ALMA MATER STUDIORUM • UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA - SCUOLA DI SCIENZE

Corso di Laurea in Scienze e Tecnologie Informatiche

DBMS BASATI SUI GRAFI:

ANALISI E PROTOTIPAZIONE DI NEO4J

Relazione finale in

Laboratorio di Basi di Dati

Relatore	Correlatore	Presentata da
Chiar.mo Prof. Matteo Golfarelli	**Dott. Simone Graziani**	**Matteo Torta**

Sessione II°

Anno Accademico 2013/2014

SOMMARIO

Introduzione

I sistemi informativi (SI) sono una componente centrale delle aziende e consentono di rivoluzionare i processi produttivi con lo scopo di migliorarne l'efficienza e la produttività. Un sistema informativo è composto dalle informazioni utilizzate, gestite e prodotte da una organizzazione. Questi dati devono poter descrivere qualsiasi sfaccettatura del mondo reale, e per far si che un sistema informativo gestisca al meglio queste informazioni, esse dovranno essere immagazzinate ed organizzate.

Data la crescente mole e varietà di informazioni che questi sistemi devono manipolare, è nata l'esigenza di appoggiarsi a sistemi informatici muniti di tecnologie di immagazzinamento dei dati sempre più efficaci e capaci di rappresentare ogni possibile aspetto della vita reale, a questo scopo sono nate le basi di dati. In informatica, il termine **database [1]**, base di dati o banca dati (a volte abbreviato con la sigla DB), indica un archivio dati, o un insieme di archivi ben strutturati, in cui le informazioni in esso contenute sono strutturate e collegate tra loro secondo un particolare modello logico. Il compito di reperire e manutenere i dati di questi archivi è affidato a delle particolari tecnologie che prendono il nome di **Database Management System** (**DBMS**). La gestione delle informazioni è la chiave di volta che a permesso all'informatica di avanzare e svilupparsi nel corso degli anni, permettendo così la nascita di una innumerevole quantità di tecnologie.

Quasi sin dalla nascita dei database, il modello relazionale è stato sicuramente quello di maggior successo e che meglio ha permesso di rappresentare e strutturare i dati. Esso consente di organizzare le informazioni schematizzandole sotto forma di entità connesse da relazioni. Con questo modello logico è stato possibile descrivere al meglio quasi l'intera totalità dei casi d'uso che si presentavano nel mondo reale. Tuttavia, la sua natura gli impone che l'organizzazione dei dati segua una serie di vincoli e regole che non consentono alla struttura di archiviazione di adattarsi a dei cambiamenti imprevisti. Questi vincoli sono la sua più grande forza e al tempo stesso la causa della sua debolezza.

Con l'avvento di Internet agli inizi deli anni '90 e più recentemente con l'affermarsi dei Social Network come Facebook e Twitter, il mondo d'oggi è sempre più connesso e sempre più tipi di informazioni vengono correlate tra loro. Quest'ultime sono sempre meno sono soggette a quei vincoli che permettevano di definire una struttura di base alla quale attenersi.

Colossi tecnologici come Google, lo stesso Facebook e altri come Ebay, hanno un po' abbandonato la "via del relazionale" per appoggiarsi a diversi tipi modelli. I dati elaborati e prodotti da queste multinazionali, sono sicuramente l'esempio più lampante che meglio può far comprendere le problematiche che derivano nel gestire un realtà fatta di connessioni. In un mondo così altamente connesso e in costante evoluzione non poteva che verificarsi la nascita di tecnologie di immagazzinamento dei dati capaci di adattarsi a questa nuova era.

I Graph Database e di conseguenza i Graph DBMS sono sicuramente la risposta più forte che è stata data dal mondo dell'informatica al nascere di queste nuove esigenze. Perciò, il modello relazione incomincia a vacillare e a perdere il suo primato di miglior metodo di rappresentazione dei dati, e nuove tecnologie incominciano a proporsi fortemente come sua alternativa.

Dato che il mondo commerciale è ancora fortemente legato agli **RDBMS** (**Relational Database Management System**), ovvero a quelle tecnologie che sfruttano il modello relazionale come logica di base nel salvare e gestire le informazioni, e dato che per ora solo le grandi compagnie possono permettersi di sviluppare ed effettuare degli studi accurati sui Graph DBMS, è assai scarsa la conoscenza generale che si ha di quest'ultime di tecnologie.

L'obiettivo di questa tesi è, appunto, quello di mettere a confronto due mondi: quello dei DBMS relazionali e quello dei DBMS a grafo, con lo scopo di comprendere meglio queste nuove tecnologie che giorno dopo giorno rafforzano la loro presenza sul mercato internazionale. Per poter raggiungere questo arduo obiettivo, si è deciso di scegliere come cavie di studio, le due tecnologie che meglio rappresentano i loro mondi: Oracle per gli RDBMS e Neo4j per i Graph DBMS. I due DBMS sono stati sottoposti ad una serie di interrogazioni atte a testare le performance al variare di determinati fattori, come la selettività, il numero di join che Oracle effettua, etc.

I test svolti si collocano nell'ambito business intelligence e in particolare in quello dell'analisi **OLAP - On-Line Analytical Processing**. Quest'ultimo è il paradigma principale impegnato per effettuare l'analisi interattiva e veloce di grandi quantità di informazioni. In una tipica sessione OLAP l'utente richiede un insieme di misure corrispondenti ad una certa prospettiva di analisi e, tramite una serie di operazioni, trasforma l'interrogazione iniziale fino ad arrivare ad un risultato per lui più interessante. Ovvero, le tecniche OLAP vengono impiegate, ad esempio, per analizzare i risultati delle vendite di un'azienda, l'andamento dei costi di acquisto merci, per misurare il successo di una campagna pubblicitaria, etc. Molto spesso accade che il database su cui viene effettuata un'analisi per mezzo di tecniche OLAP, proviene da un contesto OLTP - **Online Transactional Processing**. Gli strumenti OLAP si differenziano dagli OLTP per il fatto che i primi hanno come obiettivo la performance nella ricerca e il raggiungimento di un'ampiezza di interrogazione quanto più grande possibile; i secondi, invece, hanno come obiettivo la garanzia di integrità e sicurezza delle transazioni.

Il seguito della tesi è così organizzato:

- Il primo capitolo ha l'obiettivo di fornire delle informazioni che consentano al lettore di comprendere il mondo dei Graph DBMS, per poi affrontare al meglio i capitoli successivi.
- Nel secondo capitolo viene presentato Neo4j, mettendo in risalto le caratteristiche principali, la struttura di base, le strutture dati di rifermento e il suo linguaggio di interrogazione.
- Il terzo capitolo presenta i test effettuati, i quali vengono discussi in modo tale da mettere in risalto le caratteristiche della nuova tecnologia. In particolare vengono presentati i database su cui sono stati effettuati i test, i risultati dei test e la discussione di questi.
- Infine nel capitolo conclusivo si riassume quanto già detto precedentemente.

Capitolo I

Graph DBMS

In questo capitolo vengono introdotti i DBMS NoSQL, fornendo al lettore una panarmonica sulle loro caratteristiche principali, e le principali tipologie (sezione 1.1). Successivamente verrà presentato il mondo dei Graph DBMS, illustrando le caratteristiche e i concetti di base di questo nuovo modo di strutturare ed organizzare le informazioni (sezione 1.2).

1.1 I DBMS NoSQL

Negli ultimi anni è incredibilmente aumentata la popolarità delle tecnologie di immagazzinamento di informazioni conosciute con il nome di NoSQL, acronimo che sta per *Not only SQL* **[7]**. Ma cosa sono di preciso queste tecnologie? I NoSQL Database Management System sono sistemi software che consentono di immagazzinare e organizzare i dati senza fare affidamento sul modello relazionale, solitamente impiegato da database tradizionali .

I NoSQL DBMS sono inoltre contraddistinti dal fatto che non utilizzano un sistema transazionale ACID, il quale garantisce che ogni sua transazione soddisfi le seguenti proprietà **[6]**:

- **Atomicity** - una transazione è un'unità di elaborazione atomica, indivisibile. Ciò significa che dovrà essere eseguita totalmente oppure per niente, senza scinderla in parti più piccole.
- **Consistency** - quando viene lanciata, una transazione trova il database in uno stato consistente, al suo completamento il Database dovrà ancora godere di questa proprietà.
- **Isolation** - una transazione dovrà essere isolata completamente dalle altre. In caso di fallimento non dovrà interferire con le altre transazioni in esecuzione.
- **Durability** - gli effetti di una transazione che ha terminato correttamente la sua esecuzione devono essere persistenti nel tempo.

Infine, spesso questi tipi di DBMS sono *schema-less* **[7]**, ovvero non possiedono uno schema fisso a cui devono attenersi, evitando spesso così le operazioni di join e puntano a scalare orizzontalmente.

Le principali categorie di DBMS NoSQL sono **[7]**:

- **Key-Value store**.
- **Document-oriented**.
- **Column Family store**.
- **Graph DBMS**.

1.2 I Graph DBMS

Numerosi progetti e prodotti per la gestione, l'elaborazione e l'analisi dei grafi sono apparsi negli ultimi anni. Questa grande quantità di tecnologie rende difficile tener traccia di questi strumenti e come essi si differenziano, anche per coloro che da tempo lavorano in questo campo.

Tuttavia, il mondo dei Grafi, se visto dall'alto, è possibile divederlo in due macro categorie:

1. *Tecnologie impiegate principalmente per "transactional online graph persistence", tipicamente accedute per mezzo di applicazioni realtime*. Queste tecnologie vengono chiamate **Graph DBMS**. Esse sono l'equivalente dei sistemi OLTP del mondo relazionale. Questi sistemi sono caratterizzati da numerose ma semplici e veloci transazioni eseguite, spesso, in maniera concorrenziale **[2]**.
2. *Tecnologie impiegate principalmente per l'analisi Offline dei grafi. Solitamente eseguite come una serie di batch step*. Queste tecnologie vengono chiamate **Graph Compute Engine [2]**.

1.2.1 Graph Compute Engine

Un Graph Compute Engine è una tecnologia che permette di eseguire Algoritmi Computazionali su Grafi Globali sopra grandi dataset **[2]**. I graph compute engine sono progettati per eseguire operazioni come identificare clusters all'interno dei dati, oppure rispondere a delle domande come "Qual è la media della relazioni che posseggono gli utenti di una social network?" .

A causa dell'imponenza delle interrogazioni, i graph compute engine sono ottimizzati per scandire e processare enormi quantità di blocchi di informazione.

1.2.2 Graph DBMS

Un Graph Database Management System **[2]** è un sistema di gestione online che sottopone un modello dati a grafo, a metodi di Creazione, Lettura, Aggiornamento e Cancellazione (Create, Read, Update e Delete : CRUD). I Graph DBMS sono generalmente costruiti per sistemi transazionali OLTP. Di conseguenza vengono progettati in modo da ottimizzare le prestazioni e l'integrità delle operazioni transazionali.

Vi sono due componenti da tener in mente quando si si vuole analizzare una tecnologia di questo genere:

Underlying Storage

Sebbene sia scontato pensare che questi sistemi posseggano ogni loro componente proiettata verso il modo dei grafi, in realtà solo qualche Graph DBMS utilizza dei *native graph storage,* ovvero, possiede una piattaforma di salvataggio delle informazioni sottostante nata ed ottimizzata per salvare i dati sotto forma di grafo. Diversi graph DBMS, in effetti, traducono e salvano le informazioni in modi differenti, ovvero all'interno di un database relazionale, di un database orientato agli oggetti, o qualche altro tipo di data store.

Processing Engine

Le definizioni fornite dal mondo dei grafi richiedono che, per essere considerati tali, i Graph DBMS debbano utilizzare l'*index-free adjancency* (questo significa che ogni elemento contiene un puntatore diretto ai suoi elementi adiacenti rendendo così le ricerche via indice non necessarie). Tuttavia, come detto in precedenza, è possibile espandere la definizione di Graph Database Management System a tutti quei DBMS che permettano di eseguire delle operazioni CRUD su un modello dati a grafo. Ciò significa che, possiamo distinguere i graph DBMS in due categorie, la prima tutti quelli che sfruttano la l'index-free adjancency , processing engine nativo (più performante); la seconda quelli che non la usano, processing engine non nativo.

La figura 1.1 permette di avere un'idea delle tecnologie presenti oggi sul mercato **[2]**

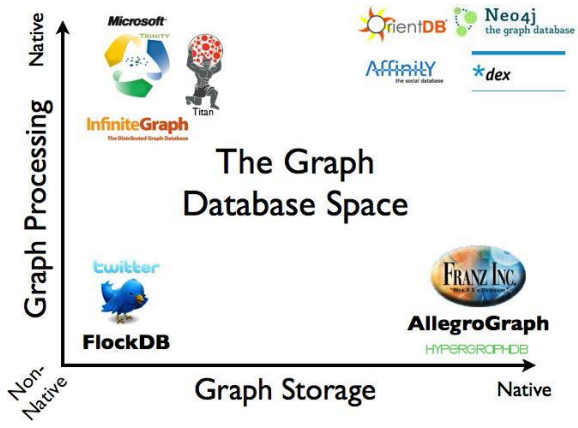

Figura 1.1: Serie di Graph DBMS presenti oggi sul mecato.

1.2.3 Che cos'è un Grafo?

I Graph DBMS organizzano le informazioni sotto forma di grafo, perciò è naturale chiedersi cosa sia un grafo **[2]**.

Sono strutture espressive che ci permettono di modellare tutti i tipi di scenari. Un grafo è una raccolta di ***vertici*** e ***archi*** , in parole semplici, è un insieme di ***nodi*** connessi da ***relazioni***. I grafi rappresentano le entità con i nodi, e il modo nel quale queste entità si rapportano con il mondo, con le relazioni.

Esempio: Le informazioni di Twitter possono essere rappresentate come un grafo. L'immagine sottostante rappresenta una piccola rete di followers.

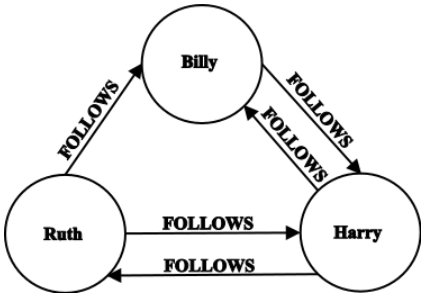

Figura 1.2: Grafo che rappresenta una catena di followers **[2]**.

Le relazioni sono la chiave per comprendere la semantica del contesto (dicono chi segue chi e chi è seguito da chi).Ovviamente, il vero grafo di twitter è centinaia di milioni di volte più grande dell'esempio. La figura 1.3 mostra il potere espressivo del modello a grafo.

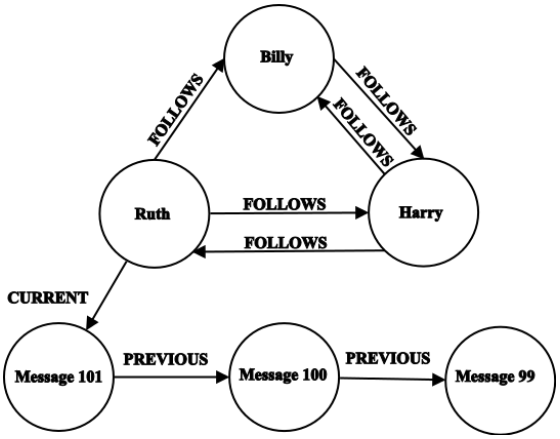

Figura 1.3: Grafo che rappresenta una catena di followers, conl'aggiunta dei messagi **[2]**

E' facile notare che Ruth ha pubblicato una sequenza di messaggi. Il più recente messaggio può essere trovato seguendo la Relazione CURRENT; la relazione PREVIOUS verrà creata mentre viene eseguito un post.

1.2.4 Il Property Graph Model

Esistono svariate tipologie di grafo e il mondo della teoria dei Grafi fornisce un'infinità di soluzioni, tuttavia l'attenzione verrà posta su un solo particolare tipo di modello, il **property graph model [2]**.

Un **property graph model** è così definito:

- Un grafo contiene nodi e relazioni.
- I nodi posseggono delle proprietà (coppie di chiave-valore).
- Le relazioni posseggono un nome e sono direzionate, ed hanno sempre un nodo di partenza e un nodo di arrivo.
- Anche le relazioni possono avere delle proprietà.

Molte persone trovano il property graph model intuitivo e facile da capire. Sebbene semplice, può essere usato per descrivere la stragrande maggioranza dei casi d'uso grafici, in modo tale che diano indicazioni utili sui nostri dati.

1.2.5 Le potenzialità dei Graph DBMS

Qualsiasi cosa può essere modellata in un grafo, e i graph DBMS fornisco delle potenti ed originali tecniche di modellazione dei dati. Essi offrono un modello dati flessibile e agile che permette di adattarsi continuamente all'evolversi della realtà. Ecco quelle che sono le loro potenzialità **[2]**.

Performance

Le performance dei Graph DBMS tendono ad essere ottimali quando i dati da archiviare sono altamente connessi e la mole del dataset è estremamente grande. Al contrario degli RDBMS (Relational Database Management System), la loro natura gli consente di evitare le onerose operazioni di join semplicemente attraversando le relazioni che connettono i nodi.

Flessibilità

I graph DBMS sono *Schema-less*, in altre parola non posseggono uno schema prefissato al quale attenersi. La loro natura gli permette di adattarsi all'evolversi del dominio applicativo senza dover rimodellare e convertire l'intera base dati. Inoltre, l'aggiunta di nuove relazioni e nodi non compromette le interrogazioni che sono state costruite per la vecchia versione del databse.

1.2.6 La Modellazione di un Grafo

Essendo i Graph DMBS una tecnologia recente, non esiste ancora una precisa e ben consolidata tecnica di modellazione. In effetti, si può affermare che nessuno possiede la "ricetta perfetta" della modellazione di uno schema a grafo. Esistono teorie differenti e a volte contrastanti sul come dovrebbe essere la tecnica di modellazione, il più delle volte essa prevede la creazione uno schema E-R in tutto per tutto.

Lo schema E-R, pur essendo la base di partenza delle tecniche di modellazione di un base dati relazione, è sicuramente il diagramma che più si avvicina al property graph model.

Pur non essendo presenti teorie ben consolidate, verrà illustrata la più accreditata ed utilizzata delle tecniche di modellazione.

Tecnica di Modellazione di un Grafo

Fase 1 - Analisi : Nelle prime fasi dell'analisi, il lavoro richiede di avere un approccio simile a quello del modello relazionale: utilizzando metodi lo-fi (a bassa fedeltà, poco professionale) viene data una descrizione approssimativa del dominio, ma che permetta di avere un'idea di come sarà poi strutturato il nostro modello finale. In questa fase viene creato un modello molto simile allo schema E-R.

Fase 2 - Arricchimento: Dopo aver fatto ciò, invece di trasformare le entità del modello in tabelle, ossia creando quello che viene chiamato Modello Logico, lo arricchiamo, con l'obiettivo di produrre un'accurata rappresentazione degli aspetti salienti del dominio. Ovvero, creiamo dal nostro schema E-R, simile ad un grafo, un modello a grafo arricchito di proprietà e relazioni che cerchi di descrivere al meglio il dominio del problema.

In molti casi in aggiunta allo schema "arricchito", si decide di non progettare uno schema generalizzato, ma rappresentare un tipico caso d'uso che permetta di dare una descrizione globale del dominio. Ossia, ci si baserà su uno schema che mostra i valori delle singole entità e delle loro relazioni, esattamente il contrario di quello che viene fatto per un database relazionale, ovvero verrà utilizzata una sotto-istanza del dominio per descriverlo al meglio.

La modellazione nella pratica: The Movie Graph

Per meglio comprendere questa tecnica verrà ora mostrato un esempio di modellazione.

Dominio: "Si vuole rappresentare il mondo cinematografico e come i vari componenti principali nella produzione di un film si relazionano con esso.

Ogni film possiede una serie di attori che interpretano un ruolo, dei produttori, registi, scrittori e chi lo ha revisionato."

Fase 1 - Analisi: Si cercherà di rappresentare per mezzo di uno schema simile all'entità relazioni la struttura di base del dominio, esso dovrà essere approssimativo e orientato alle relazioni.

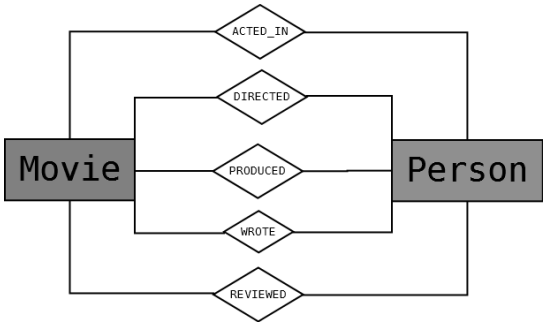

Figura 1.4: Modello simile all'E-R, prodotto dall'anailisi 1.

Fase 2 – Arricchimento:

Dallo schema nato nella fase precedente, ne verrà creato uno nuovo più ricco di informazioni e che meglio descriva la natura del dominio applicativo.

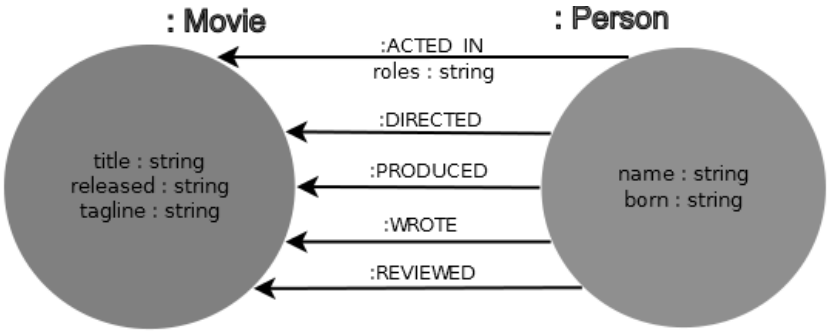

Figura 1.4: Modello generalizzato, prodotto nella fase 2.

La figura soprastante mostra il modello arricchito di particolari e permette si di avere una visione della struttura generale del grafo, ma non consente di avere una vera comprensione ”dell’aspetto” finale del database.

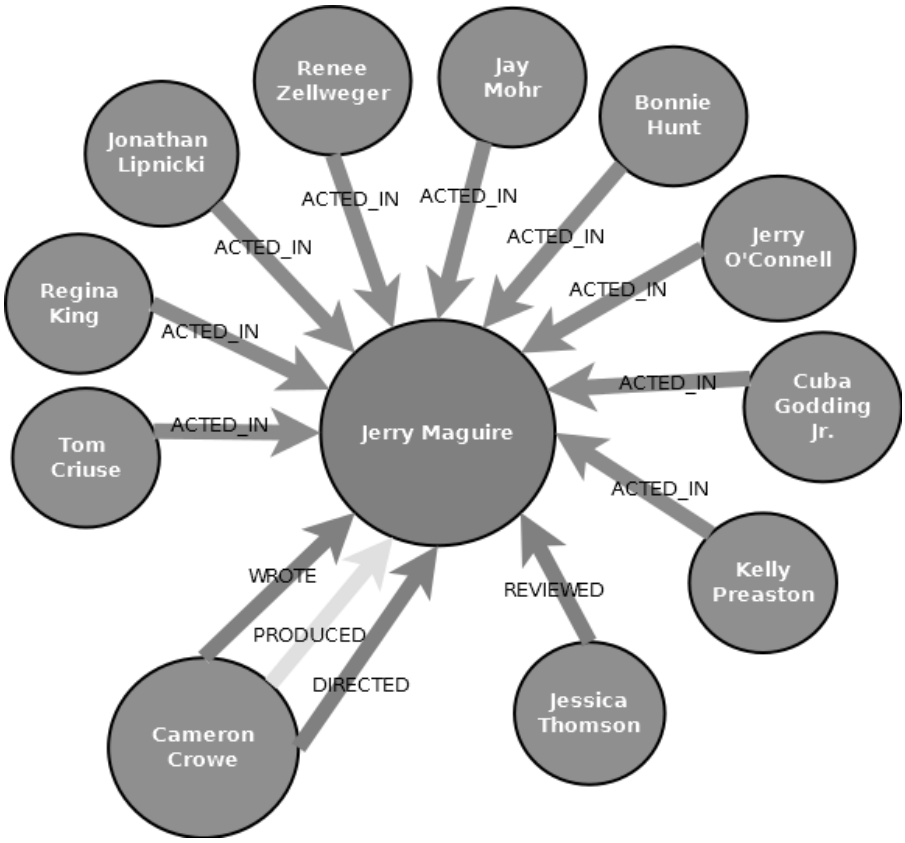

Figura 1.5: Rappresentazione di una sotto-istanza del database

Con quest’ultimo modello si può meglio comprendere la natura del database, anche se viene rappresentata solamente una sotto istanza del dominio.

Capitolo II

Neo4j

In questo capitolo viene descritto nel dettaglio Neo4j. Inizialmente – con la sezione 2.1 – viene data una descrizione di massima della tecnologia, con l'obiettivo di fornire delle conoscenze di base che permettano di affrontare meglio le parti successive del capitolo. Nella sezione 2.2 viene descritta l'architettura della tecnologia NoSQL. Con la 2.3 vengono mostrate le caratteristiche del modello dati a cui Neo4j fa affidamento. Con le sezioni 2.4 e 2.5, vengono descritte le strutture dati che il Graph DBMS sfrutta per raggiungere facilmente le informazioni e per mantenerle in ordine. Con le sezioni 2.6, 2.7, 2.8 e 2.9 viene presentato il linguaggio di interrogazione di Neo4j, e vengono descritte alcune funzionalità di impiego del linguaggio di interrogazioni. Infine, con la sezione 2.10, vengono proposti degli impieghi futuri nell'analisi OLAP.

2.1 Presentazione

Neo4j **[1]** è un Graph DBMS open source transazionale, prodotto dalla software house Neo Technology. Possiede processing engine e underlying storage nativi ed è sviluppato completamente in Java. É robusto, scalabile e ad alte prestazioni. È dotato di:

- Transazioni ACID,
- High Availability,
- può memorizzare miliardi di nodi e relazioni,
- alta velocità di interrogazione tramite attraversamenti,
- linguaggio di interrogazione dichiarativo e grafico.

È un DBMS schema-less, ciò sta a significare che i suoi dati non devono attenersi al alcuna struttura di rifermento prefissata, inoltre non possiede una politica di accesso controllata .

La index-free adjancency è alla base delle sue alte prestazioni di attraversamento, d'interrogazione e di scrittura, ed è uno degli aspetti chiave della sua architettura. L'index-free adjancency è una lista (o tabella), ove ogni suo elemento è composto da un nodo del grafo e dai puntatori ai nodi connessi ad esso.

Neo4j salva i dati dentro di una serie di *store file*, contenuti all'interno di un'unica cartella. Ognuno di questi file contiene al suo interno le informazioni relative ad una singola parte del grafo (e.g. nodi, relazioni, proprietà). Questa separazione della struttura del grafo facilita il suo attraversamento.

2.2 Architettura

La figura sottostante mostra l'architettura di base di un server Neo4j.

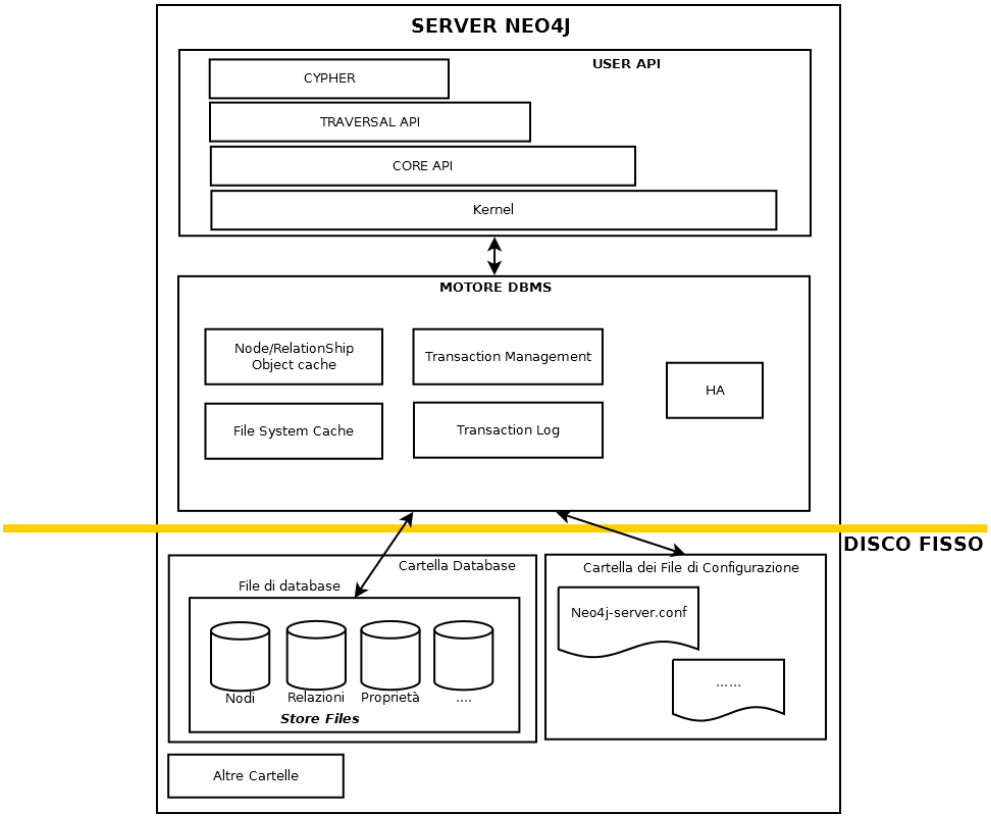

Figura 2.1: Architettura interna di Neo4j.

Tutti i dati e le informazioni del grafo che il server storicizza e gestisce vengono salvate all'interno di una serie di file che prendono il nome di S*tore File*, i quali vengono memorizzati all'interno di un'unica cartella, detta *Cartella di Database*. Ogni database o grafo possiede una propria Database Directory, e un server può gestire una sola di queste cartelle per volta. Prima di avviare il server è possibile definire da quale cartella caricare il grafo, modificando un dei file di configurazione presenti nell'albero cartelle del server (conf/Neo4j-server.conf).

Gli Store File di un grafo sono innumerevoli, ma le informazioni che ne descrivono la struttura e i dati che esso contiene sono essenzialmente tre:

- *neostore.relationshipstore.db* per le relazioni;
- *neostore.propertystore.db* per le proprietà;
- *neostore.nodestore.db* per i nodi.

2.2.1 Store File

Ogni elemento salvato all'interno degli Store File **[2]** possiede una struttura dati di memorizzazione con lunghezza fissa detta *Record*.

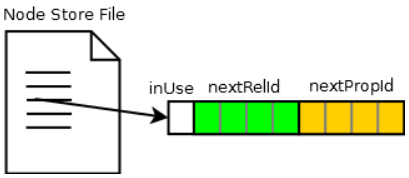

Figura 2.2 : Struttura del record dei nodi.

La figura soprastante mostra la struttura di un record dello Store File dei nodi, il quale è lungo 9 byte. Il primo byte rappresenta un flag che indica se il record è impiegato o meno per salvare i dati di un nodo, i sucessivi quattro byte rappresentano l'ID della prima relazione connessa al nodo, i restanti byte rappresentano l'ID della prima proprietà del nodo.

Il flag-byte è un denominatore comune dei record degli Store File di Neo4j. Esso consente a Neo4j di riciclare gli ID: quando viene creato un nodo, se è presente un record non utilizzato, esso verrà impiegato per salvare i dati del nuovo nodo, altrimenti verrà creato un nuovo record da posizionare in fondo al file. La lunghezza fissa dei record permette a Neo4j di effettuare delle ricerche velocissime: qual ora si voglia ricercare il nodo

con id 100 basterà scorrere i primi 900 byte del file.

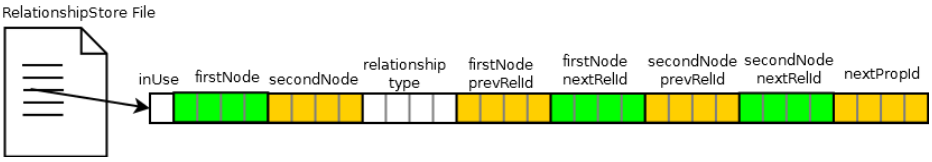

Figura 2.3 : Struttura del record delle relazioni.

I record delle relazioni sono lunghi 33 byte. Ogni record contiene gli ID del nodo di partenza e di arrivo, il puntatore al tipo di relazione, i puntatori ai record della precedente e prossima relazione del nodo di partenza e di arrivo. Gli ultimi 4 byte contengono l'ID della prima proprietà della relazione.

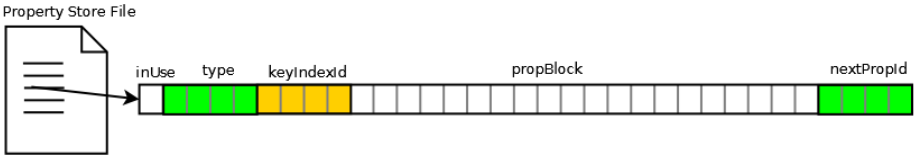

Figura 2.4 : Struttura del record delle proprietà

I record delle proprietà sono anch'essi lunghi 33 byte e sono composti dal puntatore al tipo di proprietà, dal puntatore all'indice, dall'Id della successiva proprietà dell'elemento a cui appartiene e da un blocco di memorizzazione. Quest'ultimo parte conterrà il valore assunto dalla proprietà del nodo o della relazione solo qual ora si trattasse di valori di piccole dimensioni, nel caso di lunghe stringhe ed array, i valori verranno salvati in uno Store File a parte.

Questa suddivisione fisica dei dati e il modo in cui sono memorizzati all'interno degli Store File è la chiave che sta alla base delle alte prestazioni di attraversamento di questo DBMS.

2.2.2 Cache

Neo4j **[3]** possiede due cache di diversa tipologia:

- *File System Cache*;
- *Object Cache*;

La File System cache agisce sugli Store File, caricandosi in memoria porzioni di essi. Ogni Store File viene diviso in un numero sempre uguale di parti, tutte della stessa grandezza. La politica di rimpiazzo delle porzioni di file è simile alla LRU (*Last Recently Used*).

L'Object Cache agisce ad alto livello, essa mantiene in memoria porzioni di grafo che sono state caricate precedentemente dal file sytem e non sono sotto forma di record degli Store File, ma in una che consente di migliorare la velocità di attraversamento del grafo. Per la precisione il contenuto di questa cache sono oggetti (Objects) con una rappresentazione orientata a sostenere le API di Neo4j e gli attraversamenti del grafo. Le operazioni di lettura possono essere dalle 5 a 10 volte più veloci rispetto a quelle della File System Cache. Neo4j permette di abilitare o meno questa cache, sempre agendo sui file di configurazione del server. L'Object Cache è composta a sua volta da due cache:

- *References Cache*;
- *High-Performance Cache*.

La prima cercherà di utilizzare la maggior parte delle *Java Virtual Machine Heap Memory* messa a disposizione, ed impiega una politica di rimpiazzo LRU. Questa cache sfrutta le porzioni di memoria in comune con le altre applicazioni della stessa JVM, quindi essa sarà in costante "competizione" per lo spazio. Ciò sta a significare che verrà tolta della memoria a tutti quegli oggetti che con lei condividono la JVM, ad esempio: oggetti intermedi prodotti dalle *Query Cypher* (il linguaggio di interrogazione di Neo4j), altre applicazioni create dall'amministratore del server, altre applicazioni prodotte dallo stesso server Neo4j.

L'High-Performance Cache è disponibile solamente nella versione enterprise del server. Ad essa viene adibita una porzione di memoria della JVM completamente dedicata, però di dimensione massima limitata. Questa cache carica in memoria porzioni di grafo fino ad raggiungere il limite massimo, una volta raggiunto sarà essa a rimpiazzare gli oggetti, invece di affidarsi al Garbage Collection della JVM. L'overhead della High-Performance cache è molto più piccolo rispetto alla References Cache, così come i tempi di inserimento e di ricerca.

Infine, è possibile usufruire della cache del sistema operativo che ospita il server Neo4j, anche questa volta modificando i file di configurazione.

2.2.3 Transaction Management

Il Transaction Management e il Transaction Log, racchiudono in se tutti quei meccanismi che hanno il compito di garantire le proprietà ACID alle transazioni di Neo4j.

Tutte le operazioni che vengono effettuate sul database, dall'accesso al grafo all'utilizzo di un indice, vengono eseguite all'interno di una transazione. Le transazioni di Neo4j sono di tipo "*flat nested transactions*", ovvero ogni transazione può racchiudere al suo interno una transazione annidata (di livello inferiore), la quale se non completata correttamente può comportare il *rollback* della transazione madre a cui appartiene e di tutte le altre transazioni da cui dipende quest'ultima.

Qui sotto viene riportato un tipico ciclo di iterazione **[3]** che descrive il modo in cui lavorano le transazioni:

1) Inizio della transazione.
2) Esecuzione delle operazioni sul database.
3) Segnalazione dell'avvenuto successo o meno della transazione.
4) Fine della transazione.

E' fondamentale completare una transazione perché, fino al suo completamento, essa non rilascerà i *lock* (blocchi) acquisiti sugli oggetti del database, e non libererà la memoria della JVM occupata da tutti quegli oggetti che vengono modificati, creati e cancellati. E' bene suddividere le grandi transazioni in altre più piccole in modo tale da non esaurire la JVM Heap Memory a diposizione del server.

Il Transaction Log è un processo che gestisce un "diario", all'interno del quale vengono annotate tutte le modifiche apportate dalle transazioni. Questo componente di Neo4j è fondamentale qual ora si voglia mantenere il proprio database in una stato consistente e coerente anche in caso di una Failure, la quale costringe Neo4j ad eseguire un rollback. Il Transaction Log può essere abilitato o meno modificando i file di configurazione del server.

2.2.4 API

Anche se il filesystem e le infrastrutture di caching sono molto affascinanti, i programmatori raramente interagiscono direttamente con esse, ma preferiscono appoggiarsi ad altri strumenti, quest'ultimi sono le API **[2]**. Le *Application Programming Interface - API* solitamente sono una serie di procedure / librerie messe a disposizione del programmatore, le quali gli permettono di interagire con un tecnologia sfruttando un linguaggio di programmazione o altro, senza dover conoscere e gestire i meccanismi interni della tecnologia in questione.

Neo4j mette a diposizione diverse API, e la scelta di impiegare una invece che un'altra dipende dal tipo di utilizzo che se ne vuole fare di questa tecnologia.

Kernel API

Al più basso livello dello stack delle API di Neo4j, si trova il *Kernel Transaction Event Heandler*. Questa API consente al programmatore di interagire con il ciclo di vita delle transazioni e captarne gli eventi, permettendogli di modificare e gestire il risultato che una transazione produrrà. Un tipico caso d'uso è quando si vuole evitare che i nodi vengano eliminati fisicamente, a livello di record dello Store File, ma si intende eliminarli solo dal punto di vista logico, in modo tale da poterne recuperare i dati anche in un secondo momento.

Core API

La Core API di Neo4j, chiamata anche Beans API, è una Java Api imperativa che consente al programmatore di esporre il grafo a primitive di creazione, modifica, eliminazione ed interrogazione tramite codice Java. Questa API può essere realmente veloce, ma a patto che colui che la utilizza conosca in modo approfondito la struttura del grafo, la quale dovrà essere riproposta all'interno del codice Java. Questo sta a significare che, impiegando la Core API, il programma che ne nascerà sarà molto più vulnerabile alle variazioni del dominio applicativo che si presentano con il passare del tempo.

Traversal API

La Traversal API è una Java API dichiarativa. Al contrario della Core API, con la quale bisogna riproporre la struttura del grafo all'interno del codice Java, con la Traversal API è possibile interrogare il grafo semplicemente indicando i vicoli generali che permettono di limitare l'attraversamento. Ciò sta a significare che, invece di indicare nel codice di programmazione i particolari tipi di nodi, relazioni e proprietà che si vogliono estrarre, è possibile costruire le interrogazioni semplicemente ponendo come limiti dell'attraversamento, la struttura generale del sotto-grafo di interesse. Con questa API è possibile costruire interrogazioni più generalizzate, ma meno performanti.

Cypher

Cypher è il linguaggio di interrogazione nativo di Neo4j. Esso è un linguaggio grafico, ovvero si basa sulla riproduzione grafica del sotto-grafo che si vuole estrarre. Esso consente di creare, modificare, eliminate e interrogare i dati del database.

Il sotto-grafo riprodotto nelle query viene chiamato *pattern*, e per produrlo non servono strumenti particolari, ma basta seguire delle semplici regole che permettono di disegnarlo impiegando i caratteri ASCII (i caratteri presenti sulla tastiera).

In gergo tecnico la riproduzione grafica del sotto-grafo viene chiamata "**things like this**", è una frase che tradotta significa "cose come questa". Già si può meglio comprendere il concetto che sta alla base di della costruzione delle query Cypher. Più avanti verrà descritto in maggio dettaglio questo linguaggio di interrogazione.

Altre API

Oltre alle API appena descritte, Neo4j ne mette a disposizione altre di diversa natura. La REST API è sicuramente la più importante tra quelle non menzionate precedentemente. Essa fornisce una serie di funzionalità richiamabili per mezzo di richieste http di tipo POST e GET. Su di essa si basa l'interfaccia Web RESTful che consente all'amministratore di visionare lo stato del server e di eseguire diversi tipi di operazioni, dall'esecuzione di SCRIPT Cypher, alla creazione di indici.

2.2.5 Decisioni Architetturali

Quando si vuole costruire un sistema basato su un graph DBMS, vi sono diverse decisioni architetturali che devono essere effettuate **[2]**. Queste decisioni dipendono dal prodotto finale che si vuole ottenere. Neo4j fornisce una quantità di soluzioni che permettono di soddisfare gran parte delle esigenze.

Attualmente molti DBMS vengono eseguiti come applicazioni server a se stanti, le quali vengono accedute per mezzo di altri software costruiti con librerie client. Neo4j è un DBMS inusuale, perché è possibile incorporalo all'interno dei software client oppure eseguirlo nella maniera classica, in altre parole in modalità server.

Embedded Mode

In modalità Embedded, Neo4j viene eseguito all'interno del processo dell'applicazione che si sta costruendo. Embedded Neo4j è l'ideale per computer desktop oppure Hardware Device, addirittura può essere impiegato per costruire una propria applicazione che funga da server di database. Vediamo ora quali sono i vantaggi forniti da questo tipo di architettura.

Vantaggi:

Low Latency: I tempi di risposta da parte del database sono chiaramente rapidissimi, visto che quest'ultimo è una parte integrante dell'applicazione.

Scelta delle API: In questa modalità è disponibile la totalità delle API , per creare ed interrogare i dati: le Core API, il traversal framework, e il linguaggio di interrogazione Cypher.

Transazioni esplicite: Utilizzando le Core API, è possibile controllare il ciclo di vita transazionale, eseguendo arbitrariamente una complessa sequenza di comandi a carico del database, tutto all'interno di una singola transazione. Le Java API consentono di mettere a nudo il ciclo di vita della transazioni, permettendo di aggiungere una personale gestione delle transazioni via evento, in modo tale da poter aggiungere delle logica addizionale ad ogni transazione.

Named Indexes: L'Embedded Mode fornisce una controllo completo sulla creazione e la gestione di indici muniti di nome. Questa funzionalità è anche disponibile grazie alla web-based REST interface; Cypher non ne è capace.

Quando si esegue Neo4j in modalità Embedded è buona regola tener conto anche delle seguenti note.

JVM only: Neo4j è un database basato sulla Java Virtual Machine (JVM). Diverse delle sua API sono, tuttavia, accessibili solamente per mezzo del linguaggio base della JVM, in altre parole il Java.

Comportamento della Garbage Collection: Quando viene eseguito in Embedded Mode, Neo4j è soggetto al comportamento della Garbage Collection (GC) dell'applicazione che lo ospita. Lunghe pause dovute al GC si riflettono sui tempi di risposta delle query. Addirittura, può capitare a volte, che quando una istanza in Embedded mode fa parte di un cluster Neo4j High Available (HA), una lunga pausa da parte del GC può indurre il cluster a rieleggere il proprio master (quest'ultima parte risulterà più chiara una volta che verrà affrontata l'architettura Neo4j HA).

Database life cycle: L'applicazione host è responsabile del controllo del ciclo di vita del database. Il software che incorpora il database Neo4j deve essere in grado di lanciare e stoppare il database in modo coretto, controllando le varie problematiche che ne derivano.

Server Mode

Neo4j Server è la modalità più comunemente utilizzata attualmente. Il cuore di un database Neo4j server è un istanza di tipo Embedded. Ecco alcuni dei benefici derivanti da questo modalità di esecuzione.

Vantaggi

REST API: Il server è munito di una ricca REST API che permettono ai client di spedire richieste in formato JSON per mezzo del protocollo HTTP. Le risposte vengono restituite all'interno di documenti JSON arricchiti con Hypermedia Links che mettono in risalto ulteriori caratteristiche del dataset. Sono molteplici le funzionalità messe a disposizione dalla REST API, ma il suo più grande vantaggio è quello di potervi accendere per mezzo di una semplice applicazione browser, come Firefox, Chrome o Internet Explorer.

Platform Independence: Dato che le informazioni contenute nel server vengono accedute per mezzo di documenti JSON spediti attraverso l'HTTP, le applicazioni client possono essere costruite su qualsiasi tipo di piattaforma, basta possedere delle librerie client HTTP.

Scaling Independence: Quando neo4j viene eseguito in modalità server possiamo aumentare o diminuire il numero di componenti del cluster indipendente dal tipo di applicazione.

Isolamento dal comportamento del GC delle altre applicazioni: In modalità server, Neo4j è protetto dall'influenza che potrebbe avere la Garbage Collection (GC) di qualsiasi altra applicazione. Ovviamente, essendo Neo4j una tecnologia recente e basata sulla JVM, ancora produce qualche "garbage" (sta a significare che, nel momento in cui si conclude un qualche tipo di procedura interna al database, la memoria non viene completamente rilasciata da questi processi interni). Nel corso del tempo l'impatto di Neo4j sul garbage collector è stato attentamente monitorato, e durante lo sviluppo è stato ottimizzato per rendere minimo ogni effetto.

Quando si esegue Neo4j in modalità Embedded è buona regola tener conto anche delle seguenti note.

Network Overhead: Vi è un certo overhead di comunicazione per ogni richiesta http. Dopo la prima richiesta, la connessione TCP rimane aperta fino alla chiusura da parte del client.

Per-request transactional: Ogni richiesta da parte del client viene eseguita come una singola transazione, atomicamente separata dalle altre. Tuttavia, la REST API fornisce un supporto per l'esecuzione di operazioni in batch (ovvero l'esecuzione "accorpata" delle operazioni).

Clustering – Neo4j High Available

Qualora si voglia garantire che il proprio sistema sia in grado di fornire un servizio di erogazione dei dati continuo, senza *failure point* e in grado di bilanciare e gestire un'enorme mole di richieste, Neo4j High Available (HA) è la risposta a questa esigenza **[3]**. Neo4j HA è stato progettato per rendere semplici, le transazioni da una singola macchina ad un una macchina multipla, senza dover cambiare la tipologia delle istanze che andranno a comporre il cluster.

Consideriamo un istanza di database Neo4j esistente, presente all'interno di una singola macchina, già popolato e configurato a dovere. Per replicare tale applicazione in una macchina multipla (o cluster), l'unico cambiamento richiesto è quello di cambiare un semplice parametro di configurazione dell'istanza. Sia Neo4j stand alone che HA, implementano la stessa interfaccia, e non richiedono ulteriori modifiche.

Neo4j HA è in grado di fornire le seguenti funzionalità:

1. Fornisce una ***fault-tolerant database architecture***, nella quale diverse istanze, chiamate "Slave", vengono configurate per poter essere l'esatta copia di una singola istanza, detta "Master". Questo permette al sistema utente finale di essere completamente funzionale sia lettura che in scrittura in caso di un hadware failure, in altre parola nel caso una macchina che compone il cluster si "rompa".
2. Fornisce una ***horizontally scaling read-mostly architecture*** che permette al sistema di gestire meglio il carico di lettura di una singola istanza di database Neo4j.

Ogni componente di un cluster Neo4j possiede al suo interno una copia dell'intero database. Rispetto ad altre impostazioni *master- slave replication*, Neo4j è in grado di gestire le richieste di scrittura su tutte le macchine, cosicché non ci sia il bisogno di rindirizzare specificatamente le richieste al master.

Come è stato accennato in precedenza, un database Embedded può far parte di un cluster, come se fosse una versione server. Infatti, un cluster Neo4j può essere composto sia da istanze in Server Mode che in Embedded Mode. Questa architettura "ibrida" è comune in quegli scenari in cui un impresa vuole rendere il proprio sistema completamente integrato (Enterprise Integration); i regolari aggiornamenti che vengono eseguiti sule istanze Embedded di Neo4j, vengono a loro volta applicati sui server.

Come opera Neo4j HA

Un cluster Neo4j HA opera corporativamente e ogni istanza di database contiene la logica necessaria al fine di coordinarsi con gli altri membri del cluster. All'avvio un'istanza di database Neo4j HA cercherà di connettersi a un cluster esistente specificato in fase di configurazione. Se il cluster esiste, l'istanza si unirà come uno slave. In caso contrario, verrà creato il cluster e l'istanza diventerà il suo master. Quando Neo4j viene eseguito in HA mode, il cluster che ne nascerà sarà sempre composto da almeno un singolo master e zero slave.

Scrittura

Quando si esegue un'operazione di scrittura su uno slave, ogni azione sarà sincronizzata con il master, e dovranno essere acquisiti locks (blocchi) sia sul master che sullo slave. Nel momento in cui viene eseguito il commit della transazione, sarà innanzitutto completata sul master e poi, in caso di successo, sullo slave. Per garantire la coerenza, i dati dello slave dovranno essere sempre in linea con quelli del master prima di eseguire un'operazione di scrittura. È il protocollo di comunicazione tra lo slave e il master che consente di mantenere il sistema in uno stato di coerente, in modo che gli aggiornamenti vengano applicati automaticamente a uno slave che comunica con il suo master.

Le transazioni di scrittura che vengono eseguite direttamente sul master saranno eseguite come se quest'ultimo fosse in non-HA mode (normalmente). In caso di successo della transazione, essa verrà inviata (pushed out) ad un numero configurabile di slave (di default uno). Questa procedura viene fatta con "*ottimismo*", ovvero, qualora l'operazione di replica dei dati fallisca, è comunque garantita la durabilità delle informazioni, dato che sono presenti all'interno dell'istanza master. Scrivere direttamente sul master aumenta comunque i rischi di perdita delle transazioni non ancora completate, è buona regola comunicare solamente con gli slave.

Gestione delle Failure

Ogni volta che un'istanza Neo4j non è più disponibile, ad esempio per via di un guasto hardware o interruzioni della rete, le altre istanze del cluster sono in grado di rilevarlo e segnalarlo come temporaneamente *failed*. Una istanza di database che diventa disponibile dopo l'indisponibilità verrà automaticamente inserita nel cluster. Se il master viene meno, un altro membro (il più adatto) sarà eletto da slave a master dopo che il quorum sarà stato raggiunto all'interno del cluster. Quando il nuovo master avrà cambiato il suo ruolo informerà tutti i gli altri componenti. Normalmente un nuovo master viene eletto e diviene attivo nel giro di pochi secondi e durante questo periodo nessuna operazione di scrittura può avvenire, esse vengono bloccate e in rari casi viene lanciata un'eccezione. L'unica volta che questo accade è quando un vecchio master ha apportato dei cambiamenti prima di diventare indisponibile, e i cambiamenti non sono stati replicati su nessun altro membro del cluster . Se il nuovo master viene eletto ed esegue modifiche prima che il vecchio ritorni attivo, ci saranno due "diramazioni" del database dopo il punto in cui il vecchio master è divenuto indisponibile. Il master decaduto si porterà via il proprio database (la sua "diramazione") e scaricherà una copia completa del nuovo master, per poi diventare disponibile come slave.

Tutto questo può essere riassunto:

- Operazioni di scrittura possono essere eseguite su qualsiasi istanza di database di un cluster.
- Neo4j HA è fault tolerant e può continuare ad operare sia che risultino offline una serie di macchine oppure che ne rimanga anche solo una attiva.
- Gli schiavi saranno sincronizzati automaticamente con il master durante le operazioni di scrittura.
- Se il master diviene offline (viene meno) un nuovo master sarà eletto automaticamente.
- Il cluster gestisce automaticamente le istanze che divengono indisponibili (per esempio a causa di problemi di rete), e fa in modo di accettarli come membri del cluster anche quando sono di nuovo disponibili.
- Le transazioni sono atomiche, coerenti e durevoli, e poi eventualmente propagate ad altri slave.

- Gli aggiornamenti degli slave sono coerenti per natura, ma possono essere configurati per essere “spinti ottimisticamente” da un master durante il commit.
- Se il master diviene offline, qualsiasi operazione di scrittura in esecuzione verrà bloccata e verrà eseguito il rollback. Inoltre tutte le nuove transazioni verranno bloccate o fallite fino a quando un nuovo master tornerà disponibile.
- Le letture sono HA e la capacità di gestire i carichi di lettura scalano con l’aumentare delle istanze di database che compongono il cluster.

Arbiter (arbitro)

Sono particolari istanze di server Neo4j. Gli arbitri possono essere considerati come partecipanti al cluster e il loro ruolo è quello di prendere parte alle elezioni di un master con l'unico scopo di rompere i legami che bloccano il processo di elezione del master.

Scenario: Abbiamo un cluster in cui si dispone di un gruppo di due istanze di database Neo4j e un'istanza arbitro supplementare, il cluster ancora gode della tolleranza di un singolo guasto di una delle tre istanze.

Load Balancing

Quando viene costruito un cluster, bisogna considerare di bilanciare il carico del traffico delle richieste che lo attraversa, in modo da aiutarlo a massimizzare il throughput (rendimento) e ridurre la latenza. Neo4j non possiede un Load Balancer nativo, perciò il compito del bilanciamento è addossato completamente delle infrastrutture che compongono la rete. Dato che le interrogazioni vengono spedite via HTTP, viene collocato tra la rete esterna il cluster un Server Proxy che funge da Bilanciatore di Carico. Questo server si limiterà a direzionare le richieste di lettura verso gli slave e quelle di scrittura verso il master.

Cache Sharding

Un altro metodo, sfruttato da Neo4j per bilanciare il carico di lavoro, è quello di impiegare la tecnica chiamata *Cache Sharding*. Questa tecnica si basa sul fatto che le queries vengono eseguite più velocemente, se la porzione del grafo di interesse è ancora salvata all'interno della memoria centrale. Il cache Sharding consiste nel direzionare le richieste verso quei nodi del cluster che contengo in memoria centrale i sottografi che servono a soddisfarle.

Estensioni

Le estensioni consentono di eseguire del codice Java all'interno del server. L'utilizzo delle estensioni permette di estendere la REST API oppure di rimpiazzarla completamente.

Le estensioni prendono la forma di *JAX-RS annotated classes*. Una JAX-RS è una Java API costruita per risorse RESTful, in altre parole è un API costruita per il linguaggio Java nata per interloquire con un architettura di tipo REST.

Dato che le estensioni permettono di eseguire del codice Java all'interno dell'istanza server, l'utilizzo di queste potrebbe avere qualche impatto sul comportamento del Gabarge Collection del server.

Vantaggi

Transazioni Complesse: Le estensioni permettono di eseguire arbitrariamente una complessa sequenza di operazioni all'interno di un'unica transazione.

Scelta delle API: Ogni estensione viene incorporata all'interno del cuore del server di database sotto forma di riferimento. Questo ci permette di avere un accesso totalità delle API (Core API, traversal framework, graph algorithm package, e Cypher) per poter sviluppare un'estensione personalizzata.

Incapsulamento: Siccome ogni estensione è nascosta all'interno delle RESTful interface, è possibile accresce o modificare la loro implementazione a piacimento.

Formato delle Risposte: Consentono di controllare il formato delle risposte.

2.3 Il modello dei Dati

In Neo4j **[3]** le unità fondamentali che compongono un grafo sono i nodi e le relazioni.

2.3.1 Nodi

I nodi vengono solitamente impegnati per rappresentare le *entità*, ma a seconda della sfera delle relazioni possono essere utilizzati per scopi differenti.

A parte proprietà e relazioni, i nodi possono anche essere etichettati con zero o più Label.

2.3.2 Relazioni

Le relazioni trai nodi sono una parte chiave dei database a grafo. Ci permettono di trovare le informazioni connesse. Come per i nodi, le relazioni possono avere le proprietà.

Caratteristiche:

- Una relazione connette due nodi, e possiede sempre un nodo di partenza e uno di arrivo.
- Una relazione ha sempre una direzione
- Le relazioni possono essere attraversate in entrambe le direzioni. Ciò significa che non vi è bisogno di aggiungere delle relazioni duplicate con direzione opposta.
- Un nodo può essere relazionato con se stesso.
- Le relazioni possono essere di un tipo (Type).

2.3.3 Proprietà

Sia nodi che relazioni possono avere delle proprietà. Le proprietà sono delle coppie chiave valore, dove la chiave è una stringa. Il valore delle proprietà può essere sia un tipo di primitiva che un array di un tipo di primitiva. Per esempio: String, int a int[].

Il valore NULL non è valido per le proprietà. Il valore NULL può essere implementata con l'assenza della proprietà (ovvero della chiave).

2.3.4 Labels

Una label è un "named graph construct" , viene usata per raggruppare i nodi in sottoinsiemi; tutti i nodi etichettati con la stessa label fanno parte dello stesso insieme.

Diverse query possono lavorare con questi insiemi invece che con l'intera totalità del grafo, rendendo le interrogazioni più facili da scrivere e più efficienti. Un nodo può essere etichettato con un diverso numero di Labels, inclusa nessuna, rendendole così un aggiunta opzionale al grafo.

Le label vengono usate quando si vogliono definire constraint e aggiungere indici sulle proprietà.

Un esempio: Label: User → può essere impiegata per etichettare tutti quei nodi che rappresentano un utente. In questo modo, si può chiedere a Neo4j di eseguire delle operazione solo sui quei nodi utente, come ad esempio cercare tutti gli utenti con un dato nome.

Tuttavia, le label possono essere impiegate per altri scopi. Per esempio, possono essere aggiunte o tolte in fase di runtime, ovvero possono essere impiegate per marcare temporaneamente i nodi, per indicarne uno stato. Si può creare un label "Offline" per marcare tutti i telefoni offline, oppure "Happy" per gli animali felici, e così via.

Il massimo numero di label che possono essere presenti nel database sono 2 miliardi, perchè posseggono ognuna un id, i quali sono di tipo int.

2.3.4 Percorsi (Path)

Un path è uno o più nodi connessi da relazioni, tipicamente recuperabile da una query. Il percorso più corto possibile ha lunghezza zero ed è costituito da un solo nodo senza relazioni uscenti o in arrivo (un nodo a se). Un percorso ha lunghezza 1 se è costituito da due nodi connessi da una relazione, oppure un nodo con un relazione che connette se stesso.

2.4 Gli Indici

Gli ***indici*** sono particolari strutture dati che consento un rapido accesso ad un sottoinsieme del database. Neo4j consente di indicizzare i dati, ma al contrario di molte altre tecnologie, esso possiede due categorie di indici:

- **Schema Index**,
- **Non-Schema Index** (Lucene Indexes).

2.4.1 Schema Index

Le performance vengono aumentate creando gli ***schema index***, i quali aumentano la velocità di ricerca dei nodi nel database. Una volta specificata quale proprietà di una determinata Label è da indicizzare, Neo4j manterrà i tuoi indici aggiornati e in linea con l'evolversi del grafo. Alcune operazioni di ricerca dei nodi attraverso recenti proprietà indicizzate, si mostreranno una significante spinta del rendimento.

Gli Schema Index in neo4j sono "*eventually available" (disponibili con il tempo)*. Sta a significare che quando viene creato un indice, l'operazione viene eseguita immediatamente (hai subito un riscontro/risultato), ovvero viene creato immediatamente. L'indice, però, viene popolato in background e non è subito disponibile per le interrogazioni. Con il tempo diventerà online, e quando verrà completamente popolato sarà pronto per essere utilizzato dalle queries.

Se qualcosa dovesse andare storto con l'indice, esso finirebbe in un "failed state". Quando fallisce, non può essere impiegato per velocizzare le query. Per sistemarlo, bisogna cancellarlo (DROP) e ricrearlo (CREATE). E' buona norma tenere sotto osservazione i logs per avere indizi riguardo ai fallimenti.

Per tener traccia dello stato degli indici bisogna utilizzare le API a disposizione (shell, tools, etc), perché con Cypher non è possibile farlo.

Gli Schema Index vengono definiti per mezzo del linguaggio Cypher:

```
CREATE INDEX ON :name-of-label(proprety-name)
```

Con questo commando viene creato un indice sull'attributo sfruttando l'etichetta indicata.

2.4.1 Non-Schema Index (Lucene)

Neo4j consente di implementare gli indici "comunicando" direttamente con il componente che fornisce il servizio di definizione e costruzione degli indici. Questo componente prende il nome di Lucene (neo4j-lucene-index). Lucene è integrato all'interno dello standard download di Neo4j e permette la definizione di diverse tipologie di indici.

La creazione degli indici Non-Schema viene effettuata per mezzo di chiamate POST alla REST API di Neo4j e, al contrario degli Schema Index, essi non verranno mantenuti aggiornati dal DBMS, ma sarà compito dell'amministratore aggiungere e togliere i nodi e le relazioni da indicizzare.

I Non-Schema Index sono molto importati perché vengono impiegati dalla clausola START di Cypher.

2.5 Constraint

Neo4j consente di tenere i dati "in ordine". Lo fa utilizzando le ***constraint***, le quali permettono di specificare le regole che i dati dovranno rispettare. Ogni cambiamento che violerebbe queste regole verrà NEGATO.

Neo4j consente di rafforzare l'integrità dei dati con l'uso delle *constraints*. La **unique constraints** è l'unica fornita per ora da Neo4j. Essa viene impiegata per assicurarsi che il valore di una proprietà sia unica per tutti i nodi con una specifica LABEL. La unique constraint non significa che tutti i nodi del grafo con la Label indicata, devono per forza possedere la proprietà specificata, la quale dovrà essere unique (univoco), ma che tutti i nodi della Label di interesse che posseggono anche la proprietà su cui è stato costruita la constraint, dovranno attenersi alle regole del vincolo. Ovvero, la constraint non vincola la totalità dei nodi, ma solo quel sottoinsieme di essi, che soddisfano le sue caratteristiche. Quindi, i nodi senza la proprietà o/e la label specificata nella constraint non sono soggetti a questa regola.

Aggiungere constraints è un'operazione atomica che richiede del tempo – tutti i dati esistenti devono essere visionati prima da Neo4j , il quale, una volta verificati i dati, può rendere la constraint "on" (attiva). Aggiungendo una constraint di unicità su una proprietà, verrà aggiunta anche un Schema Index. Eliminando la constraint si eliminerà anche l'indice; se si vorrà mantenere l'indice sulla proprietà, dovrà essere ricreato. Si possono avere più di una unique constraint per una data LABEL.

2.6 Cypher
Il Linguaggio di Interrogazione

Cypher (**[2], [3])** permette agli utenti (o ad una applicazione che agisce per conto dell'utente) di interrogare il database cercando i dati che corrispondono ad una specifica struttura. In termini da profano, chiediamo al database di " *cercare tutti quegli oggetti simili o che assomigliano* " un certo pattern. Il pattern è la struttura di riferimento a cui si dovrà attenere la query nella ricerca delle informazioni. Precedentemente è stato accennato questo concetto che prende il nome di "***things like this***", e il modo cui viene descritto il pattern assomiglia al disegnarlo, usando caratteri ASCII [es:"(a)-[:Rel]->(b)"].

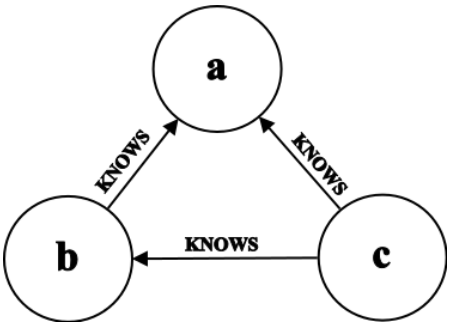

Figura 2.5: Rappresentazione a grafo di tre amici reciproci (tratta da "**The Graph Database**" di Ian Robison e Jim Wabber)..

Questa struttura descrive tre amici reciproci. Qui sotto vi è il corrispettivo disegno ASCII che viene usato da Cypher per rappresentare la struttura.

```
(a)-[:KNOWS]->(b)-[:KNOWS]->(c) ,  (a)-[:KNOWS]->(c)
```

Questo pattern descrive un *path* (percorso), che connette (a) a (b), (b) a (c), e (a) a (c). Sono stati impiegato una serie di trucchi per girare intorno al fatto che una query a solo una dimensione (la lettura del testo da sinistra a destra), mentre un diagramma a grafo può essere strutturato in più di una dimensione. In questo caso è stato separato il pattern principale in due sotto-pattern per mezzo di una virgola. Nel complesso, i pattern di Cypher seguono naturalmente il modo in cui disegniamo i grafi sulla lavagna o su un foglio di carta. Il modo i cui vengono disegnati i pattern con i caratteri ASCII è fondamentale per le query Cypher.

Una query di Cypher si ancora a una o più parti di un pattern in uno specifico punto del grafo (il punto di partenza), per poi flettersi verso le parti non ancorate che si trovano attorno, per cercare dei riscontri locali. In parole più comprensibili, data una query con una struttura di esempio (pattern), Cypher sceglie uno o più parti di questa struttura, con le quali stabilire i punti di partenza nel grafo da cui far iniziare la ricerca, la quale si sviluppa cercando riscontri nell'intorno di questi punti (ovvero si flette verso i punti non ancorati).

La posizione di partenza, "***anchor point***" (punto di ancoraggio), può essere trovate in più modi. Il metodo più comune è l'utilizzo di un indice. Neo4j usa gli indici come servizio di denominazione; ed è lo strumento di ricerca della posizioni di partenza, basato su uno o più valori delle proprietà di una determinata Label o tipo di relazione, da cui poi far iniziare la ricerca .

Come molti linguaggi di interrogazione, Cypher è composto da clausole. Le più semplici query sono composte dalla clausola START, seguita da MATCH e RETURN. Nell'esempio seguente una query Cypher usa queste tre clausole per scovare i reciproci amici dell'utente Micheal.

```
START  a = node:user(name:'Michael')
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c),(a)-[:KNOWS]->(c)
RETURN b , c ;
```

2.6.1 START

START specifica uno o più punti di partenza – nodi o relazioni – nel grafo. Questi punti di partenza sono ottenuti per mezzo di ricerche su un ***Non-Schema Index,*** o più raramente , con un accesso diretto ad un nodo o una relazione per mezzo di ID. Nella query di esempio precedente, abbiamo cercato un nodo di partenza per mezzo di un indice chiamato **user**. E' stato chiesto all' indice di trovare un nodo con una proprietà **name** che assume il valore **Micheal**. Il valore di ritorno di questa ricerca è vincolato da un *identificatore*, che prende il nome "**a"**. L'identificatore consentirà di far riferimento al nodo di partenza all'interno della nostra query.

2.6.2 MATCH

Questa è le parte di *descrizione per esempio*. Usando i caratteri ASCII per rappresentare nodi e relazioni, disegniamo i dati a cui siamo interessati. Vengono usate le parentesi tonde per indicare i nodi, e coppie di trattini (-) , per disegnare le relazioni (- - > e <- -). I segni maggiore(>) e minore(<), indicano la direzione della relazione. In mezzo ai trattini, all'interno delle parentesi quadre e prefissato dai due punti, è presente il nome del tipo di relazione.

Questo pattern potrebbe, in teoria, ricorrere molte volte all'interno del grafo; con un grande set di user, potrebbero essere presenti diversi sotto-grafi corrispondenti a questo pattern. Per circoscrivere la query dobbiamo ancorarla ad una o più parti del grafo. Lo abbiamo fatto con la clausola START, la quale ricerca un nodo all'interno del grafo – il nodo che rappresenta Michael. Questo nodo è stato vincolato all'identificatore a; ed è stato riutilizzato (trasportandolo) all'interno della clausola MATCH. Così facendo abbiamo ancorato il nostro pattern ad uno specifico punto nel grafo.

2.6.3 RETURN

Questa clausola specifica quali nodi, relazioni e proprietà nei dati riscontrati dovranno essere restituiti al client.

2.6.4 Altre clausole Cypher

Le altre clausole che possiamo usare in una query Cypher sono:

- **WHERE:** Fornisce dei criteri per filtrare i risultati dei riscontri.
- **CREATE and CREATE UNIQUE:** Crea nodi e relazioni.
- **DELETE:** Rimuove nodi, relazioni e proprietà.
- **SET:** Sets i valori delle proprietà.
- **FORECH:** Fornisce un'azione di aggiornamento per ogni nodo di una lista.
- **UNION:** Unisci i risultati di due o più query (introdotto in Neo4j 2.0).
- **WITH:** Manipola il risultato di una sotto-query prima che venga passata alla restante parte della query.
- **OPTIONAL MATCH:** Molto simile alla clausola match, la differenza sta nel fatto che, se non vengono effettuati riscontri, l'OPTIONAL MATCH restituirà il valore NULLs per rappresentare quelle parti mancanti del pattern. L'OPTIONAL MATCH potrebbe essere considerato l'equivalente del SQL outer-join del mondo Cypher.

Cypher, fornisce una ulteriore moltitudine di costrutti che non non verranno elencati, ma che permettono di listare un sottoinsieme di nodi per poi lavorarci come se fossero le tuple di una tabella.

2.7 L’attraversamento del grafo

Attraversare un grafo significa visitare i suoi nodi, seguendo le relazioni connesse da qualche regola. In molti casi solo un sotto grafo è visitabile, laddove vengono trovati i nodi e le relazioni che ci interessano.

Neo4j mette a disposizione due tecniche che permettono di percorre il grafo in cerca dei suoi nodi e delle sue relazioni.

2.7.1 Gli Algoritmi sui Grafi

La prima cosa che viene in mente quando si vuole attraversare un grafo, è sicuramente quella di impiegare uno dei tanti algoritmi di attraversamento forniti dal mondo della teoria dei grafi, Neo4j non viene meno a questo primo approccio.

Graph Algorithms **[3]** per Neo4j è un componente che contiene l’implementazione di alcuni dei più comuni algoritmi su grafi.

Include algoritmi come:

- Shortest paths,
- all paths,
- all simple paths,
- Dijkstra,
- A*.

Tutti questi algoritmi possono essere trovati all’interno del componente **neo4j-graph-algo**, il quale è presente nello standard Neo4j download.

Nota: Ci sono altri algoritmi che possono essere usati sui grafi più piccoli (es. calcolo della centralità, betwenness, closeness, accentraty, etc). Questi algoritmi non sono stati progettati per essere utilizzati in grafi molto grandi, ma possono risultare utili in alcuni scenari. Essi risiedono all’interno del package **org.neo4j.graphalgo.impl.centrality** e non vengono considerati una produzione di qualità.

Neo4j possiede una serie di build-in algoritmi su grafo. Essi vengono eseguiti su un nodo di partenza (start node). Gli algoritmi possono essere utilizzati sfruttando le varie API messe a disposizione di Neo4j, dalle REST API alle query Cypher.

2.7.2 Query

Un'altra tecnica di attraversamento del grafo messa a disposizione da Neo4j sono le query stesse. Le interrogazioni Cypher consentono di selezionare sottoinsiemi del grafo presente nel database. Questa selezione avviene, come è stato detto in precedenza, per mezzo di un pattern che riproduce una struttura presente all'interno del database. Ciò significa che, anche se si effettuano delle query di selezione, in effetti non si fa altro che percorre il grafo per trovare tutti quei sottoinsiemi che rispecchiano il pattern datogli in input.

Questo concetto sarà la chiave di volta della prossima sezione.

2.8 La selezione dei dati

Cypher fornisce una molteplicità di clausole che permettono di interrogare il database.

La maniera in cui vengono impiegate queste clausole consentono di definire diverse tecniche di selezione, che influiranno, positivamente o negativamente, sulle prestazioni delle query.

2.8.1 Tecniche di selezione

Per affrontare al meglio questa parte si è deciso di utilizzare un modello di riferimento, a cui sottoporre una serie di query SQL e poi analizzare le correspettive query Cypher.

La conoscenza della struttura dei modelli non è un problema di rilievo visto che le queriy che verranno prese in esame non saranno complesse dal punto di vista logico, infatti non verranno effettuati grandi join tra le tabelle, e la struttura dei modelli è facilmente intuibile direttamente dalle interrogazioni stesse.

Si prenda in considerazione la seguente query SQL:

```
SELECT FT.*
FROM FACT500K  FT, CITY c
WHERE FT.CITY = c.IDCITY
 AND c.CITY = 'Alameda';
```

La tabella FACT500K contiene all'interno una serie di dati statistici suddivisi per città, etnia, sesso, occupazione e anno.

La query di esempio esegue un semplice join tra la tabella FACT500K e la tabella CITY. L'interrogazione estrarrà tutte le tuple da FACT500K che contengono i dati statistici della città di Alameda. La suddetta query potrebbe essere tradotta nelle seguente query Cypher:

```
MATCH (FT:Fact500k)-[:IN_CITY]->(c:City)
WHERE c.CITY = 'Alamenda'
RETURN FT
```

La struttura del grafo è facilmente deducibile. I nodi etichettati "Fact500k", che corrispondono alle tuple della tabella FACT500K, sono connessi ai nodi con label "City" per mezzo di una semplice relazione. Chiaramente i nodi etichettati "City" corrisponderanno ai record della tabella CITY.

I risultati della query SQL e della query Cypher sono pressoché identici, non si possono considerare uguali perché le chiavi primarie ed importate non sono presenti in un database a grafo, al contrario del relazionale.

In questa prima interrogazione Cypher, si può notare che la struttura di rifermento all'interno della clausola MATCH non presenta alcuna selezione esplicita, a parte la forma del pattern. Ciò significa che prima verranno estratti tutti i Fact presenti nel database, con annessi tutti i nodi Città collegati per mezzo della relazione "IN_CITY" , e poi successivamente verrà effettuata la schermatura dei risultati in base al predicato di selezione presente nella clausola WHERE.

Chiaramente dal punto di vista prestazionale, la query appena descritta risulta assai onerosa, visto che il motore di database di neo4j dovrà prima percorre il database per estrarre tutti i sotto-grafi che corrispondono al pattern per caricarseli in memoria centrale, e successivamente scandirli sequenzialmente per eliminare tutti quei riscontri che non soddisfano il predicato di selezione.

Nel caso di grandi dataset, questo tipo di approccio grava talmente tanto sulle risorse, da rischiare di saturare tutta la memoria RAM adibita al server Neo4j (qualora non sia stata dedicata un'enorme quantità di memoria).

Un'altra query Cypher corrispondente alla query SQL potrebbe essere la seguente:

```
MATCH (FT:Fact500k)-[:IN_CITY]->(c:City{CITY: 'Alamenda'})
RETURN FT
```

In questo caso il predicato di selettività è presente all'interno del pattern, ciò sta a significare che non verranno caricati in memoria centrale tutti i Fact presenti nel database, ma la cernita verrà effettuata a livello di attraversamento del grafo. In questo modo si eviterà di sovraccaricare la memoria RAM, e di ridurre notevolmente i tempi di esecuzione della query. Anche in questo caso Neo4j dovrà scandire sequenzialmente tutti i fact presenti nel database per poi scartare quelli che non gli interessano.

La prossima query Cypher suppone che sia stato creato un indice sulla proprietà "CITY" dei nodi etichettati "City", questo perché verrà effettuata la ricerca impiegando la clausola START:

```
START c=node:City(CITY='Alameda')
MATCH (c)<-[:IN_CITY]-(FT:Fact500k)
RETURN FT
```

Dato che è stato implicitamente impiegato un indice utilizzando la clausola START, l'interrogazione risulterà sicuramente più performante. L'enorme aumento delle prestazioni non è dovuto solamente al fatto che è stato utilizzato l'indice per eseguire la query, il merito è da attribuire soprattutto alla logica della clausola START.

Come è stato detto in precedenza, la clausola START esegue una prima ricerca in base al predicato di selezione indicatogli. Questi nodi verranno impiegati come punti di partenza (*anchor point*) da cui far protendere la ricerca vera e propria. In questo modo, Neo4j non dovrà scandire tutti i fact presenti all'interno del database, ma si dovrà solamente limitare ad attraversare tutte quelle relazioni che partono dalle Citta estratte inizialmente e che gli consentono raggiungere i Fact di interesse.

Vediamo ora un'ultima query:

```
START c=node(773)
MATCH (c)<-[IN_CITY]-(FT:Fact500k)
RETURN FT
```

In quest'ultima interrogazione si può notare che il "nodo ancora" viene scelto direttamente specificandogli l'id del nodo. Il nodo ancora sarà solo uno (come nel caso precedente) e da esso partirà la ricerca. L'utilizzo degli id permette di aumentare ulteriormente la velocità di esecuzione delle nostre queries. Ogni qualvolta si vuole effettuare una ricerca molto grande, l'utilizzo di questi identificatori univoci è un buon escamotage. Tuttavia, l'impiego degli id implica che si conosca a priori l'id del nodo di interesse.

2.9 Aggregazione dei dati

L'estrazione di informazioni aggregate per un certa chiave di lettura è una delle componenti fondamentali dei linguaggi di interrogazione. Per poter mostrare nel modo corretto come Cypher offre questa funzionalità, si è deciso di adottare il metodo della sotto-sezione 2.8. Ovvero esporre delle query SQL, per poi mostrare le corrispettive interrogazioni Cypher.

Si consideri la seguente query SQL:

```
SELECT C.CITY, COUNT(FT.CITY)
FROM CITY C, FACT500K FT
WHERE C.IDCITY = FT.CITY
GROUP BY C.CITY;
```

L'interrogazione consente di contare quanti record della tabella FACT500K corrispondo ad ogni valore dell'attributo CITY della tabella CITY. Nella clausola GROUP BY viene indicato per quali attributi delle tabelle si vuole effettuare l'aggregazione. Invece la funzione di aggregazione, in questo caso COUNT(), indica qual è il valore aggregato che deve calcolare la query. Cypher offre soluzioni molto simili al GROUP BY dell'SQL, ovvero le **funzioni di aggregazione** e le **non-aggregate expression**. La precedente query SQL può essere tradotta nella seguente query Cypher.

```
MATCH (c:City)-->(ft:Fact500k)
RETURN c, count(ft);
```

Vi sono due espressioni di ritorno – **n**, e **count(ft)**. La prima , **n**, è una funzione di non aggregazione (non-aggregate expression**)**, essa sarà la chiave di raggruppamento. L'ultima, **count(ft)** , è l'espressione di aggregazione. In questo modo il sotto grafo estratto dalla query viene diviso in differenti bucket, dipendenti dalla chiave di raggruppamento. La funzione di raggruppamento lavorerà su questi bucket, da cui calcolerà il valore aggregato.

2.10 Impieghi Futuri

La seguente sezione del capitolo avrà principalmente il compito di proporre degli impieghi futuri della tecnologia NoSQL nell'analisi OLAP.

2.10.1 Viste Materializzate

Una **vista materializzata [1]** è un oggetto delle base di dati che contiene i risultati di una query. Le viste materializzate possono essere richiamate dalle interrogazioni SQL come una qualsiasi vista o tabella, e solitamente contengono i dati delle tabelle o unione di tabelle già aggregati, in questo caso di parla di viste di aggregazione.

Le viste materializzate sono state implementate per la prima volta da Oracle: dalla versione 8i. Neo4j non possiede questo tipo di oggetto, tuttavia, per questa tesi, si è tentato di produrne uno del tutto simile, che potesse in qualche modo migliorare le prestazioni del DBMS NoSQL.

Nodi Aggregati

Si suppone di voler ottenere la somma dell'attributo PERWT in base allo Stato e gruppo Etnico di appartenenza. Per ottenere queste informazioni, bisognerà eseguire la seguente query Cypher:

```
MATCH  (f:Census)-[:IN_CITY]->(:City)-[:IN_STATE]->(s:State),
(rg:RaceGroup)<-[:IN_RACEGROUP]-(:Race)<-[:IN_RACE]-(f)
RETURN id(s), id(rg), sum(f.PERWT);
```

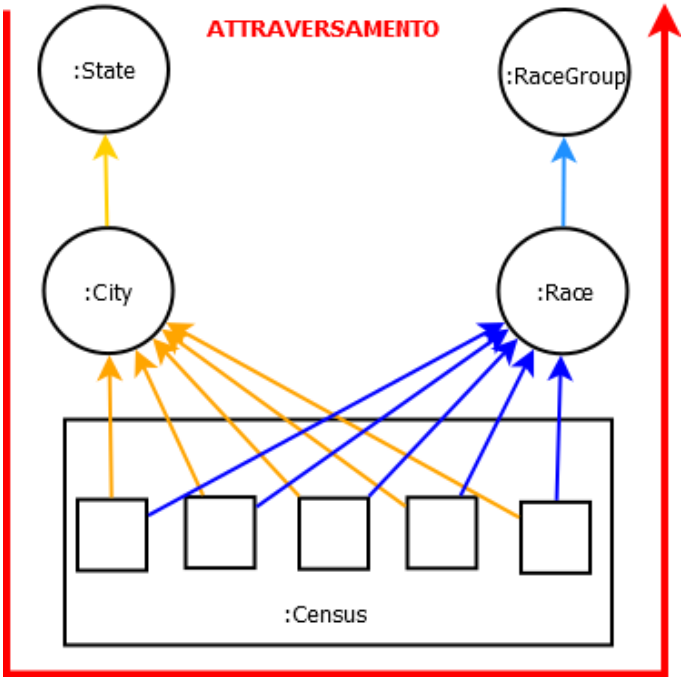

Figura 2.6: Rappresentazione dell'attraversamento che viene effettutato in assenza

La query appena descritta deve effettuare ben 4 Hop, per poter restituire il risultato richiesto.

Si suppone invece, di aver creato precedentemente dei nodi di tipo :Census che contengono al loro interno la somma di PERWT aggregato per Stato e gruppo etnico. E si suppone che questi nodi, invece di essere connessi ai nodi dimensionali di baso livello, cioè :City e :Race, verrano direttamente collegati ai nodi dimensionali per cui è stata fatta l'operazione di raggruppamento, ovvero :State e :RaceGroup. Questi particolari nodi verranno da ora in poi chiamati *Nodi Aggregati*.

Sfruttando i nodi aggregati basterà limitare l'attraversamento alla semplice selezione di questi particolari nodi.

```
MATCH (s:State)<--(f:Census)-->(rg:RaceGroup)
RETURN id(s), id(rg), sum(f.PERWT);
```

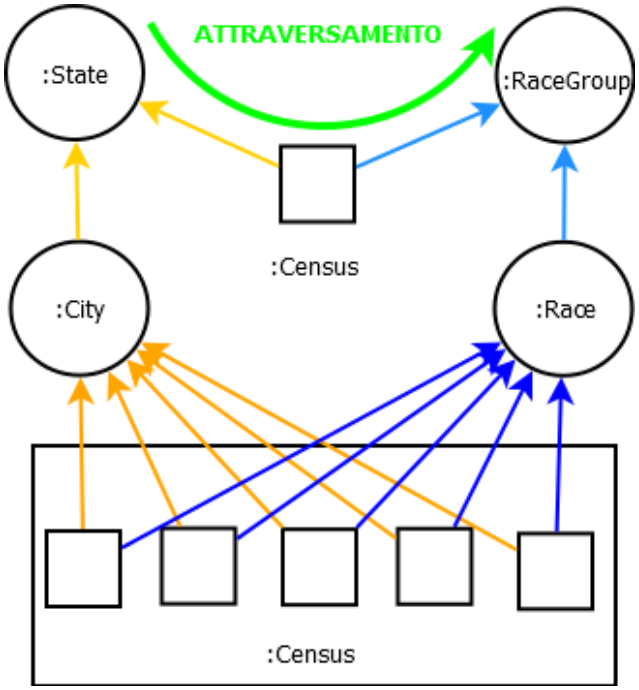

Figura 2.7: Rappresentazione dell'attraversamento che viene effettuatoin presenza dei nodi aggregati.

Costruendo invece i Nodi Aggregati è possibile ottenere lo stesso risultato scrivendo una query che ha la metà degli Hop.

Quindi, l'idea di base è quella di costruire delle query Cypher che possano fornire delle viste di taglio, le quali contengono i dati dei nodi :Census aggregati per un determinato livello di specializzazione delle dimensioni.

Con i risultati forniti da queste interrogazioni, verranno poi creati dei nodi di tipo Census che invece di essere connessi ai nodi dimensionali di basso livello (es. :City, : Race, ;Occupation, :Sex e :Year), saranno collegati direttamente ai nodi dimensionali di livello superiore (es. :State, :Branch, :Racegroup, :sex e :Year). Quindi, ogniqualvolta si desidererà ottenere dei valori aggregati in base ad un maggiore livello di raggruppamento, invece di percorre il grafo fino a raggiungere i nodi :Census connessi ai nodi dimensionali di basso livello, basterà estrarre i nodi :Census creati in secondo il criterio appena descritto.

Implementazione

Il seguente script consente di creare i Nodi Aggregati, eseguendo una vista di taglio che raggruppa i Census in base al secondo livello di aggregazione.

```
MATCH (f:Census)-[:IN_CITY]->(:City)-[:IN_STATE]-
>(s:State{STATE:'FL'}),
(f)-[:IN_RACE]->(:Race)-[:IN_RACEGROUP]-
>(rg:RaceGroup{RACEGROUP:'White'}),
(f)-[:IN_OCCUPATION]->(:Occupation)-[:IN_BRANCH]->(b:Branch),
(f)-[:IN_SEX]->(sex:Sex),
(f)-[:IN_YEAR]->(y:Year)
WITH  s, rg ,b,sex,y, sum(f.PERWT) as per, sum(f.COSTELEC) as
cost
CREATE (g:Census{PERWT:0, COSTELEC:0})
MERGE (g)-[r1:IN_STATE{IDSTATE:-1}]->(s)
MERGE (g)-[r2:IN_RACEGROUP{IDRACEGROUP:-1}]->(rg)
MERGE (g)-[r3:IN_BRANCH{IDBRANCH:-1}]->(b)
MERGE (g)-[r4:IN_YEAR{IDYEAR:-1}]->(y)
MERGE (g)-[r5:IN_SEX{IDSEX:-1}]->(sex)
ON CREATE SET g.PERWT = per,
g.COSTELEC = cost,
r1.IDSTATE = s.IDSTATE
r2.IDRACEGROUP = rg.IDRACEGROUP,
r3.IDBRANCH = b.IDBRANCH,
r4.IDYEAR = y.IDYEAR,
r5.IDSEX = sex.IDSEX
RETURN g;
```

Per creare i Nodi aggregati anche per il terzo livello di aggregazione, ovvero :Region, :Mrn, :SubCategory, :sex e :Year, basterà modificare la prima parte dello script, in cui è presente l'interrogazione che effettua il calcolo dei dati aggregati.

Vantaggi

- **Miglioramento delle performance:** Man mano che sale il livello di aggregazione aumenta anche il contributo fornito dai Nodi Aggregati. Effettuare delle query su questi particolari Nodi comporta la diminuzione del numero di Nodi e relazioni da attraversare.

- **Riduzione della complessità delle interrogazioni:** essendo Noe4j una tecnologia assai giovane, il proprio ottimizzatore non raffinato quanto quello di altre come Oracle. Ciò significa che l'impiego degli Hint all'interno delle query, è fondamentale affinché un'interrogazione possa fornire il risultato voluto in tempi ragionevoli. Diminuendo la complessità dei pattern delle query, cala anche lo sforzo che deve effettuare il programmatore per scrivere una query ottimizzata.

Svantaggi

- **Aumento delle risorse Hardware:** aggiungere i nodi aggregati significa aggiungere Nodi e Relazioni al grafo della base dati, e più è grande il database che Neo4j deve gestire più è alta la richiesta di risorse hardware da dover dedicare alla tecnologia.
- **Manutenzione**: la manutenzione dei nodi aggregati non è facile da gestire. Ogniqualvolta che un nodo :Census viene modificato/aggiunto oppure eliminato, oppure accade la stessa cosa per i nodi dimensionali, bisogna aggiornare di conseguenza tutti nodi aggregati che dipendono da queste modifiche. Ciò significa che vi è un enorme sforzo sia da parte della tecnologia, sia da parte dell'amministratore che deve scrivere il codice che andrà a gestire i vari cambiamenti.
- **Metadata**: per poter sfruttare i nodi aggregati il programmatore deve per forza conoscere l'intera struttura della base dati e lo stato in cui si trova.

2.10.2 Pattern Mining

Gli algoritmi di data mining consentono di ricercare ed estrarre pettern dal database. Nel data mining un **pattern** è una rappresentazione sintetica e ricca di semantica di un insieme di dati; esprime in genere un modello ricorrente nei dati, ma può anche esprimere un modello eccezionale.

Mentre la maggior parte degli approcci di data mining ricercano i pattern all'interno di una singola tabella, il multi-relational data mining (MRDM) consente di estrarre i pattern che coinvolgono più tabelle di un database relazionale.

Nel mondo OLAP estrarre le informazioni implicite che interessano più di una tabella significa effettuare delle onerose operazioni di join. Per questo motivo, dato che i Graph Database sono nati con lo scopo di risolvere questo problema, si vuole presentare come la tecnologia NoSQl può affrontare il mutli-relational data mining.

Si vuole contare quanti Census collegano ogni volare di un livello di una dimensione, esempio City, verso ogni altro valore livello di un'altra dimensione, esempio: Occupation, Branch, Race, Category, Sex, etc .

Estrarre questo tipo di informazioni da un Graph database, significa contare ogni percorso che collega un determinato un nodo dimensionale verso un altro nodo dimensionale. Esempio: City e Branch. La seguente query calcola il numero di percorsi che collegano ogni nodo dimensionale :City ad un nodo dimensionale :Sex.

```
START c=node:CityIdx("CITY:*")
MATCH p=(c)-[*2]-(s:Sex)
RETURN c, s, count(DISTINCT p);
```

Capitolo III
BenchMarking

Nei capitoli precedenti è stato descritto Neo4j e il mondo a cui appartiene. In questo capitolo vengono mostrati i risultati dei test effettuati sulle due tecnologie. Nella prima parte (3.1) viene presentata l'impostazione dei test e i relativi dataset utilizzati. Successivamente, viene mostrata un'analisi generalizzata (sezione 3.2), con lo scopo di evidenziare gli aspetti, del Graph DBMS, che suscitano maggior interesse. Infine, varranno analizzati dettagliatamente questi aspetti (sezione 3.3).

Per meglio comprendere le potenzialità della tecnologia NoSql, essa verrà affiancata al suo più grande rivale: Oracle. Oracle è un DBMS di proprietà dell'omonima multinazionale Americana. E' un DBMS Relazionale scritto in C ed è forse il più utilizzato al mondo. Questo prodotto è un modello di qualità ed eccellenza a cui molte compagnie fanno riferimento, ed al contrario di Neo4j , è stato affinato nel corso di svariate decine di anni.

Effettuare un confronto tra due tecnologie con un storia così diversa e con diversi livelli di maturità, è quasi sicuramente un azzardo, però dato che sono i leader dei loro mondi è sicuramente il modo migliore per comprendere la rivoluzione tecnologica che ha colpito molte multinazionali nell'ultimo decennio.

Lo scopo di questa tesi non è fare un paragone che possa in qualche modo promuovere una tecnologia invece che un'altra, ma semplicemente sottoporle ad una serie di test ed analizzarne i risultati. Per poter eseguire dei test su delle tecnologie che immagazzinano ed organizzano informazioni è fondamentale possedere un banca dati da sottoporre ai due DBMS. Il database che è stato scelto si chiama IPUMS.

3.1 Setup dei Test

Per i test è stato impiegato un estratto di un database OLAP: **IPUMS**. **Integrated Public Use Microdata Series (IPUMS) [8]** è l'archivio dati sulla popolazione a livello individuale più grande al mondo. Esso è composto da una serie di campioni statistici della popolazione degli Stati Uniti (IPUMS-USA) o della popolazione mondiale (IPUMS-International). Per la precisione, IPUMS(-USA) è stato costruito con i dati forniti dai censimenti Americani effettuati dal il 1980 e al 2000. Questo database permette di eseguire un accurata analisi sui cambiamenti a lungo termine della popolazione statunitense e mondiale, grazie alla grande quantità e qualità di informazioni che esso racchiude.

Essendo un dataset composto da un'enorme mole di dati, di svariato genere, esso permetterà di effettuare dei test di qualità che possano in qualche modo far luce sui meccanismi che sono alla base delle interrogazioni di Neo4j Graph DBMS..

3.1.1 Struttura ed organizzazione dei dati

La figura sottostante lo schema concettuale di IPUMS modellato utilizzando il formalismo Data Fact Model **[4]**:

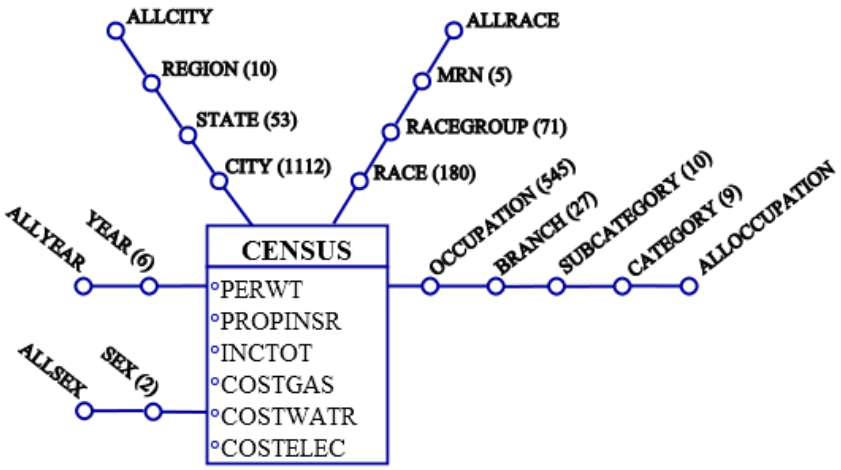

Figura 3.1: Schema concettuale IPUMS

Ogni campione statistico della popolazione è rappresentato dell'entità CENSUS, ed è classificato secondo la città di provenienza, l'anno in cui è stato censito, l'etnia di appartenenza, il sesso, e l'occupazione. Queste cinque categorie in gergo tecnico vengono chiamate *dimensioni*.

I valori di ogni dimensione sono strutturati in gerarchie, definendo così aggregazioni a diversi livelli di dettaglio. Per esempio, ogni città, rappresentata dall'attributo "CITY" , appartiene ad un particolare stato (STATE) di una determinata regione (REGION) (ALLCITY è un campo padre che rappresenta la totalità delle città). Grazie a questa organizzazione è possibile rispondere a delle domande come "qual è la media degli stipendi della popolazione di un determinato stato oppure dell'intera regione?", evitando così, di limitare l'analisi alla sola città o all'intera nazione.

Lo schema mostrato precedentemente non è il vero aspetto della struttura della base dati IPUMS, ma è uno schema che proveniente dal mondo OLAP. Il modello logico utilizzato è invece detto "star schema".

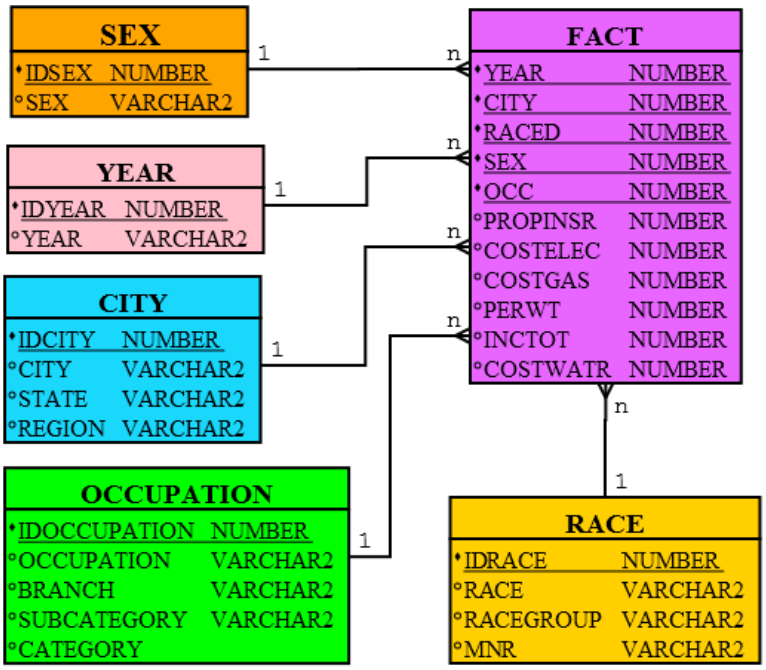

Figura 3.2: Star Schema di IPUMS

Lo schema della figura 3.2 è il vero aspetto di IPUMS. La sua struttura è nata e concepita per il mondo relazionale ed è composta da una grande tabella, detta Fact Table (Tabella dei fatti) in cui ogni tupla contiene i dati statistici della popolazione raggruppati per città, sesso, occupazione, etnia e anno di censimento, reperibili per mezzo delle cinque *tabelle dimensionali*. Inoltre, quest'ultime tabelle sono in *prima forma normale (1-NF)*, ovvero i dati descrittivi vengono ripetuti in ogni tupla. Anche se in 1-NF, il database è in un stato coerente e consistente.

Le tabelle dimensionali vengono mantenute denormalizzate perché le prestazioni degli RDBMS calano qual ora si effettua un join tra due grandi relazioni e quando si aumentano il numero di join. Il concetto appena descritto è in effetti la vera motivazione per la quale sono nati i Graph DBMS.

3.1.2 Traduzione della base dati

Il primo passo per raggiungere il fine ultimo di questa tesi è stato quello trasporre la base dati sulla nuova tecnologia, ovvero è stato tradotto IPUMS dalla sua forma relazionale ad una a grafo.

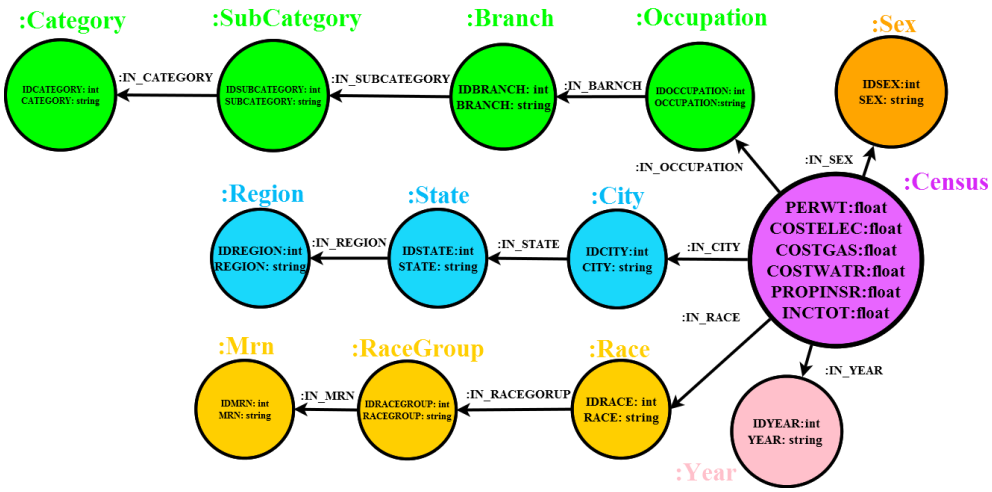

Figura 3.3: Modello a Grafo di IPUMS, nato dallo Star Schema.

La figura 3.3 mostra la struttura generale di IPUMS nella sua forma a grafo. La prima differenza che si nota tra il modello relazionale e quello a grafo è che le “tabelle dimensionali sono state normalizzate”, creando per ogni livello gerarchico un nodo con un propria Label. Un’altra differenza sono gli ID, sono stati aggiunti dei campi che fungono da identificatori univoci (IDCITY, IDSTATE, IDREGION, etc), i quali vengono impiegati da un programma, creato appositamente, per ritrasformare la base dati a grafo in una relazionale ed assicurarsi che la traduzione dei dati sia avvenuta correttamente.

Con quest’ultimo appunto è possibile notare la vera e più grande differenza tra lo schema relazionale e quello a grafo: l’assenza delle FOREING KEY. Una graph database non necessita la presenza delle chiavi importate, dato che nasce con l’idea di eliminare questo “limite”. Qual ora sorga la necessità di reperire le informazioni in base al modo con cui si relazionano tra di esse, non sarà più necessario effettuare le onerose operazioni di join tra le tabelle, ma basterà semplicemente “percorrere” la relazioni che connettono i vari nodi del grafo.

IPUMS è un database composto da milioni di record, ma dato che Neo4j è una tecnologia molto esigente dal punto di vista delle risorse hardware, è stato deciso di non lavorare con totalità dei dati di IPUMS ma solo con un sotto insieme di essi. Inoltre, per studiare meglio questa nuova tecnologia sono stati creati due diversi dataset di IPUMS, i quali sono identici dal punto di vista strutturale, però differiscono nella quantità di dati che contengono, il primo possiede 500'000 nodi/tuple Census/Fact, il secondo un milione. Effettuando i test su questi due dataset è possibile studiare come Neo4j scali all’aumentare delle informazioni da gestire, inoltre consente analizzare se e quale dei due DBMS riesce a scalare meglio.

Per poter effettuare l’operazione di conversione/creazione della base dati è stato creato un programma apposito, il quale ha la capacità di collegarsi ad un server Oracle, estrarne i dati e convertirli nella corrispettiva struttura a grafo ed inserirli all’interno di un server Neo4j.

3.1.3 I Data Set e i Database

I test consistono nel prendere nota dei tempi di esecuzione di una serie di Query (sia Sql che Cypher), le quali sono state eseguite su due dataset. Per ogni dataset è stato costruito un Databse che presentasse gli indici e uno che non gli presentasse, per un totale di quattro tipi di base dati per tecnologia:

1) ***IPUMS500K***: è il nome dato alla base dati IPUMS che possiede 500'000 elementi di tipo Census. Questo database non prevede la presenza di indici.
2) ***IPUMS500K IDX***: possiede gli stessi identici dati della precedente base dati, ma al contrario di quest'ultima, su di esso sono stati creati degli indici.
3) ***IPUMS1M***: i dati delle sue tabelle/nodi dimensionali sono gli stessi di IPUMS500K, ma al contrario di quest'ultimo contiene l suo interno un milione di Census (circa). Su di esso non sono stati costruiti indici.
4) ***IPUMS1M IDX***: analogamente ad IPUMS500K IDX, questo database è la copia di IPUMS1M, su cui sono stati costruiti degli indici.

Ad ogni esecuzione delle query sono stati sempre svuotate le memorie delle due tecnologie. Ovvero, ad Oracle è stata liberata la memoria cache e il pool SQL condiviso, per mezzo del seguente script:

```
alter system flush buffer_cache;
alter system flush shared_pool;
```

Invece, per quanto riguarda Neo4j, dato che non esisto comandi di scripting che permettano di liberare la memoria cache, prima di eseguire ogni singola interrogazione è stato riavviato il server.

Una volta segnati i tempi di esecuzione delle query, sono stati effettuati degli studi per ricavarne delle considerazioni che possano in qualche modo far luce sulle caratteristiche di Neo4j, e in generale dei Graph DBMS.

3.1.4 Configurazione

I test sono stati eseguiti su una macchina con le seguenti caratteristiche:

- ***Processore***: Intel® Core™ i5-M430 @2.27GHz 1° Generazione, con 3MB di cache di livello 2.
- ***Memoria RAM***: 8Gb (2 x SO-DIMM DDR3 SDRAM da 4096Mb)
- ***Sistema Operativo***: Windows 7 Professional 64bit.
- ***Disco Rigido***: SATA da 640 GB a 5.400 rpm.

Su questo computer è stato installato un server Oracle 11g v2:

- ***Installazione di Tipo***: Enterprice Edition
- ***Set di Caratteri***: Predefinito (WE8MSWIN1252)
- ***SID***: ORA11GMATPC

Neo4j non necessita di una installazione vera e propria, in effetti il server consiste in una cartella contenente tutta la struttura necessaria alla sua esecuzione. Per lo sviluppo di questa tesi è stato impiegato **Neo4j Enterprice Edition versione 2.0.1**. Per sfruttare il più possibile le sue potenzialità è stata abilitata la High-Performance Cache e la funzionalità di logging. Inoltre, per effettuare dei test accurati che non pregiudicassero nessuna delle due tecnologie, al server Neo4j e al server Orale sono stati dedicati lo stesso ammontare di memoria, e dato che Neo4j necessita di una determinata quantità di RAM basata sulla mole di dati che deve gestire, la scelta della quantità di memoria da dedicare ai server è stata effettuata basandosi sulle risorse hardware di base richieste da Neo4j. Il calcolo delle risorse da dedicare al server Neo4j è stato fatto per mezzo del web tool presente sul sito della NeoTecnology all'indirizzo http://www.neotechnology.com/hardware-sizing/.

Tabelle della memoria dedicata ai server:

MEMORIA IPUMS 500K		
SGA+PGA	**1400 Mb**	**Oracle**
MEM JVM	**1400 Mb**	**Neo4j**
RAM MACCHINA	**8Gb**	**Macchina Fisica**

Tabella 3.1.a

MEMORIA IPUMS 500K IDX		
SGA+PGA	**1400 Mb**	**Oracle**
MEM JVM	**1400 Mb**	**Neo4j**
RAM MACCHINA	**8Gb**	**Macchina Fisica**

Tabella 3.1.b

Tabella 3.1.a / 3.1.b: Ammontare di memoria dedicata ai server Oracle e Neo4j al momento dell'esecuzione dei test sui database IPUMS 500K e IPUMS 500K IDX.

MEMORIA IPUMS 1M		
SGA+PGA	2800 Mb	Oracle
MEM JVM	2800 Mb	Neo4j
RAM MACCHINA	8Gb	Macchina Fisica

Tabella 3.2.a

MEMORIA IPUMS 1M IDX		
SGA+PGA	2800 Mb	Oracle
MEM JVM	2800 Mb	Neo4j
RAM MACCHINA	8Gb	Macchina Fisica

Tabella 3.2.b

Tabella 3.2.a / 3.2.b: Ammontare di memoria dedicata ai server Oracle e Neo4j al momento dell'esecuzione dei test sui database IPUMS 1M e IPUMS 1M IDX.

3.2 Analisi Generale

La prima serie di interrogazioni, a cui sono state sottoposte le due tecnologie, è nata con lo scopo di fornire dei dati quantitativi che possano in qualche modo dare un'idea del comportamento della nuova tecnologia.

QUERY	Aggregazioni	Selezione	Predicato di selettività
1	City	nessuna	1
2	Region	nessuna	1
3	Region	5 Region su 10	0,5
4	City, Race	nessuna	1
5	Region, Mrn	nessuna	1
6	Region, Mrn	5 Region su 10, 3 Mrn su 10	0,15
7	City, Race, Occupation	nessuna	1
8	Region, Mrn Category,	nessuna	1
9	Region, Category, Mrn	5 Region su 10, 3 Mrn su 10, 5 Categpry su 10	0,075
10	State, Mrn, Sex Branch, Year,	5 State su 1112, 5 Branch su 27	0.000833

Tabella 3.3: Tabella che riporta le caratteristiche principali delle 10 query con cui è stata effettuata la prima analisi delle tecnologie.

Il predicato di selettività è stato calcolato supponendo che il numero delle relazioni che connettono i vari nodi :Census ai nodi dimensionali, e analogamente il numero di record della Fact Table per valore dimensionale, sia bilanciato.

Risultati:

IPUMS 500K IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	N° Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	1,104	18,491	1077	1	1	1	16,75
2	2,238	21,044	10	1	3	1	9,40
3	2,541	13,12	5	1	3	0,5	5,16
4	2,045	29,204	22740	2	2	1	14,28
5	2,462	33,595	38	2	6	1	13,65
6	2,159	19,905	10	2	6	0,15	9,22
7	15,874	224,391	347735	3	3	1	14,14
8	2,500	51,839	269	3	9	1	20,73
9	2,621	30,938	36	3	9	0,075	11,80
10	2,494	45,661	57	5	13	0,00083	18,31

Tabella 3.4: Tabella che riporta i risultati dei test eseguiti su IPUMS 500K IDX

IPUMS 500K

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	N° Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	2,041	66,019	1077	1	1	1	32,34
2	2,663	53,805	10	1	3	1	20,21
3	2,336	120,491	5	1	3	0,5	51,58
4	2,837	69,226	22740	2	2	1	24,40
5	2,529	80,746	38	2	6	1	31,93
6	2,420	51,734	10	2	6	0,15	21,38
7	16,578	226,239	347735	3	3	1	13,65
8	3,110	55,766	269	3	9	1	17,93
9	2,572	36,562	36	3	9	0,075	14,21
10	3,109	50,061	57	5	13	0,00083	16,10

Tabella 3.5: Tabella che riporta i risultati dei test eseguiti su IPUMS 500K.

IPUMS 1M IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	2,822	37,48	1077	1	1	1	13,28
2	2,791	48,812	10	1	3	1	17,49
3	2,520	31,152	5	1	3	0,5	12,36
4	3,872	47,039	30166	2	2	1	12,15
5	3,077	57,003	39	2	6	1	18,52
6	2,723	38,486	10	2	6	0,15	14,13
7	32,720	375,232	550430	3	3	1	11,47
8	3,256	113,934	286	3	9	1	34,99
9	3,266	72,756	37	3	9	0,075	22,27
10	2,931	107,268	72	5	13	0,00083	36,59

Tabella 3.6: Tabella che riporta i risultati dei test eseguiti su IPUMS 1M IDX.

IPUMS 1M

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	4,055	91,634	1077	1	1	1	22,60
2	4,530	119,809	10	1	3	1	26,45
3	4,013	259,741	5	1	3	0,5	64,73
4	5,126	153,139	30166	2	2	1	29,88
5	4,775	109,004	39	2	6	1	22,83
6	4,261	108,179	10	2	6	0,15	25,39
7	33,787	377,06	550430	3	3	1	11,16
8	4,821	110,026	286	3	9	1	22,82
9	4,533	67,696	37	3	9	0,075	14,93
10	4,160	104,522	72	5	13	0,00083	25,13

Tabella 3.7: Tabella che riporta i risultati dei test eseguiti su IPUMS 1M.

Ai tempi di esecuzione delle varie query, sono stati affiancati dei valori che riportano le caratteristiche principali delle interrogazioni:

- **Num. Elem. Output**: Numero di elementi di output restituiti dalla query.
- **Join ORACLE**: Numero di Join che Oracle deve effettuare per poter estrarre il risultato richiesto dalla query SQL.
- **N° Hop (Neo4j):** E' un valore che indica il numero di relazioni di cui è composto il pattern della query Cypher.

- **Selettività:** Viene indicato il predicato di selettività delle query. Il valore è indicativo ed è stato calcolato presupponendo che il dataset non presenti situazioni anomale.

Inoltre, per ogni singola interrogazione è stato calcolato il rapporto fra il tempo di esecuzione della query SQL e quella Cypher.

$$Rapporto\ Tempi = \frac{Tempo\ Esecuzione\ Query\ Cypher}{Tempo\ Esecuzione\ Query\ SQL}$$

Maggiore è il valore che assume questo dato, maggiore è la capacità di Oracle di affrontare meglio l'esecuzione dell'interrogazione rispetto a Neo4j.

Discussione dei risultati

Dai risultati ottenuti è già possibile fare delle piccole constatazioni. Dalle query 2 e 3, 5 e 6, 8 e 9, si può notare che a parità di join da parte di Oracle e Hop (numero di relazioni di cui è composto il pattern dell'interrogazione Cypher) da parte di Neo4j, al diminuire del predicato di selettività, diminuisce anche il rapporto dei tempi. Ovvero, Neo4j quando effettua una ricerca mirata su un sottogruppo di dati, tende a "guadagnare terreno" nei confronti di Oracle. E' inoltre possibile osservare che il miglioramento è assai più marcato nei database in cui sono presenti indici, questo perché nelle query Cypher (Neo4j) impiegate in questi DB è stata utilizzata (quasi su tutte) la clausola START, la quale consente di migliorare considerevolmente le prestazioni di una query all'aumentare della selettività.

Un altro aspetto degno di nota è il comportamento che assumono i risultati man mano che il numero di Join di Oracle aumenta. Si considerino le query 1, 4 e 7. I risultati mostrano un considerevole abbassamento dei rapporti dei tempi all'aumentare del numero di join che Oracle è costretto ad eseguire per poter fornire il risultato finale. La Tabella dei risultati che riporta i dati di IPUMS 1M mostra un aumento sostanziale del rapporto dei tempi per la query n°4, dovuto ad un peggioramento delle prestazione di Neo4j. Questo sostanziale calo è probabilmente dovuto ad un piano di esecuzione non ottimale.

Infine, un ulteriore caso di studio è come varia il rapporto dei tempi in funzione del n° di Hop (salti) di cui è composto il pattern delle query di Neo4j. Per studiare questo comportamento sono stati messi a confronto i rapporti dei tempi delle query N° 2, 5 e 8. Queste query sono caratterizzate dal sostanziale aumento di salti da una query all'altra. E' da sottolineare il fatto che il numero di hop che le query Cypher sono costrette ad effettuare per ottenere il risultato, è assai superiore rispetto al numero di join che invece Oracle deve fare per ottenere il medesimo risultato.

3.3 Analisi Dettagliata

In base alle osservazioni effettuate sui risultati delle precedenti query, si è deciso di studiare più approfonditamente i tre concetti chiave che hanno interessato la precedente discussione dei risultati ovvero:

1) La selettività.
2) Il numero di Join delle query SQL.
3) Il numero di Hop delle query Cypher.

Perciò sono state creati tre gruppi di query, uno per ogni caso di studio, composti da tre interrogazioni ciascuno. Le query appartenenti allo stesso caso di studio possiedono la stessa struttura di base, ma differiscono dalle interrogazioni in un solo aspetto, che dipende dal caso di studio stesso. Ad esempio, per la selettività, le tre query andranno a toccare le stesse tabelle/tipi di nodi, ma differiranno nel predicato di selettività, il quale dovrà essere via via sempre più piccolo.

3.3.1 Selettività

Il primo gruppo è nato con l'obiettivo di studiare il comportamento delle due tecnologie all'aumentare della selettività. Per far si che i tempi di esecuzione dipendessero solamente dal predicato di selettività e non da altri fattori, le tre query sono state strutturate in modo tale da effettuare il minor numero di join/hop, senza però dover scrivere al tempo stesso un'interrogazione estremamente lunga.

Perciò la struttura di base è la seguente:

```
SELECT C.REGION, sum(FT.PERWT)
FROM CITY C, FACT500K FT
WHERE C.IDCITY = FT.CITY
AND (Selettività)
GROUP BY C.REGION;
```

L'interrogazione sql è stata tradotta nelle seguenti query Cypher:

Server Neo4j con indici

```
START
r=node:RegionIdx(Selettività)
MATCH (f:Census)-[*3]->(r)
RETURN  id(r), sum(f.PERWT);
```

Server Neo4j senza indici

```
MATCH
(f:Census)-[*3]->(r:Region)
WHERE Selettività
RETURN  id(r), sum(f.PERWT);
```

La seguente tabella mostra le caratteristiche principali delle query.

ID	Aggregazioni	Predicato di Selettività	Descrizione
1	**Region**	**0,5**	**selezione 5 Region su 10**
2	**Region**	**0,3**	**selezione 3 Region su 10**
3	**Region**	**0,1**	**selezione 1 Region su 10**

Tabella 3.8: Caratteristiche delle interrogazioni.

IPUMS 500K IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	**1,6347**	**17,486**	**5**	**1**	**3**	**0,5**	**10,7**
2	**1,6865**	**14,461**	**3**	**1**	**3**	**0,3**	**8,58**
3	**1,5489**	**9,971**	**1**	**1**	**3**	**0,1**	**6,44**

Tabella 3.9: Risultati del gruppo di query eseguite su IPUMS 500K.

IPUMS 500K

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	1,6318	99,621	5	1	3	0,5	61,05
2	1,8533	88,343	3	1	3	0,3	47,67
3	1,7294	62,525	1	1	3	0,1	36,15

Tabella 3.10: Risultati del gruppo di query eseguite su IPUMS 500K..

IPUMS 1M IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	3,6099	29,269	5	1	3	0,5	8,11
2	3,2853	26,021	3	1	3	0,3	7,92
3	3,4696	17,651	1	1	3	0,1	5,09

Tabella 3.11: Risultati del gruppo di query eseguite su IPUMS 1M IDX.

IPUMS 1M

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Selettività	Rapporto Tempi
1	1,7755	240,319	5	1	3	0,5	135,353
2	2,017	174,462	3	1	3	0,3	86,496
3	1,7316	118,466	1	1	3	0,10	68,414

Tabella 3.12: Risultati del gruppo di query eseguite su IPUMS 1M..

I risultati confermano quello che è stato detto in precedenza. A parità del numero di Join da parte di Oracle e Hop di Neo4j, al diminuire del predicato di selettività diminuisce anche il rapporto dei tempi, ovvero Neo4j non solo migliora le proprie performance qualora si effettua una ricerca mirata, ma risulta avere un approccio più efficacie di quello utilizzato dal proprio rivale.

I seguenti grafici permettono di farsi un'idea migliore.

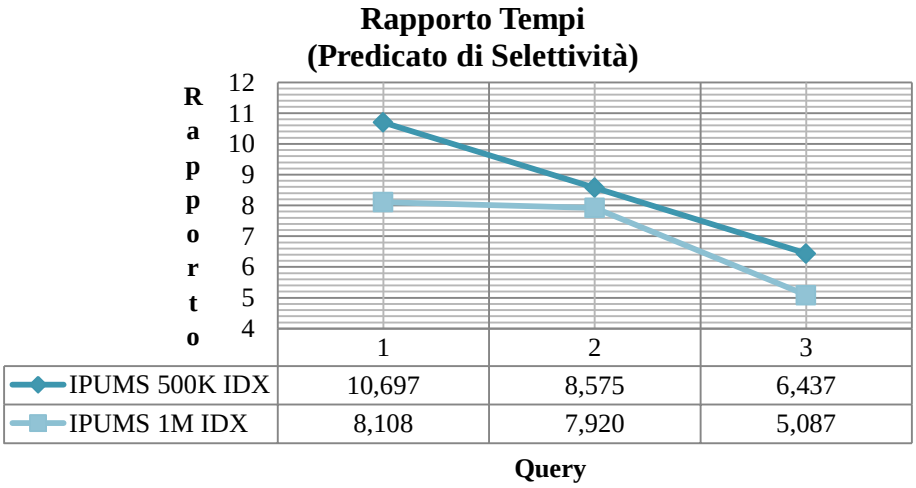

Figura 3.4: Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K IDX e IPUMS 1M IDX.

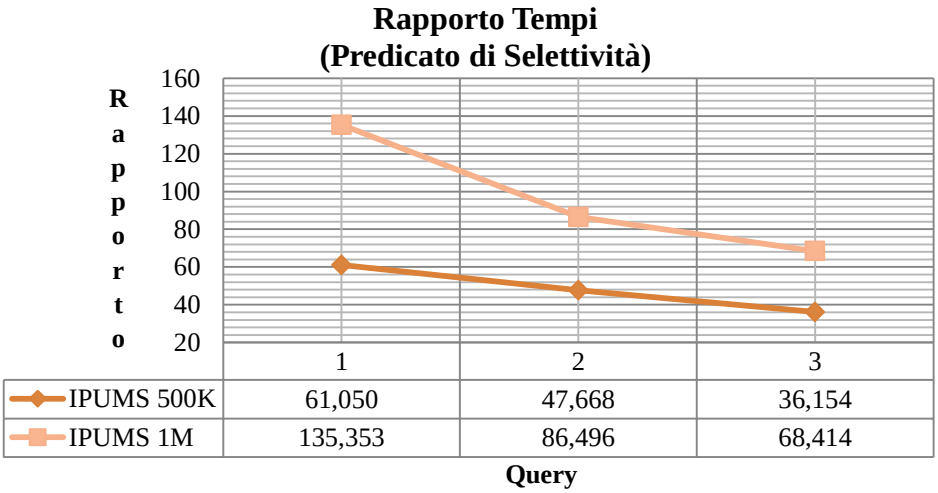

Figura 3.5 Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K e IPUMS 1M.

E' da notare che, i valori dei rapporti dei tempi dei database in cui non vi sono indici sono molto più alti rispetto a quelli in cui sono presenti. Questa sostanziale differenza dipende dal fatto che per le query Cypher scritte per le basi dati in cui sono stati costruiti gli indici, è stata impiegata la clausola START. I meccanismi di questa clausola sono uno dei maggiori punti di forza di Neo4j.

Inoltre, nel primo grafico, i rapporti dei tempi del database IPUMS 1M IDX sono più bassi rispetto a quelli del suo fratello più piccolo IPUMS 500K IDX. Invece, nel grafico che riporta i valori dei rapporti dei tempi dei due database senza indici avviene esattamente il contrario. Questo comportamento dipende sempre dal fatto che, per uno sono state impiegate delle query munite della clausola START, per l'altro no. Ciò sta significare che, la clausola START non solo consente di migliorare le prestazioni, ma consente a Neo4j di scalare efficacemente. Comunque, indifferentemente dalla clausola START, il Graph DBMS risulta essere più efficacie ogniqualvolta ti tenta di recuperare un piccolo sottogruppo di dati su cui poi operare.

3.3.2 N° di Join di Oracle

IL secondo gruppo di query è stato sviluppato con lo scopo di verificare se Neo4j guadagna realmente terreno nei confronti di Oracle all'aumentare dello stesso numero di join da parte dell'RDBMS, e hop da parte del Graph DBMS. Per evitare che altri fattori potessero influire sul comportamento dell'interrogazione e di conseguenza sui tempi di esecuzione, le query del secondo gruppo sono caratterizzate da aggregazioni che interessano solamente il livello più basso di specializzazione delle tabelle dimensionali, ovvero le Città per i luoghi, la Razza per l'etnia, e così via.

Esempio:

```
SELECT C.IDCITY, R.IDRACE, sum(FT.PERWT)
FROM FACT500K FT, RACE R, CITY C
WHERE R.IDRACE = FT.RACED
AND C.IDCITY = FT.CITY
GROUP BY C.IDCITY, R.IDRACE;
```

Server Neo4j con indici

```
START c=node:CityIdx("CITY:*")
MATCH (c)<--(f:Census)-->(r:Race)
RETURN id(c), id(r), sum(f.PERWT);
```

Server Neo4j senza indici

```
MATCH
(c:City)<--(f:Census)-->(r:Race)
RETURN id(c), id(r), sum(f.PERWT);
```

La seguente tabella mostra le caratteristiche principali delle query.

ID	Aggregazioni	N° Join Oracle	Descrizione
1	City	1	Join tra FACT e CITY
2	City, Race	3	Join tra FACT e CITY Join tra FACT e RACE
3	City, Race, Occupation	3	Join tra FACT e CITY Join tra FACT e RACE Join tra FACT e OCCUPATION

Tabella 3.13: Caratteristiche delle interrogazioni.

IPUMS 500K IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	2,1544	20,539	1077	1	1	1	9,53
2	3,223	54,186	22740	2	2	1	16,81
3	27,7889	197,094	347735	3	3	1	7,09

Tabella 3.14: Risultati del gruppo di query eseguite su IPUMS 500K.

IPUMS 500K

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	1,7768	51,495	1077	1	1	1	28,98
2	2,5381	80,5329	22740	2	2	1	31,73
3	28,4372	225,264	347735	3	3	1	7,92

Tabella 3.15: Risultati del gruppo di query eseguite su IPUMS 500K.

IPUMS 1M IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	3,245	48,661	1077	1	1	1	14,99
2	3,8145	93,147	30166	2	2	1	24,12
3	31,6489	324,34	550430	3	3	1	10,25

Tabella 3.16: Risultati del gruppo di query eseguite su IPUMS 1M IDX.

IPUMS 1M

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Selettività	Rapporto Tempi
1	1,7359	100,839	1077	1	1	1	58,09
2	2,5913	149,245	30166	2	2	1	57,60
3	33,5415	387,146	550430	3	3	1	11,54

Tabella 3.17: Risultati del gruppo di query eseguite su IPUMS 1M.

Anche in questo caso i risultati confermato quello che è stato detto nella discussione dei risultati della sezione 3.2. All'aumentare del numero di Join che Oracle deve effettuare per ottenere il risultato richiesto dalla query, e a parità di hop di cui è composto il pattern dell'interrogazione Cypher, il rapporto dei tempi cala.

I seguenti grafici consentono di apprezzare meglio questo fatto.

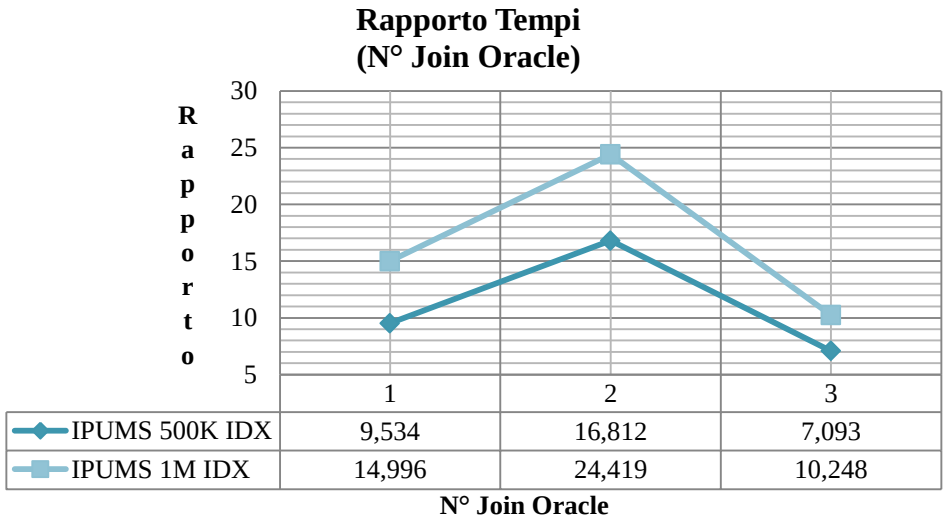

Figura 3.6: Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K IDX e IPUMS 1M IDX.

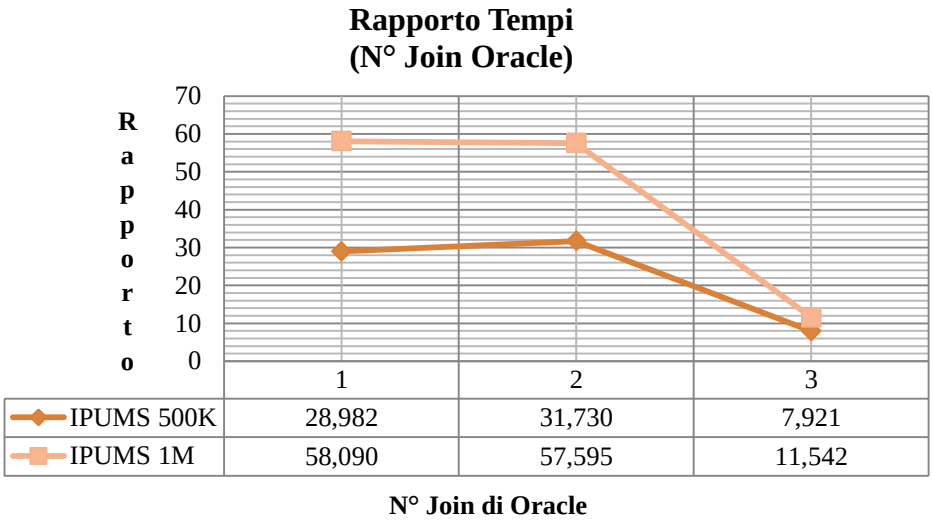

Figura 3.7: Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K e IPUMS 1M.

Anche se tendenzialmente risultata vero ciò che è stato detto in precedenza, si nota l'impressionante impennata dei rapporti dei tempi delle query n°5, eseguita sui database in cui vi sono indici. Questo drastico aumento non è dovuto ad un peggioramento da parte di Neo4j, il quale risulta mantenere lo stesso comportamento in tutti i casi, bensì è causato da il modo con cui Oracle affronta l'esecuzione di questa particolare interrogazione. Per ognuna, delle tre query, eseguite sul basi dati con gli indici, Oracle crea un piano di esecuzione che non ha nulla che fare con quello delle altre. Al contrario per le query eseguite sui database in assenza di indici Oracle assume più o meno lo stesso approccio per tutte e tre le interrogazioni.

Più precisamente, accade che per i database in cui non vi sono indici, Oracle esegue sempre i join in cascata accedendo alle tabelle per mezzo di un FULL SCAN, perciò i join vengono eseguiti senza l'ausilio di indici che consento di raggiungere rapidamente i record da associare. Ciò significa che il costo di esecuzione di una query eseguita sui database senza indici, cresce linearmente con l'aumentare dei join. Neo4j, invece, esegue le operazioni di associazione dei dati semplicemente percorrendo le relazioni che connetto i vari nodi della base dati, quindi anche se non ci sono indici di supporto alla sue interrogazioni, riesce comunque ad effettuare questa operazione in maniera ottimale.

Nel caso dei database in cui sono stati costruiti gli indici la situazione cambia. Per la query n°4, Oracle non deve far altro che effettuare un Join tra la tabella CITY e la tabella FACT. Per fa ciò, usufruisce del NESTED LOOP, ponendo come relazione esterna la tabella dei dati statistici e come interna quella delle città, però prima di porle in join effettua l'aggregazione dei dati direttamente quando accede alla FACT Table, e sfrutta l'indice per raggiungere rapidamente i record della tabella dimensionale.

Piano di esecuzione query 4:

- **SELECT STATEMENT**
  - **HASH (GROUP BY)**
  - **NESTED LOOPS (JOIN)**
    - **VIEW**
      - **HASH (GROUP BY)**
      - **TABLE ACCESS FULL – FACT TABLE**
    - **INDEX (UNIQUE SCAN) – CITY**

NOTA: L'operazione di raggruppamento effettuata quando viene eseguito l'accesso sequenziale alla tabella dei FACT è un escamotage di Oracle che gli consente di effettuare l'operazione di GROUP BY senza dover però prima accedere alla tabella delle città. Ovvero, Oracle è talmente "intelligente" da capire che, raggruppare per IDCITY della tabella CITY , oppure per la chiave CITY della tabella dei FACT è la stessa cosa, perciò decide di creare un vista intermedia con già i dati raggruppati per IDCITY. In effetti il costo della query risulta limitarsi a questa operazione di aggregazione e creazione della vista.

Quindi, a parte l'escamotage del group by, non vi è nulla di strano. Il costo della query per il database IPUMS 500K IDX è pari a 1339, invece per IPUMS 1M IDX è 2548.

Nel caso della query n° 5, Oracle per evitare di perdere colpi dovuti al crescere del numero di join decide di affrontare la query con un approccio totalmente differente:

Piano di esecuzione query n°5:

- **SELECT STATEMENT**
  - **HASH (GROUP BY)**
  - **MERGE JOIN**
    - **Sort (JOIN)**
    - **VIEW**
      - **HASH (GROUP BY)**
      - **HASH JOIN**
        - **INDEX FULL SCAN - RACE**
        - **TABLE ACCESS FULL – FACT TABLE**
    - **Sort (JOIN)**
    - **INDEX (FAST FULL SCAN) – CITY**

In questo caso, prima viene effettuato il Join tra la tabella FACT e la tabella RACE, dopo di che viene eseguito il join tra il risultato intermedio nato dal precedente join (a cui è stato effettuata una prima operazione di raggruppamento) e la tabella CITY, sula quale è stato effettuato un accesso rapido via indice. Perciò Oracle non esegue dei join in cascata, ma trova altre alternative per evitare che di incombere nel problema che nasce dal principio stesso di Join tra tabelle. Il costo finale dell'interrogazione è 2665 per IPUMS 500K IDX, e 4811 per IPUMS 1M IDX.

Piano di esecuzione query n°6:

- **SELECT STATEMENT**
  - **HASH (GROUP BY)**
  - **HASH JOIN**
    - **INDEX (FAST FULL SCAN) – OCCUPATION**
    - **VIEW**
      - **HASH (GROUP BY)**
      - **HASH JOIN**
        - **INDEX FULL SCAN - CITY**
        - **VIEW**
          - **HASH (GROUP BY)**
          - **HASH JOIN**
            - **INDEX FULL SCAN - RACE**
            - **TABLE ACCESS FULL – FACT TABLE**

Per quanto riguarda l'ultima delle tre query, la numero 6, avviene l'inevitabile. Ovvero Oracle non più fare a meno di porre i join tra le tabelle in cascata. In questo caso si avrà un costo di 8858 per IPUMS 500K IDX e 17576 per IPUMS 1M IDX.

In definitiva, si può notare che finché il numero di join è basso, Oracle riesce in qualche modo a contenere i costi, quando il numero diviene elevato, non è più in grado di limitarli. Al contrario Neo4j, riesce in ogni caso a contenere l'attraversamento del grafo solo a quei nodi e quelle relazioni che davvero interessano la query, perciò riesce a guadagnare terreno nei confronti dell'RDBMS.

3.3.3 N° Hop Neo4j

L'ultimo gruppo è stato strutturato in modo tale da poter studiare quanto il numero di hop di cui sono composti i pattern delle query Cypher, possano influire sulle performance. A questo scopo è stato scelto di concentrare le query su un'unica dimensione, in questo modo le tre query di cui è composto questo gruppo, andranno a "toccare" gli stessi Census, eliminando la variante della quantità di dati che le varie interrogazioni andranno ad estrarre. Inoltre, con quest'ultimo set di query è possibile anche studiare come reagisce la tecnologia NoSQL, man mano che il livello di aggregazione aumenta.

ID	Aggregazioni	N° Hop Neo4j	Descrizione
1	City	1	(City)<--(Census)
2	State	2	(State)<--(City)<--(Census)
3	Region	3	(Region)<--(State)<--(City)<--(Census)

Tabella 3.18: Caratteristiche delle interrogazioni.

IPUMS 500K IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	2,1544	20,694	1077	1	1	1	9,61
2	1,8901	22,935	52	1	2	1	12,13
3	2,2029	24,367	10	1	3	1	11,06

Tabella 3.19: Risultati del gruppo di query eseguite su IPUMS 500K.

IPUMS 500K

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	1,7294	50,939	1077	1	1	1	29,46
2	1,494	50,996	52	1	2	1	34,13
3	1,9672	54,773	10	1	3	1	27,84

Tabella 3.20: Risultati del gruppo di query eseguite su IPUMS 500K.

IPUMS 1M IDX

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Predicato di Selettività	Rapporto Tempi
1	3,245	35,756	1077	1	1	1	11,02
2	3,1327	45,864	52	1	2	1	14,64
3	3,0636	45,443	10	1	3	1	14,83

Tabella 3.21: Risultati del gruppo di query eseguite su IPUMS 1M IDX.

IPUMS 1M

QUERY	Tempi ORACLE (secondi)	Tempi NEO4J (secondi)	Num. Elem. Output	JOIN Oracle	N° HOP (Neo4j)	Selettività	Rapporto Tempi
1	1,7359	97,454	1077	1	1	1	56,14
2	1,6783	103,225	52	1	2	1	61,50
3	2,0982	105,909	10	1	3	1	50,48

Tabella 3.22: Risultati del gruppo di query eseguite su IPUMS 1M.

Per quest'ultimo caso di studio è meglio affrontare in modo distaccato la discussione dei risultati delle query che sono state eseguite sui database con indici e quelle no.

Database con indici

Il prossimo grafico mostra che all'aumentare del numero di Hop, di cui sono composti i pattern delle query, cresce anche il rapporto dei tempi.

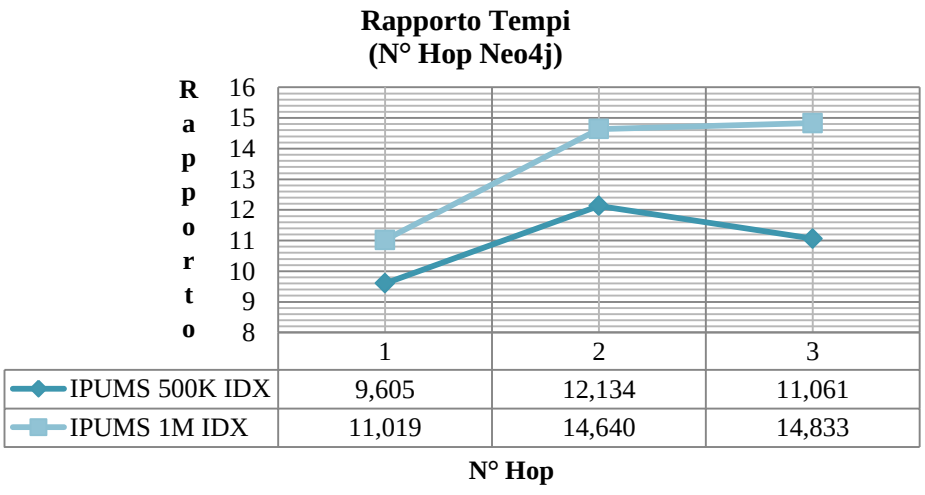

Figura 3.8: Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K IDX e IPUMS 1M IDX.

La crescita è causata da un solo fattore: lo schema di IPUMS di Oracle non è normalizzato, perciò l'RDBMS è deve effettuare un solo join per tutte e tre le interrogazioni, inoltre in presenza di indici effettua questi join in modo ottimale. Tuttavia, è da notare una caratteristica dei risultati ottenuti. Ovvero, anche se i rapporti dei tempi aumentano, la crescita non così marcata, e le motivazioni che sono alla base di questo fatto sono 2.

Per prima cosa, le query Neo4j accedono via START clausole ai nodi dimensionali, dai quali fanno partire l'attraversamento, e per costruire delle interrogazioni che non dovessero essere influenzate da altri fattori oltre quello del caso di studio, è stato scelto di salire la struttura gerarchica di una delle cinque dimensioni . Questo significa che la prima query farà partire la propria ricerca dai 1112 nodi :City, la seconda dai 53 di tipo :State e la terza dai 10 :Region. Questo avvantaggia Neo4j, proprio perché man mano che si sale la struttura diminuisco i nodi ancora di partenza.

In secondo luogo, il motivo per cui il rapporto dei tempi cresce lievemente, interessa il processo di caching delle informazioni più utilizzate. Le query, quando vengono eseguite per la prima volta (ovvero in assenza di dati in memoria cache), per fornire l'output devono creare e caricare in memoria tutti gli oggetti utili. Questo processo di creazione e mantenimento in memoria centrale di questi oggetti è fondamentale affinché Neo4j possa incrementare le proprie prestazioni al presentarsi di successive richieste. Però se la mole di dati da dover caricare è alta, si ha una diminuzione della velocità di restituzione dell'output. Ritornando al caso di studio, man mano che aumentano gli hop delle query si sale anche l'albero gerarchico della dimensione, quindi le interrogazioni forniscono un minor numero di elementi di output, perciò devono caricare in memoria meno oggetti, e di conseguenza è più rapida la restituzione dell'output.

Database senza Indici

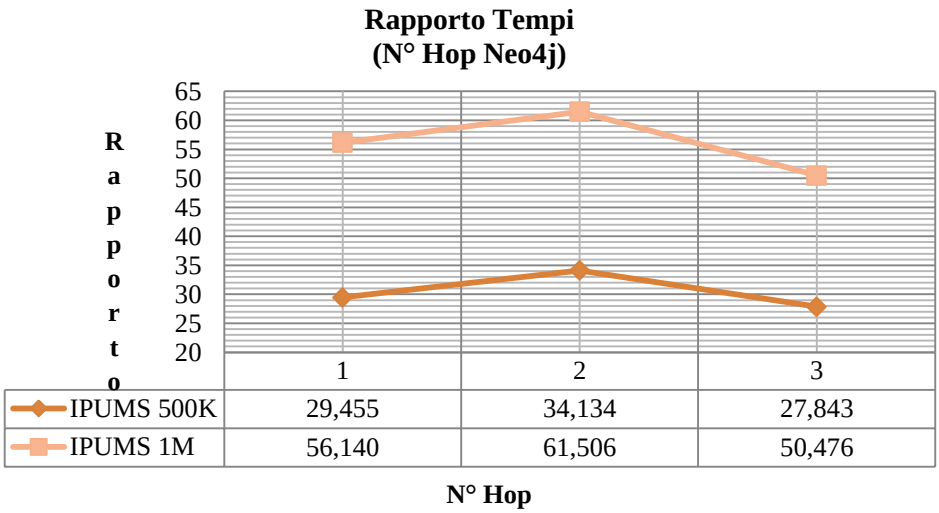

Figura 3.9: Grafico che mostra il rapporto dei tempi delle query eseguite su IPUMS 500K e IPUMS 1M.

I risultati ottenuti per i database in cui non vi sono gli indici mostrano un cambio totale di direzione. All'aumentare del numero di Hop diminuisce il rapporto dei tempi. Questo accade perché il motivo per il quale Neo4j perdeva terreno, erano i join di Oracle effettuati con l'ausilio degli indici. Perduti gli indici, Oracle è costretto ad accedere alle tabelle per mezzo di uno FUL SCAN TABLE, perciò i sui join non sono più ottimizzati. Al contrario Neo4j, continuerà ad relazionare i dati semplicemente attraversando il grafo.

In definitiva, dai risultati si può constatare che Neo4j gestisce in modo eccellente gli attraversamenti.

Le aggregazioni

I rapporti dei tempi ci permettono di constatare che maggiore è il livello di aggregazione, migliore è la reazione della tecnologia NoSQL. Questo accade perché, le aggregazioni di alto livello tendono a fornire meno elementi di output, e come è già stato ribadito diverse volte in precedenza, più è limitato l'output e migliore è il tempo di risposta del Graph DBMS.

Capitolo IV
Conclusioni

I test eseguiti mettono in luce una serie di comportamenti di cui si è ampiamente discusso nel precedente capitolo dei test, tuttavia non è stato ancora fatto notare l'aspetto più eclatante: l'enorme divario tra i tempi di esecuzione di Oracle quelli di Neo4j.

Questo abisso tra i tempi delle due tecnologie è dovuto alla base dati impiegata per i test. IPUMS è un database nato è strutturato per il mondo OLAP – Relazionale. Ciò significa che la struttura è stata concepita per far fronte ad ogni possibile debolezza degli RDBMS. Un esempio eclatante è il fatto che le tabelle dimensionali sono de normalizzate. Inoltre è stato strutturato per poter eseguire poche operazioni di estrazione delle informazioni che interessano grandi quantità di dati alla volta. Al contrario, Neo4j è un DBMS transazionale, quindi nasce con l'idea di gestire una enorme mole di transazioni che vanno a toccare solo un piccolo sottogruppo di dati. Oltretutto, non è un base dati in cui le relazioni ne fanno da padrone, anche se è vero che ci sono 5 relazioni per ogni record della Fact Table. Tuttavia, le relazioni tra le tabelle sono tutte del tipo 1 a N. I Graph DBMS sono stati progettati per far fronte al problema che nasce dall'interrogare basi dati la cui struttura è composta da tante e grandi entità connesse da una vasta gamma di relazioni del tipo N a N. In poche parole IPUMS non è un database altamente connesso, ma è l'esatto contrario.

Un altro argomento che è stato menzionato raramente nei precedenti sotto-capitoli, è la *scalabilità*. Dai risultati ottenuti, si nota che i rapporti dei tempi delle query eseguite sulle basi dati IPUMS da un milione di Census, sono quasi sempre più alti rispetto a quelli ottenuti dai tempi di esecuzione delle query lanciate su IPUMS 500K. Tutto ciò, indicherebbe che la capacità di Oracle nel gestire maggiori informazioni è migliore rispetto a quella di Neo4j. Tuttavia, questo dato indica solamente che Oracle è in grado di scalare meglio all'aumentare della mole di informazioni di cui è composta la basi dati rispetto a Neo4j, solo per questa tipologia di data base.

Come è stato detto più e più volte in precedenza, IPUMS è un dateset nato e strutturato per il DBMS della tipologia di Oracle, quindi risulta ovvio che in questo caso, Oracle è in grado di scalare meglio rispetto a Neo4j. In generale, dai risultati ottenuti, non è possibile affermare che gli RDBMS siano in grado di scalare si meglio rispetto ai Graph DBMS, e viceversa.

In ogni caso, anche se i tempi di esecuzione sono a sfavore della tecnologia NoSQL, i rapporti dei tempi confermano tutto ciò che viene dichiarato sulle potenzialità dei Graph DBMS:

- Sono efficaci qualora si effettua una ricerca mirata.
- Sono ottimizzati per percorrere le relazioni che connettono le informazioni.

Si consideri Ebay, il quale sfrutta proprio Neo4j come piattaforma di immagazzinamento e gestione dei dati, e si consideri l'esempio degli articoli che vengono proposti ad un tipico utente della piattaforma web americana. Questa lista degli articoli viene solitamente ricavata, dagli articoli che ha ricercato in precedenza l'utente, dagli articoli associati a tutti quelli che sono stati visionati dall'utente nelle precedenti ricerche e dagli articoli della stessa categoria che hanno suscitato l'interesse della maggior parte degli utenti di Ebay.

A un Graph DBMS gli basterebbe ricercare il Nodo che rappresenta l'utente di interesse, effettuando così una ricerca mirata all'interno di un'enorme mole di informazioni, e contare il numero maggiore di relazioni che connettono l'utente di Ebay agli articoli visionati, per poi da li verificarne tutti gli altri articoli associati e contemporaneamente, in base alla categoria di articolo, effettuare una stima degli articoli con il maggior numero di relazioni che li connettono ai nodi utenti.

Invece, un RDBMS dovrebbe mettere in join diverse volte, la tabella degli articoli e quella degli utenti, delle categorie e così via, tutte tabelle estremamente grandi in un contesto come quello di Ebay, finendo per soccombere.

Quindi, risulta chiaro il perché questo nuovo modo di organizzare e strutturare le informazioni stia prendendo piede all'interno dei grandi colossi tecnologici.

Bibliografia

[1] *Wikipedia*. (http://www.wikipedia.it)

[2] Ian Robison, Jim Wabber. *The Graph Database*. O'Reilly Media Inc., 2013.

[3] *Manuale Online di Neo4j v2.0M*. (Febbraio - Marzo 2014) (http://docs.neo4j.org/chunked/stable/)

[4] M. Golfarelli, S. Rizzi. *Data Warehouse Design: Modern Principles and Methodologie*s. McGraw-Hill, 2009.

[5] Săso Džeroski. Jŏzef . Articolo: *Multi-Relational DataMining: An Introduction*. Stefan Institute Jamova 39, SI-1000 Ljubljana, Slovenia.

[6] R. A. Elmasri, S.B. Navathe. *Sistemi di basi di dati - Complementi*. Pearson, 2005.

[7] *NOSQL Databases*. http://www.nosql-database.org/

[8] Minnesota Population Center. *Integrated public use microdata series* (2008). http://www.ipums.org