



# UNIVERSITÀ DI PISA

Corso di Laurea in Informatica

## IMPLEMENTAZIONE CON NEO4J DI UN DATABASE A GRAFO CONTENENTE MOVIMENTI FINANZIARI

Candidato: Giorgio Pirina

Relatore

Prof.Paolo Ferragina

Contro-relatore

Prof. Alessio Conte

Anno Accademico 2019/2020

# Indice

1	Introduzione	3
1.1	Cos'è un graph database?	4
1.2	Basi di dato a grafo	5
1.3	Casi d'uso di un graph database	10
1.4	Obiettivo della tesi	14
2	Capitolo 2 : Neo4j	15
2.1	Cos'è neo4j?	15
2.2	Rappresentazione e ricerca dei dati in Neo4j	16
2.3	Query con Cypher	19
2.4	Pattern e relazioni in Cypher	22
2.5	Interrogazione e aggiornamento del grafo	23
3	Capitolo 3: Implementazione di un grafo in Neo4j	25
3.1	Creazione del grafo e delle relazioni	25
3.2	Funzione neighbor a livello k e Shortest path	27
3.3	Indegree e Outdegree di un grafo	33
3.4	BFS e DFS	35
4	Capitolo 4: Creazione e interrogazione di un big GDB	39
4.1	Dal file csv a grafo in Neo4j	39
4.2	Esecuzione algoritmi standard sul grafo	44
4.2.1	Grafi(reti) a invarianza di scala.	47
4.2.2	Distribuzione del grafo e Indegree/Outdegree	48
5	Conclusioni e ringraziamenti	52

# Introduzione

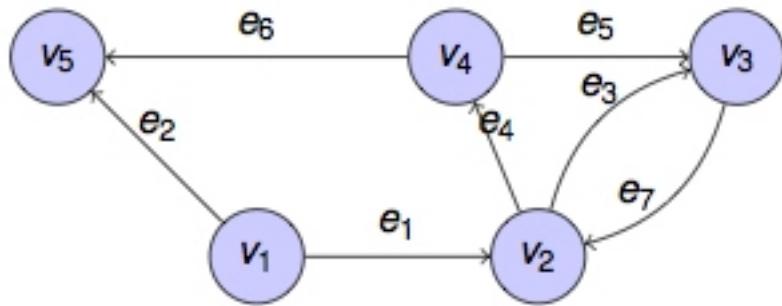
La seguente relazione ha lo scopo di illustrare come la scelta di un database a grafo di tipo NoSQL sia in certi casi la scelta migliore per la rappresentazione di dati e relazioni tra essi di come Neo4j sia uno strumento potente ed affidabile per la memorizzazione e indicizzazione di queste relazioni e per la realizzazione di interrogazioni complesse su di essi.

Si andrà ad illustrare brevemente in cosa consiste un database a grafo e quali siano i campi più interessanti in cui l'utilizzo di un GDB (Graph Database) risulti indispensabile: I social network come Twitter, Instagram e Facebook utilizzano database a grafo da centinaia di miliardi di nodi e archi. Successivamente si focalizzerà la attenzione su Neo4j, un software per basi di dati a grafo open source sviluppato interamente in Java. La scelta è caduta su Neo4j poiché a livello di efficienza, grafica e documentazione è uno dei migliori al mondo, inoltre utilizza Cypher, un linguaggio di query grafico dichiarativo che consente di eseguire interrogazioni molto espansive ed efficienti. Proseguendo si illustrerà brevemente su un grafo di prova le potenzialità di Neo4j e gli svariati algoritmi che si possono eseguire facilmente su di esso come Shortest path, BFS, DFS e molti altri. Come step finale si mostrerà come, anche con un semplice portatile e con Neo4j, sia possibile gestire un GDB da circa 9Gb di dati, con circa 180000 nodi e più di 265 milioni di archi. Si analizzeranno vari aspetti di quel grafo attraverso l'utilizzo di Neo4j browser, che fornisce una interfaccia grafica user friendly che permette di visualizzare i nodi e gli archi in modo molto efficace. L'obiettivo ultimo di questa relazione è anche quello di fornire un metodo robusto ed efficace per la gestione di un database a grafo di grandi dimensioni svelando quali possono essere le insidie in cui si può incorrere se non poniamo attenzione ad alcuni dettagli progettuali e realizzativi.

## 1.1 Cos'è un graph database?



I database a grafo sono un tipo di database in grado di indicizzare dati in forma di grafo. Un grafo  $G=(V,E)$  è una struttura matematica formata da un insieme di nodi ( $V$ ) e un insieme di archi ( $E$ ) che connettono coppie di nodi. Un database a grafo generalmente utilizza i nodi per memorizzare entità in grado di interagire tra loro instaurando un legame di qualsiasi tipologia (transazione bancaria, following, amicizia, legame giuridico...etc).



La rappresentazione dei dati mediante grafi offre un'alternativa al modello relazionale; i sistemi relazionali, pur essendo tutt'ora una risorsa sempre valida sono spesso economicamente molto costosi e richiedono le tipiche e onerose operazioni di unione (join) [1].

I database a grafo invece sono spesso più veloci di quelli relazionali nell'associazione di insiemi di dati, mappano in maniera più diretta le strutture presenti nelle applicazioni orientate agli oggetti, scalano più facilmente a grandi quantità di dati e non richiedono le operazioni di join; inoltre dipendono meno da un rigido schema entità-relazione e sono molto più adeguati a gestire dati mutevoli con schemi evolutivi [1].

### Modelli di riferimento

Quando si parla di modelli dei dati basati su grafo è inevitabile fare riferimento alla teoria dei grafi. In origine i modelli di riferimento per l'implementazione dei database a grafo erano due: il property graph model e il resource description framework graph (RDF). Il primo fa riferimento principalmente al progetto Tinkerpop, mentre il secondo è il modello di riferimento del Web semantico. I database a grafo che utilizzano il modello RDF sono anche noti come Triple Store, Quad Store, o RDF Store. I due modelli non sono del tutto coincidenti, anche se solitamente il passaggio da uno all'altro è molto intuitivo. Per entrambi esistono dei linguaggi di interrogazione specifici, ma solo per RDF esiste uno standard riconosciuto in SPARQL. Successivamente ne sono stati sviluppati altri come i seguenti :

- Modello dei dati di grafo basico: grafo diretto con nodi e archi etichettati da qualche vocabolario.
- Modello dei dati a ipernodo: Si basa sulla generalizzazione di grafo con ipernodi e iperarchi, permettendo la creazione di oggetti complessi, dipendenze funzionali e eredità strutturale multipla.
- Modello dei dati a ipernodo con grafi annidati: modello in cui ad un ipernodo può essere esso stesso un grafo .
- Modello dei dati RDF: modello raccomandato dal W3C per rappresentare metadati .
- Modello dei dati del grafo di Proprietà: modello di multigrafo diretto, etichettato, con attributi (proprietà) e con archi [1].

## 1.2 Basi di dato a grafo

I sistemi di gestione delle basi di dato a grafo (Graph Database Management System) si classificano in due categorie: Basi di dato a grafo, e framework elaboratori di grafi (graph processing framework). I primi hanno l'obiettivo di gestire in maniera persistente il dato permettendo di archiviare transazionalmente e di accedervi in maniera persistente i secondi invece forniscono processi di batch e analisi su grandi grafi spesso in un ambiente distribuito con molte macchine [16].

Tra i due si andrà ad approfondire le basi di dato a grafo in quanto maggiormente adatte al tipo di obiettivo prefissato, per il quale serve che la gestione dei dati sia persistente.

Di seguito vari esempi di basi di dato a grafo native:



1. AllegroGraph: Resource Description Framework (RDF) e database a grafo . AllegroGraph è un triplestore closed source progettato per memorizzare tripli RDF, un formato standard per Linked Data. Funziona anche come archivio di documenti progettato per archiviare, recuperare e gestire informazioni orientate ai documenti, in formato JSON-LD. AllegroGraph è attualmente utilizzato in progetti commerciali [e in un progetto del Dipartimento della Difesa degli Stati Uniti. È anche il componente di archiviazione per il progetto TwitLogic che sta portando il Web semantico ai dati di Twitter [10].



2. InfiniteGraph: Distribuito e abilitato per il cloud Database grafico, InfiniteGraph è un database a grafo implementato in Java e fa parte di una classe di tecnologie di database NOSQL ("Not Only SQL") che si concentrano sulle strutture di dati a grafo. Sviluppato da Objectivity, Inc., è in fase di migrazione in ThingSpan. Gli sviluppatori utilizzano InfiniteGraph per trovare relazioni utili e spesso nascoste in set di big data altamente connessi. InfiniteGraph è multipiattaforma, scalabile, abilitato per il cloud ed è progettato per gestire un throughput molto elevato. InfiniteGraph è adatto per applicazioni e servizi che risolvono problemi con grafici o rispondono a domande come "Come sono connesso a Kevin Bacon?" oppure "Quali sono i voli di andata e ritorno più economici dalla California a New York con non più di 2

scali, almeno 30 minuti tra i voli e che partono alle 8:00 di martedì e tornano entro le 18:00 di venerdì?" [11].



The #1 Database for Connected Data

3. Neo4j: La mia scelta per numerose caratteristiche favorevoli che andrò ad approfondire nel secondo capitolo Open source, supporta ACID, ossia garantisce le 4 proprietà per la gestione dei dati di availability, consistency, isolation e durability, ha un clustering ad alta disponibilità per le distribuzioni aziendali e viene fornito con un'amministrazione basata sul Web che include il supporto completo delle transazioni e un esploratore grafico di collegamenti dei nodi con un interfaccia user friendly; accessibile dalla maggior parte dei linguaggi di programmazione utilizzando la sua interfaccia API Web REST incorporata e un protocollo Bolt, un browser Web per smartphone in grado di eseguire applicazioni Java ME, proprietario con driver ufficiali [3].



4. OrientDB: OrientDB è un visualizzatore ed editor di grafi scritto in Java, ma in cui l'interfaccia si basa su comandi utente in SQL. Nato da una software house italiana viene rilasciato gratuitamente con licenza Apache 2.0. Orient nacque dall'esigenza di avere uno strumento semplice e prestante per rendere persistenti gli oggetti applicativi [12].

È un database documentale in cui le relazioni sono gestite principalmente come in un database a grafo con connessioni dirette tra i singoli dati. OrientDB supporta modalità senza schema, con schema oppure miste. La cosa notevole rispetto agli altri programmi NoSQL è che utilizza comandi SQL fin dove possibile,

a cui aggiunge comandi NoSQL solo dove i corrispondenti SQL non esistono già. Ha HTTP REST, ossia il protocollo di comunicazione basato su http e sui principi rest in cui, per utilizzare le risorse le componenti di una rete (componenti client e server) comunicano attraverso una interfaccia standard (HTTP) per scambiare rappresentazioni di queste risorse.



5. Microsoft Azure Cosmos DB : Il database cloud Azure Cosmos DB è un progetto ambizioso. Ha lo scopo di emulare più tipi di database come tabelle convenzionali, documenti, column family e grafo, il tutto attraverso un servizio unificato con un insieme coerente di API.

Un database a grafo è quindi solo una delle varie modalità in cui può operare Cosmos DB, che utilizza il linguaggio di query di Gremlin e API per le query di tipo grafo, oltre a supportare la console Gremlin creata per Apache TinkerPop [13.]

Un altro grande punto di forza di Cosmos DB è che l'indicizzazione, il ridimensionamento e la geo-replicazione vengono gestiti automaticamente nel cloud di Azure, senza alcuna manipolazione da parte dell'utente. Non è ancora chiaro come l'architettura all-in-one di Microsoft si adatti ai database a grafo nativi in termini di prestazioni, ma Cosmos DB offre sicuramente un'utile combinazione di flessibilità e scalabilità.



Amazon Neptune è un servizio di database a grafo rapido, affidabile e completamente gestito che semplifica la creazione e l'esecuzione di applicazioni che funzionano con set di dati altamente connessi. Il centro nevralgico di Amazon Neptune è un motore di database a grafo ad alte prestazioni appositamente

ottimizzato per archiviare miliardi di relazioni ed eseguire query al grafo con una latenza di pochi millisecondi. Amazon Neptune supporta i modelli di grafi più diffusi Property Graph e RDF di W3C, con i relativi linguaggi di query Apache TinkerPop Gremlin e SPARQL, permettendoti di creare in modo semplice query efficaci su dataset altamente connessi. Neptune consente i casi d'uso dei grafi come motori di raccomandazioni, rilevamento di attività fraudolente, grafi della conoscenza, scoperte di farmaci e sicurezza delle reti.

Amazon Neptune è altamente disponibile, con repliche di lettura, ripristino point-in-time, backup continuo su Amazon S3 e replica nelle zone di disponibilità. Neptune è sicuro, con supporto per connessioni client HTTPS criptate e crittografia dei dati inattivi. Neptune è una soluzione completamente gestita, pertanto non dovrai più preoccuparti di attività di gestione del database come provisioning dell'hardware, applicazioni di patch del software, impostazione, configurazione o backup [14].



7. janusGraph: è un fork nato dal progetto TitanDB e ora è sotto il controllo della Linux Foundation. Utilizza uno dei numerosi back-end supportati (Apache Cassandra, Apache HBase, Google Cloud Bigtable, Oracle BerkeleyDB) per archiviare i dati del grafo, supporta il linguaggio di query Gremlin (così come altri elementi dallo stack di Apache TinkerPop) e può anche incorporare la ricerca full-text tramite i progetti Apache Solr, Apache Lucene o Elasticsearch. IBM, uno dei sostenitori del progetto JanusGraph, offre una versione di JanusGraph su IBM Cloud chiamata Compose per JanusGraph. Come Azure Cosmos DB, Compose per JanusGraph offre scalabilità automatica e disponibilità elevata, con prezzi basati sull'utilizzo delle risorse [15].

## 1.3 Casi d'uso di un graph database

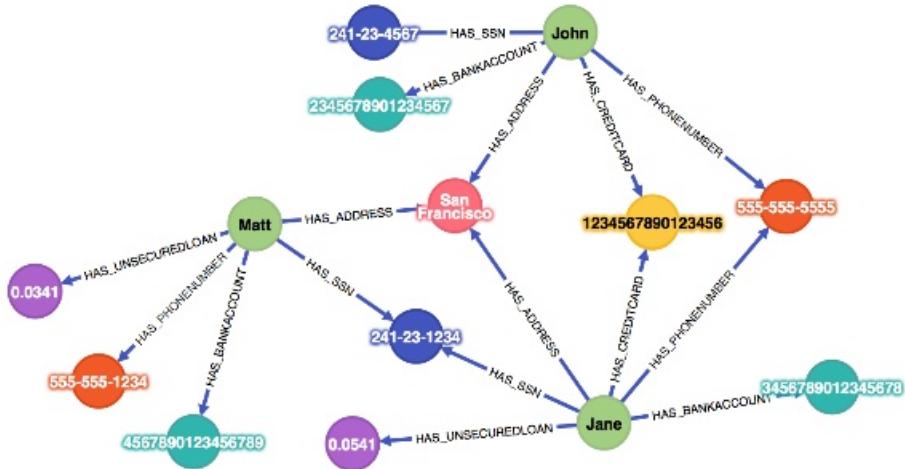
I casi d'uso per i database a grafo sono innumerevoli ecco i principali:



### Social network

Ormai nel nostro quotidiano l'utilizzo delle icone nella figura sopra è diventato ormai parte fondamentale della nostra vita sia lavorativa sia sociale. Meno intuitivo è per noi comprendere come tali aziende siano riuscite a gestire milioni/miliardi di utenti che attraverso queste piattaforme si scambiano miliardi di contenuti musicali, foto, messaggi ogni giorno. In tutto questo aziende come Facebook, Instagram, Twitter e molte altre si sono affidate ad una gestione dei dati attraverso un database a grafo. Un database a grafo è infatti in grado di elaborare in modo rapido e semplice moltissimi profili e interazioni degli utenti per costruire applicazioni di social network. Linguaggi di interrogazione appositamente studiati e standard come SPARQL, consentono query al grafo altamente interattive e con un elevato throughput. Ad esempio, se si costruisce un feed social nella propria applicazione, si può utilizzare una query sul database a grafo per fornire risultati che assegnino priorità, mostrando ai tuoi utenti gli ultimi aggiornamenti dei loro familiari, amici che mettono "mi piace" e amici che vivono vicino a loro, oppure suggerimenti di amicizia in base alla vicinanza tra due nodi (Es: Alberto conosce Brando, Brando conosce Carlo, suggerimento di amicizia: Alberto-Carlo) [3].

## Fraud Detection



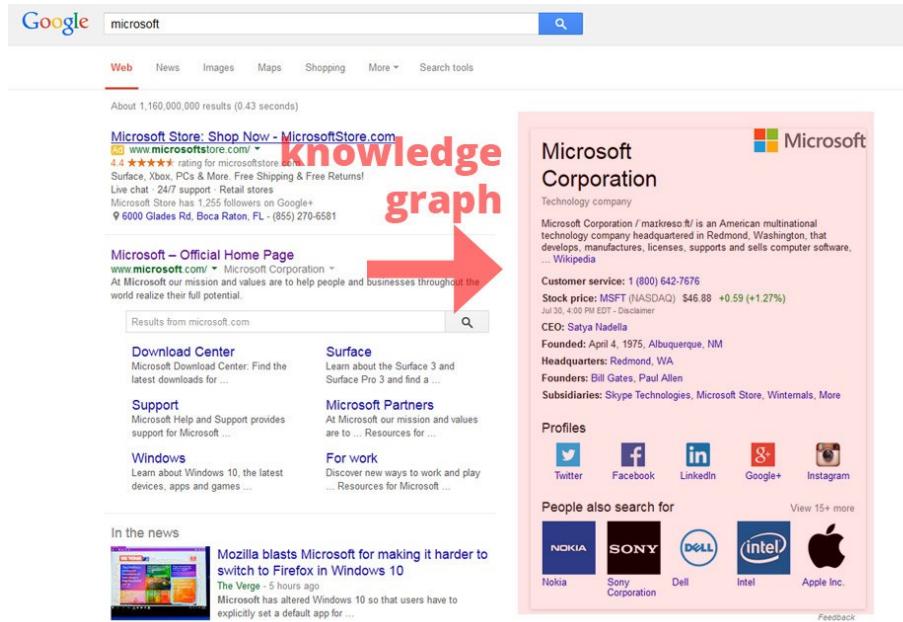
- Source: Neo4j graph gist



### Fraud detection

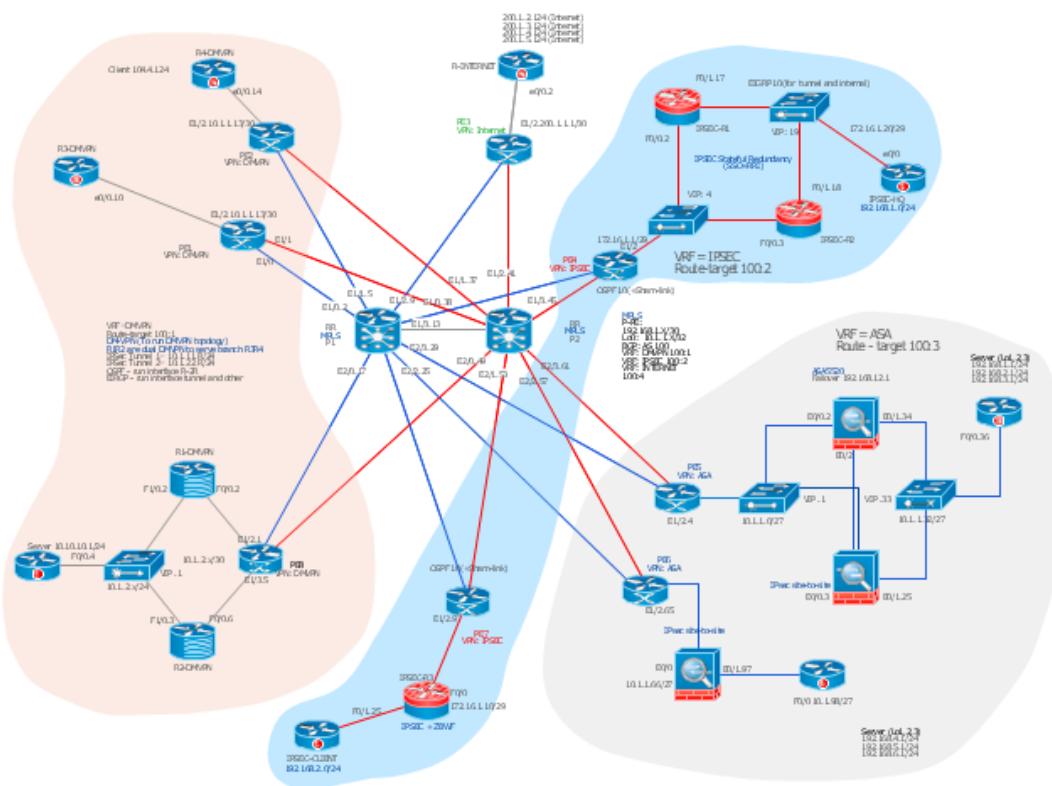
La rappresentazione mediante un GB offre notevoli vantaggi in ambito di frodi finanziarie. Permette di lavorare in un sottografo in cui magari alcuni nodi risultano sospetti e quindi di isolarli e cercare legami con altri nodi vicini. Si può inoltre utilizzare le relazioni per elaborare transazioni finanziarie e di acquisto praticamente in tempo reale, per rilevare con facilità gli schemi di attività fraudolente. Un database a grafo consente di rilevare che un potenziale acquirente utilizza lo stesso indirizzo e-mail e carta di credito di un noto caso di frode. Se si sta creando un'applicazione di rilevamento di attività fraudolente nel settore retail, si possono costruire query al grafo per rilevare con facilità casi di più persone associate a stesso indirizzo e-mail personale o più persone che condividono lo stesso indirizzo IP ma risiedono in indirizzi fisici differenti [3].

## Google Knowledge graph



Le applicazioni di grafi della conoscenza sono tra le più importanti applicazioni oggi esistenti. Il 16 maggio 2012 Google lancia il Knowledge graph. Nella versione italiana (google.it) la funzionalità è stata attivata il 4 dicembre 2012. Il Knowledge Graph è il primo passo verso una ricerca semantica: grazie a questa funzione, il motore di ricerca di Google associa alle parole cercate un oggetto e mette in relazione altri oggetti in modo da avere una ricerca più veloce e accurata. Il risultato è un riquadro che sintetizza le informazioni principali sull'oggetto della query, velocizzando dunque la ricerca dell'utente e soddisfacendo il search intent. Utilizzando un grafo della conoscenza, si possono dunque aggiungere informazioni a cataloghi di prodotti, costruire ed eseguire query complesse o estrarre modelli di informazioni generali, come Wikidata [3].

## IT operations e network security



Un altro utilizzo molto comune di un database a grafo per archiviare la topologia della tua rete e utilizzare query al grafo per rispondere a domande come quanti host eseguono una specifica applicazione. Il database a grafo è infatti in grado di archiviare ed elaborare miliardi di eventi per gestire e proteggere la tua rete. Se rilevi un evento anomalo, il database a grafo per comprendere rapidamente come può influire sulla tua rete, eseguendo query per uno schema a grafo utilizzando gli attributi dell'evento. Puoi eseguire query per individuare altri host o dispositivi potenzialmente compromessi. Ad esempio, se rilevi un file maligno su un host, il database a grafo può aiutarti a individuare i collegamenti tra gli host che diffondono il file maligno e consentirti di tracciarlo fino all'host originario che lo ha scaricato. Anche in fase di progettazione la rappresentazione a grafo spesso evidenzia limiti di scalabilità della nostra applicazione [3].

## **1.4 Obiettivo della tesi**

Dopo aver esaminato esaurientemente cosa sia un graph database, quali siano le sue principali applicazioni e quali siano i principali GDB ad oggi in commercio, andrò a porre la mia attenzione su Neo4J. Esso verrà descritto in modo più approfondito nel capitolo 2, per poi considerare nel capitolo 3 uno studio approfondito di un grafo complesso contenente movimenti finanziari tra vari clienti. Illustrerò un metodo che, partendo da un file .csv di grandi dimensioni, riesce con Neo4j a creare un grafo in grado di rispondere a interrogazioni complesse. Cercherò di soffermarmi sulla gestione della memoria e sulla creazione di indici di ricerca su particolari proprietà dei nodi. Infine, attraverso l'uso di grafici excel, cercherò di evidenziare alcune specificità del grafo in questione, soffermandomi anche sulle prestazioni di ogni ricerca e sulle particolari configurazioni dei nodi nel grafo. Evidenzierò attraverso l'utilizzo di grafici di tipo Scatter e Bubble l'andamento dell'in-degree e dell'out-degree, l'andamento delle componenti connesse, la centralità degli utenti, e i nodi (utenti) con maggior importanza a livello di interconnessioni. Tutto questo utilizzando Neo4j browser e qualche accorgimento che illustrerò nei prossimi capitoli.

# Capitolo 2

## Neo4j

Questo capitolo si propone di offrire una panoramica di Neo4j, come ambiente di lavoro reale preso in considerazione, e come strumento efficiente ed affidabile per il nostro scopo.

### 2.1 Cos'è Neo4j?

Neo4j è un software per basi di dati a grafo open source sviluppato interamente in Java. È un database totalmente transazionale, che viene integrato nelle applicazioni permettendone il funzionamento stand alone e memorizza tutti i dati in una cartella. È stato sviluppato nel 2007 dalla Neo Technology, una startup di Malmö, Svezia e della San Francisco Bay Area [1].

È dotato di:

- Transazioni ACID, ossia che garantiscono le 4 proprietà Availability, Consistency, isolation, durability
- High Availability,
- può memorizzare miliardi di nodi e relazioni,
- alta velocità di interrogazione tramite attraversamenti del grafo,
- linguaggio di interrogazione dichiarativo e grafico.

È un DBMS schema-less, ciò significa che i suoi dati non devono attenersi ad alcuna struttura di riferimento prefissata, inoltre non possiede una politica di accesso controllata [3].

### Modalità d'uso

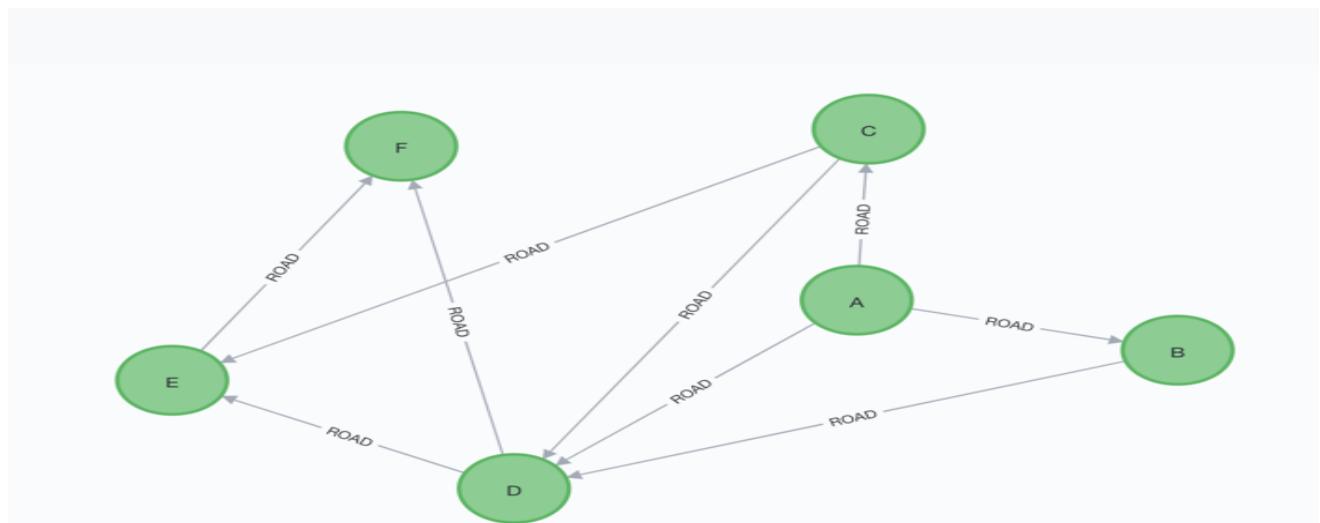
Il database può essere usato sia in modalità embedded che su server. Nella modalità embedded si incorpora il database nell'applicazione (con maven o includendo i file JAR) e questo viene eseguito all'interno della JVM, quindi nello stesso processo ma accettando vari thread concorrenti. Nella modalità server invece il database è un processo a sé stante a cui si accede tramite REST facendo delle query e ricevendo i dati in remoto; il server permette l'utilizzo di plugin che filtrano i dati in input e output e offrono servizi aggiuntivi, per esempio il supporto alle query spaziali. Neo4j permette la modalità batch, non concorrente, per l'importazione massiva di dati da altri database o da file, ma per l'uso comune si basa sulle transazioni. Una volta aperta una transazione è possibile creare nodi e assegnarvi delle

proprietà, ossia dei valori corrispondenti ai tipi di dato elementari di Java (più le String e gli array) identificati grazie a un nome e unire dei nodi tramite le relazioni, i cui tipi sono definiti dal programmatore, che possono essere direzionali o meno. Anche le relazioni possono avere delle proprietà come i nodi. Il grafo è quindi schema-less, il che da un lato permette di definire dati molto eterogenei con il minimo sforzo e dall'altro può creare problemi di consistenza dei dati, che è interamente responsabilità dell'applicazione.

### Indicizzazione

Neo4j integra un servizio di indicizzazione basato su Lucene che permette di memorizzare nodi facendo riferimento a una etichetta assegnata arbitrariamente, per poi accedere all'iteratore dei nodi con una certa etichetta recante un certo valore. È possibile anche effettuare le ricerche fulltext, una funzione centrale di Lucene, e esistono dei plugin del server che permettono di indicizzare automaticamente i nodi. È inoltre disponibile un servizio di indicizzazione basato sui timestamp che permette di ottenere i nodi corrispondenti a una ora e una data comprese in un certo intervallo [1].

## 2.2 Rappresentazione e ricerca dei dati in Neo4j



Neo4j basa l'intera struttura dati su due entità fondamentali: i nodi e le relazioni. I nodi sono appunto entità in grado di rappresentare molteplici oggetti, carte di credito, clienti, case, attori, macchine, a cui possono essere assegnate delle proprietà, come ad esempio un nodo di tipo macchina ha come proprietà la targa. Nel nostro esempio i nodi rappresentano dei luoghi, e relazioni invece rappresentano strade mediante una freccia direzionale. Vi è comunque la possibilità di percorrere le

relazioni in modo bidirezionale, permettendo in fase di ricerca di muoversi in entrambi i sensi come se il grafo fosse non orientato. Nel nostro caso il collegamento è di tipo road, e viene definito in fase di creazione della relazione. I legami fra due nodi possono essere molteplici e di svariato tipo; è inoltre possibile assegnare una proprietà anche al tipo di relazione. In questo caso la relazione di tipo road che collega A e C potrebbe avere una proprietà di tipo string che indica il nome della via.

### **Property graph model**

Un Property graph (o grafo di proprietà) è un tipo di modello di grafo in cui le relazioni non sono solo connessioni, ma portano anche un nome (tipo) e alcune proprietà. Un grafo delle proprietà eccelle nel mostrare le connessioni tra i dati sparsi in diverse architetture di dati e schemi di dati. Forniscono una visione più ricca di come i dati possono essere modellati su molti database diversi e di come si relazionano i diversi tipi di metadati. I grafi delle proprietà mostrano anche le dipendenze dei dati non visualizzate in uno schema di database relazionale o tramite altri strumenti [17].

In sostanza possiamo riassumere un Property graph attraverso le seguenti caratteristiche:

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

### **Vantaggi**

La struttura a grafo di Neo4j è estremamente comoda ed efficiente nel trattare strutture come gli alberi estratte ad esempio da file XML, filesystem e reti, che ovviamente vengono rappresentate con naturalezza da un grafo poiché sono esse stesse dei grafi. L'esplorazione di queste strutture risulta in genere più veloce rispetto a un database a tabelle perché la ricerca di nodi in relazione con un certo nodo è un'operazione primitiva e non richiede più passaggi, in genere impliciti in un join di SQL, su tabelle diverse. Ogni nodo contiene l'indice delle relazioni entranti e uscenti da esso, quindi la velocità di attraversamento del grafo non risente delle dimensioni complessive ma solo della densità dei nodi attraversati. Rilevante è anche il fatto che esistono delle implementazioni già pronte per le operazioni più comuni sui grafi, come la ricerca del cammino minimo tra due nodi tramite l'algoritmo di Dijkstra, la ricerca di cicli e il calcolo del diametro della rete.

## Neo4j: ricercare i dati

Una volta definiti i dati in termini di nodi e relazioni, arriva il momento di effettuare le ricerche su di essi. In Neo4j esistono diverse strategie per poterlo fare: le API Core Java e Traversal e le query con Cypher. Vediamo le prime due brevemente di seguito, andando poi ad approfondire Chyper [5].

### API Core Java

L'API Core Java a basso livello consente di attraversare le relazioni grazie alla programmazione "manuale". Con questa API, un programma Java potrebbe partire da un nodo, percorrere tutte le sue relazioni, esaminarne il tipo e poi recuperare i nodi all'altro capo di tali relazioni. Si tratta di una modalità molto adatta alla mentalità dei programmatore Java, al loro modo di interpretare i nodi e le relazioni. Il problema sta nel fatto che è richiesto di implementare ciascun passaggio dell'attraveramento dei nodi e ciò porta a dover scrivere metodi di ricerca piuttosto corposi [5].

### Traversal

Traversal è una API di livello più alto, che i programmatore Java possono impiegare per la ricerca sui dati. Facendo riferimento all'esempio di ricerca di cui parlavamo sopra, con la API Traversal, la si scriverebbe così:

```
TraversalDescription traversalDescription =
TraversalDescription()
.relationships( "IS_FRIEND_OF", Direction.OUTGOING )
.evaluator( Evaluators.atDepth( 2 ) )
.uniqueness(Uniqueness.NODE_GLOBAL );
Iterable nodes =
traversalDescription.traverse(nodeById).nodes();
```

Il programma definisce un oggetto TraversalDescription che dice a Neo4j il modo in cui costruire il suo insieme di risultati e poi gli passa il nodo da cui partire e gli richiede un'interfaccia iterable che legga i suoi nodi [5].

Dopo aver brevemente illustrato le prime due andrò a descrivere il linguaggio di programmazione dichiarativo Chyper, il cui stile ricorda MySql ed è stato importantissimo per la mia esperienza con

Neo4j. È un linguaggio semplice ma potente e ha una sintassi molto elegante che sin da subito mi ha aiutato nella fase di implementazione del database a grafo.

## 2.3 Query con Cypher

Cypher è un linguaggio di query su grafo dichiarativo che consente di eseguire query sui dati espansive ed efficienti in un grafo di proprietà. Cypher è stato in gran parte un'invenzione di Andrés Taylor mentre lavorava per Neo4j, Inc. (ex Neo Technology) nel 2011. Cypher era originariamente concepito per essere utilizzato con il database grafico Neo4j , ma è stato reso open tramite il progetto openCypher nell'ottobre 2015. Il linguaggio è stato progettato pensando alla potenza e alle capacità di SQL (linguaggio di query standard per il modello di database relazionale ), ma Cypher poi si è evoluto sui componenti e sulle esigenze di un database costruito sui concetti della teoria dei grafi [1].

### Modello grafico

Cypher si basa sul Property Graph Model, il quale organizza i dati in nodi e relazioni, e ne permette la visualizzazione tramite un interfaccia user friendly e allo stesso tempo chiara e sintetica. Permette la visualizzazione grafica di nodi potendoci interagire col cursore, potendo mostrare i nodi uscenti e entranti dal nodo, oltre che alle proprietà attribuitegli in fase di creazione del nodo. Permette inoltre di visualizzare sottografi, nodi singoli, pattern che collegano due nodi, attributi delle relazioni e proprietà delle stesse. I nodi inoltre possono essere contrassegnati con zero o più etichette (come tag o categorie), che rappresentano i loro diversi ruoli in un dominio [9]. Le etichette possono raggruppare nodi simili assegnando zero o più etichette di nodo. Le etichette sono una specie di tag e consentono di specificare determinati tipi di entità da cercare o creare. Le proprietà sono coppie chiave-valore: dove la chiave è una stringa e il valore è selezionato dal sistema di tipi Cypher. Le query Cypher sono assemblate con modelli di nodi e relazioni con qualsiasi filtro specificato su etichette e proprietà per creare, leggere, aggiornare ed eliminare i dati trovati nel modello specificato.

### Sintassi

Il linguaggio di query Cypher descrive modelli di nodi e relazioni e filtra tali modelli in base a etichette e proprietà. La sintassi di Cypher è basata sull'arte ASCII , che è arte visiva basata su testo per computer, introdotta per la prima volta dall'IBM nel 1979. Ciò rende il linguaggio molto visivo e di facile lettura perché rappresenta visivamente e strutturalmente i dati specificati nella query. Ad esempio, i nodi sono rappresentati con parentesi attorno agli attributi e alle informazioni riguardanti

l'entità. Le relazioni sono rappresentate con una freccia (diretta o non orientata) con il tipo di relazione tra parentesi.

```
//node (variable:Label {propertyKey: 'PropertyValue'})  
//relationship -[variable:RELATIONSHIP_TYPE]->  
//Cypher pattern (node1:LabelA)-[rel1:RELATIONSHIP_TYPE]->(node2:LabelB)
```

### Key words

Cypher contiene una varietà di parole chiave per specificare modelli, filtrare modelli e restituire risultati. Tra quelli più comuni ci sono: MATCH, WHERE e RETURN. Questi funzionano in modo leggermente diverso rispetto a SELECT e WHERE in SQL ; tuttavia, hanno scopi simili. MATCH viene utilizzato prima di descrivere il modello di ricerca per trovare nodi, relazioni o combinazioni di nodi e relazioni insieme. WHERE in Cypher viene utilizzato per aggiungere ulteriori vincoli ai pattern e filtrare eventuali pattern indesiderati. RETURN di Cypher formatta e organizza i risultati. Proprio come con altri linguaggi di query, si possono restituire i risultati con proprietà, elenchi, ordinamenti e altro ancora. Utilizzando le parole chiave con la sintassi del modello mostrata sopra, la query di esempio sotto cercherà il modello del nodo (etichetta dell'attore e proprietà chiamata nome con valore di 'Nicole Kidman') connesso da una relazione (tipo ACTED\_IN e direzione di uscita dal primo nodo) a un altro nodo (etichetta del film). La clausola WHERE quindi filtra per mantenere solo i modelli in cui il nodo Movie nella clausola match ha una proprietà year inferiore al valore del parametro passato. Nell'output, la query specifica di restituire i nodi del film che si adattano al modello e filtraggio dalla corrispondenza e clausole where [4].

```
neo4j$ MATCH (nicole:Actor {name: 'Nicole Kidman'})-[:ACTED_IN]→(movie:Movie)  
WHERE movie.year < $yearParameter  
RETURN movie Cypher
```

Cypher contiene anche parole chiave per specificare clausole per la scrittura, l'aggiornamento e l'eliminazione dei dati. CREATE e DELETE vengono utilizzati per creare ed eliminare nodi e relazioni. SET e REMOVE vengono utilizzati rispettivamente per impostare o rimuovere i valori sulle proprietà e aggiungere o togliere etichette sui nodi. MERGE viene utilizzato per creare nodi in modo univoco senza duplicati. I nodi possono essere eliminati solo quando non hanno altre relazioni ancora esistenti. Per esempio:

```
neo4j$ MATCH (startContent:Content)-[relationship:IS RELATED_TO]→(endContent:Content)
WHERE endContent.source = 'user' OPTIONAL MATCH (endContent)-[r]-()
DELETE relationship, endContent
```

In Cypher, un'istruzione si compone generalmente di tre parti, una di interrogazione, opzionalmente uno o più comandi, e un'istruzione di ritorno per indicare eventuali valori da restituire. Come vedremo in seguito, più istruzioni possono essere concatenate e si possono usare valori precedentemente letti dal database. Questo significa che si possono scrivere query come la seguente, in cui abbiamo l'interrogazione, la ricerca di alcuni nodi, a seguire la creazione di una nuova relazione con un nuovo nodo e infine la restituzione dei nodi trovati e creati.

```
1 MATCH (u)-[i:ISCRITTO_A]→(:Corso {nome: 'Corso di Neo4j'})
2 WHERE i.scadenza < '2017'
3 CREATE (u)-[:LEGGE]→(l:Libro { titolo: 'Neo4j Cookbook' })
4 RETURN u,l
5
```

Andiamo nel dettaglio della query precedente. Con la prima riga abbiamo cercato tutti i nodi che abbiamo collegato alla variabile u, aventi una relazione di tipo ISCRITTO\_A che abbiamo collegato alla variabile i, con un qualsiasi nodo avente la proprietà nome uguale al valore “Corso di Neo4j”. In questa prima parte della query vengono selezionate quelle porzioni di grafo che soddisfano il pattern appena specificato (pattern matching). Con la seconda riga abbiamo filtrato i risultati selezionando solo quelli in cui la proprietà scadenza è specificata ed è minore di 2017, come in SQL. Nella terza riga, invece, abbiamo creato, per ogni risultato trovato, una relazione di tipo LEGGE tra il nodo u e un nuovo nodo avente etichette “libro” e proprietà “titolo” uguale al testo “Neo4j Cookbook”. Infine, con `<code>RETURN</code>` abbiamo indicato di restituire i nodi u e l. Alcune annotazioni:

se la prima istruzione non trova alcun nodo, non ci sarà alcuna creazione e quindi nessuna riga restituita; per ogni risultato trovato verrà creato un nuovo nodo con etichetta Libro. Più avanti vedremo come è possibile creare pattern assicurandosi che siano unici.

Il risultato della query è il seguente:

u	1
{username: verdi@html.it} {titolo: Neo4j Cookbook}	

```
| {username: rossi@html.it}|{titolo: Neo4j Cookbook} |
```

Vediamo appunto che abbiamo nella prima colonna (u) il nodo trovato sul database, mentre nella seconda (l) il nodo creato [4].

## 2.4 Pattern e relazioni in Chyper

### I pattern

Abbiamo visto che per valorizzare alcune variabili che rappresentano ciò che stiamo cercando nel database, dobbiamo specificare una parte di grafo da creare (CREATE) o da far corrispondere (MATCH).

La sintassi usata da Cypher per rappresentare questi pattern è molto semplice. Vediamone alcuni esempi.

#### Nodi

Tutto ciò che è tra parentesi tonde è un nodo. Detto ciò, è possibile esprimere varie condizioni; ad esempio:

- `1()` fa match con qualsiasi nodo;
- `(:User)` corrisponde a qualsiasi nodo con etichetta User. Per convenzione, le etichette hanno la sola iniziale maiuscola;
- `{ titolo: 'Neo4j Cookbook' , pagine: 210 }` corrisponde a qualsiasi nodo avente le proprietà titolo e pagine entrambe con i valori indicati. Per convenzione generalmente le proprietà sono tutte in minuscolo;
- il pattern che specifica sia l'etichetta sia le proprietà: `(:Libro { titolo: 'Neo4j Cookbook' , pagine: 210 })` corrisponde ai nodi che rispettano tutte le condizioni: sia l'etichetta sia le proprietà valorizzate con i valori indicati;
- per indicare che vogliamo collegare il valore di una variabile ad ogni nodo che corrisponde al pattern, dobbiamo specificare il nome della variabile subito dopo l'apertura della parentesi: `(cookbook:Libro { titolo: 'Neo4j Cookbook' , pagine: 210 })`. In questo caso abbiamo dato il nome cookbook ai nodi trovati. Le variabili sono comode successivamente per referenziare gli elementi corrispondenti, come abbiamo visto in precedenza nella clausola WHERE.

## Relazioni

Tutto ciò che è tra parentesi quadre è una relazione, o una catena di relazioni. Ad esempio:

- -[]- fa match con qualsiasi relazione, e qualsiasi orientamento dell'arco;
- -[]-> fa match con qualsiasi relazione, e verso dal nodo specificato a sinistra a quello di destra. Ovviamente la sintassi <[]- indica il verso opposto;
- -[i:ISCRITTO\_A] corrisponde a qualsiasi relazione di tipo ISCRITTO\_A. Per convenzione le etichette hanno la sola iniziale maiuscola;
- -[{ scadenza: '2017-11-01'}]-> corrisponde a qualsiasi relazione avente la proprietà scadenza con il valore indicato. Anche qui per convenzione le proprietà sono in minuscolo;
- il pattern che specifica sia il tipo sia le proprietà corrisponde a relazioni che rispettano tutte le condizioni, come abbiamo visto per i nodi. Per esempio: [:ISCRITTO\_A { scadenza: '2017-11-01'}];
- per indicare che vogliamo collegare il valore di una variabile ad ogni relazione che corrisponde al pattern, anche qui dobbiamo specificare il nome della variabile subito dopo l'apertura della parentesi: -[i:ISCRITTO\_A]->;
- se vogliamo indicare che tra due nodi ci può essere una catena con al massimo 2 relazioni, possiamo usare la sintassi -[\*1..2]->. Naturalmente, possiamo combinare il pattern con quelli usati in precedenza e scrivere pattern fatti così: -[:CONOSCE\*2..]- per indicare che vogliamo catene di relazioni di tipo CONOSCE con almeno 2 relazioni [4].

## 2.5 Interrogazione e aggiornamento del grafo

Cypher può essere utilizzato sia per l'interrogazione che per l'aggiornamento di un grafo.

Andrò a elencare la struttura della query in Chyper con i suoi vantaggi e le sue limitazioni.

Una parte di query Cypher non può abbinare e aggiornare il grafo allo stesso tempo. Ogni parte può leggere e abbinare sul grafo o apportarvi aggiornamenti. Se si legge dal grafo e poi si aggiorna il grafo, la query ha implicitamente due parti: la lettura è la prima parte e la scrittura è la seconda parte. Se la query esegue solo letture, Cypher sarà "pigro" e non genererà effettivamente al modello finché non si richiederanno i risultati. In una query di aggiornamento, la semantica è che tutta la lettura verrà eseguita prima che avvenga effettivamente qualsiasi scrittura.

L'unico modello in cui le parti della query sono implicite è quando si legge e poi si scrive. Le parti vengono separate utilizzando l'istruzione WITH che è come un orizzonte degli eventi: è una barriera tra un piano e l'esecuzione finale di quel piano. Quando si desidera filtrare utilizzando dati aggregati è necessario concatenare due parti di query di lettura: la prima esegue l'aggregazione e la seconda filtra i risultati provenienti dalla prima.

```
MATCH (n {name: 'John'}) - [:FRIEND] - (amico) WITH n, count(friend) AS friendsCount  
WHERE friendsCount > 3  
RETURN n, friendsCount
```

Usando WITH, specifichi come desideri che avvenga l'aggregazione e che l'aggregazione deve essere completata prima che Cypher possa iniziare a filtrare.

Ecco un esempio di aggiornamento del grafico, scrivendo i dati aggregati nel grafico:

```
MATCH (n {name: 'John'}) - [:FRIEND] - (amico) WITH n, count(friend) AS friendsCount  
SET n.friendsCount = friendsCount  
RETURN n.friendsCount
```

È possibile concatenare il numero di parti di query consentito dalla memoria disponibile [4].

### **Restituzione dei dati**

Qualsiasi query può restituire dati. Se la tua query si limita a leggere, deve restituire dati: non serve a nulla se non lo fa e non è una query Cypher valida. Le query che aggiornano il grafico non devono restituire nulla, ma possono. Dopo tutte le parti della query arriva una clausola RETURN finale. RETURN non fa parte di alcuna parte della query: è un simbolo di punto alla fine di una query. La clausola RETURN ha tre sottoclausole che la accompagnano: SKIP / LIMIT e ORDER BY. Se restituisci elementi grafici da una query che li ha appena cancellati, stai tenendo un puntatore che non è più valido. Le operazioni su quel nodo non sono definite.

# Capitolo 3

## Implementazione di un grafo in Neo4j

Dopo aver illustrato le caratteristiche generali di Neo4j e aver introdotto il linguaggio di query Cypher, utilizzato da Neo4j per le interrogazioni al grafo, andrò a realizzare un grafo di prova sul quale eseguirò alcuni algoritmi di ricerca interessante; che poi utilizzerò in seguito sul grafo di elevate dimensioni contenente movimenti finanziari.

### 3.1 Creazione del grafo e delle relazioni

La Query per la creazione del grafo ha 28 righe di codice elementare in cui nella prima parte creo i 13 nodi con il label Loc e come proprietà la lettera dell'alfabeto corrispondente (es: primo nodo-lettera A, secondo nodo B etc). Nelle righe successive vado a creare 19 relazioni tra i nodi che daranno vita al grafo mostrato nella terza figura.

```
1 CREATE (a:Loc {name: 'A'}),  
2   (b:Loc {name: 'B'}),  
3   (c:Loc {name: 'C'}),  
4   (d:Loc {name: 'D'}),  
5   (e:Loc {name: 'E'}),  
6   (f:Loc {name: 'F'}),  
7   (g:Loc {name: 'G'}),  
8   (h:Loc {name: 'H'}),  
9   (i:Loc {name: 'I'}),  
10  (j:Loc {name: 'J'}),  
11  (k:Loc {name: 'K'}),  
12  (l:Loc {name: 'L'}),  
13  (m:Loc {name: 'M'}),
```

Qui creo i 13 nodi con i nomi delle prime 13 lettere dell'alfabeto.

Qui sotto vado invece a creare le relazioni con il nome di ROAD assegnando una proprietà alla relazione di nome cost, che ovviamente indica il costo per andare dal primo nodo al secondo, ricordando che ogni relazione ha un nodo origine, un nodo destinazione e una direzione. L'assegnazione della proprietà alla relazione è del tutto facoltativa.

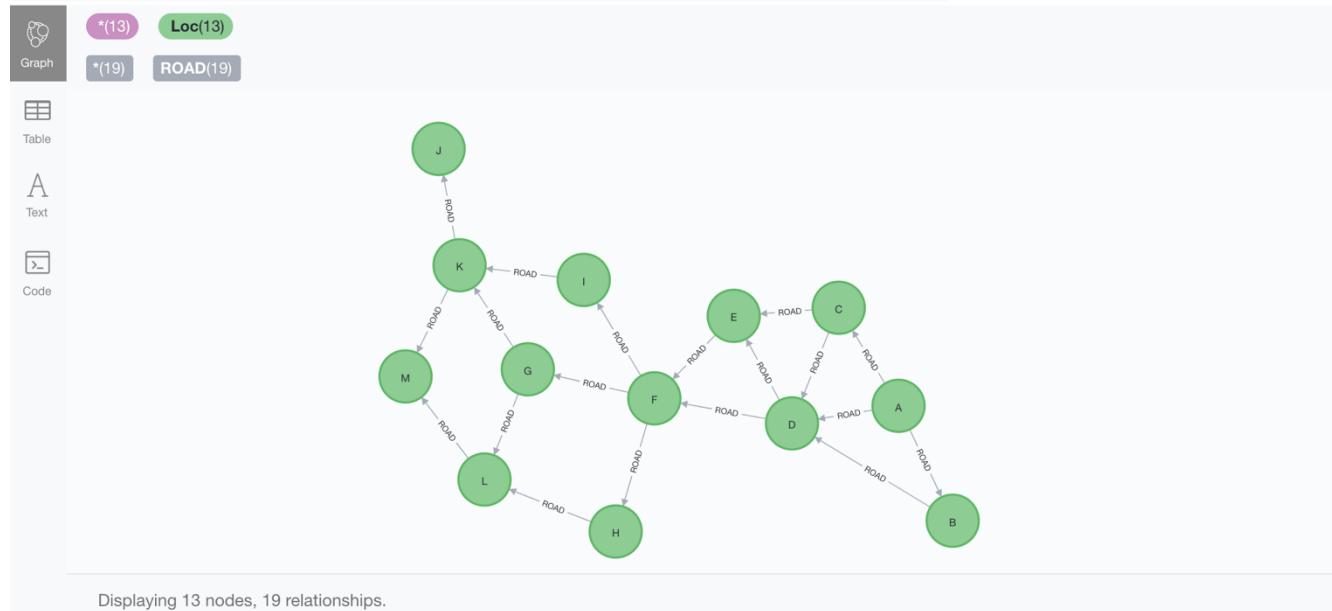
Vedremo in seguito che è possibile trattare il grafo senza costi oppure anche non orientato.

```

14 (a)-[:ROAD {cost: 50}]→(b),
15 (a)-[:ROAD {cost: 50}]→(c),
16 (a)-[:ROAD {cost: 100}]→(d),
17 (b)-[:ROAD {cost: 40}]→(d),
18 (c)-[:ROAD {cost: 40}]→(d),
19 (c)-[:ROAD {cost: 80}]→(e),
20 (d)-[:ROAD {cost: 30}]→(e),
21 (d)-[:ROAD {cost: 80}]→(f),
22 (e)-[:ROAD {cost: 40}]→(f),
23 (f)-[:ROAD {cost: 50}]→(h),
24 (f)-[:ROAD {cost: 50}]→(i),
25 (f)-[:ROAD {cost: 100}]→(g),
26 (g)-[:ROAD {cost: 40}]→(l),
27 (g)-[:ROAD {cost: 40}]→(k),
28 (h)-[:ROAD {cost: 80}]→(l),
29 (i)-[:ROAD {cost: 80}]→(k),
30 (k)-[:ROAD {cost: 80}]→(j),
31 (k)-[:ROAD {cost: 30}]→(m),
32 (l)-[:ROAD {cost: 30}]→(m);

```

Il risultato dell'esecuzione delle 32 righe di codice precedenti è il seguente



Ottenuto con un tempo di completamento di 35 ms.

## 3.2 Funzione neighbour a livello k e Shortest path

### Neighbor a livello k

La funzione neighbor fa parte della libreria APOC di Neo4j. APOC è l'acronimo di Awesome Procedures on Cypher. Prima del rilascio di APOC, gli sviluppatori dovevano scrivere le proprie procedure e funzioni per funzionalità comuni che Cypher o il database Neo4j non avevano ancora disponibili. Quindi, uno degli sviluppatori Neo4j ha creato la libreria APOC come libreria di utilità standard per procedure e funzioni comuni. Ciò ha consentito agli sviluppatori di piattaforme e settori diversi di utilizzare una libreria standard per procedure comuni e di scrivere solo le proprie funzionalità per la logica aziendale e le esigenze specifiche del caso d'uso. Si ritiene che la libreria APOC sia la libreria di estensioni più grande e più utilizzata per Neo4j. Comprende oltre 450 procedure standard, fornendo funzionalità per utilità, conversioni, aggiornamenti grafici e altro ancora. Sono ben supportati e sono molto facili da eseguire come funzioni separate o da includere nelle query Cypher.[6]

Andrò adesso a dare un esempio su come si utilizza tale funzione neighbor.

Essa viene chiamata specificando un nodo di cui si vuole conoscere i vicini e si può specificare il livello di vicinanza desiderata fino ad un intero k. Ad esempio se A è AMICO di B e B è AMICO di C ma non di A, A e C sono amici con livello di vicinanza 2, poiché hanno un amico in comune B ma non sono amici direttamente, cioè a livello 1.

```
1 MATCH (p:Loc{name: "G"})
2 CALL apoc.neighbors.athop(p, "ROAD", 1)
3 YIELD node
4 RETURN node
```

In questo caso si fa match sul nodo G e poi si va a vedere a livello 1 quali nodi si possono raggiungere. In pratica si vanno a vedere gli archi uscenti ed entranti di un nodo con risultato i nodi F, L, K.

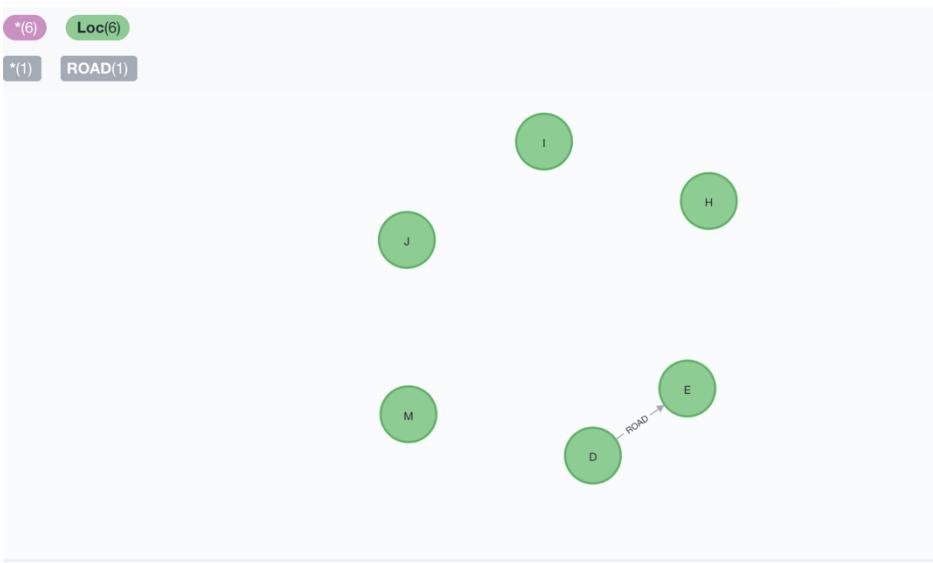


Se invece aumento il livello a due, la query setta semplicemente il numero dei salti a 2 e il risultato sarà quello in figura

```

1 MATCH (p:Loc{name: "G"})
2 CALL apoc.neighbors.athop(p, "ROAD", 2)
3 YIELD node
4 RETURN node
    
```

Query in cui cambia semplicemente il parametro hop con risultato:



Da notare che F, L, K non vengono restituiti perché la query restituisce SOLO quelli a livello k=2.

Sapere i vicini a livello k è molto importante, per riuscire a scoprire dei legami nascosti tra nodi.

## Shortest Path

L'algoritmo Shortest Path calcola il percorso più breve (pesato) tra una coppia di nodi. In questa categoria, l'algoritmo di Dijkstra è il più noto. È un algoritmo grafico in tempo reale e può essere utilizzato come parte del normale flusso utente in un'applicazione web o mobile [6].

Il primo esempio sarà l'algoritmo classico che presi due nodi ne calcola il cammino minimo, e crea una successione di nodi intermedi che fanno parte del percorso minimo con i loro relativi costi intermedi. La seconda funzione invece chiamata ShortestPath () riceve in ingresso due nodi e plotta a schermo il percorso più corto non tenendo conto del costo. Utile per cercare intermediari in transazioni illecite, oppure passaggi di denaro tra due persone.

La query utilizzata nel primo caso è la seguente:

```
1 MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'K'})  
2 CALL gds.alpha.shortestPath.stream({  
3   nodeProjection: 'Loc',  
4   relationshipProjection: {  
5     ROAD: {  
6       type: 'ROAD',  
7       properties: 'cost',  
8       orientation: 'UNDIRECTED'  
9     }  
10   },  
11   startNode: start,  
12   endNode: end,  
13   relationshipWeightProperty: 'cost'  
14 })  
15 YIELD nodeId, cost  
16 RETURN gds.util.asNode(nodeId).name AS name, cost
```

Ho trattato il grafo come non orientato. È possibile però cambiare orientazione come si vuole.

In questo caso viene restituita una tabella che in ordine forma il cammino minimo e nell'ultima casella vi è il costo totale del cammino.

name	cost
"A"	0.0
"B"	50.0
"D"	90.0
"E"	120.0
"F"	160.0
"I"	210.0
"K"	290.0

d streaming 7 records after 2 ms and completed after 2120 ms.

Il cammino minimo è quindi A->B->D->E->F->I->K con costo di 290. Interessante notare che i cammini più corti ADFGK e ADFIK hanno costo rispettivamente 320 e 310, e non sono quindi il cammino di costo minimo. Eseguiamolo adesso come se il grafo fosse non orientato e senza costi, avvicinandosi di più al tipo di grafo che andrò ad approfondire nel capitolo successivo.

La query per il grafo non orientato e senza costi è la seguente.

```

1 MATCH (start:Loc {name:'A'}), (end:Loc {name:'K'})
2 CALL gds.alpha.shortestPath.stream({
3   nodeProjection: 'Loc',
4   relationshipProjection: {
5     ROAD: {
6       type: 'ROAD',
7
7       orientation: 'natural'
8     }
9   },
L0   startNode: start,
L1   endNode: end
L2
L3
L4 })
L5 YIELD nodeId
L6 RETURN gds.util.asNode(nodeId).name AS name

```

E come di consueto ritorna i nodi nell'ordine del cammino più corto:

name

1 "A"

2 "D"

3 "F"

4 "I"

5 "K"

ted streaming 5 records after 1 ms and completed after 39 ms.

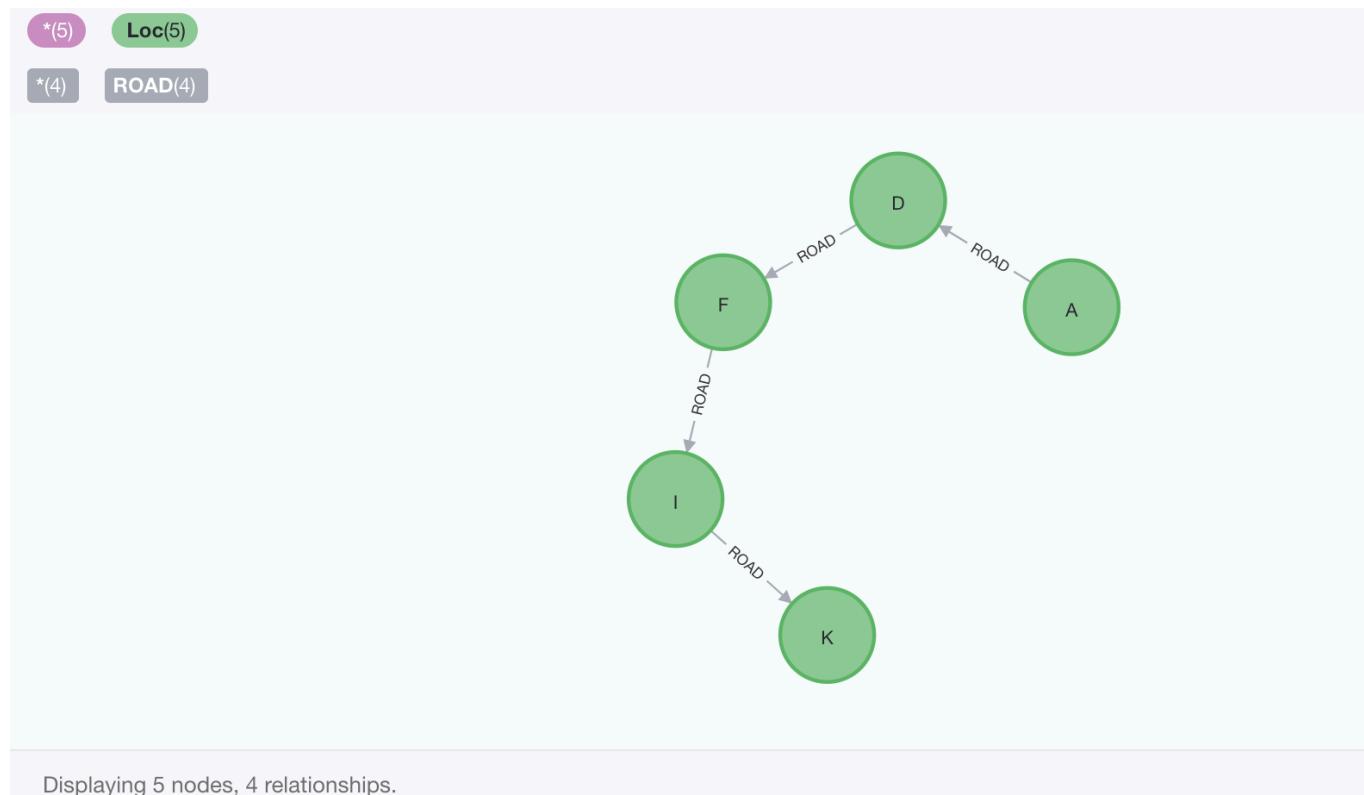
Il cammino più corto da A a K è ADFIK.

Nel caso i cui volessi vedere graficamente il path, la query è la seguente:

```
1 MATCH (start:Loc {name: 'A'}), (end:Loc {name: 'K'})  
2 MATCH p=shortestPath((start)-[:ROAD*]→(end))  
3 RETURN p
```

Qui si va a invocare la funzione Shortest path che tratta il grafo in modo orientato e va a trovare se esiste un cammino minimo tra i due nodi.

Il risultato è il seguente:



In questo modo ho la possibilità di visualizzare a schermo i nodi e posso interagirci direttamente.

Mentre negli altri casi è possibile sia esportare il file csv contenente i nodi che creare un sottografo utilizzando la funzione gds.graph.create in grado di visualizzare graficamente il path corrispondente.

### 3.3 Indegree e Outdegree dei nodi di un grafo

Dato un nodo v, l'indegree di v è il numero di archi entranti in v; mentre l'outdegree è il numero di archi uscenti da v.

L'indegree di v è indicato con  $\text{deg}^-(v)$  e il suo outdegree è indicato con  $\text{deg}^+(v)$ .

Un vertice con  $\text{deg}^-(v) = 0$  è chiamato sorgente, poiché è l'origine di ciascuna delle sue frecce in uscita. Allo stesso modo, un vertice con  $\text{deg}^+(v) = 0$  è chiamato sink, poiché è la fine di ciascuna delle sue frecce in arrivo [1].

Andrò adesso a calcolare indegree e outdegree del grafo precedentemente creato usando Chyper. La query è molto semplice. Vado semplicemente a contare con `size()` il numero di relazioni entranti e uscenti da un nodo :

```
1 MATCH (u:Loc)
2 RETURN u.name AS name,
3 size((u)-[:ROAD]→()) AS follows,
4 size((u)←[:ROAD]-()) AS followers
```

Ottenendo un risultato di questo tipo:

	name	follows	followers
1	"A"	3	0
2	"B"	1	1
3	"C"	2	1
4	"D"	2	3
5	"E"	1	2
6	"F"	3	2
7	"G"	2	1
8	"H"	1	1
9	"I"	1	1
10	"J"	0	1
11	"K"	2	2
12	"L"	1	2

ted streaming 13 records after 12 ms and completed after 20 ms.

In cui per ogni nodo vi è il numero delle persone seguite e il numero dei followers. La tabella mostrata qui sopra sarà molto importante per il grafo finanziario in cui andrò a studiare la distribuzione di in-degree e out-degree.

## 3.4 DFS (Depth first Search) E BFS (Breath First Search)

### DFS

Nella teoria dei grafi, la ricerca in profondità (in inglese depth-first search, con acronimo DFS), è un algoritmo di ricerca su alberi e grafi. A differenza della ricerca in ampiezza, ha la caratteristica di essere intrinsecamente ricorsivo.

Il nome deriva dal fatto che in un albero, ancora prima di avere visitato i nodi adiacenti, l'algoritmo può ritrovarsi a visitare nodi lontani dalla radice, andando così "in profondità". Non a caso, se fatto girare su un grafo, l'algoritmo individua un albero che ne è un sottografo (ovvero che ne contiene tutti i vertici e tutti e soli gli archi che sono stati seguiti). Possiamo vedere l'algoritmo come una visita in ampiezza in cui invece che una coda utilizziamo una pila (ovvero invece di aggiungere gli elementi nuovi in fondo li aggiungiamo in cima). La strategia di ricerca esplora il grafo andando, in ogni istante dell'esecuzione dell'algoritmo, il più possibile in profondità: gli archi del grafo vengono esplorati a partire dall'ultimo vertice scoperto  $v$  che abbia ancora degli archi non esplorati uscenti da esso. Una volta terminata l'esplorazione di tutti gli archi non esplorati del vertice  $v$  si ritorna indietro per esplorare tutti gli archi uscenti a partire dal vertice da cui  $v$  era stato precedentemente scoperto. Il processo di esplorazione continua fin quando tutti i vertici del grafo non siano stati esplorati. Il sottografo dei predecessori, che viene generato dalla visita in profondità può essere costituito da più alberi: tale sottografo è definito foresta DFS, ed è composta, quindi, da diversi alberi DFS [1]. La ricerca DFS, oltre a generare la foresta DFS marca ogni vertice con ben precise informazioni temporali, in particolar modo aggiorna due etichette per ogni nodo:

$d[v]$  che registra quando il generico vertice  $v$  è stato scoperto ed  $f[v]$  che registra quando è stata esplorata l'intera lista di adiacenza di  $v$  (ovvero tutti i vertici che sono raggiungibili a partire da esso). I nodi vengono colorati in modo differente a seconda dei casi: white, colore che ogni nodo assume prima del tempo  $d[u]$ , grey tra il tempo  $d[u]$  ed  $f[u]$  e black nel seguito. Per ogni nodo  $u$  vale la disequazione  $d[u] < f[u]$ .

Dopo aver ricordato brevemente cosa sia la DFS vado ad esegirla col grafo al livello che desidero.

Prima di tutto chiamo il grafo di esempio mygraph1 su cui poi andrò a lavorare:

```
CALL gds.graph.create('myGraph1', 'Loc', 'ROAD', { relationshipProperties: 'cost' })
```

Adesso chiamo la DFS dal nodo A e poiché non specifico la profondità essa visita tutto il grafo

```
1 MATCH (a:Loc{name: 'A'})  
2 WITH id(a) AS startNode  
3 CALL gds.alpha.dfs.stream('myGraph1', {startNode: startNode})  
4 YIELD path  
5 UNWIND [ n in nodes(path) | n.name ] AS tags  
6 RETURN tags  
7
```

Ritornando la seguente tabella ordinata per ordine di visita sul nodo (dal primo nodo visitato all'ultimo)

tags
1 "A"
2 "D"
3 "F"
4 "I"
5 "K"
6 "M"
7 "J"
8 "H"
9 "L"
10 "G"
11 "E"
12 "C"

ted streaming 13 records after 1 ms and completed after 18 ms.

Con tempo di completamento della query di 18ms.

Più significativamente possiamo troncare la BFS a un livello preciso settando la variabile maxDepth. Nel nostro esempio maxDepth=1.

```
1 MATCH (a:Loc{name:'A'})  
2 WITH id(a) AS startNode  
3 CALL gds.alpha.dfs.stream('myGraph1', {startNode: startNode, maxDepth: 1})  
4 YIELD path  
5 UNWIND [ n in nodes(path) | n.name ] AS tags  
6 RETURN tags  
7
```

In questo caso si trovano solo 4 nodi, che sono A e i vicini di A

tags	
1	"A"
2	"D"
3	"C"
4	"B"

ted streaming 4 records after 1 ms and completed after 25 ms.

Per arrivare al livello 2 si fissa maxDepth = 2

```

1 MATCH (a:Loc{name:'A'})
2 WITH id(a) AS startNode
3 CALL gds.alpha.dfs.stream('myGraph1', {startNode: startNode, maxDepth: 2})
4 YIELD path
5 UNWIND [ n in nodes(path) | n.name ] AS tags
6 RETURN tags
7

```

Con risultato oltre A i nodi D, F, E, C, B

tags	
1	"A"
2	"D"
3	"F"
4	"E"
5	"C"
6	"B"

ted streaming 6 records after 1 ms and completed after 7 ms.

## BFS

Nella teoria dei grafi, la ricerca in ampiezza (in inglese breadth-first search, con acronimo BFS) è un algoritmo di ricerca per grafi che partendo da un nodo detto sorgente permette di cercare il cammino minimo nel numero di archi fino ad un altro nodo scelto e connesso al nodo sorgente. BFS è un metodo di ricerca “non informato”, il cui obiettivo è quello di espandere la frontiera della ricerca al fine di esaminare tutti i nodi del grafo sistematicamente, fino a trovare il nodo cercato. In altre parole,

se il nodo cercato non viene trovato, la ricerca procede in maniera esaustiva su tutti i nodi del grafo. La BFS costruisce un albero dei cammini minimi. Se si volesse restituire l'albero breadth-first sarebbe necessario tenere nota di tutti i nodi visitati e del predecessore tramite il quale si è arrivati a loro. A tale scopo, a seconda dello stadio di elaborazione, sarebbe utile marcare i nodi con delle etichette quali "visitato", "in corso di visita" e "non visitato" [1].

Andrò quindi ad eseguire sul grafo di esempio una BFS, che è analogo alla DFS:

```
1 MATCH (a:Loc{name:'A'})  
2 WITH id(a) AS startNode  
3 CALL gds.alpha.bfs.stream('myGraph1', {startNode: startNode, maxDepth: 2})  
4 YIELD path  
5 UNWIND [ n in nodes(path) | n.name ] AS tags  
6 RETURN tags  
7
```

Con risultato i primi 6 nodi:

	tags
1	"A"
2	"B"
3	"C"
4	"D"
5	"E"
6	"F"

gds streaming 6 records after 1 ms and completed after 11 ms.

Abbiamo visto come queste operazioni operano efficientemente su un grafo di piccole dimensioni. Nel prossimo capitolo cercheremo di trattare un grafo di grandi dimensioni su cui eseguire gli stessi algoritmi senza incontrare problemi di heap space o loop, che in un grafo composto da 13 nodi non si presentano ovviamente.

# CAPITOLO 4

## Creazione e interrogazione di un Big GDB

L'interrogazione e la creazione di un grande database a grafo è un compito davvero interessante.

In questo capitolo andrò a creare un grafo composto da 182510 nodi e circa 270 milioni di archi partendo da un file di nome ARCHI.csv di 8.82 Gb, dove ciascuna riga è una transazione che collega 2 nodi, a cui è associato un tempo in cui avviene tale transazione.

Andrò quindi a fare un analisi del grafo cercando di comprenderne la struttura e studiarne le caratteristiche come Indegree, outdegree, degree distribution, Shortest path e altre ancora, per poi prestare attenzione alle prestazioni di ogni esecuzione.

### 4.1 Dal file csv a grafo in Neo4j

Il file ARCHI.csv come già accennato è un file di dimensione 8.82Gb contenente esattamente 267.962.619 archi. Ogni riga contiene è così strutturata:

ID\_originw; ID\_destinazione; tempo della transazione

Nei primi due campi trovo un intero che varia da 1 a 182510(ID dei nodi connessi), mentre nell'ultimo campo ho una stringa che rappresenta un ora ben precisa:

Esempio: 1327; 12349; 13:09:27

All'interno del file ogni riga è unica, ossia ogni coppia di nodi è univoca. Il grafo viene quindi costruito creando prima 182510 nodi numerati a partire da 1, e poi avverrà la lettura del file csv riga per riga andando a creare le relazioni tra essi.

Per semplicità e per risparmio di memoria ho deciso di chiamare ogni relazione "transazione" senza assegnarle alcuna proprietà. Ricordiamo infatti che ad ogni relazione può essere associata una proprietà, che in questo caso sarebbe stata "tempo", ma poiché volevo snellire il grafo e incrementarne le prestazioni anche in fase di visualizzazione ho preferito evitare.

Quindi per prima cosa andrò a creare i nodi con la seguente query:

```
1 UNWIND range(1, 182510) as id
2 CREATE (a:User {id:id})
3
```

La query è davvero molto semplice, UNWIND attua una specie di ciclo for in cui crea per ciascun numero intero n da 1 a 182510 un nodo con proprietà id, alla quale è associato l'intero n.

Andrò quindi a creare il primo nodo con id associato 1, il secondo con id associato 2 e così via.

La prossima fase è la più delicata, ossia il caricamento del file csv su Neo4j. In Neo4j esiste una funzione apposita LOAD CSV che viene utilizzato, appunto per importare dati da file CSV.

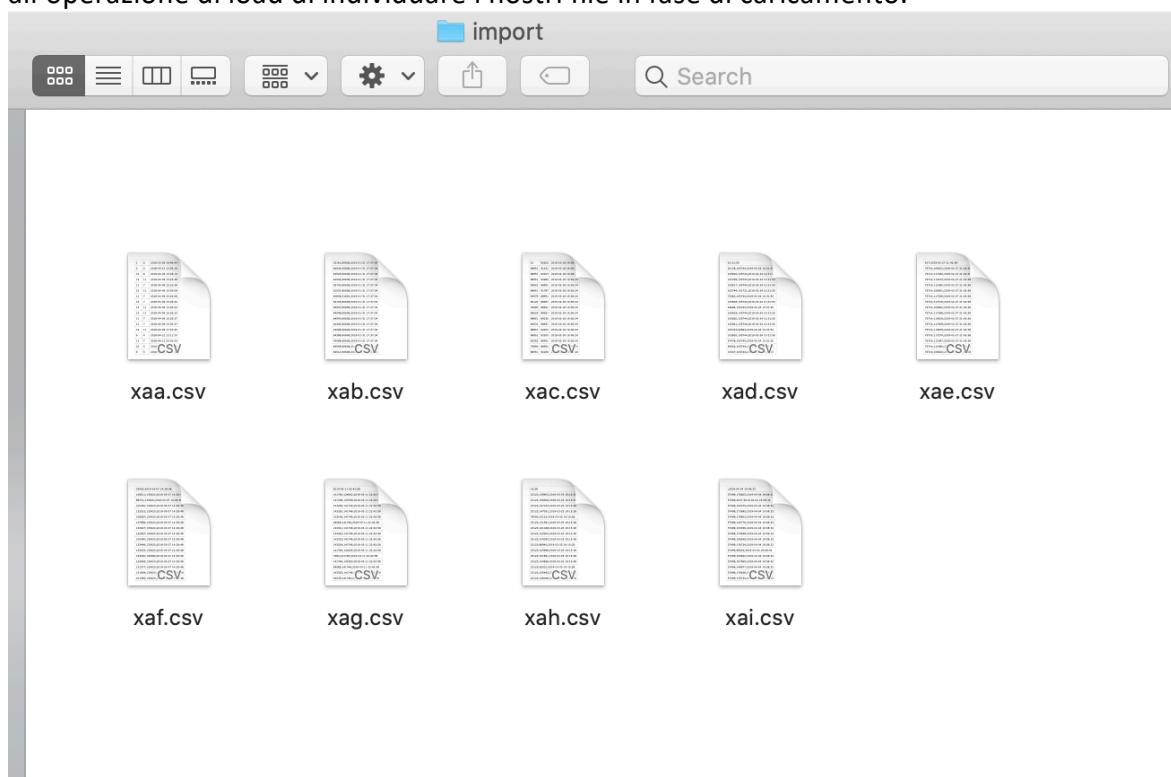
Prima però di eseguire tale funzione, dopo alcuni tentativi di caricare il grafo per intero, ho trovato la strategia giusta per affrontare un file così grande. Ho deciso infatti di dividerlo in 9 file da circa 1 gb ciascuno così da facilitare la query di creazione.

La procedura utilizzata è la seguente:

```
Last login: Fri Jan 29 02:59:23 on ttys001  
giorgiopirina@Giorgios-Air ~ % split -b 1000m ARCHI_copy.csv
```

La funzione split prende in ingresso una dimensione file, in questo caso 1 gb, e divide il file in file da 1 gb l'uno, con l'eccezione dell'ultimo file che potrebbe essere minore di 1 Gb.

Una volta creati i 9 file è necessario mettere i file nella cartella Import del nostro progetto per permettere all'operazione di load di individuare i nostri file in fase di caricamento.



Dopo aver eseguito tale operazione, un altro passaggio fondamentale per caricare questi file in memoria è stata la creazione di un indice di ricerca sul campo id dei nodi.

Andrò adesso a spiegare cos'è un indice di ricerca e come si usa.

## Indici di ricerca

Un indice (nel campo dei database) è una struttura dati realizzata per migliorare i tempi di ricerca (query) sui dati. Se una tabella non ha indici, ogni ricerca obbliga il sistema a leggere tutti i dati presenti in essa. L'indice consente invece di ridurre l'insieme dei dati da leggere per completare la ricerca.

Ad esempio, se si ha un insieme di dati disordinato, è possibile crearne un "indice" in ordine alfabetico, e sfruttare le proprietà dell'ordine alfabetico per arrivare prima al dato o ai dati cercati. Si potrebbe pensare, ad esempio, di applicare una ricerca binaria all'indice ordinato per reperire in tempi più brevi le informazioni richieste.

Gli indici possono essere anche degli effetti negativi in quanto rendono più lente le operazioni di inserimenti e modifica (update), ed aumentano l'uso della memoria di massa.

Prima di definire su quali campi creare gli indici occorre valutare quali siano le operazioni di selezione più frequenti. La giusta scelta degli indici in uno schema può migliorare le prestazioni dell'operazione di lettura anche dell'80%. Strumenti idonei all'individuazione degli indici spesso sono disponibili quali strumenti dei DBMS. In particolare, data una query di selezione (abbastanza complessa), lo strumento di ottimizzazione degli indici individua eventuali indici da creare ed effettua una stima percentuale del miglioramento che potrebbe essere ottenuto. Al contrario una errata scelta può comportare un degrado delle prestazioni del sistema dovuto all'overhead introdotto per il mantenimento delle informazioni corrette, infatti ad ogni operazione di inserimento, aggiornamento e cancellazione di record indicizzati il dbms deve intervenire anche sul/sui file indice.

Un indice si può pensare che gestisca  $\langle K_i, P_i \rangle$  dove  $K_i$  è il valore dell'attributo chiave e  $P_i$  è il puntatore al record di dati. Generalmente il file indice è ordinato secondo i valori del campo  $K_i$  affinché sia possibile effettuare una ricerca binaria.

Le tipologie di indici sono le seguenti:

Indici primari: I DBMS tendono a far ricadere, erroneamente, in questa categoria gli indici definiti su attributi a valore univoco (ovvero su una chiave), in realtà gli indici primari, a differenza di quelli secondari, contengono direttamente i dati (o sono realizzati su file ordinati sugli stessi campi su cui è definito l'indice stesso). Sono detti primari poiché non solo garantiscono l'accesso in base alla chiave, ma anche le locazioni fisiche necessarie per memorizzare i dati.

Indici secondari: Sono gli indici che hanno lo scopo di favorire gli accessi ai dati senza contenere i dati stessi. Sono generalmente definiti su attributi che possono avere valori ripetuti.

Indici clustered: Sono gli indici definiti sull'attributo secondo i cui valori il file di dati è ordinato

Indici unclustered: Sono gli indici definiti sull'attributo secondo i cui valori il file di dati non è ordinato

Indici densi: Sono gli indici il cui numero di coppie  $\langle K_i, P_i \rangle$  è uguale al numero di valori chiave dei record

Indici sparsi: Sono gli indici il cui numero di coppie  $\langle K_i, P_i \rangle$  è inferiore al numero di valori chiave dei record

Esistono vari tipi di indici in base al tipo di tabella utilizzata, ad esempio: hash, btree, rtree, ecc...

Nel mio caso sono quindi andato a creare un indice di ricerca sulla chiave di ogni nodo, ossia l'id, poiché come si vedrà dopo nella query per creare il grafo, vi è una fase di matching in cui si cerca il nodo corrispondente all'id\_origine e quello del id\_destinazione. La creazione di un index sulla chiave id permette un notevole miglioramento delle prestazioni in fase di matching [1].

Un indice su una singola proprietà per tutti i nodi che hanno una particolare etichetta può essere creato con CREATE INDEX index\_name FOR (n: Label) ON (n.property). Da tenere presente che l'indice non è immediatamente disponibile, ma verrà creato in background.

La query in CHYPER per creare un indice di ricerca sul campo id è la seguente:

```
1 CREATE INDEX btree FOR (n:User)  
2 ON (n.id)
```

La query crea un indice di ricerca di tipo btree sull'id di ogni nodo che in questo caso è chiave di ricerca.

Andiamo adesso a caricare il grafo con la seguente query che commenteremo brevemente:

```
1 :auto Using periodic commit 10000  
2 load csv from "file:///xaa.csv" as line fieldterminator ';'  
3 MATCH (a:User{id:toInteger(line[0])})  
4 MATCH(b:User{id:toInteger(line[1])})  
5  
6 CREATE (a)-[r:transazione]→(b)
```

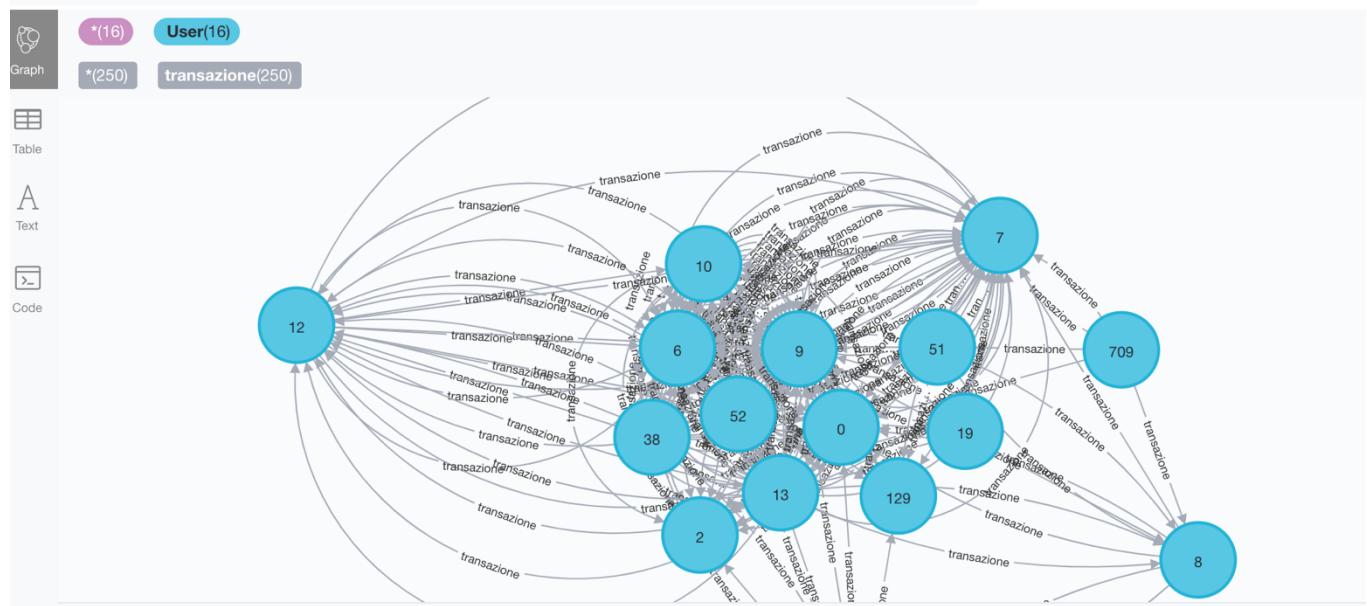
Questa è la query che andrà eseguita 9 volte su ogni file spartito in precedenza da circa 1 GB. Viene utilizzato nell'analisi a blocchi di 10000 righe ciascuno evitando così un sovraccarico del sistema e dell'heap space. Vado quindi ad eseguire la load csv dal primo dei nove file specificando il terminatore di campo; Ecco la fase di matching in cui prendo il primo campo che ricordiamo rappresenta l'id del nodo e vada a prendere il nodo a e il nodo b che sono rispettivamente il nodo origine e il nodo destinazione nella transazione. Dopo che le variabili a e b vengono assegnate, procedo alla creazione di una relazione di tipo :transazione per ogni coppia di nodi per riga, essendo ogni coppia univoca.

Ripeto il processo per nove volte cambiando semplicemente il nome del file (xaa,xab,xac... ecc)

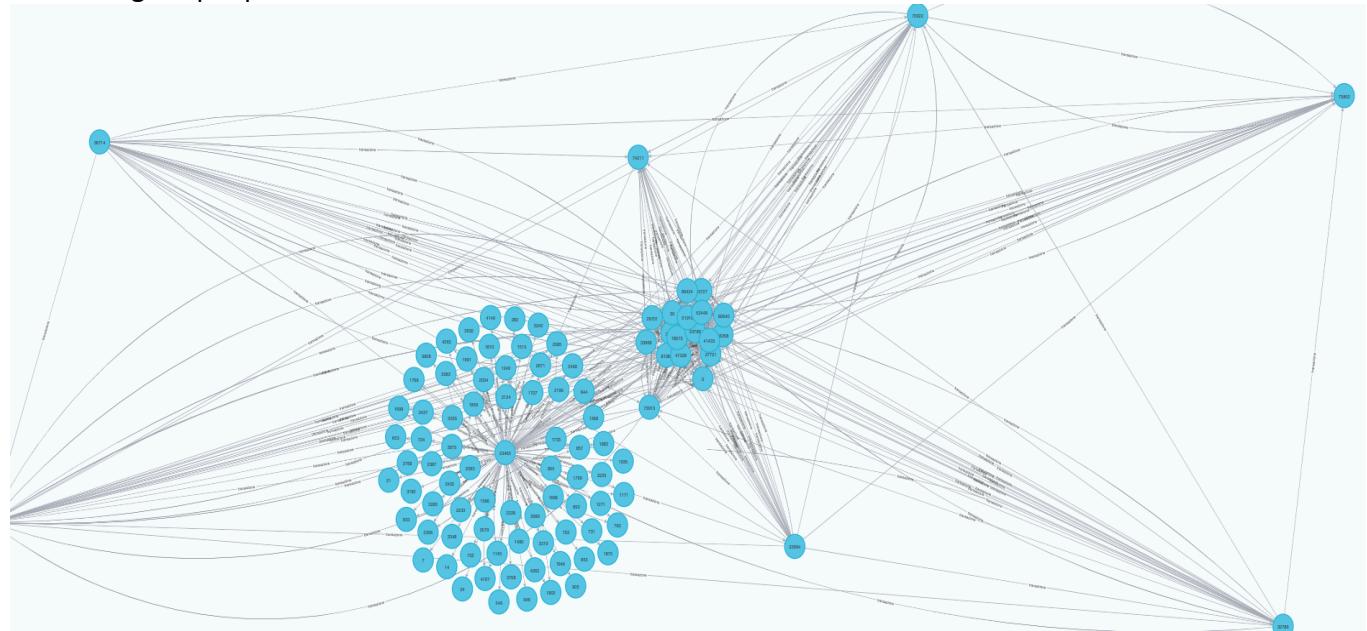
Il risultato è chiaramente un grafo ampio e complesso e quindi graficamente impossibile da visualizzare chiaramente.

Ecco il risultato in cui nel primo caso mostro solo 25 nodi e nel secondo provo ad aumentare il numero di nodi visualizzati:

```
neo4j$ MATCH p=()-[r:transazione]→() RETURN p LIMIT 25
```



Come si vede bene vi sono talmente tanti archi che è impossibile distinguerli. Bisognerà quindi lavorare con sottografi più piccoli.



Il tempo di completamento per la creazione del grafo è di circa 22 minuti. Considerando che comunque è un grafo enorme e che l'operazione viene eseguita soltanto una volta è un tempo ragionevole per un grafo di tali dimensioni.

## 4.2 Esecuzione e prestazioni degli algoritmi standard sul grafo

Adesso andrò a sperimentare alcuni degli algoritmi visti nel capitolo 3, andando naturalmente a creare il grafo myGraph:

```
neo4j$ CALL gds.graph.create('myGraph', 'User', 'transazione')
```

Adesso eseguo una BFS con maxDepth =5, prendendo un nodo a caso;

```
1 MATCH (a:User{id:1069})
2 WITH id(a) AS startNode
3 CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxDepth: 5})
4 YIELD path
5 UNWIND [ n in nodes(path) | n.id ] AS ids
6 RETURN ids
7 ORDER BY ids
```

Essa termina in appena 13ms, facendo streaming di 118167 nodi, ossia tutti i vicini di 1069 a livello 5.

Adesso prendo due nodi a caso ed eseguo lo shortest path;

```
1 MATCH (start:User {id:1}), (end:User {id:182510})
2 CALL gds.alpha.shortestPath.stream({
3   nodeProjection: 'User',
4   relationshipProjection: {
5     ROAD: {
6       type: 'transazione',
7
7       orientation: 'natural'
8     }
9   },
10  },
11  startNode: start,
12  endNode: end
13
14 })
15 YIELD nodeId
16 RETURN gds.util.asNode(nodeId).id AS id
```

Il risultato è il seguente:

	id
1	1
2	176736
3	182510

ted streaming 3 records after 1 ms and completed after 254213 ms.

Tempo di Completamento di 254213 ms, circa 4 minuti per trovare lo shortest path 1->176736->182510.

Prendo altri due nodi in cui lo shortest path è da 4:

```

1 MATCH (start:User {id:2}), (end:User {id:673})
2 CALL gds.alpha.shortestPath.stream({
3   nodeProjection: 'User',
4   relationshipProjection: {
5     ROAD: {
6       type: 'transazione',
7
7       orientation: 'natural'
8     }
9   },
10  startNode: start,
11  endNode: end
12
13
14 })
15 YIELD nodeId
16 RETURN gds.util.asNode(nodeId).id AS id

```

In questo caso si ottiene il risultato:

	id
1	2
2	601
3	669
4	673

ted streaming 4 records after 1 ms and completed after 361670 ms.

Il path è infatti formato dai nodi 2->601->669->673. Il tempo di completamento è di circa 6 minuti.

### BFS a livello K

Eseguiamo adesso una bfs troncata al livello k=3:

```
1 MATCH (a:User{id:1})
2 WITH id(a) AS startNode
3 CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxDepth: 3})
4 YIELD path
5 UNWIND [ n in nodes(path) | n.id ] AS tags
6 RETURN tags
7
```

Con risultato:

	tags
1	1
2	2
3	18
4	33
5	36
6	57
7	...

ted streaming 118310 records after 1 ms and completed after 6 ms,

Vengono individuati 118310 nodi e si completa in 5 ms, un tempo molto interessante.

La funzione stream permette di scrivere i dati su un file esportabile di tipo csv, con il quale si possa lavorare con excel o programmi di analisi di dati

#### 4.2.1 Grafi(reti) a invarianza di scala

Viene definita rete a invarianza di scala (in inglese scale-free network) un grafo che gode della seguente proprietà: se si considera la relazione tra il numero di nodi ed il numero delle loro connessioni si vede che il suo grafico è di tipo esponenziale negativo, e quindi invariante per cambiamenti di scala. Questa invarianza di scala significa che paragonando il numero di due tipi di nodi, ad esempio quelli con 10 connessioni e quelli con 15, si vede che la proporzione fra i due è  $e^{-a(N_b-N_a)}$ , dove Nb ed Na sono il numero di nodi del denominatore e numeratore rispettivamente mentre a è un parametro del tipo di rete considerato. Questa legge è detta legge di potenza, di cui a è il parametro.

Il termine fu coniato da Albert-László Barabási con Rèka Albert dell'Università di Notre Dame (USA) nel 1999 [8].

La nascita di una rete a invarianza di scala è molto semplice: si stabilisce che quando un nodo deve stabilire un nuovo collegamento, preferisce farlo verso un nodo che ne ha già molti, portando questi ad una crescita esponenziale con l'aumentare del numero dei collegamenti della rete. Riassumendo è una situazione del tipo: il ricco diventa sempre più ricco mentre il povero sempre più povero (in proporzione). Nodi di tipo "ricco" vengono detti hub. Il meccanismo del "ricco che diventa sempre più ricco" è inoltre molto resistente contro altri meccanismi di crescita della rete e porta spesso alla preservazione della proprietà dell'invarianza di scala [18].

Le reti a invarianza di scala sono interessanti anche per il loro comportamento nei confronti di situazioni aggressive. Paragoniamo una rete di tipo casuale ed una a invarianza di scala sulla base delle reazioni a due tipi di attacco: l'attacco casuale e l'attacco mirato. Un attacco casuale non sceglie i nodi da sopprimere, andando a caso; in una rete casuale la perdita di funzionalità sarà quindi proporzionale al danno inflitto mentre una rete ad invarianza, avendo la sua funzionalità concentrata in pochissimi centri, sarà virtualmente insensibile a danni di questo tipo fintanto che il numero di nodi non-hub rimane alto. Un attacco mirato invece sceglie con cura i nodi da attaccare per massimizzare il danno; nel caso di una rete casuale, avendo tutti i nodi approssimativamente la stessa importanza (con un margine di errore di  $\frac{1}{\sqrt{N}}$  dove N è il numero dei nodi della rete) un attacco mirato non ha quindi differenze sostanziali da un attacco casuale. L'effetto su di una rete ad invarianza di scala è invece l'opposto: con pochi singoli attacchi mirati agli hub è possibile abbattere la funzionalità praticamente del 100% (anche con milioni di nodi!). L'effetto è pertanto studiato per via degli effetti che un eventuale attacco informatico da parte di malintenzionati potrebbe avere sulla struttura di internet, o di come tossine mirate possano distruggere interi ecosistemi [1].

#### 4.2.2 Legge di potenza e Indegree/Outdegree

Una legge di potenza (power law) è una qualsiasi relazione del tipo:

$$f(x) = ax^k + o(x^k), \quad x \rightarrow 0,$$

dove a e k sono costanti e  $o(x^k)$  è una funzione asintoticamente più piccola di  $x^k$ , dove k è di solito chiamato esponente di scala.xk [1].

Leggi di potenza ricorrono nelle distribuzioni di probabilità di molti fenomeni fisici (ad esempio la magnitudo dei terremoti, il diametro dei crateri dei pianeti, la dimensione dei frammenti degli oggetti che si infrangono per urti, l'intensità delle esplosioni solari), sociali (il numero dei morti nelle guerre, la popolazione delle città, il numero di collegamenti ai siti web, il numero di citazioni) ed economici (la distribuzione della ricchezza, le vendite di libri e cd, ecc.); così come ricorrono in altri tipi di relazioni, come quella tra il tasso metabolico di una specie e la sua massa corporea (cosiddetta legge di Kleiber), o quella tra la forza di gravità e la distanza tra le masse [1].

Nel caso delle distribuzioni di probabilità, una distribuzione che obbedisce alla legge di potenza è denominata power law distribution, o anche distribuzione di Pareto - dal nome dell'economista Vilfredo Pareto, che per primo la individuò nella distribuzione del reddito - o infine legge di Zipf - dal linguista George Kingsley Zipf che la individuò studiando la frequenza d'uso delle parole nei testi. quattro con reddito pari a 1 trilione [7].

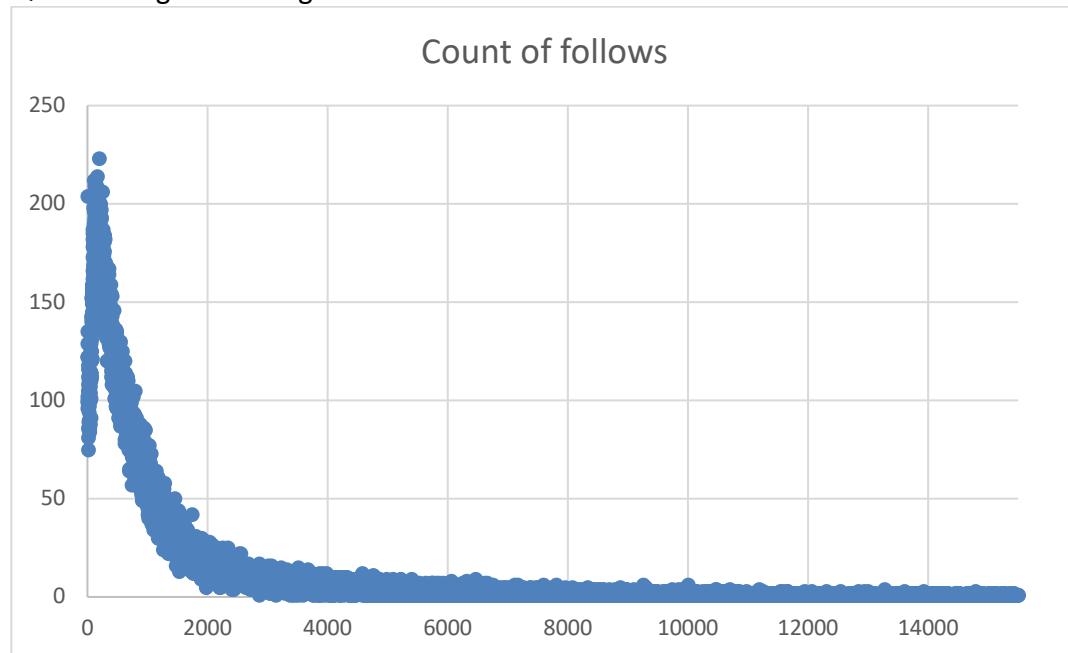
Andrò quindi a calcolarmi il grado uscente e il grado entrante per ogni nodo, usando excel, creando una tabella pivot in cui nella prima colonna ho messo i gradi uscenti del nodo, e a sinistra il numero di nodi che ha quel grado uscente/entrante.

0	204
1	96
2	135
3	100
4	122
5	102
6	122
7	99
8	129
9	75
10	116
11	86
12	118
13	105
14	105

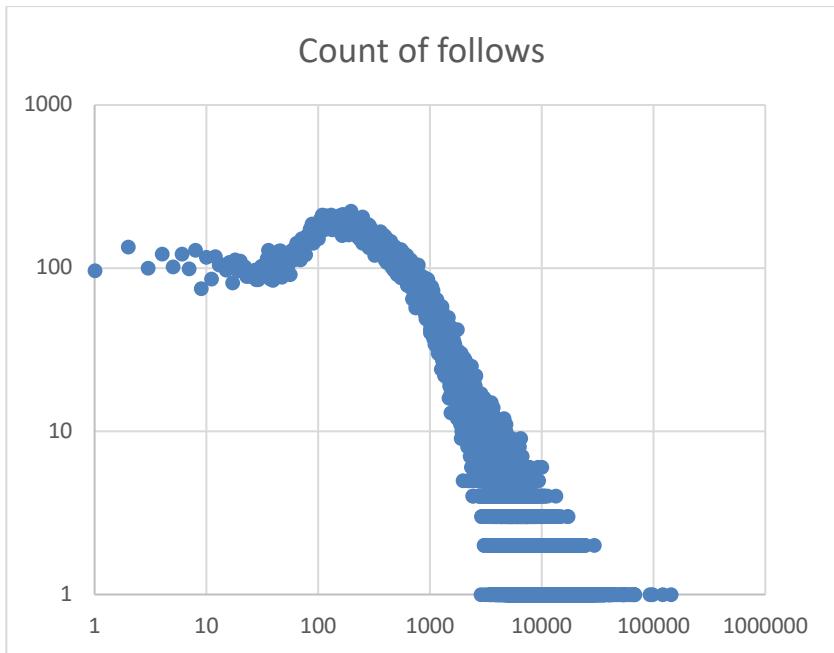
Ossia ci sono 204 user che non seguono nessuno, 96 User che seguono una persona soltanto e così via fino a 182510.

Di seguito riporterò i grafici eseguiti con excel di entrambi i gradi (uscente/entrante) ed andrò a osservare il tipo di distribuzione che assumono. Laddove il grafo avesse una distribuzione dei gradi che segue la legge di potenza, concluderei che il sistema ha un elevata attack tolerance. Se invece avesse una distribuzione esponenziale il sistema risulterebbe esposto ad attacchi su nodi critici.

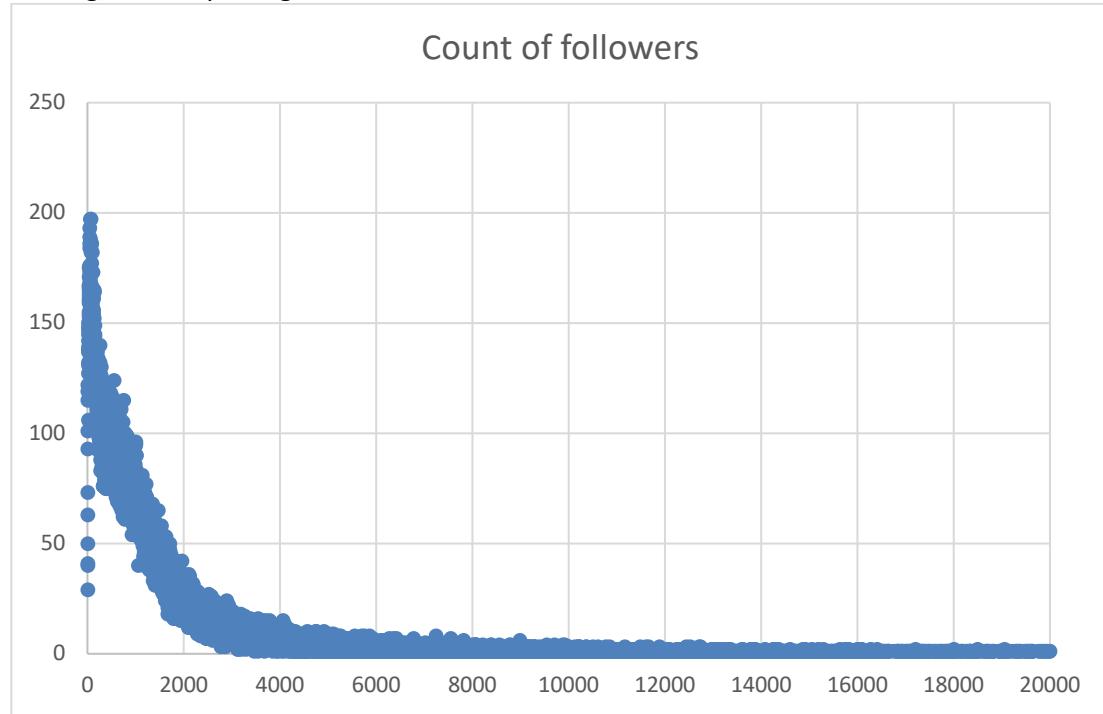
Questo è il grafico del grado uscente:



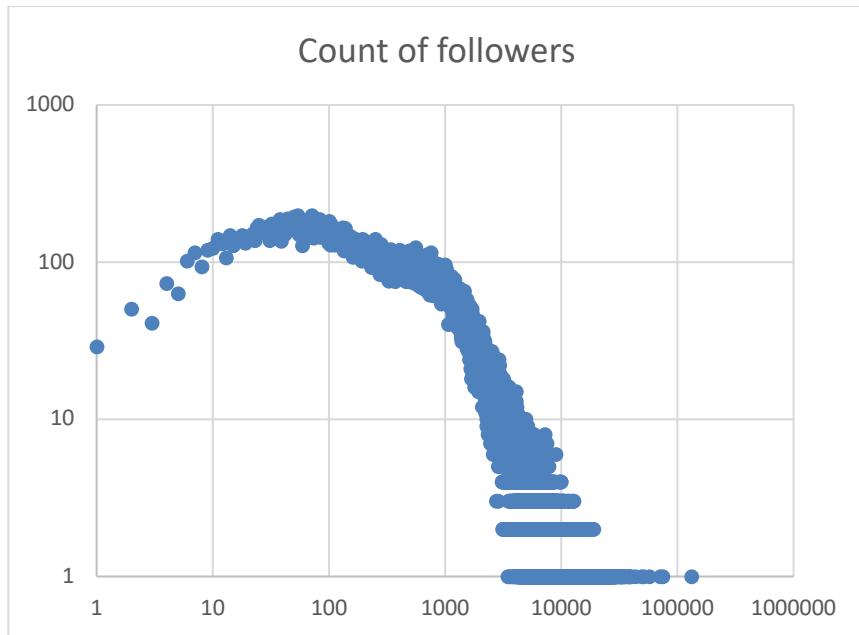
In cui sull' asse delle ascisse ho la colonna a sinistra mentre sulle ordinate ho la colonna a destra.  
Osserviamo la distribuzione di grado in scala log/log nel grafico sottostante. Notiamo che non si tratti di una legge a potenze in quanto i punti non si dispongono su una retta.



Analogamente per il grado entrante:



E visualizziamo anche il grafico log/log. Anche qui non si riscontra un andamento con legge a potenze.



### Conclusioni sul tipo di distribuzione

Il tipo di distribuzione non suggerisce una rete ad invarianza di scala, infatti come è ben visibile entrambe le distribuzioni non seguono la legge di potenza (power law distribution) bensì un andamento esponenziale. In questo caso la robustezza del grafo ne risente sensibilmente. Nel contesto di reti complesse, la tolleranza agli attacchi è la robustezza della rete, ovvero la capacità di mantenere la connettività e il diametro complessivi della rete quando i nodi vengono rimossi.

Il sistema si espone quindi ad attacchi mirati su nodi specifici, attacchi basati sulla vicinanza temporale o sulla persistenza del nodo, i quali si evitano se la rete è a invarianza di scala.

# Conclusioni e ringraziamenti

Possiamo a questo punto riassumere brevemente il percorso di tesi appena terminato per rendere ancor più evidenti i risultati ottenuti. Dopo aver presentato brevemente le principali basi di dati a grafo, ho focalizzato il mio studio su Neo4j per l'incredibile efficienza delle librerie graph data science e APOC che esso offre e le numerose funzioni adatte per il tipo di obiettivo della tesi di determinare il grado di robustezza del grafo e di eseguire algoritmi di ricerca su esso. La tesi ha voluto dimostrare come da un file contenente solo relazioni e nodi si possa creare un grafo complesso ma allo stesso tempo osservabile, su cui si possano compiere studi sui nodi, sulla distribuzione di grado, individuare quali siano i nodi più vulnerabili ad attacchi mirati ed eseguire efficientemente algoritmi come Shortest path fra due nodi o le classiche BFS e DFS. Un altro punto interessante, che mette in luce Neo4j come solida piattaforma su cui lavorare, è la facilità con cui i nostri risultati possano essere esportati in file di tipo csv e viceversa. I risultati di ogni query possono essere sia esportati come csv, lavorandoci con excel o altri programmi simili, oppure possiamo creare un sottografo con i nodi ottenuti come risultato e lavorare su quella porzione di grafo applicando gli stessi algoritmi e procedure del grafo principale.

Questa esperienza di tesi mi ha permesso di conoscere la realtà dei Graph DB apprezzando la loro efficienza ed efficacia nel contesto di dati che contengono relazioni. La mia relazione che mostra inoltre la semplicità con cui si può cambiare modello a seconda del tipo di studio da eseguire sui diversi dataset.

Concludo quindi la tesi affermando con decisione che la scelta di rappresentare i dati con un database a grafo offre delle opportunità nuove e interessanti sia in ambito di sicurezza informatico che di analisi dei big data.

Un ringraziamento speciale va al Prof. Paolo Ferragina il quale mi è stato vicino in questo momento difficile da noi tutti affrontato, riuscendo a cogliere ogni volta aspetti positivi e negativi del mio operato dandomi continuamente nuovi contributi e idee.

Dal punto di vista affettivo ringrazio la tutta la mia famiglia che mi è stata sempre vicino in ogni momento di difficoltà, con un saluto particolare a mia nonna Anna e mio nonno Evandro che, nonostante si siano ormai abbandonati all'eterno ritorno dell'identico (Nietzsche), spero di aver reso entrambi felici.

# Bibliografia

[1] Wikipedia, <https://it.wikipedia.org>

[2] database a grafo

<https://it.linkeddata.center/b/database-a-grafo/>

[3] Ajit Singh, Anant Kumar 2020-2021

Graph Database with Neo4j, pp 75-89

[4] Onofrio Panzarino

<https://www.html.it/pag/67730/cypher-sintassi-e-struttura/>

[5] Trasversal API

<http://www.mokabyte.it/2016/01/neo4j-1/>

[6] Libreria graph data science (gds)

<https://neo4j.com/docs/graph-data-science/current/>

[7] Redis Newman, M. E. J.

Power laws, Pareto distributions and Zipf's law, in Contemporary Physics, vol. 46, 2005, pp. 323-351

[8] Barabási, A-L., Albert, R., Jeong, H.

"Mean-Field Theory for Scale-Free Random Networks", Physica A 272, pp 173-187

[9] Modello a grafo di proprietà

"Property Graph Model". GitHub. Retrieved 2019-11-08.

[10] Allegro Graph

<https://allegrograph.com>

[11] Infinite Graph

<https://objectivity.com/products/infinitegraph/>

[12] OrientDB

<https://www.orientdb.org>

[13] Azure cosmos DB

<https://azure.microsoft.com/it-it/services/cosmos-db/>

[14] Amazon Neptune

<https://aws.amazon.com/it/neptune/>

[15] JanusGraph

<https://janusgraph.org>

[16] Renzo Angles e Claudio Gutierrez

An introduction to Graph Data Management

[17] Michelle Knight

<https://www.dataversity.net/what-is-a-property-graph>

[18] Barabási, A-L. e Albert R.

"Emergence of Scaling in Random Networks" Science 286, pp 509-512, 1999