
POLITECNICO DI TORINO
Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

# Implementazione di database grafo Neo4j in un CRM aziendale

**Relatore**
prof. Paolo Garza

Marco Musso

Anno accademico 2017-2018

# Riassunto

Da un'analisi iniziata dello stato dell'arte, è emerso che, attualmente, non esiste una soluzione CRM open source, completamente free, sviluppata e focalizzata per il mercato delle aziende, organizzazioni e liberi professionisti nello scenario italiano. La stessa analisi di mercato, ci ha mostrato che attualmente non sembra esserci nulla che spicchi come grado di innovazione tra le piattaforme presenti.

Ciò che esiste è una serie di software CRM, più o meno famosi, sviluppati da aziende straniere, che occupano circa la stessa quota di mercato [17]. Il sistema di licensing più diffuso è quello della doppia licenza, free per la versione community, a pagamento (con varie formule) per la versione commerciale.

Parallelamente all'analisi di mercato precedente, ne è stata condotta un'altra, riguardante le caratteristiche tecniche, ponendo particolare attenzione al concetto di relazione tra gli "oggetti" del CRM. Da questa analisi, risulta che, attualmente, nessuna soluzione stia implementando le innovazioni introdotte dai DB a grafo applicate alle relazioni di un CRM. Studiando le features di ciascuno, infatti, non siamo riusciti ad individuare soluzioni simili a quella oggetto del progetto cmWallet. CmWallet infatti, pone particolare attenzione al tracciamento delle relazioni tra gli elementi interni ed esterni di un'azienda senza andare ad appesantire la struttura database.

L'uso di un database a grafo come Neo4j permette di ottimizzare inserimenti ed estrazioni delle relazioni, mentre normali database relazionali presentano dei limiti che andremo ad analizzare.

L'obiettivo dell'elaborato dunque è quello di capire come mappare un problema aziendale su una struttura a grafo Neo4j e come estrapolarne poi importati risultati.

# Indice

# Capitolo 1

# Introduzione al CRM cmWallet

## 1.1 Cos'è un CRM

Per definizione [14] un CRM (Customer Relationship Management) è un'applicazione progettata e studiata per gestire le relazioni con i clienti. Negli ultimi anni, il mercato non gira solamente attorno al singolo cliente, ma a tutto ciò che circonda il cliente stesso. E' dunque necessario che l'azienda valorizzi l'individuo cliente, ma anche la società e l'ambiente.

Il CRM è uno strumento sempre più utilizzato dalle aziende, il quale deve soddisfare tutte le esigenze gestionali che la riguardano. Lo strumento CRM permette di raccogliere dati ed informazioni, elaborarli ed emettere preziosi risultati i quali verranno poi analizzati al fine di trarre importati considerazioni.

Due obiettivi fondamentali di un CRM sono Customer Acquisition e Customer Retention, i quali esprimono rispettivamente l'abilità di un'azienda ad acquisire e mantenere un cliente nel tempo. In particolare, aumentare la customer retention è un obiettivo importante per la maggior parte delle implementazioni di CRM. Un altro importante punto che esamineremo in questa tesi sarà non solo il trattamento delle relazioni tra azienda e cliente, ma anche le relazioni tra gli elementi interni all'azienda. E' molto importante infatti sapere come gestire ed organizzare le risorse umane di un'azienda al fine di ottimizzarne la produttività. Un database a grafo come Neo4j può quindi aiutarci a costruire una struttura dati che soddisfi queste richieste.

## 1.2 Il progetto CmWallet

CmWallet è un CRM che pone particolare attenzione agli aspetti precedentemente descritti.Una delle tante particolarità di questa applicazione è la completa customizzazione da parte dell'utente: in particolare, l'utente può personalizzare le funzionalità del CRM in base alla sua strategia di business. Questa soluzione permette agli sviluppatori di non concentrarsi sulle singole richieste degli utenti, ma di progredire con lo sviluppo e di migliorare le potenzialità dell'applicazione. L'utente, può scegliere come organizzare i suoi dati, stabilire dei permessi, stabilire dei ruoli e la struttura layout. Oltre a questo aspetto di completa personalizzazione, CmWallet pone particolare attenzione alle funzionalità di **ricerca e ottimizzazione**: sfruttando un database a grafo come Neo4j si è potuto estrapolare importanti risultati che sarebbero stati improbabili utilizzando un database relazionale.

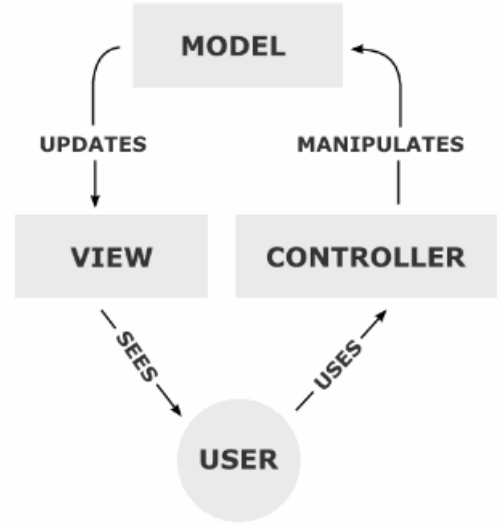
### 1.2.1 Il Framework

Il CRM è stato sviluppato in linguaggio php con Zend Framework 2. Zend Framework [19] (al quale ci riferiremo nel seguito di questa trattazione con la semplice abbreviazione di "ZF") è un web application framework open source creato per semplificare e rendere più efficace sotto ogni punto di vista la produzione di applicativi e servizi web PHP-based. [20] Come vedremo, ZF contiene un insieme di componenti riutilizzabili che rispondono nella sostanza a tutti i requisiti richiesti per un completo applicativo web-based: questi componenti sono scritti interamente in PHP seguendo le migliori pratiche della programmazione ad oggetti.

Quello che rende differente questo framework PHP da altri è la sua particolare struttura "a compartimenti stagni": l'indipendenza dei suoi moduli permette di utilizzare solo il necessario per il proprio progetto, ma allo stesso tempo, se utilizzate insieme, esse si integrano in un substrato incredibilmente potente e semplice da imparare, indirizzando il lavoro dello sviluppatore verso soluzioni basate sui principi di riutilizzo del codice, di estendibilità, leggibilità e manutenzione. ZF ha reso possibile la totale scalabilità e flessibilità dell'applicazione CmWallet, permettendo di "alleggerire" allo sviluppatore il compito di aggiungere moduli e farli interagire tra di loro. In generale, Zend Framework permette agli sviluppatori di concentrasi interamente sulle funzionalità dell'applicazione web, senza preoccuparsi di dover ottimizzare al meglio il codice.

### 1.2.2 Il modello MVC

Zend Framework si basa sul pattern MVC (Model-View-Controller). MVC è un pattern architetturale, sfruttato soprattutto per la programmazione object oriented, in grado di separare la logica di business dalla logica di presentazione. Le tre componenti [21] del pattern rappresentano rispettivamente il modello dei dati (Model), il rendering del modello dei dati (la View) e la logica che accetta gli input e che manipola il Model e la View (il Controller).




Anche se MVC è stato originariamente progettato per il personal computing, è stato adattato ed è ampiamente utilizzato dagli sviluppatori web grazie alla sua enfasi sulla separazione delle competenze: infatti è possibile suddividere in BitBucket

il branch "master", il quale penserà ai Controller e Model, dal branch "design" che penserà alla View, mantenendo quindi le competenze separate in modo netto durante lo sviluppo del progetto.

- Model: Il model è la parte del framework che permette l'accesso ai dati permanenti dell'applicazione ed è definito come "ponte" tra Controller e View. Questo modulo viene definito come "cieco", ovvero non è a conoscenza di quale trattamento verrà applicato ai dati una volta restituiti al Controller e successivamente alla View. Quindi il suo unico scopo è quello di memorizzare i dati nella sua memoria permanente (ad esempio un Database) o prepararli e restituirli a terze parti.

- Controller: Il suo compito è gestire i dati che l'utente inserisce o invia e aggiornare di conseguenza il Model. Nel modello MVC, ad un Controller corrisponde una ed una sola View.

- View: La View è dove vengono visualizzati i dati richiesti al Model. In particolare, dopo che i dati vengono prelevati dal Model e manipolati dal Controller, vengono infine visualizzati nella View. Tradizionalmente, nelle Web Applications create con MVC, la vista è la parte del sistema in cui viene generato e visualizzato l'HTML.

Di seguito possiamo osservare una classica interazione tra Client, View, Controller e Model.

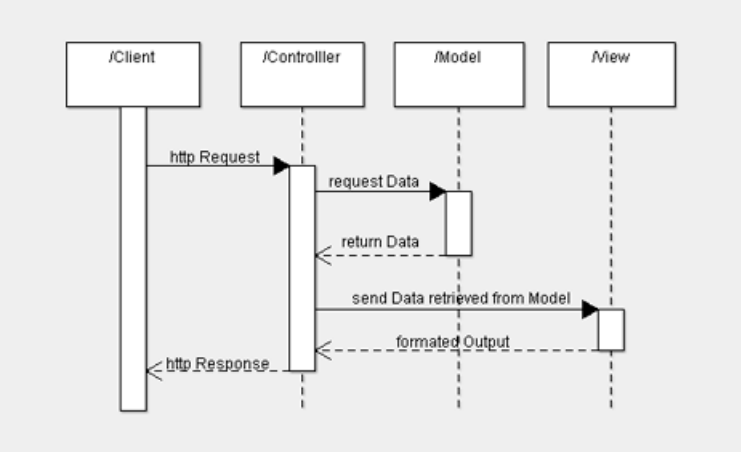



Figura 1.1: Interazione tra gli elementi di un sistema MVC.

E' chiaro dunque come Zend Framework soddisfi perfettamente con le esigenze del progetto CmWallet, nel quale è stato indispensabile mantenere una certa indipendenza dei moduli per migliorare al massimo la scalabilità e flessibilità.

## 1.3 CmWallet e Neo4j

Come accennato in precedenza, CmWallet pone maggiore attenzione sugli aspetti di ottimizzazione delle risorse aziendali. In particolare è stato indispensabile trovare uno strumento per immagazzinare i dati in un modo tale per cui possano essere restituiti già in un modo ottimizzato, senza aggiungere tempi di elaborazione aggiuntivi.

Nei capitoli successivi infatti, vedremo dei casi di studio che riguardano problemi di ottimizzazione di costi di produzione di automobili, ricerche di sottogruppi di persone in base a precisi criteri di ricerca e la gestione di adozioni per una Onlus torinese. Questi tre casi evidenzieranno la necessità di utilizzare uno strumento che immagazzini i dati già in modo ottimizzato e che metta a disposizione giù lui stesso delle funzioni di ricerca, senza trattare i dati successivamente con degli algoritmi costruiti ad hoc.

A tal proposito, è stato indispensabile usare uno strumento come Neo4j. Neo4j, come vedremo, è un database a grafo che permette di immagazzinare dati in un modo totalmente differente da quello che siamo abituati a vedere nei database RDBMS o altri NoSQL come MongoDB. Inizialmente CmWallet non è stato pensato per interagire con un database a grafo: solo dopo uno studio approfondito delle richieste da parte dei clienti e interne all'azienda, ci siamo resi conto che un RDBMS non sarebbe più stato in grado di soddisfare richieste come quelle descritte precedentemente. L'alternativa sarebbe potuta essere l'implementazione di un sistema OLAP (On Line Analytical Processing) ma la nostra idea era quella di trovare uno strumento OLPT in grado anche di analizzare e processare i dati al fine di rispondere alle esigenze di un'azienda.

Tuttavia, i tipici RDBMS in alcune situazioni sono ancora vantaggiosi. Per alcune tipi di interrogazioni, come vedremo, Neo4j non presenta alcun vantaggio in termini ti prestazioni, anzi è addirittura più lento di un normale database relazionale. Nei capitoli successivi infatti analizzeremo Neo4j andando a definire gli aspetti positivi e negativi: in particolare in che modo può migliorare il sistema su cui viene implementato. In una futura release di CmWallet, si è pensato di implementare Machine Learning attraverso delle specifiche librerie di Neo4J. In particolare Neo4j permette l'interfacciamento con Apache Spark, in quale permette di eseguire elaborazioni parallele di dati, distribuite su più cluster.

Quando si vuole costruire un sistema basato su un graph DBMS, vi sono diverse decisioni architetturali che devono essere affrontate [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 (Embedded Mode) oppure eseguirlo nella maniera classica, ovvero in Server Mode.

### 1.3.1 Server Mode

Come detto precedentemente, è stata questa la modalità di interfacciamento tra i due sistemi. Neo4j Server sfrutta delle HTTP API (Transaction Cypher HTTP endpoint) [9] ed è la modalità più comunemente utilizzata attualmente.

Il cuore di un database Neo4j server è un istanza di tipo Embedded. Il vantaggio principale di questa modalità è che l'applicazione Neo4j restituisce un json di risposta al client: questo permette l'interfacciamento con qualsiasi tipo di applicazione client senza porsi il problema del linguaggio di programmazione o dell'infrastruttura che lo ospita.

In questa modalità di utilizzo inoltre, Neo4j viene "protetto" dall'influenza che potrebbe avere la Garbage Collection (GC) di qualsiasi altra applicazione attiva

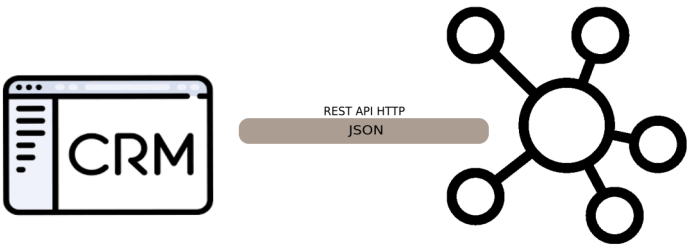



Figura 1.2: Neo4J utilizzato in modalità Server.

sulla macchina che lo ospita.

Le Transactional Cypher HTTP endpoint permettono l'esecuzione di una serie di Cypher statements all'interno dello scope di una singola transazione. Una singola transazione può essere mantenuta aperta anche per multiple richieste HTTP, fino a che il client decide di eseguire commit o roll back.

# Capitolo 2

# Neo4j

## 2.1 Il database a grafo

Le applicazioni innovative di ieri sono state guidate dai BigData, le applicazioni innovative di domani saranno guidate dalle connessioni tra dati.[12] Le connessioni tra i dati vengono conosciute come grafi. Lo scopo dei database a grafo è quello di mappare, analizzare, archiviare e attraversare reti di dati connessi, al fine di rilevare contesti e relazioni invisibili ai comuni strumenti di elaborazione dati.

Neo4j è in grado di fare tutto questo, cercando di fornire uno strumento intelligente per affrontare le sfide aziendali più complesse, tra cui:

- Intelligenza Artificiale (AI) e Machine Learning
- The Internet of Things (IoT)
- Raccomandazioni in tempo reale
- Master Data Management (MDM)
- Operazioni di rete e IT
- Identità e gestione degli accessi

I nodi sono gli elementi principali dei grafi e sono collegati tra di loro tramite le relazioni. I nodi possono godere di proprietà, ovvero degli attributi memorizzati come coppie chiave/valore all'interno di essi. Questi nodi possono includere anche

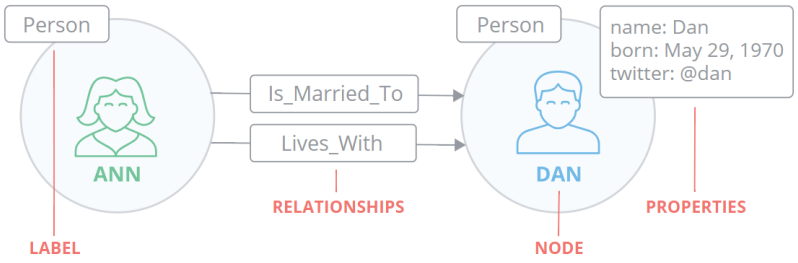



Figura 2.1: Le entità principali del grafo Neo4J.

una o più etichette, le quali mi identificano il ruolo del nodo nel grafo. Come i nodi, anche le relazioni possono godere di proprietà in forma chiave/valore.

## 2.2 Neo4j vs RDBMS

I database relazionali presentano una notevole rigidità dell'organizzazione dei dati. In alcuni casi, questo comporta quindi a delle performance scadenti occupando risorse del sistema ospitante. Come è noto infatti, i RDBMS organizzano i loro dati in tabelle predefinite che rendono i dati disgiunti.

Si pensi ad esempio ad una semplice operazione di JOIN su due tabelle: questa operazione comporta un appesantimento delle prestazioni, le quali decadono in modo esponenziale con il numero JOIN nella stessa query.

Supponiamo di voler eseguire una JOIN (Nested Loop JOIN) tra la tabella R e la tabella S. Ipotizziamo che R sia la outer table e che la S sia la inner table, lo pseudo codice corrispondente all'operazione di JOIN risulta essere:

```
for each tuple r in R do
  for each tuple s in S do
    if r and s join then output (r,s)
```

Figura 2.2: Pseudo codice Nested Join tra R e S

Chiamiamo come B(R) e B(S) il numero di pagine in memoria necessarie per contenere la tabella R e S rispettivamente. Chiamiamo inoltre come T(P) il numero di tuple contenute in una pagina P.

L'algoritmo inizia caricando nell'input buffer una pagina $P_0(R)$ contenente la prime tuple della outer table R e di una seconda pagina $P_0(S)$ contenente le prime tuple della inner table S. Nell'output buffer avremo, nel caso in cui la condizione di JOIN venga rispettata per ogni tupla della inner table, una pagina$P_0([R \bowtie S])$ contenente al massimo T(P) tuple:

$$\text{IN}[\ P_0(R)\ ] + \text{IN}[\ P_0(S)\ ] + \text{OUT}[\ P_0([R \bowtie S])\ ]$$

Mantenendo fissa la $P_0(R)$ si caricano le successive B(S) pagine per la tabella S:

$$\text{IN}[\ P_0(R)\ ] + \text{IN}[\ P_{B(S)}(S)\ ] + \text{OUT}[\ P_{B(S)}([R \bowtie S])\ ]$$

Si prosegue facendo:

$$\text{IN}[\ P_1(R)\ ] + \text{IN}[\ P_{B(S)}(S)\ ] + \text{OUT}[\ P_{B(S)}([R \bowtie S])\ ]$$

Infine otterremo:

$$\text{IN}[\ P_{B(R)}(R)\ ] + \text{IN}[\ P_{B(S)}(S)\ ] + \text{OUT}[\ P_{B(S)}([R \bowtie S])\ ]$$

Il costo di questa operazione di JOIN quindi è pari a:

$$B_{TOT} = B(R) + B(S)*B(R) + B(R \bowtie S)$$

dove $B(R \bowtie S)$ è il numero di pagine sull'output buffer a fine algoritmo e $B_{TOT}$ è il numero complessivo di pagine caricate in memoria per eseguire questa operazione di JOIN. A questo punto occorre poi considerare le operazioni di swap-in e swap-out, nel caso in cui non ci siano un numero di pagine sufficienti in memoria principale.

E' chiaro quindi che il costo computazionale di una JOIN non è affatto da sottovalutare.

I database a grafo riescono a ridurre notevolmente questi costi. Le relazioni non sono più determinate da indici univoci posti in tabelle differenti, ma sono un'entità concreta e differenziabile, esattamente come le tuple in un RDBMS. Questo implica che se si vuole ottenere delle relazioni dal database a grafo lo posso chiedere direttamente in modo esplicito, come se fosse una semplice "SELECT" di un RDBMS.

Vediamo in figura 2.2 un classico esempio di pattern che potremmo utilizzare per mappare delle relazioni tra le entità ingegneri e le entità progetti. Esiste una terza tabella Relazioni, nella quale vengono riportate tutte le relazioni tra Ingegnere e Progetto.

**Engineer**

| id_e | Nome | Cognome | Ruolo |
|---|---|---|---|
| 232 | N1 | C1 | R1 |
| 421 | N2 | C2 | R2 |
| 901 | N3 | C3 | R4 |
| 200 | N4 | C4 | R0 |
| 774 | N5 | C5 | R3 |
| 252 | N6 | C6 | R1 |
| 881 | N7 | C7 | R2 |
| 952 | N8 | C8 | R5 |
| 988 | N9 | C9 | R1 |
| 334 | N10 | C10 | R2 |

**Projects**

| id_p | Titolo | Ore | N.partecipanti |
|---|---|---|---|
| 1827 | T1 | 1500 | 3 |
| 4665 | T2 | 2500 | 7 |
| 7124 | T3 | 800 | 4 |
| 5572 | T4 | 3000 | 9 |
| 2765 | T5 | 1000 | 5 |
| 6111 | T6 | 1000 | 5 |

**Relations**

| id_r | progetto | ingegnere |
|---|---|---|
| 23 | 1827 | 421 |
| 24 | 1827 | 901 |
| 45 | 1827 | 200 |
| 77 | 2765 | 988 |
| 11 | 2765 | 200 |
| 87 | 2765 | 952 |
| 21 | 2765 | 334 |
| 87 | 2765 | 774 |
| 34 | 2765 | 232 |

Figura 2.3: Mappatura delle relazioni tra la tabella Ingegneri e la tabella Progetti.

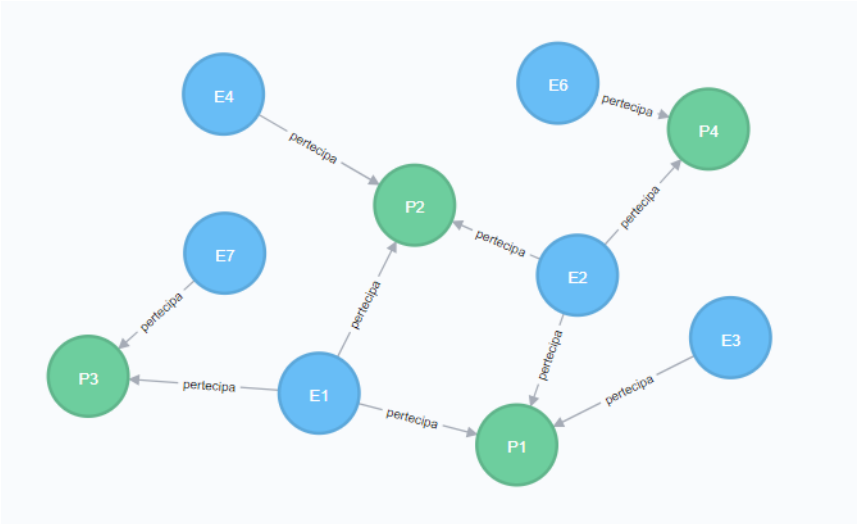



Figura 2.4: Grafo Neo4j che mappa le relazioni tra Ingegneri e Progetti, attraverso la relazione "partecipa".

Se volessimo sapere il Titolo dei progetti ai quali ha partecipato l'ingegnere [N1,C1] occorrerebbe eseguire operazioni di JOIN tra queste tre tabelle.

Vediamo ora la stessa situazione ma mappata su Neo4j. Le relazioni "partecipa" uniscono direttamente l'entità Ingegnere e l'entità Progetto: in questo modo posso interrogare il database ed estrapolare direttamente le relazioni. Per maneggiare i dati all'interno di Neo4j, si utilizza il linguaggio Cypher, che permette di interrogare il database usando una semantica semplice e intuitiva. Di seguito è riportata l'interrogazione Cypher che permette di restituire tutti i progetti ai quali ha partecipato l'ingegnere E1 e le rispettive relazioni "partecipa" che uniscono E1 ai progetti.

```
{ MATCH (n1:Engineer {name: "E1"})-[r:partecipa]->(n2)
  RETURN r, n1, n2 LIMIT 25 }
```

Di seguito, è riportato l'output di Neo4j a seguito delle precedente interrogazione.

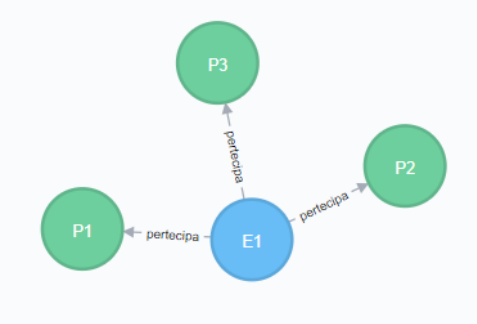



Figura 2.5: Output di Neo4j.

Questa estrapolazione diretta delle relazioni è stata possibile perché Neo4j mantiene la forma naturale dei dati, senza doverli mappare su tabelle.[13] Ogni nodo del database a grafo contiene un elenco di record, che mi rappresenta tutte relazioni con gli altri nodi e specifica il tipo, la direzione (se uscente o entrante nel nodo) ed eventuali attributi della relazione stessa. Ogni volta che si esegue l'equivalente di un'operazione JOIN , il database grafico utilizza questo elenco, accedendo direttamente ai nodi connessi ed eliminando la necessità di costosi calcoli di ricerca e corrispondenza.

Come è possibile immaginare dalle differenze strutturali discusse sopra, i modelli di dati relazionali rispetto alla struttura a grafo sono molto diversi. La struttura lineare del grafo consente di ottenere modelli di dati molto più semplici e più espressivi rispetto a quelli prodotti utilizzando database relazionali tradizionali o altri database NoSQL. L'individuazione di relazioni tra nodi non risulta più essere un'operazione costosa, ma ha un costo lineare nel numero di relazioni entranti/uscenti dal nodo.

## 2.3 L'architettura di Neo4j

Come già accennato precedentemente, Neo4j è un DBMS schema-less, questo vuol dire che i dati non vengono organizzati in base ad una struttura dati prefissata. A partire dalla versione 3.0, questo database gode delle seguenti proprietà:

- Linguaggio Cypher chiaro e dichiarativo (come SQL).
- Transazioni ACID.
- Compressione dinamica dei puntatori, che permette il salvataggio di più di 34 milioni di nodi.
- Utilizzo del protocollo Bolt per l'iterfacciamento con molti linguaggio di programmazione.
- Java Stored Procedures, per permettere l'accesso di basso livello al grafo.
- Neo4j Browser Sync, un'interfaccia molto utile ed intuitiva per eseguire delle query Cypher attraverso un qualsiasi browser.
- Embeddable, grazie a delle specifiche Java API.

A differenza di normani DBMS, in Neo4j le relazioni tra dati diventano quasi più importanti dei dati singoli: questo implica che le relazioni e le connessioni vengono mantenute attraverso ogni parte del ciclo di vita dei dati e che non ci siano operazioni out of band come MapReduce.

Alla base delle prestazioni di attraversamento, d'interrogazione e di scrittura esiste un elemento che rende l'architettura Neo4j unica: la **index-free adjancency**, ovvero una lista dove ogni elemento rappresenta un nodo del grafo con i puntatori ai nodi adiacenti. La index-free adjancency è tipica dei database a grafo **nativi**, ovvero di un sistema che non ha necessità di elaborazioni complesse per estrapolare risultati, ma gli basta semplicemente andare a percorrere liste di puntatori e leggere degli attributi per determinare il sottoinsieme di nodi richiesto. Quindi se volessimo sapere quali nodi sono adiacenti dato un nodo di partenza, occorre semplicemente percorrere la lista dei puntatori ai nodi adiacenti [18][7].

Esistono due elementi principali che distinguono la tecnologia del grafo nativo: archiviazione ed elaborazione.

Per l'archiviazione dei grafi si riferisce comunemente alla struttura sottostante del database che contiene i dati del grafo. Quando si parla di database a grafo nativi, si parla di database con memoria strutturata appositamente per un grafo, garantendo che i dati vengano salvati in modo efficiente, ovvero che nodi e relazioni siano fisicamente vicini. I database RDBMS ad esempio, sono strutturati in modo non nativo per un grafo; è possibile tuttavia memorizzare un grafo su questo tipo di database ma le prestazioni decadrebbero in modo esponenziale.

L'architettura di Neo4j è ottimizzata al massimo per operare su un grafo, a partire dalla query Cypher fino al modo con cui si organizzato i dati tra disco e memoria principale.

Per prima cosa, andiamo a contestualizzare quello di cui abbiamo discusso, guardando l'architettura in figura 2.5.

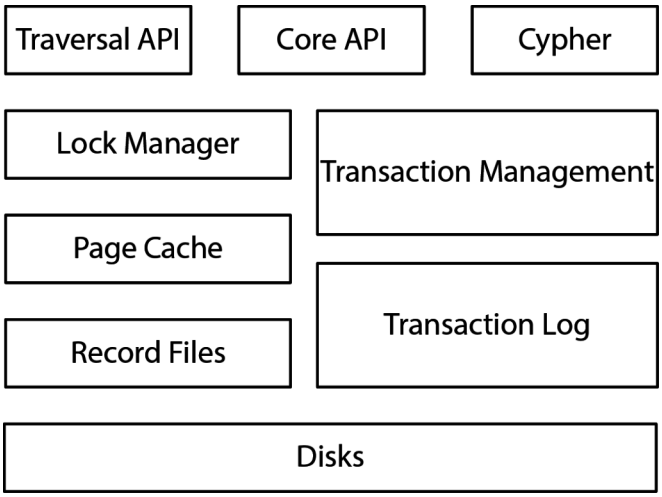



Figura 2.6: Architettura di Neo4j.

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 Store File (contenuti nel modulo *Disk*), i quali vengono memorizzati all'interno di un'unica cartella, detta Cartella di Database.

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.

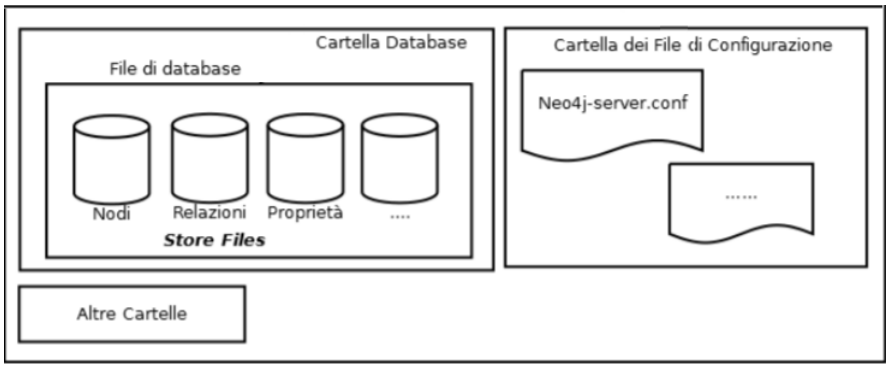



Figura 2.7: Struttura interna del disco.

Questa separazione netta tra i tipi di informazioni permette di percorrere il grafo in modo rapido e scalabile.

Lo Store File dei nodi (neostore.nodestore.db) memorizza tutti i nodi creati dall'utente come record. Come tutti gli Store file di Neo4j, anche quello dedicato ai nodi ha i record con **dimensione fissa di 9 Byte**, come viene mostrato in Figura 2.7 [8] [7]. I record hanno dimensione fissa consentono ricerche rapide dei nodi dello Store File. Se ad esempio si ha un nodo con id pari a 100, questo implica che il suo record inizia sicuramente al byte 900 all'interno dello Store File dei nodi. Basandosi su questo fatto, il database può determinare in quale punto troverà il nodo con

id 100 attraverso un'operazione con costo unitario O(1), quindi accesso diretto al record senza dover effettuare una scansione lineare con costo O(log N), dove N è il numero di record.

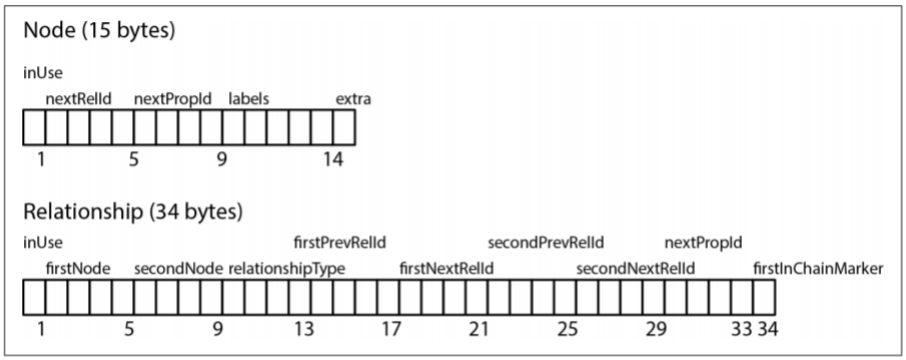



Figura 2.8: Struttura dei record.

La Figura 2.8 mostra la struttura di un record dello Store File dei nodi, 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 successivi quattro byte rappresentano l'ID della prima relazione connessa al nodo, i restanti byte rappresentano l'ID della prima proprietà del nodo. Nelle ultime versioni (Figura 2.7) sono stati aggiunti anche 5 Byte per le etichette più un Byte extra utilizzato indicare quali nodi sono fortemente connessi.

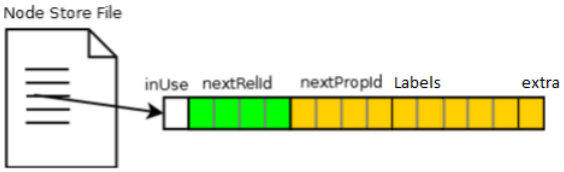



Figura 2.9: Struttura del record Nodo.

Lo Store File delle relazioni *neostore.relationshipstore.db*, come quello dei nodi, presenta dimensione fissa. In Figura 2.7 si può osservare che i primi 5 Byte indicano il nodo di "testa" della relazione (firstNode), mentre i successivi 5 Byte indicano il nodo di "coda" (secondNode). Dal Byte numero 9 in poi ci sono i Byte che indicano rispettivamente: tipo di relazione, i puntantori alla relazione successiva e a quella precedente sia per il nodo di "testa" sia per in nodo di "coda" delle relazione stessa. L'ultimo Byte è quello che mi indica se quella relazione è la prima nella cosiddetta *relationship chain*. Il tutto è mostrato in modo più chiaro in Figura 2.9

In Figura 3.0, viene rappresentanto graficamente come sono organizzati i puntatori e come il database li percorre per riuscire a recuperare l'informazione richiesta. Ad esempio, notiamo che per recuperare la proprietà "age" del Node 1, basta seguire la lista delle proprietà. Allo stesso modo, per cercare una relazione del Node 1, occorre percorrere la lista doppiamente concatenata delle relazioni partendo dalla relazione "LIKES". Una volta trovata la relazione, vorremmo ad esempio leggere una proprietà del nodo di "coda" della relazione stessa. Per fare ciò occorre leggere l'id del nodo e accedere direttamente al Byte (del *neostore.nodestore.db*) corrispondete a quell'id; il tutto a costo O(1).

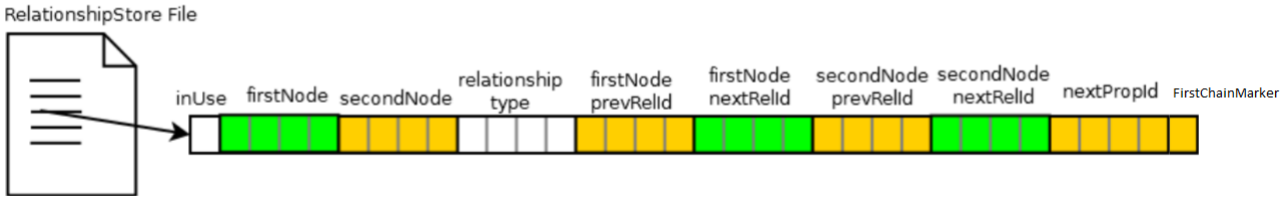



Figura 2.10: Struttura del record Relazione.

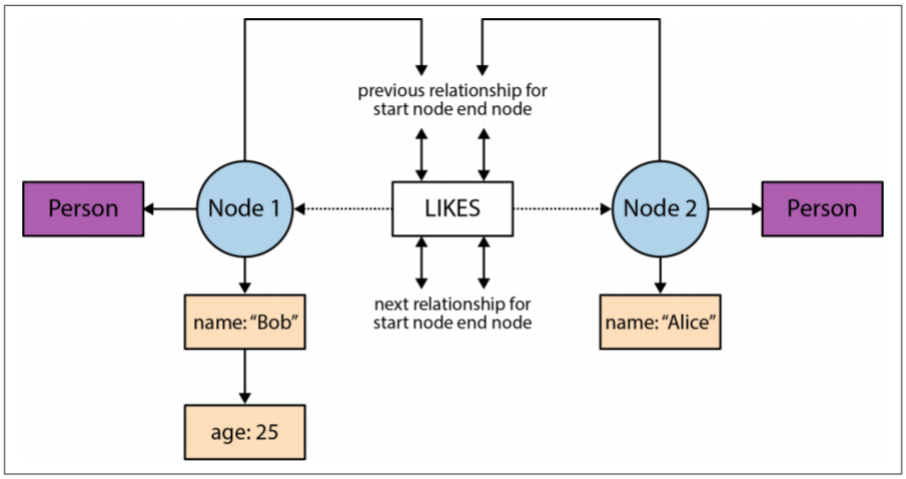



Figura 2.11: Come un grafo è fisicamente salvato in Neo4j.

E' chiaro ora come Neo4j riesca a percorrere un grafo in modo sorprendentemente veloce: il costo di attraversamento del grafo è dato solamente dal calcolo dell'offset, a partire dall'id del nodo, per ogni nodo attraversato. Questo è stato possibile proprio perché Neo4j gode di struttura a grafo nativa.

Otre gli Store File *neostore.nodestore.db* e *neostore.relationshipstore.db*, i quali mi creano di fatto la struttura nativa del database Neo4j, esiste anche lo Store File *neostore.propertystore.db* che mantiene i dati dell'utente, ovvero le proprietà dei nodi e delle relazioni. Come per gli Store File dei nodi e relazioni, anche lo Store File *neostore.propertystore.db* presenta record con dimensione fissa. Ogni record delle proprietà è costituito da quattro blocchi con l'aggiunta di un ultimo blocco che mi rappresenta l'id della proprietà successiva, all'interno della catena di proprietà. Notiamo che questa catena non è doppiamente collegata come la catena delle relazioni.

Uno dei blocchi più importanti del record è il blocco che mi contiene i quattro blocchi da 5 byte ciascuno: questi mi contengono il valore vero e proprio delle proprietà. Per ogni record proprietà si ha anche un puntatore al file di indice delle proprietà (*neostore.propertystore.db.index*), nel quale vengono memorizzati i nomi di tutte le proprietà. Il valore delle proprietà possono essere sia stringhe che array; per proprietà di grandi dimensioni si usano Store File dinamici a parte, ovvero il neostore.propertystore.db.strings e il *neostore.propertystore.db.arrays*. Un array o una stringa di grandi dimensioni possono occupare più di un record di questi Store File dinamici. Dai vari test condotti in azienda, si è visto che dati interi o stringhe inferiori ai 20 caratteri (come codici postali, numeri di telefono ecc...) vengono codificati in modo da essere contenuti direttamente all'interno dei quattro blocchi

del record nel file *neostore.propertystore.db*. Ciò si traduce in operazioni di I/O ridotte e throughput migliorato, poiché è richiesto solo un singolo accesso ai file.

Un'altra ottimizzazione di Neo4j è quella di utilizzare un solo record del file *neostore.propertystore.db.index* per le proprietà con lo stesso nome. Si pensi ad esempio ad un grafo che mi implementa un social media: tutti i nodi con label "persona" avranno la proprietà "nome" e la proprietà "cognome". Nel file *neostore.propertystore.db.index* ci sarà un solo record per la proprietà "nome" e un solo record per la proprietà "cognome". Questo si traduce in un notevole risparmio di spazio su disco e di operazioni di I/O [8][7].

Nonostante i file di archivio siano stati ottimizzati per gli attraversamenti rapidi, le considerazioni sull'hardware possono comunque avere un impatto significativo sulle prestazioni. La capacità di memoria è aumentata significativamente negli ultimi anni; tuttavia, i grafi molto grandi superano ancora le capacità di tenerli interamente nella memoria principale. Con l'utilizzo di dischi allo stato solido (SSD) le prestazioni sono notevolmente migliorate, in quanto non c'è una latenza dovuta alla ricerca del dato su disco. Tuttavia, i database a grafo nativi come Neo4j cercano di sfruttare al massimo la cache: in cache si memorizzano porzioni di Store File in base alle politiche dell'accesso probabilistico. In questo modo si cerca di limitare l'accesso al disco e alla memoria principale, quindi si riduce la latenza dovuta al tempo di transito dei dati sul bus. L'algoritmo adottato per lo swap-in e swap-out delle pagine è LRU-K, il quale mantiene le pagine più utilizzate in cache, mentre quelle meno utilizzate risiedono su disco. Nel momento in cui si genera un cache miss, la pagina meno recentemente utilizzata viene riportata su disco e viene portata in cache la pagina richiesta [8][7][18].

## 2.4 Caratteristiche non funzionali

A questo punto abbiamo capito cosa significa costruire un database a grafo nativo e abbiamo visto come alcune di queste funzionalità native del grafico sono implementate. Ma per essere considerato affidabile, qualsiasi tecnologia di archiviazione dei dati deve fornire un certo livello di garanzia sulla durabilità e accessibilità dei dati memorizzati.

Una misura comune con cui i database relazionali vengono tradizionalmente valutati è il numero di transazioni al secondo che possono elaborare. Nel mondo relazionale, si presume che queste transazioni mantengano le proprietà ACID (anche in presenza di guasti) in modo tale che i dati siano coerenti e recuperabili. Almeno ad alto livello, lo stesso vale per i database a grafo. Devono garantire la coerenza, recuperare gli arresti anomali e prevenire il danneggiamento dei dati. Inoltre, devono ridimensionarsi per fornire un'elevata disponibilità e aumentare le prestazioni. Nelle sezioni seguenti esamineremo cosa significa ognuno di questi requisiti per un'architettura di database grafico. Ancora una volta, ci espanderemo su alcuni punti scavando nell'architettura di Neo4j come mezzo per fornire esempi concreti. Va sottolineato che non tutti i database di grafici sono completamente ACID. È importante, quindi, comprendere le specifiche del modello di transazione del database scelto. Per le transazioni, la logica ACID di Neo4j mostra considerevoli livelli di affidabilità, livelli che siamo abituati ad ottenere dai sistemi di gestione di database relazionali di classe enterprise [8][7].

### 2.4.1 Transazioni

Le transazioni sono state un fondamento di sistemi informatici affidabili per decenni. Nonostante si pensi che le transazioni non permettano una buona scalabilità nei sistemi NOSQL, le transazioni rimangono un'astrazione fondamentale per l'affidabilità nei database a grafo contemporanei, incluso Neo4j. Le transazioni in Neo4j sono semanticamente identiche a transazioni di database tradizionali. Le scritture si verificano all'interno di un contesto di transazione e per ogni nodo e relazione coinvolta nella transazione si utilizza un'area di appoggio su disco, prima che i dati vengano scritti definitivamente su disco con l'operazione di commit. In caso di completamento positivo della transazione, le modifiche vengono scaricate su disco per garantire la **durabilità** e vengono rilasciati i lock di scrittura. Queste azioni mantengono **l'atomicità** della transazione.

Se la transazione fallisce per qualche motivo, le scritture vengono scartate e i lock di scrittura rilasciati, mantenendo così il grafico nel suo stato coerente precedente, garantendo la **consistenza**. Se due o più transazioni tentano di modificare contemporaneamente gli stessi elementi del grafico, Neo4j rileva una potenziale situazione di deadlock e serializza le transazioni. Le scritture all'interno di un singolo contesto transazionale non saranno visibili ad altre transazioni, mantenendo quindi **l'isolamento**.

In Neo4j, ogni transazione è rappresentata da un oggetto in memoria, in cui stato rappresenta le scritture sul database. Questi oggetti vengono gestiti da un lock manager, il cui applica dei lock di scrittura mano a mano che i nodi e le relazioni vengono creati, aggiornati ed eliminati. Neo4j utilizza un registro di log per tracciare ogni transazione eseguita sul database: ogni volta che una transazione va a buon fine, ovvero un commit su disco, viene aggiunta una entry nel registro di log.

### 2.4.2 Recoverability

Non dobbiamo dimenticare che i database non sono diversi da altri sistemi software e quindi potenzialmente soggetti a bug, oppure sensibili a malfunzionamenti dell'hardware ospitante. Anche se il sistema, nel suo complesso, è progettato in modo diligente, occorre quindi valutare la possibilità di arresto anomalo.

In un sistema ben progettato, un arresto del server ospitante il database, seppur fastidioso, non dovrebbe influire sulla disponibilità del servizio ma per lo più sul throughput. Nel momento in cui il server riprende a funzionare, deve garantire che i dati non siano corrotti, indipendentemente dalla natura del failure e dal tempo di down del server [8][7][1]. Quando Neo4j viene ripristinato dopo un arresto anomalo, esso esegue un controllo delle transazioni attive più recenti, per poi eseguirle. E' possibile tuttavia che alcune di queste transazioni siano già state eseguite precedentemente, ma dato che la riproduzione di una transazione è un'azione idempotente, la riproduzione della stessa azione non comprometterebbe la consistenza del database e quindi sarà coerente con lo stato appena prima del crash.

### 2.4.3 Availability

Le caratteristiche di transazione e di recoverability, inducono a Neo4j anche un'ottima availability. Questa capacità permette al grafo di ritornare disponibile dopo

un crash, anche senza l'intervento umano. Ovviamante, se si volesse aumentare ulteriormente l'availability occorre aggiungere delle istanze del database. Infatti, in uno scenario di produzione, è raro che si desideri creare singole istanze di database disconnesse. Neo4j utilizza una disposizione cluster **master-slave** al fine di garantire la ridondanza del database su più macchine. Le scritture del database vengono replicate dal master su ogni slave ad intervalli frequenti, garantendo che in ogni momento ci siano copie identiche del database su ogni macchina.[8][7][1] Il pattern più utilizzato per sfruttare al meglio la configurazione master-slave è quello in cui si utilizza il master per le scritture su database, mentre gli slave per le letture. Questa configurazione permette una scalabilità asintotica per le scritture e una scalabilità lineare per le letture. Infatti, aumentando il numero di richieste di lettura, posso incrementare orizzontalmente in numero di slave, ottenendo quindi un incremento lineare della scalabilità; all'aumentare delle scritture sul master invece, si deve tener conto dei limiti hardware della macchina, ottenendo così una scalabilità asintotica.

Sebbene il pattern write-master e read-slave sia quello più utilizzato, Neo4j permette anche di effettuare scritture sugli slave. La scrittura su ogni slave viene riportata sempre in modo sincrono anche sul master. Questa soluzione aumenta la flessibilità del sistema, a discapito della latenza, che aumenterebbe per una continua sincronizzazione con il master [8][15].

La disponibilità del dato dipende moltissimo dal tipo di query che andiamo a svolgere sul Neo4j. In Neo4j infatti è importate capire la differenza tra query **ideomatiche** e query **non idiomatiche**. Le query ideomatiche sono caratterizzate dal fatto che hanno uno o più punti di partenza per l'esplorazione del grafo. L'architettura di Neo4j, incluso l'utilizzo della cache, è stata pensata ed ottimizzata per supportare questo tipo richieste da parte del client. In particolare, il caching è allineato proprio con questo tipo di ricerche sul grafo in modo tale da eseguire le ricerche in modo rapido, liberando velocemente le risorse per la query successiva, aumentando così il throughput. Viceversa, Le query non ideomatiche non hanno un punto di partenza in particolare e non eseguono attraversamenti. Si pensi ad esempio di cercare un sottoinsieme di nodi e relazioni che soddisfano una specifica proprietà: in questo caso, i nodi e le relazioni che soddisfano questa richiesta potrebbero essere sparsi nel grafo senza nessuna logica. Questo implica mediamente un gran numero di operazioni di I/O da cache a disco.

### 2.4.4 Scalabilità

Con l'aumento dei volumi di dati, il database a grafo ha bisogno di scalabilità. Possiamo definire la scalabilità come un insieme di Capacità, Latenza e Throughput:

- **Capacità:** Come già detto nel Paragrafo 2.3, Neo4j usa dimensioni di puntatori fissi. Questo permette un maggiore velocità di esplorazione del grafo, a discapito però del numero di nodi e relazioni che può memorizzare. Gli sviluppatori di Neo4j hanno voluto mantenere un trade-off tra prestazioni e capacità tale per cui l'impatto in memoria e in numero di I/O al secondo siano contenuti. La versione attuale 3.4 di Neo4j, permette di memorizzare decine di miliardi di nodi, relazioni e proprietà. Ciò consente di avere grafici con un set di dati di social networking all'incirca delle dimensioni di Facebook.

- **Latenza:** In un database a grafo nativo come Neo4j, le prestazioni non dipendono dalla dimensione totale del grafo. Al contrario, nei database relazionali esiste una forte correlazione tra grandezza del grafo e tempo necessario per eseguire una o più join. In Neo4j, una query che mi comprende la visita di un grafo comporta semplicemente ad una individuazione nel nodo (o nodi) di partenza e, successivamente, da un continuo matching tra nodo corrente e puntatore al nodo successivo attraverso degli indici. Questo comporta ad avere una latenza pressoché costante, ovvero non dipende totalmente dal numero di nodi e relazioni che abbiamo nel grafo. Notare che questa considerazione sulla latenza è corretta solamente per query ideomatiche: se utilizzassimo Neo4j al solo scopo di restituire sottoinsiemi di nodi o relazioni all'interno del grafo, avremmo tempi di latenza anche peggiori di un RDBMS.

- **Read and write throughput:** In generale, per un database relazionale, le operazione di I/O richiedono la correlazione di più dati al fine di crearne l'input o l'output. Si pensi ad esempio quando vogliamo estrarre dei record composti da dati che provengono da tabelle diverse; attraverso delle JOIN possiamo "raccogliere" i dati ed aggregarli sotto forma di record. Questo non accade per i database a grafo come Neo4j. Nei database a grafo infatti, tutte le sotto operazioni di aggregazione dei dati (come le join) non sono necessarie e vengono aggregate in operazioni più grandi e coese, ottenendo uno sforzo computazionale ridotto a pari risultato. Ad esempio, si immagini uno scenario editoriale in cui vorremmo leggere l'ultimo articolo di un autore. In un RDBMS andremmo a selezionare tipicamente le opere dell'autore unendo la tabella degli "autori" ad una tabella "articoli" basate sull'identificativo dell'autore corrispondente, quindi ordinando i libri per data di pubblicazione. A seconda delle caratteristiche dell'operazione di ordinamento, potrebbe trattarsi di un'operazione O(log (n)). Proviamo ora a mappare il problema su un grafo.

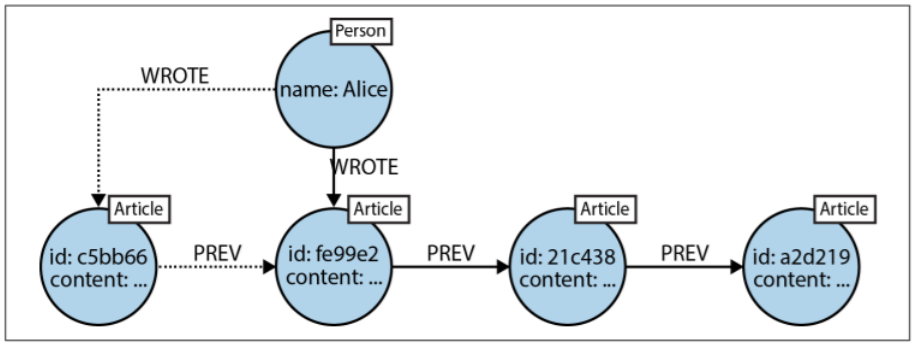



Figura 2.12: Mappatura su Neo4j.

Dalla Figura 2.11 è chiaro che l'operazione di estrazione dell'articolo più recente risulti avere complessità (O(1)), in quanto basta selezionare l'ultimo articolo identificato con la relazione "WROTE". Se si volesse ricercare gli articoli precedenti occorrerebbe seguire le relazioni "PREV". L'operazione di scrittura dell'ultima opera è anch'essa a costo O(1), in quanto si inserisce sempre in testa alla lista. Sia l'operazione di lettura che quella di scrittura sono a tempo costante e non dipendono dal numero di dati memorizzati nel grafo.

Se un solo cluster non è più sufficiente a soddisfare tutte le richieste, si potrebbe scalare orizzontalmente Neo4j su più cluster, creando una logica si sharding. La scalabilità del sistema dipende molto da come è costruito il grafo: un grafo sconnesso, ovvero composto da tanti grafi sconnessi tra loro, si presta molto bene per scalare orizzontalmente. Purtroppo non tutti i grafi sono suddivisibili in modo naturale, ma dobbiamo decidere noi dove andare a dividere il grafo. In questo caso, l'approccio che si usa è simile a quello che viene adottato su un normale database NOSql come MongoDB: si utilizzano delle chiavi che mettono in relazione i record a livello applicativo. Neo4J risulta essere nettamente più veloce per attraversamenti sulla stessa istanza, mentre per attraversamenti su istanze diverse non si ha nessun vantaggio significativo rispetto a MongoDB. Mediamente però, le prestazioni complessive di Neo4j circa gli attraversamenti risultano essere più veloci rispetto a MongoDB, in quando si tende ad organizzare il grafo tale per cui l'attraversamento avvenga il più possibile sulla stessa istanza.[8][7][1]

# Capitolo 3

# Casi di studio e implementazioni in CmWallet

In questo capitolo analizzeremo i campi applicativi in CmWallet. Uno dei nostri obiettivi è quello di capire come definire dei **pattern di trasformazione** da un RDBMS su un grafo. A tal proposito, in fase di studio, sono stati fatti molti mapping di database relazionali in grafi; tuttavia, in questo capitolo andremo a vedere i tre casi più significativi che tuttora sono utilizzati da diverse aziende o organizzazioni. Questi tre casi sono riportati nei paragrafi:

- **Ottimizzazione dei costi di produzione**
- **Diagramma reticolare**
- **Adozioni**

Sono stati scelti questi tre cosi di studio in quanto rappresentano rispettivamente tre metodi operativi differenti di mapping di un RDBMS in un grafo e, ad oggi, rappresentano circa il 90% dei casi reali.

Prima però, nel Paragrafo 3.1 del capitolo, faremo una breve descrizione circa le **regole generali** per modellare al meglio un qualsiasi struttura dati RDBMS in un grafo. Queste best practies sono state utilizzate per CmWallet.

## 3.1 Data Model Transformation

E' essenziale capire come modellare i dati. Una modellazione errata e confusionale dei dati sul grafo porta inesorabilmente a prestazioni deludenti, con tempi di risposta prolungati. A tal proposito, esiste un pattern per modellare in modo corretto i dati, provenienti ad esempio da un database relazionale, in modo che le interrogazioni future possano restituire risultati sensati e in un tempo limitato. Nell'esempio di seguito troviamo il tipico scenario relazionale in cui si hanno due tabelle messe in relazione da una terza tabella, detta JOIN Table. [5]. Come già descritto nei capitoli precedenti, l'operazione di join tra le due tabelle risulta essere onerosa; occorre quindi portare i dati su una struttura meno rigida, ovvero un grafo.

Vediamo quindi i punti principali per tradurre un database relazionale in un grafo:

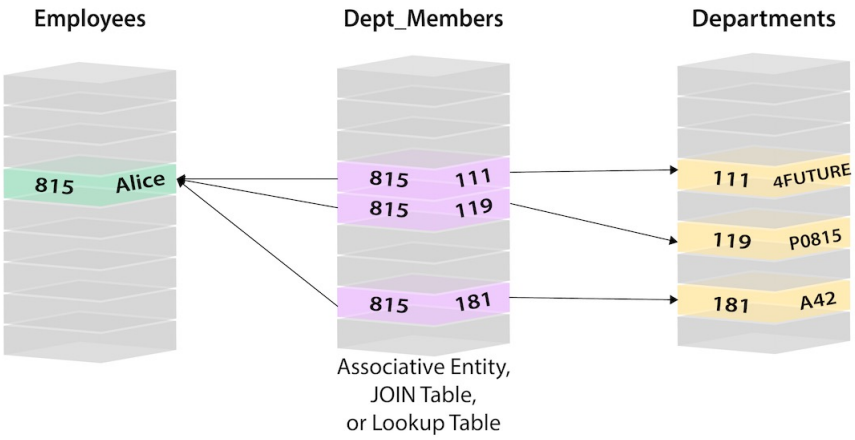



Figura 3.1: Struttura relazionale.

- **I nomi delle tabelle diventano label:** I nomi delle tabelle danno il nome alle label dei nodi. Non è necessario creare una label per ogni tabella, ma è importate individuare le tabelle che mi definiscono delle entità a se stanti, come le tabelle *Employees* e *Departments* della Figura 3.1
- **Da tupla a nodo:** Determinate le tabelle nel punto precedente, si trasforma ogni tupla della tabella in un nodo. In base alla tabella, si avrà un nodo con una label differente.
- **Da colonna a proprietà:** Ogni colonna della tabella diventa una proprietà del nodo.
- **Solo natural key:** Mantenere solamente le natural key ed eliminare le tecnical key.
- **Aggiungere constraints e indici:** occorre applicare dei constraints alle primary key e indicizzare gli attributi del nodo che verranno spesso ricercati.
- **Eliminare le foraign key:** Le foraign key verranno trasformate in relazioni.
- **Eliminare i valori di default:** Rimuovere tutti i valori di default in quanto sono assolutamente inutili.
- **Colonne simili:** Le colonne che mi identificano lo stesso tipo di informazione (ad esempio *email1,email2,email3*) possono essere raggruppate in un array che farà parte delle proprietà del nodo.
- **Join table in relazioni:** Le tabelle di Join diventano relazioni. Ogni tupla di una Join table diventa una singola relazione che unisce univocamente due nodi. Le altre colonne della Join table possono diventare proprietà delle relazione. All'interno di una relazione possiamo memorizzare svariati campi, che possono essere campi di stringe di testo, bool, interi, double, float. In numero di questi parametri non ha un limite implementativo, ma nel momento in cui abbiamo truppe proprietà all'interno delle relazioni, ci dobbiamo obbligatoriamente chiederci se il mapping che stiamo seguendo è corretto oppure possiamo considerare altre alternative.

Se si applicano tutte le precedenti best practies avremo il seguente risultato:

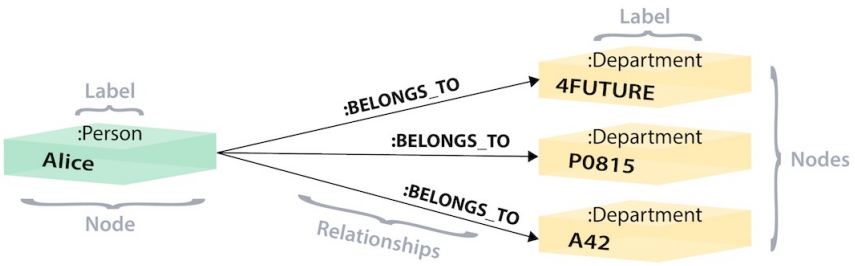



Figura 3.2: Risultato del data modelling della Figura 3.1.

Andando più nel concreto, proviamo ad analizzare il modello UML seguente e tradurlo nel corrispettivo modello a grafo, seguendo i passi precedentemente elencati [5]. Individuiamo prima di tutto quali sono le tabelle che daranno il nome alle label.

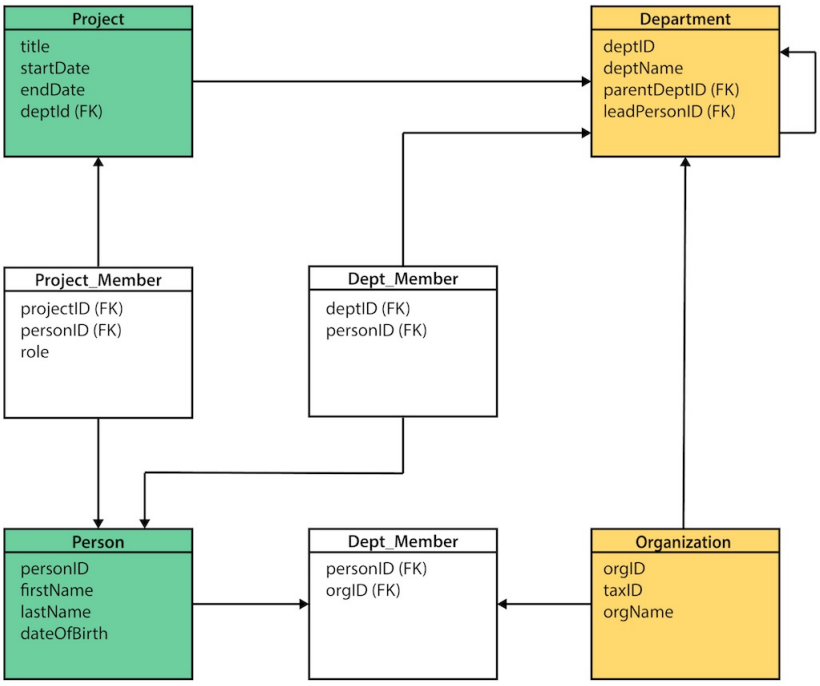



Figura 3.3: Schema relazionale.

Sono le tabelle che mi identificano le entità principali del diagramma, ovvero *Project*, *Department*, *Person* e *Organization*. Queste tabelle daranno il nome alle Label del grafo. Le righe delle tabelle saranno i Nodi del grafo, mentre le colonne diventeranno le proprietà del nodo. Eliminiamo ora tutte le chiavi tecniche: questo significa eliminare le chiavi (la colonna) appositamente create per rendere la riga della tabella univoca. Ad esempio *PersonID*, *orgID* e *deptID* vengono eliminate. Aggiungiamo a questo punto i constraints per le natural key, in modo da evitare che quella chiave possa essere duplicata nel grafo. Le JOIN table riportate in bianco in Figura 3.3 sono quelle che mi identificano le relazioni del grafo. Questo modo di modellare i dati sul grafo non va preso per il pattern definitivo. Spesso infatti si ha necessità

di mappare sul grafo dei dati che un RDBMS non sarebbe in grado di gestire. Nei prossimi paragrafi infatti vederemo delle specifiche implementazioni di Neo4j che non per forza seguono questo pattern di modellazioni dei dati. In particolare, se il nostro intento è quello di avere un grafo sul quale eseguire gli algoritmi di ricerca di Neo4J (*Shortest Path algorithm, Strongly Connected Components algorithm, ecc...*) è plausibile che si mappano i dati in un modo totalente diverso. Questo perchè spesso i dati non vengono caricati da un RDBMS ma il grafo viene popolato poco alla volta. Al contrario, esistono casi in cui non avrebbe nessun vantaggio modellare un RDBMS su un grafo: ad esempio se non abbiamo necessità di mantenere delle tabelle relazioni il grafo risultante avrebbe solo dei nodi senza nessuna relazione. Come abbiamo spiegato nel capitolo precedente, non si avrebbe nessun vantaggio con un grafo di questo tipo, ma addirittura un possibile peggioramento delle prestazioni.

A questo punto avremo ottenuto un grafo di questo tipo:

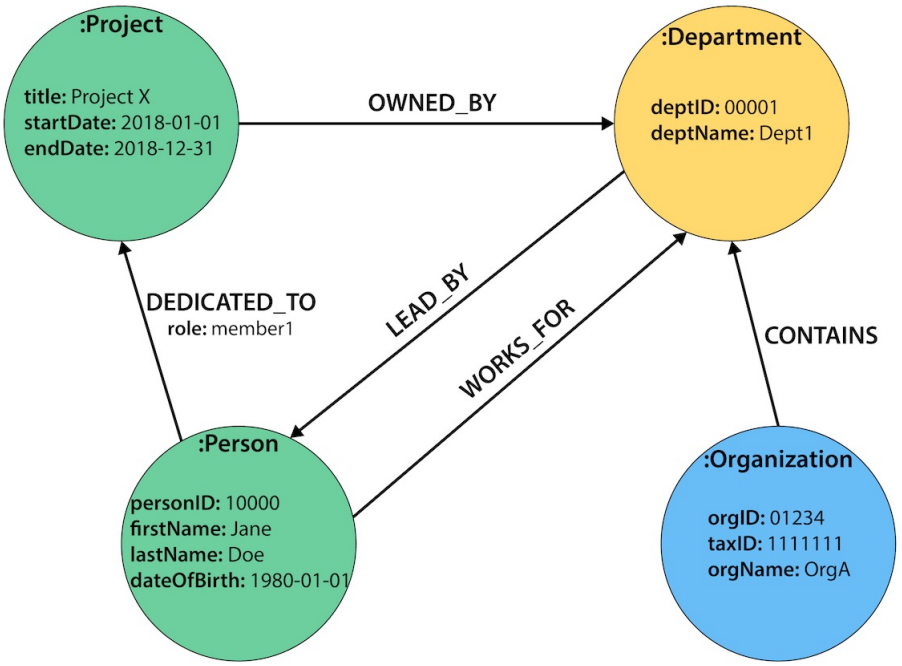



Figura 3.4: Schema relazionale.

## 3.2 Ottimizzazione dei costi di produzione

La prima nostra sperimentazione di Neo4j nasce da una richiesta, da parte di un cliente, di ottimizzazione dei costi. In particolare, questa azienda assembla automobili per disabili con installatori in tutta Italia. Le vetture vengono completamente trasformate ed adattate in base alle necessità del cliente e in base al tipo di disabilità. Questa trasformazione richiede un numero di pezzi meccanici notevoli, in parte prodotti internamente all'azienda e in parte acquistati da altri Paesi. In numero di pezzi possibili per assemblare una vettura vanno da un minimo di 50 ad un massimo di 300 pezzi, in base alla complessità della trasformazione.

Ogni pezzo meccanico può essere acquistato da produttori differenti con prezzi e caratteristiche qualitative diverse. Ovviamente queste parti meccaniche devono

essere scelte in base al budget del cliente e in base alle particolari esigenze che la sua disabilità impone.

L'azienda, senza l'uso di CmWallet, impiegava circa due settimane per la creazione di un preventivo e non sempre quest'ultimo era definitivo. Questi tempi erano dettati dal fatto che il loro database relazionale proponeva una serie di soluzioni ma non ottimizzate; era poi compito di alcuni commerciali e tecnici valutare la soluzione migliore per il cliente in base alle sue richieste e al budget. L'obbiettivo dunque, era quello di trovare un modo per ottimizzare la ricerca dei pezzi meccanici più congrui alle esigenze del cliente. L'azienda partiva da un database RDBMS MySQL il quale è stato poi modellato in parte su un grafo Neo4j.

### 3.2.1 L'obbiettivo

In nostro obbiettivo era quello di creare un grafo che permettesse, attraverso gli algoritmi messi a disposizione dalla libreria di Neo4j, di trovare la soluzione a costo minimo, in base a dei vincoli impostati dall'utente.

All'interno della libreria di Neo4j troviamo l'algoritmo **shortestPath** che permette la ricerca del cammino a costo minimo tra un nodo di partenza e un nodo di arrivo. L'algorimo shortestPath restituirà un sottoinsieme di nodi e relazioni, che corrisponderanno al commino a costo minimo, ovvero all'insieme di pezzi meccanici necessari per assemblare il prodotto finale per il cliente.

### 3.2.2 Modellazione

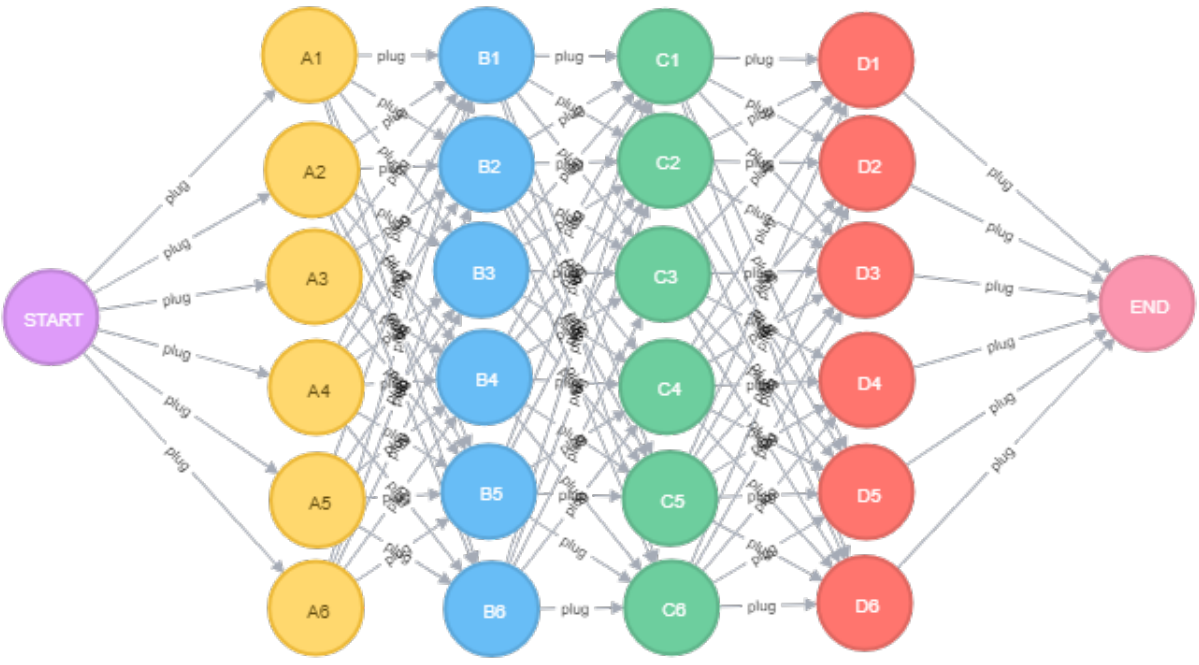



Figura 3.5: Modellazione dei prodotti su un DAG.

La scelta dell'algoritmo **shortestPath** ha portato a una modellazione piuttosto inusuale, ovvero l'utilizzo di un DAG. Un DAG (*Directed acyclic graph*) è, per definizione, un grafo aciclico. L'algoritmo di **shortestPath** necessita in input un

nodo di partenza e un nodo di arrivo; per questo motivo abbiamo creato due nodi fittizi "START" e "END" rispettivamente.

Nella Figura 3.5 vediamo una piccola simulazione di quello che è stato fatto realmente. In questa rappresentazione abbiamo quattro tipi di label (quattro colori diversi) che mi identificano quattro categorie di componenti meccanici. Ad esempio la label "LFDX" (colore giallo) mi identifica le *leve freno destra*, mentre la "LFSX" (colore blu) mi identifica le *leve freno sinistra*, e così per le altre due categorie di pezzi. In questo esempio vengono solo riportati, per ovvi motivi di spazio, solo 6 componenti per ogni categoria.

Notiamo che il grafo è fortemente connesso: le connessioni plug rappresentano tutte le possibili combinazioni all'interno del dominio di ricerca che dovrà esplorare l'algoritmo shortestPath. Inoltre queste relazione rappresentano la compatibilità tra un componente e il successivo. Ad esempio, il Figura 3.5 notiamo che la relazione tra A6 e B6 non esiste; questo significa che non esiste compatibilità tra i due componenti e quindi l'algoritmo di ricerca escluderà quella combinazione. Il codice Cypher per la creazione dei nodi è semplicemente il seguente:

```
MERGE (a:LFDX {name:'A1'})
MERGE (b:LFDX {name:'A2'})
MERGE (c:LFDX {name:'A3'})
MERGE (d:LFDX {name:'A4'})
MERGE (e:LFDX {name:'A5'})
MERGE (f:LFDX {name:'A6'})
MERGE (a1:LFSX {name:'B1'})
MERGE (b1:LFSX {name:'B2'})
MERGE (c1:LFSX {name:'B3'})
MERGE (d1:LFSX {name:'B4'})
MERGE (e1:LFSX {name:'B5'})
MERGE (f1:LFSX {name:'B6'})
MERGE (a2:PDV {name:'C1'})
MERGE (b2:PDV {name:'C2'})
MERGE (c2:PDV {name:'C3'})
MERGE (d2:PDV {name:'C4'})
MERGE (e2:PDV {name:'C5'})
MERGE (f2:PDV {name:'C6'})
MERGE (a3:ACCV {name:'D1'})
MERGE (b3:ACCV {name:'D2'})
MERGE (c3:ACCV {name:'D3'})
MERGE (d3:ACCV {name:'D4'})
MERGE (e3:ACCV {name:'D5'})
MERGE (f3:ACCV {name:'D6'})
//start
MERGE (bg:begin {name:'START'})
//end
MERGE (fn:fin {name:'END'})
//CARICAMENTO RELAZIONI
MERGE (bg)-[:plug {cost:50}]->(a)
MERGE (bg)-[:plug {cost:80}]->(b)
MERGE (bg)-[:plug {cost:79}]->(c)
```

```
MERGE (bg)-[:plug {cost:83}]->(d)
MERGE (bg)-[:plug {cost:80}]->(e)
MERGE (bg)-[:plug {cost:81}]->(f)
//PRIMO LIVELLO
MERGE (a)-[:plug {cost:50}]->(a1)
MERGE (a)-[:plug {cost:50}]->(b1)
MERGE (a)-[:plug {cost:40}]->(c1)
MERGE (a)-[:plug {cost:45}]->(e1)
MERGE (a)-[:plug {cost:55}]->(f1)
...
//TERZO LIVELLO
MERGE (f2)-[:plug {cost:34}]->(a3)
MERGE (f2)-[:plug {cost:35}]->(b3)
MERGE (f2)-[:plug {cost:32}]->(c3)
MERGE (f2)-[:plug {cost:40}]->(d3)
MERGE (f2)-[:plug {cost:37}]->(e3)
MERGE (f2)-[:plug {cost:32}]->(f3)
//CHIUSURA su END
MERGE (a3)-[:plug {cost:3}]->(fn)
MERGE (b3)-[:plug {cost:3}]->(fn)
MERGE (c3)-[:plug {cost:3}]->(fn)
MERGE (d3)-[:plug {cost:3}]->(fn)
MERGE (e3)-[:plug {cost:3}]->(fn)
MERGE (f3)-[:plug {cost:3}]->(fn);
```

Per questioni di ripetitività sono state omesse alcune righe di codice. Le relazioni **plug** sono caratterizzate dalla proprietà *cost* che mi identifica il costo di acquisto del componente meccanico. Notiamo che questa modellazione è piuttosto inusuale: il costo del nodo non è stato posizionato all'interno del nodo stesso, ma è un attributo della relazione che mi porta ad esso. Quindi ogni nodo può possedere fino a 6 archi entranti con un attributo *cost* sempre uguale. Ad esempio, il nodo D3 ha 6 archi entranti con un cost = 32 che partiranno rispettivamente da C1,C2,C3,C4,C5 e C6.

Questa scelta è stata fatta perché l'algorimo Shortest Path (Cammini Minimi), per definizione, restituisce il percorso che collega due vertici e che minimizza la somma dei costi associati all'attraversamento di ciascun arco. Per questo motivo dunque è stato necessario dissociare la proprietà *cost* dal nodo ed associarla all'arco entrante nel nodo stesso [4].

Notiamo che alcune relazioni non sono presenti. Questo vuol dire che tra quei due componenti meccanici non c'è compatibilità. Questa informazione era mappata inizialmente come una tabella "compatibilità" del database relazionale. Di conseguenza abbiamo ritenuto ragionevole eliminare definitivamente la tabella "compatibilità" dalla struttura database globale.

L'importazione dei dati è stata fatta manualmente attraverso una query Cypher in quanto è stato ritenuto meno dispendioso rispetto all'importazione di file CSV costruiti ad hoc.

### 3.2.3 Shortes Path algorithm

Come già anticipato in precedenza, l'algoritmo Shortes Path permette la ricerca del percorso a costo minimo tra due nodi. Scoperto dallo studioso Dijkstra, questo algoritmo era stato pensato inizialmente per trovare la strada alternativa più breve nel caso la via principale fosse bloccata. Tutt'ora, viene utilizzato per i Social Network come Linkedin per trovare i gradi di separazione tra due persone, oppure da tutti i software che implementano funzioni di navigazione sulla mappa stradale.

Abbiamo dimostrato che questo algoritmo, grazie all'implementazione di Neo4j, può essere adottato per **casi d'uso alternativi** come quello riportato in questo paragrafo. La Cypher che ha permesso la restituzione del cammino a costo minimo è la seguente:

```
MATCH (start{name:'START'}), (end{name:'END'})
CALL algo.shortestPath.stream(start, end, 'cost')
YIELD nodeId, cost
MATCH (other) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```

Senza andare troppo nel dettaglio della sintassi Cypher, in quanto non è l'oggetto di studio principale di tale elaborato, descriviamo cosa abbiamo fatto con questa query. Partiamo subito dalla chiamata della proceduta `algo.shortestPath.stream(start, end, 'cost')`.

Questa procedura permette la restituzione del percorso a costo minimo tra i nodi `start` e `end`, in base al "peso" dell'arco chiamato `cost`. I nodi `start` e `end` vengono prima ricercati nel grafo attraverso l'istruzione `MATCH`. Infine, ritorniamo il nome di ogni nodo attraversato e il costo per raggiungere tale nodo. L'output della console di Neo4J è il seguente:

| name | cost |
|---|---|
| "START" | 0.0 |
| "A1" | 50.0 |
| "B3" | 90.0 |
| "C2" | 280.0 |
| "D6" | 312.0 |
| "END" | 315.0 |

Started streaming 6 records after 3 ms and completed after 26 ms.

Figura 3.6: Output console Neo4j.

Nella parte di sinistra della tabella vediamo i nodo attraversati, tra il nodo di START e il nodo di terminazione END, mentre a destra i rispettivi costi di attraversamento. Il costo complessivo del percorso a costo minimo è di 315. Un altro dato interessante è il tempo di esecuzione, che risulta essere di 26 ms complessivi.

Purtroppo gli algoritmi messi a disposizione da Neo4j presentano alcuni limiti, come ad esempio non poter inserire dei criteri di ricerca del percorso minimo e poter dunque discriminare alcuni nodi o relazioni da altri. Ad esempio, nel nostro caso, avremmo necessità considerare alcuni nodi obbligatori all'interno del cammino a costo minimo. Il codice Cypher Neo4j di ricerca del cammino minimo è il seguente:

```
MATCH p=(a{name:'START'})-[:plug*1..7]->
(b1)-[:plug*1..7]->(b2)-[:plug*1..7]->(n{name:'END'})
WHERE ALL (n IN nodes(p) WHERE n.name <> 'A1'
and n.name <> 'A3'
and b1.name = 'B3'
and b2.name = 'C1')
RETURN p AS shortestPath,
reduce(accum_cost=0, r IN relationships(p)
| accum_cost+r.cost) AS totalCost
ORDER BY totalCost ASC
LIMIT 1
```

Con questa Cypher si specifica che si deve escludere il componente meccanico A3, ma sono obbligatori i componenti B3 e C1. Con questa strategia è stato implementato un modo per specificare le ricerca in base all'automobile sulla quale si andranno a installare i componenti, il prezzo e alcuni altri campi di ogni nodo.

### 3.2.4 Architettura della soluzione

Inizialmente, si era pensato ad una soluzione nella quale il database relazionale venisse completamente abbandonato per passare definitivamente ad un db a grafo come Neo4j. Come abbiamo esaminato nel Capitolo 2, i database a grafo come Neo4j sono molto performanti nel momento in cui si applicano delle ricerche sui percorsi, ovvero **query ideomatiche**.

L'azienda presa in considerazione utilizza il software CmWallet per il 70% per query non ideomatiche, ovvero la semplice ricerca di report e documentazione. Questa analisi ci ha permesso di valutare la possibilità di utilizzare il database relazionale e Neo4j contemporaneamente. In particolare, le ricerche per i cosi di produzione, ovvero il cammino minino, viene effettuato grazie a Neo4j, mentre normali interrogazioni non ideomatiche vengono eseguite sul database relazionale.

La comunicazione tra CmWallet e Neo4j avviene grazie alle HTTP API, ovvero delle API Rest messe a disposizione da Neo4j. La particolarità di queste API è la completa flessibilità, in quanto sono indipendenti dal linguaggio di programmazione utilizzato.

Si utilizzano stringhe in formato JSON con dei campi specifici, come mostrato di seguito.

```
url = 'http://neo4j:xxx@proxmox:7474/db/
data/transaction/commit';
J = '{
"statements" : [ {
"statement" : "MATCH (start {name: {start} }),
(end{name: {end}})
CALL algo.shortestPath.stream(start, end, {cost})
```

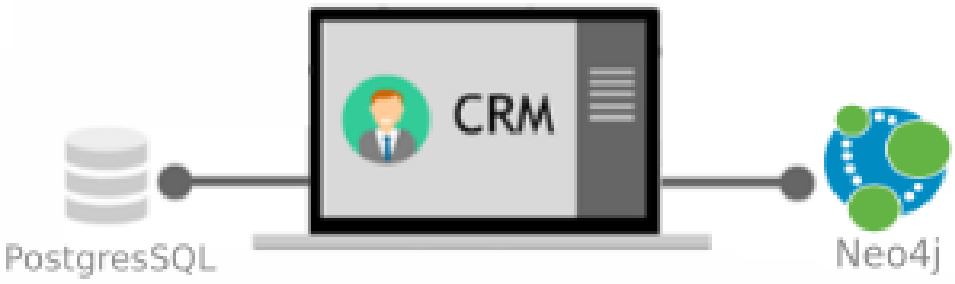



Figura 3.7: Architettura definitiva con l'aggiunta di Neo4j.

```
YIELD nodeId, cost MATCH (other)
WHERE id(other) = nodeId RETURN other.name AS name, cost",
"parameters": {
"start" : "START",
"end" : "END",
"cost" : "cost"
}}]}';
```

E' possibile tuttavia, eseguire più "statement" con una sola richiesta in modo da non dover effettuare chiamate in sequenza e allungare il tempo di latenza che si interpone tra la richiesta di un risultato e l'output del risultato stesso.

Una possibile soluzione master-slave è stata per adesso esclusa, in quando il numero di letture sul database non è tale da dilatare il tempo di latenza.

## 3.3 Diagramma reticolare - CPA

Sempre di più si sente la necessità di pianificare ed organizzare i progetti con metodi scientifici e pattern che ormai sono famosi nell'ambito del project management. Uno di questi è sicuramente il Diagramma Reticolare, detto anche Critical Path Analysis (CPA). Il CPA è sostanzialmente una rappresentazione visiva dell'ordine delle attività in base alle loro dipendenze [10]. Le attività vengono divise ed organizzate a livello temporale, formando cosi un grafo orientato aciclico. Ogni attività ha una sua durata in base alle sue difficoltà intrinseche. Il percorso critico è il percorso che si interpone tra l'inizio e la fine del progetto, avendo durata massima rispetto a tutti gli altri percorsi possibili. Notare che è possibile avere anche più percorsi critici in un diagramma reticolare. Come dice il Dott. Larry Bennett, project manager e autore di una serie di libri: *"Questo processo genera un programma per guidare il*

*team di progetto e costituisce la base per tenere traccia della performance mettendo a confronto le attività programmate con quelle attualmente in corso".*

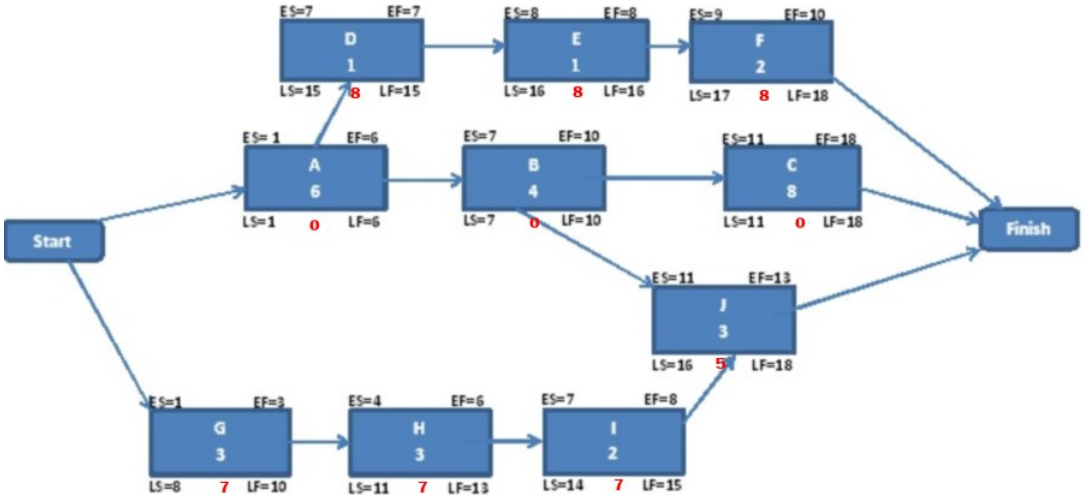



Figura 3.8: Esempio di diagramma reticolare.

In Figura 3.8 viene mostrato un diagramma reticolare contenente tutte le attività del progetto. All'interno delle attività vengono riportate le corrispondenti durate, espresse in termini di giorni. Su questo diagramma si possono fare molte considerazioni per il project management ed arrivare ad importanti stime di tempi e costi. Esistono infatti molte formule per calcolare svariati indici; noi ci concentreremo su pochi concetti al fine di mostrare le potenzialità di Neo4j e di come CmWallet può integrare un modulo per il project management.

### 3.3.1 L'obbiettivo

Esistono svariati software sul mercato, sia a pagamento che gratuiti, che implementano gran parte delle funzionalità indispensabili per la gestione a 360 di un progetto. Si pensi ad esempio Microsoft Project o OpenProject, i due software di riferimento per il project management. Tuttavia però, questi software sono ben lontani dalle funzionalità di un CRM, come ad esempio gestione ordini, prodotti, clienti, gestione documentale, gestione livelli di accesso e molto altro. L'obbiettivo è quindi quello di implementare all'interno di un CRM completamente gratuito un modulo per il project management, in modo che l'azienda abbia su uno stesso strumento sia le tipiche funzionalità che offre un CRM, ma anche la possibilità di gestire progetti e attività in modo smart.

### 3.3.2 Modellazione

Il diagramma reticolare, essendo nativamente un grafo, è piuttosto semplice la mappatura su Neo4j. Di seguito è riportato il grafo che rappresenta un diagramma reticolare.

Come si vede in Figura 3.9, il diagramma inizia con il nodo START e termina END e tutte le attività intermedie sono organizzate in base alle dipendenze tra loro. Ad esempio l'attività 2 non può essere svolta prima del completamento dell'attività 1, l'attività 7 non può essere svolta prima del completamento delle attività 3 e 4,

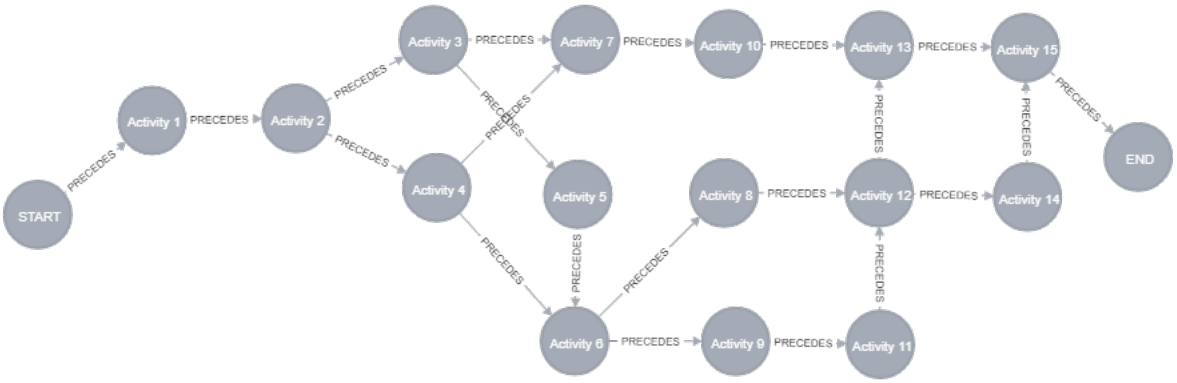



Figura 3.9: Diagramma reticolare su Neo4j.

e così via. Le relazioni che determinano l'ordine delle attività sono state chiamate *PRECEDES*. Di seguito è tiportato il codice Cypher per la costruzione del grafo.

```
CREATE (START:Activity {id:1, name:'START', length:0,
es:0, ef:0, ls:0, lf:0}),
 (A:Activity {id:2, name:'Activity 1', length:11}),
 (B:Activity {id:3, name:'Activity 2', length:5}),
 (C:Activity {id:4, name:'Activity 3', length:5}),
 (D:Activity {id:5, name:'Activity 4', length:13}),
 (E:Activity {id:6, name:'Activity 5', length:5}),
 (F:Activity {id:7, name:'Activity 6', length:4}),
 (G:Activity {id:8, name:'Activity 7', length:5}),
 (H:Activity {id:9, name:'Activity 8', length:13}),
 (I:Activity {id:10, name:'Activity 9', length:3}),
 (J:Activity {id:11, name:'Activity 10', length:15}),
 (K:Activity {id:12, name:'Activity 11', length:5}),
 (L:Activity {id:13, name:'Activity 12', length:4}),
 (M:Activity {id:14, name:'Activity 13', length:9}),
 (N:Activity {id:15, name:'Activity 14', length:11}),
 (O:Activity {id:16, name:'Activity 15', length:5}),
 (END:Activity {id: 17, name:'END', length:0})

CREATE (START)-[:PRECEDES]->(A),
 (A)-[:PRECEDES]->(B),
 (B)-[:PRECEDES]->(C),
 (B)-[:PRECEDES]->(D),
 (C)-[:PRECEDES]->(E),
 (C)-[:PRECEDES]->(G),
 (D)-[:PRECEDES]->(F),
 (D)-[:PRECEDES]->(G),
 (E)-[:PRECEDES]->(F),
 (F)-[:PRECEDES]->(H),
 (F)-[:PRECEDES]->(I),
 (G)-[:PRECEDES]->(J),
 (H)-[:PRECEDES]->(L),
 (I)-[:PRECEDES]->(K),
```

```
(J)-[:PRECEDES]->(M),
(K)-[:PRECEDES]->(L),
(L)-[:PRECEDES]->(M),
(L)-[:PRECEDES]->(N),
(M)-[:PRECEDES]->(O),
(N)-[:PRECEDES]->(O),
(O)-[:PRECEDES]->(END)
```

Per ogni nodo che rappresenta l'attività è stata assegnata la label *Activity* ed assegnate le proprietà id, *lenght* e *name*. Più in dettaglio si può osservare che al nodo di start sono state assegnate anche le proprietà *es,ef,ls,lf* che corrispondono rispettivamente a Early Start *(inizio al più presto)*, Early Finish *(fine al più presto)*, Late Start *(inizio al più tardi)* e Late Finish *(fine al più tardi)*. Questi valori sono indispensabili per ottenere il **percorso critico**. Il percorso critico, per definizione, è la sequenza delle attività concatenate che si sommano dando luogo alla più lunga durata complessiva del progetto [11]. Il calcolo di questo percorso critico rende possibile determinare quali sono le attività "critiche" (cioè, sul percorso più lungo) e quelle che permettono un possibile slittamento, che possono essere cioè dilatate senza influenzare i tempi di progetto, ovvero il calcolo del **total float**.

### 3.3.3 Forward Pass

Dopo aver definito la durata di ogni singola attività sul diagramma reticolare, si procede con la fase di Forward Pass, nella quale si definiscono i Early Start e Early Finish.

Per definizione, Early Start e Early Finish si calcolano come:

$$EF_j = ES_j + d_j$$

$$ES_j = MAX(EF_i)$$

dove d è la durata di quella specifica attività j. Per $MAX(EF_i)$ si intende il valore massimo di EF che si può trovare nelle attività subito precedenti alla j-esima attività. A titolo di esempio, possiamo osservare l'immagine di seguito, ed analizzare l'attività

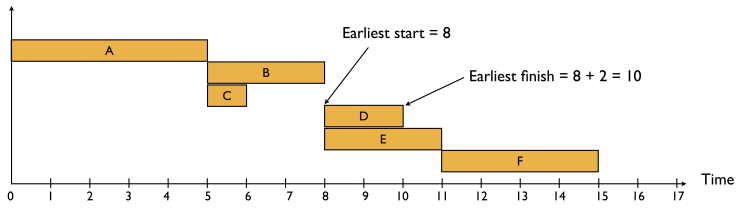



Figura 3.10: Time Line di una sequenza di task.

D per la quale sono stati definiti ES e EF.

Determiniamo ora, in base al grafo in Figura 3.9, una query Cypher in grado di calcolare automaticamente questi valori per ogni nodo del grafo. Per il calcolo degli EF useremo la seguente Cypher:

```
MATCH path = (:Activity {name:'START'})-[:PRECEDES*]->
(j:Activity)
WITH j,
MAX(REDUCE(ac = 0, a IN NODES(path) | ac + a.length))
AS ef
SET j.ef = ef
```

Ritorniamo nella variabile *path* tutti i percorsi possibili tra il nodo START e il generico nodo j. Introduciamo poi la funzione "reduce" di Neo4j, la quale esegue un'espressione per ogni elemento di una lista e ne salva il risultato in una variabile *ac* di accumulazione. La clausola "SET" permette di settare la proprietà *ef* per il nodo j. Dato che di tutti i percorsi dal nodo START al nodo j devo considerare solo quello di costo massimo (il percorso critico tra START e j) uso la clausola MAX, in modo tale da forzare la selezione al solo percorso massimo tra START e j. Proseguiamo con il calcolo degli ES:

```
MATCH (i:Activity)-[:PRECEDES]->(j:Activity)
WITH j, MAX(i.ef) AS es
SET j.es = es
```

In questo caso non è necessario usare funzioni di calcolo sul percorso quindi non usiamo una variabile *path* per salvare il percorso corrente. Ci limitiamo quindi a trovare il valore massimo di EF su tutti gli immediati predecessori di un nodo j.

### 3.3.4 Backward Pass

Successivamente si passa alla fase di Backward Pass, nella quale di definiscono i Last Start e Last Finish.

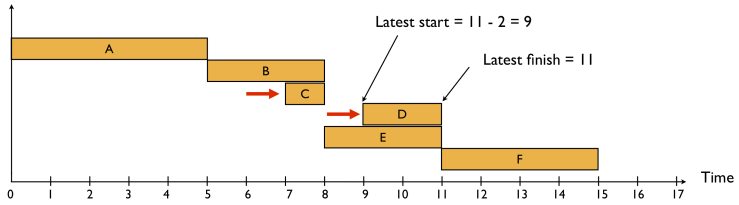



Figura 3.11: Time Line di una sequenza di task.

I tempi di LS e LF definiscono quindi gli ultimi istanti in cui un task può iniziare e finire rispettivamente, senza incidere sulla durata complessiva del progetto. Per definizione, Last Start e Last Finish si calcolano come:

$$LS_j = LF_j - d_j$$

$$LF_i = MIN(LS_j)$$

Per ricavare i valori LS e LF per ogni nodo del grafo, occorre partire dal nodo END ed eseguire le operazioni speculari rispetto alla fase di Forward Pass. Quindi, per determinare LF per un nodo $i$ occorre determinare il minimo tra gli LS dei nodi immediatamente successivi al nodo $i$. La Cypher per il calcolo degli LS è dunque:

```
MATCH p = (j:Activity)-[:PRECEDES*]->
(f:Activity {name:'END'})
WITH j,
MIN(REDUCE(ac = f.es, a IN NODES(p) | ac - a.length))
AS ls
SET j.ls = ls
```

Si noti che la variabile *ac* non parte da 0 come nel caso del Forward Pass, ma viene inizializzata al valore di ES del nodo j. Vediamo la Cypher per il calcolo del valori LF:

```
MATCH (i:Activity)-[:PRECEDES]->(j:Activity)
WITH i, MIN(j.ls) AS lf
SET i.lf = lf
```

### 3.3.5 Calcolo del Percorso Critico

Come già anticipato, il percorso critico è indispensabile per sapere il tempo totale di progetto e quali attività compongono tale percorso, ovvero le attività critiche. Utilizziamo la seguente Cypher per il calcolo del percorso critico sul nostro grafo di Figura 3.9:

```
MATCH p = (:Activity {name:'START'})-[:PRECEDES*]->
(:Activity {name:'END'})
WITH p, REDUCE(x = 0, a IN NODES(p) | x + a.length)
AS accum
ORDER BY accum DESC
LIMIT 1
RETURN p
```

Con questa Cypher stiamo trovando tutti i percorsi tra il nodo START e il nodo END e ne stiamo ritornando quello a costo maggiore in base alla variabile di accumulazione *accum*. Il percorso critico restituito risulta essere la sequenza START,1,2,4,6,8,12,14,15 e END di costo totale 66.

### 3.3.6 Calcolo del Float o Slack

Per definizione [6], il Float è la quantità di tempo che un attività può ritardare senza dilatare l'intero progetto. Le attività sul percorso critico sono per definizione con Float uguale a 0, mentre le attività non sul percorso critico possono essere ritardate di un Float maggiore di 0. Il Float per ogni nodo j, si calcola come:

$$\text{Float}_j = \text{LS}_j \text{ - } \text{ES}_j$$

La query Cypher corrispondente risulta essere:

```
MATCH (node:Activity)
SET node.float = a.ls - a.es
```

Con questa serie di Cypher abbiamo ottenuto l'analisi di un Diagramma Reticolare, il tutto riassumibile nella seguente rappresentazione grafica:

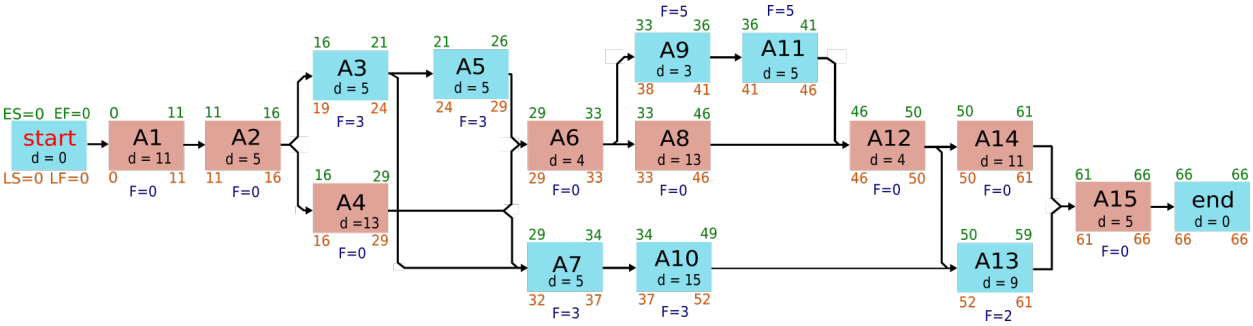



Figura 3.12: Diagramma reticolare riferito al grafo di Figura 3.9 con in evidenza il percorso critico in rosso.

### 3.3.7 Aggiunta di nuove entità

Fino a ora abbiamo implementato le funzionalità indispensabili per il project management. Ora proviamo a spingerci oltre e sfruttare al meglio le potenzialità di Neo4j. In un grafo come in Figura 3.9 abbiamo solo un tipo di nodi "Activity" e un solo tipo di relazioni "PRECEDES". Come abbiamo descritto nel capitolo precedente, in un grafo possiamo avere molte Label e molti tipi di relazioni; questo permette una totale flessibilità e andare a memorizzare un grafo anche molto complesso, con svariati tipi di nodi (Labels) e relazioni.

Detto questo proviamo ora a costruire un grafo che, oltre a rappresentare un diagramma reticolare, possa restituire altre informazioni. Potrebbe essere utile associare ogni singola attività a delle risorse umane, quindi si creano dei nodi "Employee" e li si associano ai nodi Activity in base alla collocazione delle risorse. Alle relazioni che uniscono i nodi Activity ai nodi Emaployee, gli si attribuisce il nome di "TAKEPART".

Aumentiamo ancora la complessità del grafo inserendo i nodi "Skill". Questi nodi rappresentano le skills necessarie per affrontare quella specifica attività. Alle relazioni che uniscono le Activity e le Skill prendono il nome di "ASK".

Le relazioni ASK mi definiscono anche una dipendenza tra i nodi Skill: ad esempio, il nodo *Skill 4* mi richiede la *Skill 9*, che a sua volta mi richiede la *Skill 2*. Vediamo una possibile rappresentazione grafica in Figura 3.13.

I colori rosso, verde e blu indicano rispettivamente i nodi Activity, Skill e Employee. In questo esempio ci siamo limitati ad avere tre entità di nodi differenti, ma potremmo immaginare di espandere il problema e trovare nuovi modi per mappare più entità sullo stesso grafo.

Osservando questa implementazione, è possibile osservare la scalabilità del database Neo4j: con un database relazionale avremmo dovuto costruire tre tabelle per mappare i tre tipi di relazioni, mentre in questo modo abbiamo abbattuto la tipica rigidità dei database relazionali. A questo punto possiamo sfruttare questo grafo per effettuare le Cypher e ottenere importanti informazioni che, con altri database, sarebbe stato laborioso ottenere.

Poniamo il caso che per l'azienda sia indispensabile saper ottimizzare la disposizione delle risorse umane. Ipotizziamo di voler saper come sostituire alcune risorse se queste ultime venissero meno. Ad esempio, se il nodo *Emp1* venisse meno occorrerebbe sostituirlo immediatamente con un'altra risorsa. Questa risorsa potrebbe essere scelta in base alla sue Skills: lo scopo è dunque trovare chi soddisfa le skills


Figura 3.13: Grafo con i nodi di tipo Activity, Employee e Skill.

richieste dalla *Activity 1*. Su un grafo ridotto come quello in esame, un'analisi visiva potrebbe bastare per riuscire determinare la soluzione al problema. Ad esempio osserviamo che l'*Activity 5* richiede la *Skill 4*, che a sua volta richiede la *Skill 9* che a sua volta richiede la *Skill 2*. Quindi per la proprietà transitiva su un insieme, posso affermare che l'*Activity 5* richiede la *Skill 2*. In conclusione quindi, possiamo dire che una qualsiasi risorsa a cui è stata assegnata l'*Activity 5* possiede anche la *Skill 2* e dunque può sostituire potenzialmente il nodo *Emp1*. Tuttavia, questa analisi visiva è stata possibile per le ridotte dimensioni del grafo. Si pensi ad esempio di avere un grafo con un numero di Nodi,Label e Relazioni che non sia rappresentabile visivamente. Abbiamo dunque necessità di automatizzare questa ricerca con una Cypher creata ad hoc.

Vediamo di seguito una query Cypher che, partendo dalla Activity 1 e rendendo obbligatorio il passaggio dal nodo Skill 2, restituisce tutti i percosi possibili sulle relazioni di tipo ASK che terminano su un nodo di tipo Activity. In aggiunta restituiamo anche le relazioni TAKEPART e i nodi terminali E.

```
MATCH p = (:Activity {description:'Activity 1'})
-[:ASK]->(sk:Skill{description:'Skill 2'})<-[:ASK*]-
(:Activity)-[:TAKEPART]->(E)
return p,E
```

Il risultato restituito da tale interrogazione è il seguente:

In questo caso però abbiamo imposto un constraint sul fatto che la soluzione passasse per *Skill 2*.

Rendiamo la query ancora più generale non imponendo il constraint su *Skill 2*.

```
MATCH p = (:Activity {name:'Activity 1'})
-[:ASK]->(sk:Skill)<-[:ASK*]-(:Activity)
-[:TAKEPART]->(r)
return p,r
```

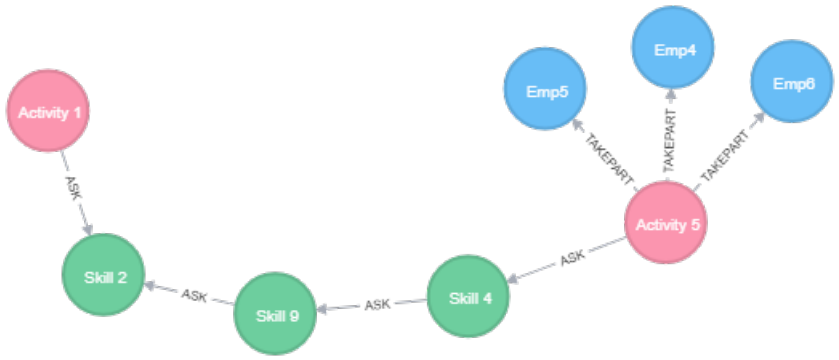



Figura 3.14: Risultato prodotto da Neo4j.

Otterremo così il seguente soluzione:

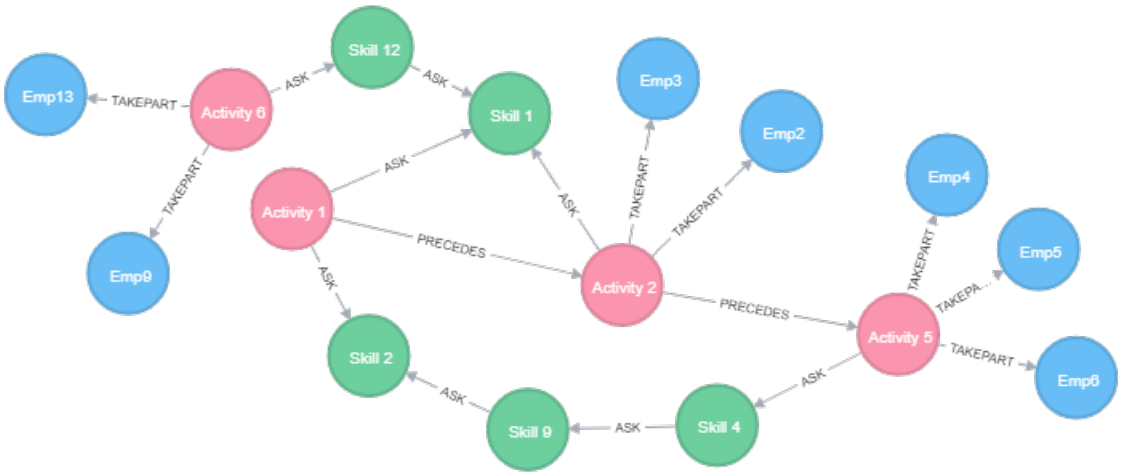



Figura 3.15: Risultato prodotto da Neo4j.

Come abbiamo visto fino ad ora, l'organizzazione delle risorse umane ricopre un ruolo molto importante nel sistema azienda. Spesso si ha interesse a raggruppare elementi non solo in base alla caratteristiche proprie dell'elemento, ma anche in base a caratteristiche indotte da altri elementi. Ad esempio, in un contesto di Social Network possiamo pensare di ottenere informazioni circa un hobby di una persona in base agli hobby che hanno i suoi amici. Quindi non è una caratteristica definita direttamente dall'utente, ma in base ad un'analisi delle sue relazioni e quindi del suo grafo di amicizie posso ricavare importanti informazioni.

Oppure possiamo pensare di raggruppare delle persone in base al loro *feeling* e in base alle loro soft skills. Sempre di più infatti, le aziende tendono a valutare una persona non solo in base alle sue competenze tecniche ma anche in base alle sue le competenze trasversali. Si pensi ad esempio l'importanza di creare un team con le persone più adatte a quello scopo, non solo in base alle loro competenze tecniche ma anche in base al loro feeling nel contesto azienda. Il feeling tra le persone è determinabile mediante più fattori, come ad esempio in numero di progetti svolti insieme, le competenze in comune, il numero di micro-task risolti insieme, ecc... Secondo Jeef Sutherland [16], l'inventore del metodo SCRUM, non basta ordinare ai componenti di un team di essere più organizzati e trascendenti; la motivazione deve arrivare dall'interno e tentare di imporla significa rovinare tutto.

Secondo uno studio effettuto dall'università di Yale, gli studenti più rapidi a volgere un compito battevano quelli più lenti per 10 a 1, ottenendo un voto altrettanto buono. Lo stesso studio è stato svolto per 3.800 team, dal lavoro delle società di revisione allo sviluppo software alle tecnologie IBM. E' stata esaminata la performance dei team e se il miglior team è riuscito a terminare un progetto in una settimana, il team più lento l'ha completato in duemila settimane. Questa è la differenza, in media, tra un team migliore da un team peggiore. Questo studio ci fa capire quanto è importante puntare a migliorare la qualità del team nella sua totalità e non focalizzarsi solo su alcune persone, magari quelle che consideriamo le più importanti. Per definizione, i team devono essere:

- **Trascendeti:** I team devono avere un senso di finalità che va oltre all'ordinario. Il solo fatto di rifiutare la mediocrità per puntare alla grandezza modifica l'idea che hanno di loro stessi, e di ciò che possono realizzare.

- **Autonomi:** Un team che funziona deve essere autonomo e autogestito per la maggior parte delle decisioni che deve prendere.

- **Interfunzionali:** I team hanno tutte le competenze necessarie per poter portare a termine il progetto. E' molto importante che ci siano nelle relazioni autentiche tra i componenti del team in modo che quelle competenze si alimentano a vicenda. Come disse un membro di Canon, *"Quando tutti i membri del team sono riuniti in una sola grande sala, le informazioni di qualcun altro diventano automaticamente tue".*

Quando si guarda ai team eccellenti come in Google, Toyota o Amazon non esiste mai una netta separazione tra i ruoli dei componenti di un team [16]. Per fare questo occorre curare le relazioni fin del primo momento e mai dare nulla per scontato. Un project manager deve occuparsi di tutti questi aspetti e cercare di riunire un numero più o meno grande di persone non solo in base alle competenze tecniche ma anche in base a innumerevoli altri fattori, che mi possono determinare le 3 caratteristiche precedentemente riportate.

Quello che si vuole fare infatti con Neo4j, è quello di fornire degli strumenti di analisi per chi ha l'arduo compito di fare tutto questo. Un esempio pratico di implementazione, potrebbe essere quello di trovare un numero dato di persone con un maggiore feeling per determinare quali potrebbero essere i potenziali componenti di un team. A tal proposito possiamo definire come feeling in numero di progetti svolti insieme in azienda o in aziende diverse.

In figura 3.16 è riportato un piccolo esempio di come si può mappare un contesto in cui esistono dei *progetti* e gli *ingegneri* che ne partecipano. In verde sono rappresentati i nodi con Label ingegnere e in grigio i nodi Label progetto. L'informazione "partecipa al progetto" è rappresentata dalla relazione *TAKE PART*, la quale è uscente dal nodo ingegnere ed è entrante nel nodo progetto.

A livello grafico è possibile osservare che i nodi E2, E8 ed E5 hanno un'importante proprietà. Se si considerano gli insiemi

$$T(P4) = \{E7, E8, E6, E2\} \tag{3.1}$$

$$T(P1) = \{E1, E2\} \tag{3.2}$$

$$T(P3) = \{E3, E8, E4, E2, E5\} \tag{3.3}$$

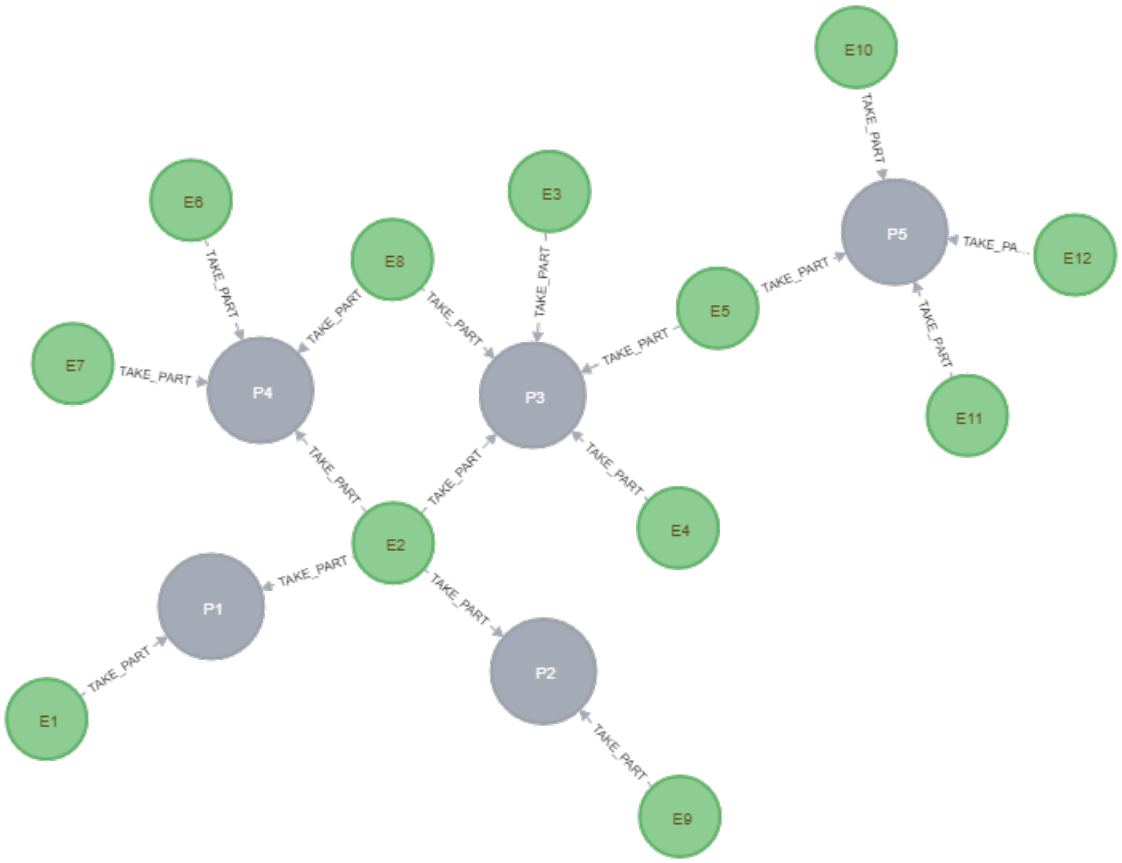



Figura 3.16: Mapping tra progetti e componenti del progetto.

$$T(P2) = \{E2, E9\} \tag{3.4}$$

$$T(P5) = \{E5, E11, E10, E12\} \tag{3.5}$$

si osserva che:

$$E2 = T(P4) \cap T(P3) \cap T(P2) \cap T(P1) \tag{3.6}$$

$$E8 = T(P4) \cap T(P3) \tag{3.7}$$

$$E5 = T(P5) \cap T(P3) \tag{3.8}$$

Questa informazione ci fa capire che E2, E8 ed E5 sono l'intersezione di questi insiemi e può essere interpretata in svariati modi, come ad esempio:

- Conoscenze trasversali: il fatto che delle risorse siano l'intersezione di più progetti può implicare che quelle risorse abbiamo acquisito conoscenze che sono al di fuori delle loro competenze principali.

- Potenziali leader: Avere più esperienze in contesti diversi aumenta per definizione la capacità di adattamento. Queste risorse potrebbero essere in grado di farsi carico di responsabilità maggiori rispetto agli altri componenti del team.

- Composizione di un nuovo team: un project manager potrebbe essere interessato a capire immediatamente quali sono dei potenziali componenti di un team, data una serie di constraints come skills richieste, lingue conosciute ecc. . .

Vediamo ora la query Cypher che ci ha permesso di ricavare questi risultati.

```
MATCH p1=(proj1:Project)<-[:TAKE_PART]-(eng1:Engineer)
with eng1, proj1, collect(eng1.name) as intorno,
collect(proj1.name) as p1
MATCH p2=(proj2:Project)<-[:TAKE_PART]-(eng2:Engineer)
where proj1.name<>proj2.name and eng2.name in intorno
WITH collect(distinct eng2.name) as
engPerProgetto,proj1,p1,collect(proj2.name) as p2
WITH collect(engPerProgetto) as progetti,p1+p2 as tot
WITH  reduce(commonProgetti = head(progetti),
  progetto in tail(progetti) |
  apoc.coll.intersection(commonProgetti, progetto))
  as Engineers,tot
RETURN Engineers, tot as commonProjects
```

La Cypher sopra riportata risulta essere già di notevole complessità, quindi la esamineremo punto per punto. Mostriamo prima di tutto il resultato della Cypher:

| Engineers | commonProjects |
|---|---|
| ["E8", "E2"] | ["P4", "P3", "P1", "P2", "P3"] |
| ["E8", "E2", "E5"] | ["P3", "P4", "P1", "P2", "P4", "P5"] |
| ["E5"] | ["P5", "P3"] |
| ["E2"] | ["P2", "P1", "P3", "P4"] |

Figura 3.17: Output della query Cypher.

In Figura 3.17 viene rappresentato l'output di Neo4j senza nessun altra elaborazione aggiuntiva. Nella colonna *commonProjects* vengono rappresentati i progetti ai quali hanno partecipato gli *Engineers* della colonna di sinistra. Con questa interrogazione riusciamo ad ottenere tutti i tipo di intersezioni: ad esempio vediamo che E8 ed E2 sono l'intersezione dei team che hanno partecipato a P4,P3,P1,P2, mentre E5 è l'intersezione dei team che hanno partecipato a P5 e P3.

Si parte con la prima clausola MATCH che ci restituisce tutti i percorsi che partono da un nodo Engineer e terminano in un nodo Project, attraverso una sola relazione TAKEPART. La clausola WITH ci permette di iniettare in input dei dati alla seconda parte della Cypher. In particolare passiamo i valori *eng1*, *prog1*, *collect(eng1.name)* rinominato come *intorno* e *collect(proj2.name)* rinominato come p1.

La funzione *collect* mi permette di collezionare in una lista i nodi passati come parametro. Se esegue poi una seconda MATCH equivalente alla prima e si impone nella clausola WHERE che i nodi proj1 e proj2 siano diversi, in modo da escludere gli stessi percorsi e ridurre il dominio di ricerca. L'obbiettivo è trovare l'intersezione tra i due insiemi p1 e p2, creiamo dunque una lista di liste con *collect(engPerProgetto)* rinominandola *progetti*. Per ottenere la colonna *commonProjects*, sommiamo i nodi Project contenuti nella lista p1 con quelli contenuti nella lista p2, con *p1+p2 as tot*.

La parte più significativa della Cypher è sicuramente la funzione *reduce*. Questa funzione applica un'espressione per ogni elemento *progetto* della lista *tail(progetti)*.

Il risultato di tale espressione viene accumulato nella variabile *commanProgetti*, la quale viene inizializzata con *head(progetti)*. Le funzioni *head* e *tail* restituiscono rispettivamente la testa e la coda di una lista di nodi. L'espressione all'interno della funzione reduce è *intersection*, la quale mi restituisce l'intersezione tra gli insiemi *commonProgetti* e *progetto*. Questa funzione *intersection* fa parte della libreria APOC di Neo4j [2]. Ritorniamo infine **Engineers** e **commonProjects**. In conclusione quindi, abbiamo ottenuto una serie di insiemi che rappresentano tutti i migliori modi possibili per raggruppare i nodi Engineer.

## 3.4 Adozioni

Il terzo caso di studio che abbiamo considerato riguarda il problema delle adozioni e sostegno umanitario. La Congregazione Suore Carmelitane *di Santa Teresa di Torino* ha costituito la "FONDAZIONE Missioni Suore Carmelitane di S. Teresa di Torino ONLUS" la quale si occupa di promuovere e realizzare progetti per il sostegno a distanza di bambini e giovani in condizioni economiche, sociali e familiari svantaggiate.

### 3.4.1 L'obbiettivo

Questa organizzazione utilizzava già un CRM prodotto dalla stessa Atlante Informatica. Un attento studio della vecchia base dati ha portato la nostra attenzione sull'importante presenza di molti tipi di entità relazione e alla necessità di poter estrapolare dati significativi dalle stesse. Ad oggi, il nostro obiettivo è quello di fornire un nuovo prodotto che implementi funzionalità incentrate sullo studio dei grafi.

### 3.4.2 Modellazione

La modellazione del problema sul grafo è stata effettuata quasi totalmente mediante importazione di CSV. Attraverso il tool **neo4j-import**, situato nella cartella di istallazione path/to/neo4j/bin/neo4j-import, eseguiamo l'importazione sia delle tabelle riportanti i nodi sia per quelle riportanti le relazioni. Questo comando è attualmente il modo più veloce per eseguire degli import, riuscendo a caricare su Neo4j fino a 1.000.000 di record al secondo.

A tal proposito, sono stati creati dei file CSV in modo da poterli utilizzare con questo comando. Le tabelle che sono state importate sono:

- **Tabelle nodi**

  Child bambini interessati per le adozioni.

  Mission missioni alle quali fanno parte i bambini.

  Agent intermediari tra gli animatori e le missioni.

  Parent genitori adottivi.

  Animator suore della Fondazione Suore Carmelitane di S. Teresa di Torino.

  Educator educatori presenti nelle scuole.

  School scuole costruite per la missione umanitaria.

- **Tabelle relazioni**

  Child_Mission relazioni tra bambino e missione.

  Child_Paret relazioni tra bambino e genitore adottivo.

  Child_School relazioni tra bambino e scuola frequentata.

  Educator_School relazion tra educatori e scuola nella quale insegnano.

  Animator_Child relazioni tra animatore e bambini.

  Agent_Mission relazioni tra agenti e missioni di cui si occupano.

Nella vecchia base dati, sono presenti molte altre tabelle ma che non erano significative per il nostro caso di studio. Vediamo ora come sono sta impostati gli Header delle tabelle **Nodi** di importazione per permettere al tool Neo4j di importarle correttamente.

| **MissionId:ID(Mission)** | **Name** | **Start** | **Country** | **...** |
|---|---|---|---|---|

Tabella 3.1: missioni.csv

| **Agent:ID(Agent)** | **Name** | **Surname** | **Country** | **...** | **MissionId:IGNORE** |
|---|---|---|---|---|---|

Tabella 3.2: agent.csv

| **Child:ID(Child)** | **Name** | **Surname** | **Country** | **...** | **AgentId:IGNORE** |
|---|---|---|---|---|---|
| **MissionId:IGNORE** | **SchoolId:IGNORE** | | | | |

Tabella 3.3: child.csv

| **Animator:ID(Animator)** | **Name** | **Surname** | **Country** | **Onlus** | **...** |
|---|---|---|---|---|---|
| **AgentId:IGNORE** | **ChildId:IGNORE** | | | | |

Tabella 3.4: animator.csv

| **ParentId:ID(Parent)** | **Name** | **Surname** | **Country** | **...** |
|---|---|---|---|---|

Tabella 3.5: parent.csv

| **EducatorId:ID(Educator)** | **Name** | **Surname** | **Country** | **...** | **EducatorId:IGNORE** |
|---|---|---|---|---|---|

Tabella 3.6: educator.csv

Riportiamo di seguito le tabelle **Relazioni**.

| SchoolId:ID(School) | Name | Address | Country | ... |
|---|---|---|---|---|

Tabella 3.7: Tabella Scuole

| :START_ID(Child) | :END_ID(Mission) | :TYPE |
|---|---|---|

Tabella 3.8: Tabella relazioni Bambini Missioni

| :START_ID(Parent) | :END_ID(Child) | :TYPE |
|---|---|---|

Tabella 3.9: Tabella relazioni Bambini Genitori adottivi

| :START_ID(Child) | :END_ID(School) | :TYPE |
|---|---|---|

Tabella 3.10: Tabella relazioni Bambini Istituti

| :START_ID(Educator) | :END_ID(School) | :TYPE |
|---|---|---|

Tabella 3.11: Tabella relazioni Educatori Istituti

| :START_ID(Animator) | :END_ID(Child) | :TYPE |
|---|---|---|

Tabella 3.12: Tabella relazioni Animatori Bambini

| :START_ID(Agent) | :END_ID(Mission) | :TYPE |
|---|---|---|

Tabella 3.13: Tabella relazioni Agenti Missioni

Esaminiamo ora i vari Header nel dettaglio descrivendone il significato. Nella Tabella 3.3 vediamo l'header *MissionId:ID(Mission)* il quale indica che quella sarà una colonna nella quale sanno riportati tutti gli ID delle missioni. In automatico, se non specificato diversamente attraverso apposito header :LABEL, a quel nodo verrà assegnata la label *Mission*.

I successivi header *Name, Start, Country* indicano rispettivamente il nome, inizio e paese in cui si svolge la missione umanitaria. Queste tre colonne verranno memorizzate come Proprietà all'interno del nodo Mission. Notare che per brevità sono state omesse dalla rappresentazione grafica le altre colonne, ma comunque importate su Neo4j regolarmente.

Se analizziamo la Tabella 3.5, vediamo che ci sono degli header del tipo *:IGNORED*; questi header verranno ignorati dal tool Neo4j durante il caricamento. Avremmo potuto eliminare direttamente le colonne del file CSV in alternativa del comando *:IGNORED*.

Per le tabelle **Relazioni** invece, abbiamo necessità di identificare il nodo sorgente e il nodo destinazione della relazione. A tal proposito si utilizzano gli header *:START_ID* e *:END_ID*, mentre per assegnare il Tipo alla relazione, si usa l'header *:TYPE*.

### 3.4.3 Script di importazione

Vediamo ora la prima riga del comando dalla schell di Neo4j per eseguire l'importazione massiva della base dati sopra descritta.

```
bin/neo4j-import --into adogest.db --id-type string \
```

Il comando quindi interpreta gli header di ogni file csv ed esegue l'upload dei nodi e delle relazioni. L'importazione di file CSV si potrebbe eseguire più facilmente attraverso una query Cypher ma su grandi quantità di dati non risulta essere il metodo migliore per un'importazione massiva.

La prima parte del comando specifica il database *adogest.db* sul quale si vuole eseguire l'upload con a seguire la specifica *id-type string* la quale indica che gli id dei nodi sono di tipo alfanumerico. Successivamente, attraverso il comando *–node:<Label>* si specificano le tabelle che riportano i nodi e con il comando *–relationship:* per specificare te tabelle relazioni. Da notare che con il comando *–node:* si specifica anche la *Label* che sarà associata a quei nodi. Come già detto nei capitoli precedenti, si può paragonare la Label come al nome della tabella per un database relazionale.

### 3.4.4 Grafo delle adozioni

Riportiamo in Figura 3.18 una riproduzione ridotta del grafo originale. Possiamo subito notare come Neo4j abbia creato automaticamente Nodi, Label e Relazioni in base ai file CSV specificati.

In questa implementazione vediamo la vera potenzialità di Neo4j e di quanto posso ridurre la complessità della base dati, aggiungendo inoltre funzionalità ulteriori che i normali database relazionali non offrono. In questo esempio infatti, si percepisce la scalabilità e la flessibilità di un database a grafo nativo come Neo4j. In Neo4j infatti possiamo non preoccuparci di quanti tipi di relazioni e nodi dobbiamo creare per la nostra base dati: essendo creato a basso livello con il principio della **index-free adjacency** possiamo permetterci di creare molteplici entità senza aumentare esponenzialmente la complessità computazionale di una Cypher in produzione.

Ricordiamo però che per queries che non sono usate per attraversamenti sul grafo, queste possono impattare negativamente sull'utilizzo della memoria, come già spiegato nei precedenti capitoli. A tal proposito si valuta anche in questa implementazione, la possibilità di avere due tipi di database; un database relazionale che contiene tutti i dati di "contorno" (indirizzo, scuola, lingua ecc...) dei nodi e un database a grafo che contiene le proprietà indispensabili per le Cypher (id,nome,cognome) tra cui sicuramente l'indice del nodo che mi corrisponde all'indice sul database relazionale.

Questa soluzione sembra essere quella più logica ma occorre tenere in considerazione la parte di sincronizzazione dei due database, rischiando di aumentare la complessità per poi non avere un grande incremento di prestazioni. L'implementazione del grafo delle adozioni, tuttavia, è stata realizzata abbandonando totalmente il database relazionale ma caricando tutti i dati nel grafo Neo4j.

Una delle ultime richieste da parte della Fondazione è stato quello di poter associare i bambini non solo ad un Animatore ma anche a altre figure professionali che si sono formate nel tempo. Quindi un bambino può essere associato ad un Animatore,

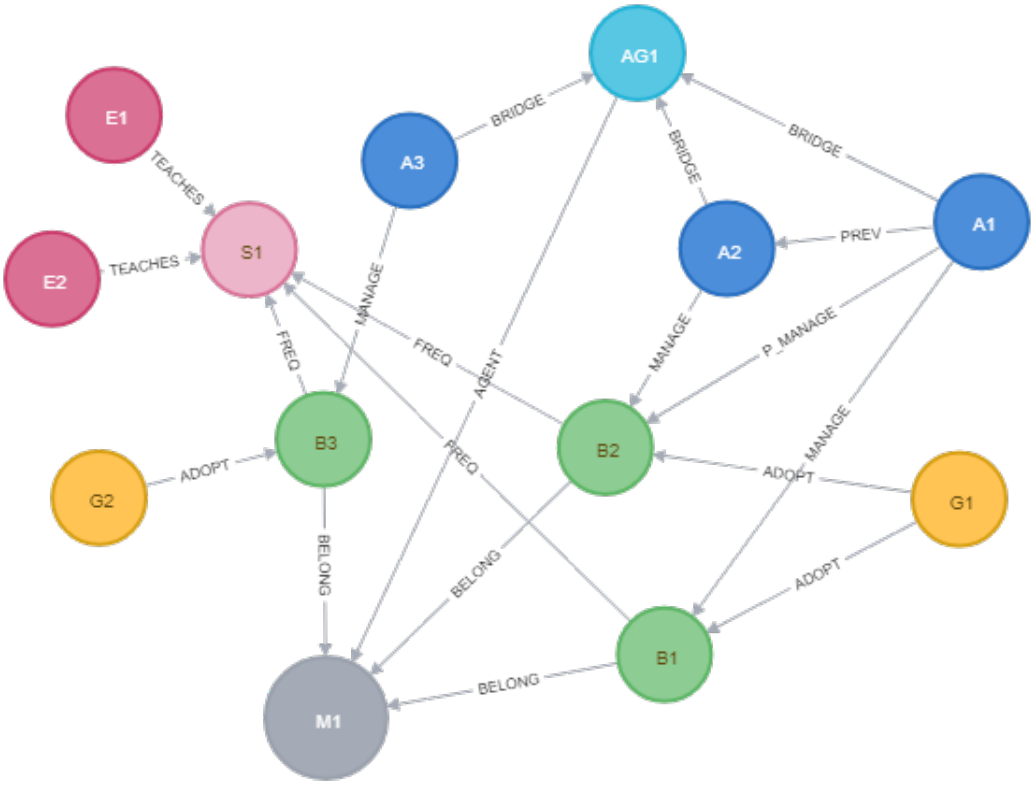



Figura 3.18: Grafo adozioni.

ad un Mediatore o ad un Responsabile Famiglia. In un database relazionale questo comporterebbe a delle modifiche considerevoli in quanto la struttura rigida non permetterebbe di aggiungere in modo flessibile queste tipo di relazioni. Si dovrebbe quindi creare una tabella come si vede in Tabella 3.16.

| **idChild** | **idAnimator** | **idMiddleman** | **idFamily** |
|---|---|---|---|
| 123 | null | 78 | null |
| 124 | 564 | null | null |
| 125 | 588 | null | null |
| 126 | 394 | null | null |
| 127 | null | null | 192 |
| 128 | null | 510 | 112 |

Tabella 3.14: Tabella delle relazioni equiparabile ad una matrice sparsa

Questa risulta essere una soluzione totalmente inefficiente e con grandi sprechi di memoria, anche se risulta essere l'unico modo per ampliare lo schema relazionale senza stravolgere totalmente la struttura dati. Con un database NOSQL come Neo4j invece, si riesce a scalare senza dover modificare la struttura dati ma andando solamente ad aggiungere nuove entità Nodo e Relazione con una semplice interrogazione ad database. Quindi si aggiungono al database i nodi Middelemen, e Responsabile Famiglia con le rispettive Label *Middleman* e *Family*.

Un aspetto che non abbiamo ancora trattato è quello di come mappare le **informazioni temporali nel grafo**. Ad esempio, potremmo avere necessità di assegnare dei livelli professionali agli educatori; quindi un educatore alle prime armi avrà un

livello 1 per poi aumentare di livello negli anni. L'aspetto fondamentale è che a questi livelli professionali non coincide solamente l'informazione circa le capacità acquisite dell'educatore ma a quel livello corrisponde anche un determinata "storia" dell'educatore: ad esempio, l'educatore E1 a livello 1, esercitava in una scuola S1 la quale faceva parte di una determinata missione M1 la quale era gestita dall'agente AG1. Quindi, per ogni livello professionale, si vuole avere una chiara "fotografia" di quella che era la situazione in quel momento.

Se volessimo riprodurre questa informazione su un database relazionale, dovremmo creare sicuramente una nuova tabella degli "stati" nella quale memorizzare, per ogni Educatore, i livelli che ha raggiunto e i rispettivi elementi (Scuola, Missione, Agente, ecc...) associati in quell'istante.

Potremmo aver necessità di applicare lo stesso criterio anche per i Bambini, ovvero è possibile che ci sia interesse a memorizzare le assegnazioni di un bambino da una missione ad un'altra. Vediamo in Figura 3.18, come si potrebbe mappare il problema sul grafo.

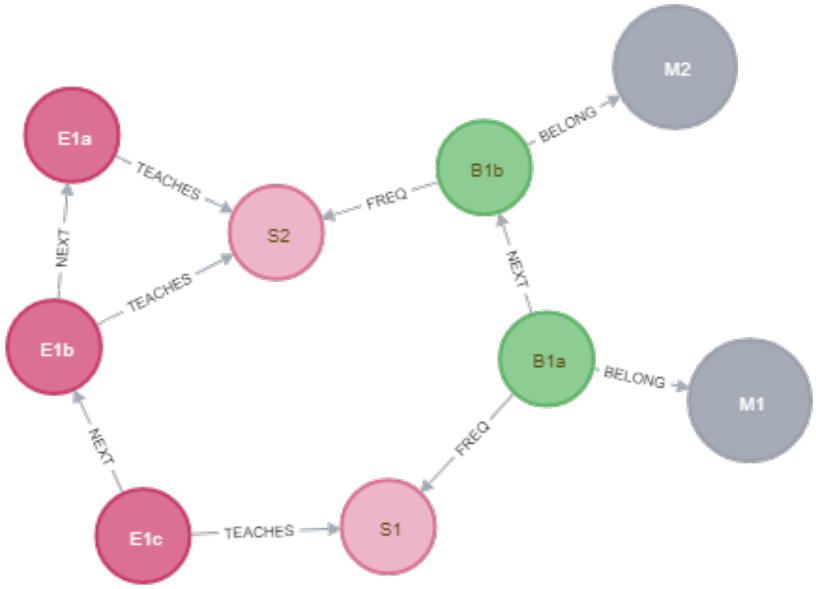



Figura 3.19: Grafo adozioni con informazioni temporali.

Vediamo chiaramente come il bambino B1 sia stato assegnato prima ad una missione M1 e una scuola S1 e successivamente ad una missione M2 e scuola S2. Per motivi di correttezza dei dati, abbiamo assegnato il nome B1a e B1b per differenziare i due stati. Stesso ragionamento viene fatto per l'educatore E1, il quale viene diviso in E1c, E1b ed E1a che mi corrispondono ai livelli professionali di quell'educatore. Quindi nel primo livello E1c insegnava nella scuola S1, nella quale era presente B1a che a sua volta apparteneva alla missione M1.

In generale quindi, possiamo applicare questo pattern tutte le volte in cui ci troviamo nella condizione di dover mappare degli stati che cambiano nel tempo; in questo modo il grafo avrà le informazioni in merito sia alle relazioni attuali tra le entità sia quelle passate. All'interno dei nodi potremmo inoltre memorizzare una proprietà che mi identifichi la data del cambio di stato in modo da dare anche un riferimento temporale per eventuali query Cypher future.

A questo punto siamo pronti a rispondere facilmente a domande del tipo *"Il bambino B1, è mai stato in una scuola con l'educatore E1?"* oppure *"La missione M2 ha mai coinvolto Educatori che siano stati assegnati alla scuola S1?"*, e così via.

# Capitolo 4

# Load Balancing e Dimensionamento

In questo ultimo capitolo analizzeremo quali possono essere le strategie architetturali e implementative che migliorano le prestazioni di un database a grafo come Neo4j. Partiremo con l'analizzare il concetto di Load Balancing per finire parlando del dimensionamento dell'hardware.

## 4.1 Clustering

In generale, per permettere ad un sistema OLTP di avere hight availability e fault-tolerance a fronte di un carico di richieste elevato, si deve obbligatoriamente scalare orizzontalmente.

Come abbiamo già discusso nel Paragrafo 2.4.2 sulla Availability, Neo4j permette una disposizione Clustering master-slave per una corretta ridondanza. Abbiamo detto che le scritture possono essere fatte non solo attraverso il Master ma anche attraverso gli Slave avendo però un tempo di scrittura di almeno un ordine di grandezza più lento dovuto al traffico di rete e tutti i protocolli aggiuntivi di coordinamento.

L'unico motivo per cui si debba scrivere attraverso gli slave è quello di aumentare la durabilità di ogni scrittura e per un fattore di cache condivisa come vedremo successivamente. Tuttavia, Neo4j raccomanda di usare sempre il Master per le scritture, configurando il sistema usando le impostazioni di configurazione ha.tx_push_factor e ha.tx_push_strategy.

In contesti nei quali il carico di scritture è molto elevato si potrebbero usare delle **code per bufferizzare le scritture**. In questo modo si maschererebbero dei periodi di manutenzione del database, andando a memorizzare le richieste dei client senza rifiutarle [3].

## 4.2 Load Balancing

Quando utilizziamo Neo4j su più cluster, è auspicabile porci il problema di come eseguire una corretta funzione di Load Balancer di smistamento del traffico tra i cluster, al fine di massimizzare il throughput riducendo la latenza.

Come ci suggerisce letteratura di Neo4j, possiamo quindi dividere il traffico in traffico in scrittura e traffico in lettura. A tal proposito quindi, se implementiamo

Neo4j per una web-application è sufficiente distinguere le richieste POST, DELETE e PUT da una semplice richiesta GET. Infatti, le prime tre richieste sicuramente modificheranno la base dati e quindi saranno operazioni di scrittura, mentre GET sarà un'operazione di lettura.

L'attuale implementazione di CmWallet ha un'integrazione di Neo4j fatta interamente con HTTP API di Neo4j, ovvero in Server Mode. Questo implica che attualmente non è possibile discriminare le operazioni di scrittura da quelle di lettura. Infatti uno dei prossimi sviluppi del progetto CmWallet sarà quello di creare un'applicazione scritta in Java che sfrutti le REST API in modo da poter trattare le singole richieste GET,POST,PUT e DELETE in modo differente le une dalle altre e quindi poter creare funzionalità di Load Balancing.

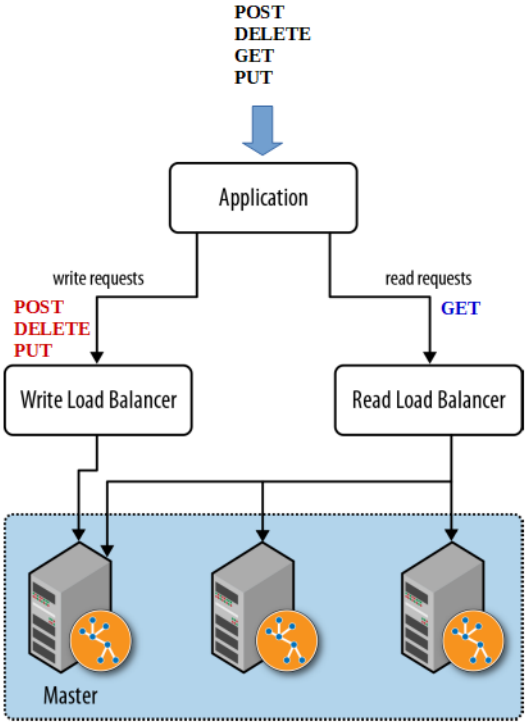



Figura 4.1: Load Balancing in base al tipo di richiesta [9].

### 4.2.1 Cache frammentata - Cache sharding

Come già accennato nel Capitolo 2, le interrogazioni al database avranno elevate prestazioni se la partizione del grafo è già contenuta nella memoria principale. E' facile intuire che questa situazione si potrebbe presentare per grafi di piccole e medie dimensioni, ma quando si parla di grafi di milioni di nodi, relazioni e proprietà occorre considerare delle opportune strategie per fare in modo che le interrogazioni non abbiamo una latenza eccessiva, dato che è del tutto improbabile che la partizione del grafo che ci interessa sia già presente in memoria.

La strategia del **Cache sharding** è quello di instradare le richieste su uno dei cluster disponibili in base al punto di accesso al grafo necessario a quella richiesta. Quindi le query che partono da uno stesso punto del grafo si tende a direzionarle sempre su uno stesso cluster. In questo modo si aumenta la probabilità che quella partizione del grafo necessaria per quella richiesta, sia già presente in memoria

principale di quel cluster, evitando quindi operazioni di swap tra disco e memoria principale.

Vediamo in Figura 4.2 una rappresentazione di quello fin'ora spiegato. Le richieste A, B e C vengono dirette su tre cluster diversi, ognuno dei quali ha in cache una partizione specifica del grafo.

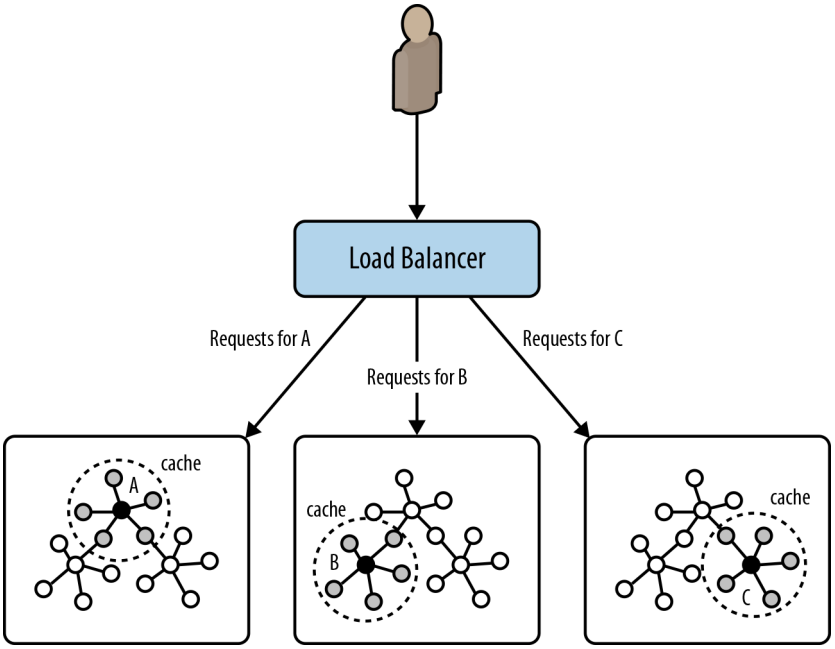



Figura 4.2: Cache Sharding [9].

Questo sistema di cache sharding potrebbe essere sfruttato da un social network. Se si immagina di avere un grafo che contiene persone di tutto il mondo, ha senso pensare che un utente italiano faccia accesso al cluster che mi contiene in cache la parte degli utenti italiani del grafo.

In tutti i nostri casi di studio non abbiamo adottato questa soluzione in quanto la grandezza dei grafi era tale da poter essere contenuto tutto all'interno della memoria principale.

## 4.3 Dimensionamento

Dato il funzionamento di Neo4j visto nei precedenti capitoli, abbiamo eseguito un dimensionamento tale per cui il grafo fosse contenuto interamente in memoria principale. Questo è stato possibile perché i grafi in produzione hanno dimensioni tali da poter evitare l'utilizzo di più cluster. In tutti i casi di studio visti, non si ha mai in produzione un aumento drastico del numero di nodi e quindi le prestazioni non avrebbero mai dei cali improvvisi, a meno di eventi che vanno oltre al corretto dimensionamento.

Nel caso in cui la dimensione del grafo sia troppo grande per un unico cluster occorre adottare la soluzione di Cache Sharding precedentemente descritta. Durante la fase di dimensionamento quindi occorre tenere presente la quantità di Heap e di Buffer Cache che si vuole allocare. Avere però Heap troppo grandi comporterebbe a compromettere le prestazioni del Garbage Collector e quindi ad incidere sui tempi di risposta dell'applicativo.

Durante la fase di test dell'applicativo infatti è stata fatta un'analisi sul comportamento del Garbage Collector in modo da evitare spiacevoli inconvenienti in produzione.

# Capitolo 5

# Test e analisi dei risultati

In questo capitolo andremo ad analizzare i risultati dei test dei tre casi di studio, provando a trarre alcune considerazioni.

## 5.1 Ottimizzazione dei costi di produzione

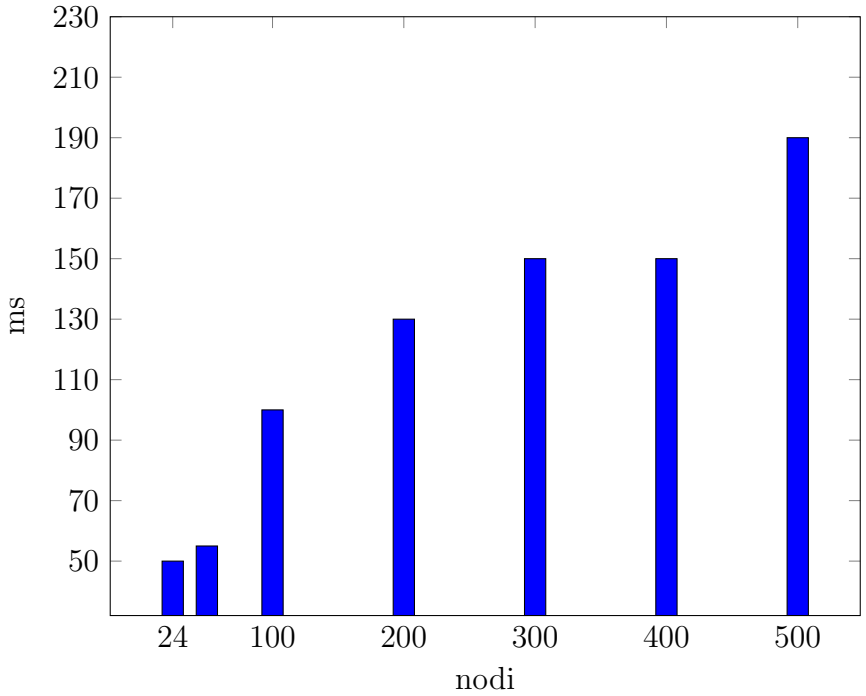
### 5.1.1 Test

Durante il periodo di sviluppo si sono svolti test circa la velocità di risposta del grafo. Iniziamo a descrivere con la tabella di seguito le caratteristiche della macchina virtuale che ospitava Neo4j.

| **Componente** | **Dettagli** |
|---|---|
| **System/Chassis** | 1U Intel |
| **RAM** | 64 GB |
| **Disk** | 1 TB SSD x 6 |
| **CPU** | XEOM E7-8855 V4 |
| **Hypervisor** | Proxmox |

Tabella 5.1: Configurazione della macchina virtuale

I risultati ottenuti per quanto riguarda l'attraversamento del grafo sono molto soddisfacenti. Nel caso reale, utilizzando 300 nodi (componenti) e circa 9000 relazioni, si ha un tempo di risposta di circa 150 ms per la ricerca del commino minimo su Neo4j. Andando ad aumentare il numero di nodi e di relazioni il tempo di risposta ri-




mane piuttosto contenuto.

In fase di test siamo partiti da un valore basso di nodi e relazioni fino ad ad arrivare fino a 500 nodi e 2500 relazioni. Quello che più influisce sul tempo di risposta è il numero di relazioni. Infatti, soprattutto in un grafo DAG fortemente connesso, il numero di relazioni aumenta le possibili combinazioni che sono all'interno dello spazio di ricerca dell'algoritmo.

## 5.2 Diagramma reticolare

### 5.2.1 Test

I test sono stati svolti grazie ad un generatore randomico di grafi disponibile sul sito *www.graphgen.graphaware.com*. In Tabella 3.2 vengono riportati i risultati dei test. Dall'applicazione Graphaware sono stati estrapolati dei file formato json e caricati sul database Neo4j attraverso uno script php, il quale scandiva il json e creando uno statemet alla volta.

Dai risultati ottenuti si osserva che il tempo di elaborazione aumenta notevolmente quando il numero di relazioni incrementa. Questo implica che nella maggior parte dei casi, il tempo di elaborazione non dipende strettamente dal numero di nodi, ma dal numero di relazioni.

Le relazioni, da punto di vita matematico, mi esprimono le combinazioni possibili nel dominio di ricerca. Il *grado* medio dei nodi Engineer è di 6 (6 relazioni TAKE_PART). Per definizione, il grado di un nodo è il numero di relazioni entranti più in numero di relazioni uscenti del nodo stesso.

Nel test numero 6 e 8, è stato impostato il caso limite in cui un nodo Enginner ha una sola relazione uscente (grado 1) verso un noto Project; questo implica che il numero di relazioni sia pari al numero di nodi Enginner. Possiamo quindi generalizzare il concetto dicendo che se **E** é il numero di archi e **V** è il numero di nodi di un dato grafo **G**, distinguerò due casi limite:

$$G_{denso}: |E| \simeq |V|^2$$

$$G_{sparso}: |E| \ll |V|^2$$

Per un grafo denso si avrà in generale che il numero di archi (relazioni) E è circa uguale al quadrato del numero di vertici (nodi) V. Per un grafo sparso invece, il numero di archi è molto minore del quadrato del numero di vertici. Questo implica quindi che i tempi di un qualsiasi attraversamento per un grafo denso saranno mediamente sempre maggiori rispetto ad un grafo sparso, a parità di numero di nodi. Infatti, nei test 6 e 8 abbiamo generato volontariamente un grafo sparso ottenendo tempi di elaborazione minori rispetto al test 7. In particolare notiamo che nonostante il test 8 sia eseguito su un numero di nodi maggiore rispetto al test 7, ha ottenuto un tempo di elaborazione nettamente inferiore.

| **test** | **Nodi E.** | **Nodi P.** | **TAKE_PART** | **Time** |
|---|---|---|---|---|
| 1 | 20 | 10 | 98 | 10 records completed in 144 ms. |
| 2 | 100 | 20 | 284 | 20 records completed in 652 ms. |
| 3 | 300 | 50 | 1167 | 50 records completed in 1494 ms. |
| 4 | 300 | 50 | 1885 | 50 records completed in 5070 ms. |
| 5 | 300 | 50 | 2731 | 50 records completed in 7788 ms. |
| 6 | 600 | 100 | 600 | 100 records completed in 574 ms. |
| 7 | 600 | 100 | 5708 | 100 records completed in 35981 ms. |
| 8 | 1200 | 200 | 1200 | 169 records completed in 1929 ms. |

Tabella 5.2: Risultati dei test

## 5.3 Adozioni

### 5.3.1 Test

Anche in questo caso è stato creato un grafo sovradimensionato attraverso in tool *www.graphgen.graphaware.com*. Per questo caso di studio non si sono applicati dei pesanti algoritmi di ricerca, ma sostanzialmente delle Cypher che restituiscano un percorso tra Animatore e Bambino o tra Genitore e Educatore.

Anche aumentando la complessità del grafo, i tempi di restituzione dei risultati è pressoché costante. Questo è un'importate risultato ottenuto: possiamo quindi concludere che un grafo nativo come Neo4j ha tempi di restituzione che aumentano di pochissimo all'aumentare del numero di nodi e di relazioni. Attenzione però che

non si tratta di ricerche di ottimizzazione o analisi, ma semplicemente di una ricerca tipica di un sistema OLTP.

Di seguito sono mostrati i risultati in merito al grafo delle Adozioni:

| **test** | **Nodi** | **Relazioni** | **Time** |
|---|---|---|---|
| 1 | 100 | 289 | completed in 12 ms. |
| 2 | 250 | 468 | completed in 16 ms. |
| 3 | 350 | 691 | completed in 13 ms. |
| 4 | 450 | 843 | completed in 27 ms. |
| 5 | 700 | 1322 | completed in 54 ms. |
| 6 | 800 | 1398 | completed in 52 ms. |
| 7 | 900 | 1655 | completed in 60 ms. |
| 8 | 1000 | 2119 | completed in 72 ms. |

Tabella 5.3: Risultati dei test

I questo ultimo caso di studio abbiamo verificato la grande differenza tra la ricerca di un cammino da un nodo Agente e un nodo Educatore eseguita su un RDBMS rispetto ad eseguirla su Neo4j.

In particolare, mantenendo circa sempre lo stesso numero di nodi, vediamo in Tabella 5.4 che per un database relazionale il tempo di esecuzione non risulta essere mai inferiore al secondo.

| **test** | **Nodi** | **Relazioni** | **Time** |
|---|---|---|---|
| 1 | 100 | 250 | completed in 1.5 s. |
| 2 | 250 | 400 | completed in 1.5 s. |
| 3 | 350 | 650 | completed in 2.2 s. |
| 4 | 450 | 800 | completed in 2.3 s. |
| 5 | 700 | 1300 | completed in 3.1 s. |
| 6 | 800 | 1400 | completed in 3.5 s. |
| 7 | 900 | 1700 | completed in 4 s. |
| 8 | 1000 | 2000 | completed in 6 s. |

Tabella 5.4: Risultati dei test per RDBMS

Per gli altri due casi di studio invece, avendo affrontato dei problemi di ricerca e ottimizzazione per i quali un RDBMS non è progettato, non è stato possibile confrontare i tempi di Neo4j.

# Capitolo 6

# Conclusioni

In questo capitolo finale proveremo a fare alcune considerazioni sulle implementazioni precedentemente descritte, ponendo l'attenzione sugli aspetti positivi e negativi della soluzione. Descriveremo anche quando effettivamente un'azienda o un'organizzazione dovrebbe investire in un database a grafo come Neo4j e quando invece non è necessario.

## 6.1 Considerazioni finali

In base ai tipi di implementazione visti, possiamo quindi definire Neo4J come un sistema *On Line Transaction Processing (OLTP)*. Però è chiaro che con gli strumenti che mette a disposizione Neo4j si possa anche utilizzare Neo4j come un sistema *On Line Analytical Processing (OLAP)*.

Infatti, la libreria di algoritmi **APOC** messa a disposizione da Neo4j comprende una serie di strumenti come Commini minimi, Componenti connesse, Rilevazione di cicli, Spanning Tree e molti altri, che possono essere utilizzati come strumenti di analisi dei dati raccolti in un'azienda, restituendo risultati in tempo reale. E' bene però che le funzionalità di OLTP e OLAP siano ben distinte sulla nostra infrastruttura. Sebbene infatti Neo4j permetta un'analisi immediata dei dati, è opportuno che le interrogazioni volte estrapolare risultati tipici di un sistema OLAP siano rivolte verso un'istanza nel database dedicata solamente e questo tipo di interrogazioni. Ad esempio, nella prima implementazione nel Paragrafo 3.2 *Ottimizzazione dei costi*, abbiamo visto Neo4j come uno strumento di analisi. In questo caso infatti la nostra soluzione è stata quella di mantenere il database relazionale e di creare un database a grafo che permetta solamente questo tipo di interrogazioni, che ovviamente il RDBMS non sarebbe in grado di gestire.

Al CRM CmWallet quindi sono state applicate soluzioni architetturali differenti in base al tipo di problema che si voleva affrontare. Alla domanda *"Neo4J quindi è la soluzione definitiva andando a sostituire i database relazionali?"* possiamo quindi dare una risposta. Come abbiamo analizzato in questo elaborato, Neo4j essendo un grafo nativo non è stato progettato per le query non ideomatiche, ovvero query che non sfruttano l'architettura di Neo4j. Questo vuol dire che se Neo4j venisse usato come un normale database relazionale (senza pensanti operazioni di JOIN) non ci sarebbe un incremento delle prestazioni ma, al contrario, si potrebbe percepire una latenza maggiore per tutte le ragioni descritte nei precedenti capitoli. Neo4J è uno strumento molto potente, ma la chiave per ottenere un'implementazione della

soluzione accettabile è quella di riuscire a mappare il problema sul grafo. La qualità del mapping infatti determinerà la latenza delle risposte.

## 6.2 Sviluppi futuri

Il progetto CmWallet viene attualmente utilizzato sia internamente all'azienda ma anche esternamente da diversi clienti. Il nostro obbiettivo è quello di continuare lo studio di Neo4j, analizzando tutte le strategie possibili in campo aziendale per sfruttare al meglio le potenzialità di Neo4J.

# Bibliografia


[1] *ACID Transaction*. 2018. URL: https://it.wikipedia.org/wiki/ACID.

[2] *APOC User Guide 3.4.0.4*. 2017. URL: https://neo4j-contrib.github.io/neo4j-apoc-procedures/.

[3] Rik Van Bruggen. "Learning Neo4j". In: (2017).

[4] *Cammino Minimo*. 2018. URL: https://it.wikipedia.org/wiki/Cammino_minimo.

[5] *Data Modelling*. 2018. URL: https://neo4j.com/developer/relational-to-graph-modeling.

[6] *Float (project management)*. 2017. URL: https://en.wikipedia.org/wiki/Float_(project_management).

[7] *Graph Databases for Beginners: Native vs. Non-Native Graph Technology*. 2018. URL: https://neo4j.com/blog/native-vs-non-native-graph-technology/?ref=blog.

[8] Jom Webber Emil Eifrem Ian Robinson. *Graph Databases - New apportunities for connected data*. 2018.

[9] Ian Robison e Jim Wabber. "The Graph Database. O'Reilly Media". In: (2013).

[10] Project Management Institute. *A Guide to the Project Management Body of Knowledge*. 2017.

[11] *Metodo del percorso critico*. 2017. URL: https://it.wikipedia.org/wiki/Metodo_del_percorso_critico.

[12] *Neo4 DBMS*. 2018. URL: https://neo4j.com/product.

[13] *Neo4 DBMS*. 2018. URL: https://neo4j.com/developer/graph-db-vs-rdbms/.

[14] J. Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *ArXiv e-prints* (giu. 2015). arXiv: 1506.02640 [cs.CV].

[15] *scalability*. 2018. URL: https://it.wikipedia.org/wiki/Scalabilit%C3%A0.

[16] Jeff Sutherland. *Fare il doppio in metà tempo*. 2017.

[17] Retal Tic. *I CRM fondamentali*. 2017. URL: http://www.retail-tic.com/crm_fondamentali.html.

[18] *Why Graph Technology Is the Future*. 2018. URL: https://neo4j.com/blog/why-graph-databases-are-the-future/.

[19] *Zend Framework*. 2017. URL: https://framework.zend.com/about.

[20] *Zend Framework*. 2017. URL: http://www.html.it/pag/19100/i-vantaggi-di-zend-framework/.

[21] *Zend Framework*. 2017. URL: https://www.sitepoint.com/.