURN n, count
LIMIT 3;
```

QUERY 3

Restituire i 3 libri che è sono stati più apprezzati - ovvero quelli col rating medio più alto

```
MATCH ()-[r]->(n:Libro)
RETURN n, AVG(r.rating) as rating_medio
ORDER BY rating_medio DESC
LIMIT 3
```

QUERY 4

Restituire tutte le coppie di utenti simili - ovvero utenti che hanno recensito lo stesso libro.
Restituire anche il nome del libro.

```
MATCH (a:Person)-->(n:Libro)<--(b:Person)
RETURN a.nome, b.nome, n.nome_libro
```

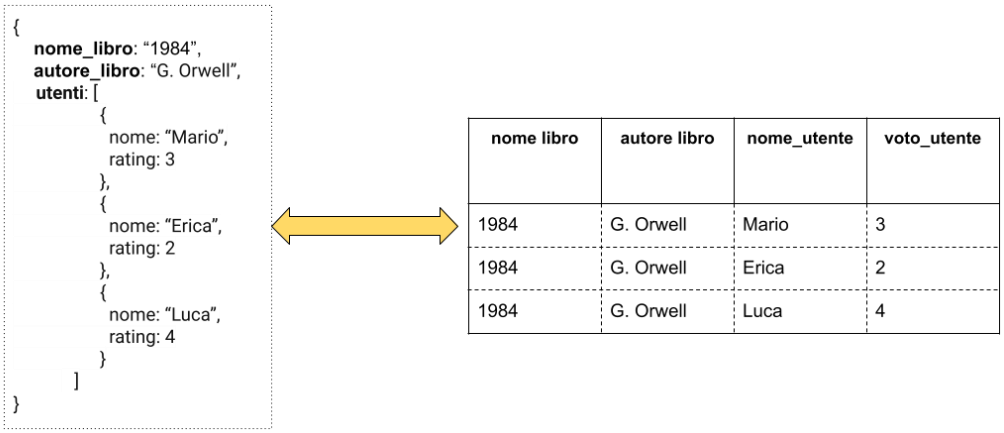
TOWARDS POLYGLOT DATA STORES – POLYBASE

Polybase è un potente strumento messo a disposizione da Microsoft per l'import, export e query dei dati di una organizzazione.
Esso permette di virtualizzare i dati: è possibile interrogare i dati di sorgenti esterne in una istanza SQL Server senza che questi siano effettivamente memorizzati nell'istanza stessa – e senza che abbiano specificatamente una struttura relazionale.

Al solo titolo di esempio si mostra come sia possibile realizzare una virtual table in SQL Server 2019 per virtualizzare in modo relazionale i dati contenuti all'interno di un document store – che nel caso specifico è MongoDB.
Questa operazione porta diversi vantaggi che sono descritti brevemente nel seguito

- la virtualizzazione non è la copia dei dati.
  I dati risiedono in una unica locazione fisica e non si ha la ridondanza dovuta alla duplicazione in diverse sorgenti. Inoltre i dati sono disponibili immediatamente senza attendere il tempo di una eventuale copia.

- Spesso nei PDW - Parallel Data Warehouse, piattaforme di Microsoft per la gestione dei big data – si necessita l'accesso e la scrittura di dati in Hadoop. Il query optimizer di Polybase può decidere se sfruttare le caratteristiche intrinseche della sorgente esterna dei dati al fine di miogliorarne le performance




```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'password@123';

CREATE DATABASE SCOPED CREDENTIAL myMongoDBCredential
WITH IDENTITY = 'admin', Secret = 'admin123';

CREATE EXTERNAL DATA SOURCE MongoDBSource
WITH (
     LOCATION = 'mongodb://127.0.0.1:27017',
     CREDENTIAL = myMongoDBCredential,
     CONNECTION_OPTIONS = 'ssl=false;'
);

CREATE EXTERNAL TABLE mongodbtable (
    nome NVARCHAR(MAX) NULL,
    autore NVARCHAR(MAX) NOT NULL,
    nome_utente  NVARCHAR(MAX),
    voto_utente INT )
WITH (
    LOCATION='admin.utenti_libri',
    DATA_SOURCE= MongoDBSource
);
```