

Alla scoperta dei Graph Database

Matteo Pani

24 ottobre 2015

GULCh

Gruppo Utenti Linux Cagliari h...?



One size doesn't fit all
Modellare le relazioni

I Graph Database

II Labeled Property Graph Model

I Graph-DBMS

Neo4j

Neo4j Internals

Cypher

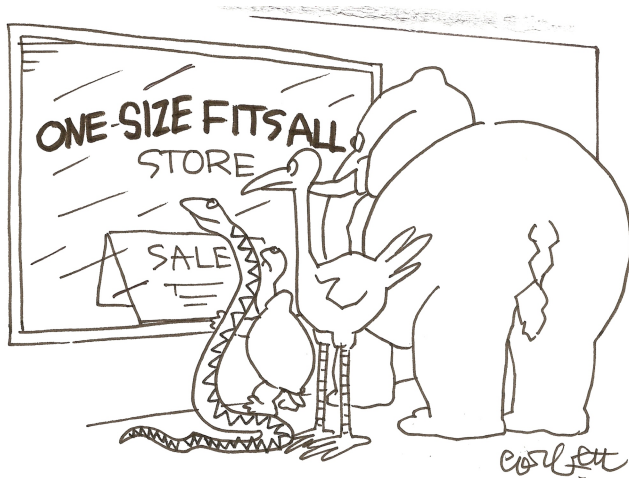
Interagire col DB

Profiling delle query

Bibliografia

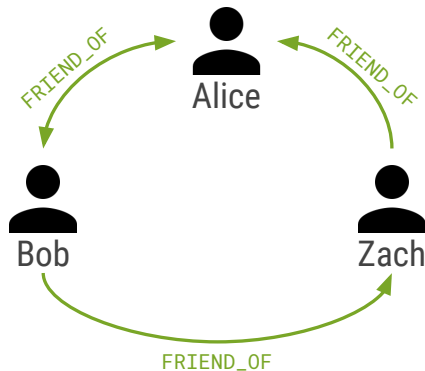


One size doesn't fit all



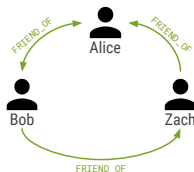
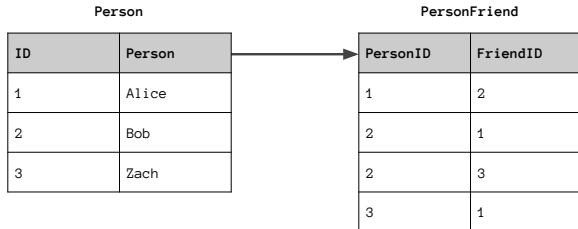
Modellare le relazioni

Un esempio di rete



Modellare le relazioni

Con i database relazionali



Modellare le relazioni

Con i database relazionali

"Chi sono gli amici di Bob?"

```
SELECT p1.Person
FROM Person p1
JOIN PersonFriend
ON PersonFriend.AmicoID = p1.ID
JOIN Person p2
ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```



Modellare le relazioni

Con i database relazionali

*"Chi sono gli amici **degli amici** di Bob?"*

```
SELECT p1.Person AS PERSON, p2.Person AS  
      FRIEND_OF_FRIEND  
FROM PersonFriend pf1  
JOIN Person p1  
ON pf1.PersonID = p1.ID  
JOIN PersonFriend pf2  
ON pf2.PersonID = pf1.FriendID  
JOIN Person p2  
ON pf2.FriendID = p2.ID  
WHERE p1.Person = 'Bob' AND pf2.FriendID  
      <> p1.ID
```



Modellare le relazioni

Con i database relazionali

Non soddisfacente:

- ▶ Join ricorsivi → Query in profondità troppo onerose
- ▶ Difficoltà nel dare giusta espressività alle relazioni
- ▶ Il sistema di chiavi esterne costa di per sé
- ▶ Tabelle sparsamente popolate → Gestione casi NULL
- ▶ Grande rigidità → Talvolta complicato adattare schema a sopraggiunte esigenze
- ▶ Query reciproche troppo costose (“Chi è amico con”)



Modellare le relazioni

I database NoSQL

NoSQL (Not Only SQL)

- ▶ Key-Value Store
- ▶ Document Store
- ▶ Column-oriented
- ▶ Graph Database

Key/Value Store, Document Store e Column-oriented sono anche detti database **Aggregate-oriented**



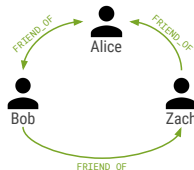
Modellare le relazioni

Con i database NoSQL Aggregate-Oriented

name: Alice

name: Bob

name: Zach



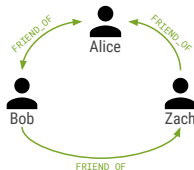
Modellare le relazioni

Con i database NoSQL Aggregate-Oriented

```
name: Alice  
friends: [Bob]
```

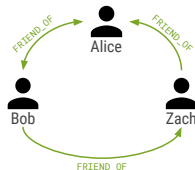
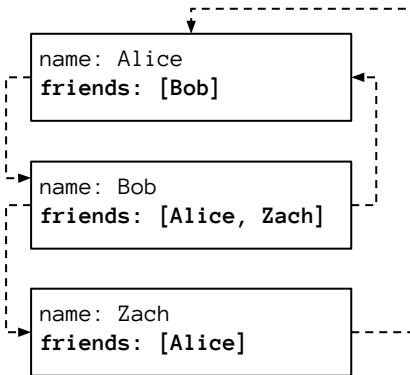
```
name: Bob  
friends: [Alice, Zach]
```

```
name: Zach  
friends: [Alice]
```



Modellare le relazioni

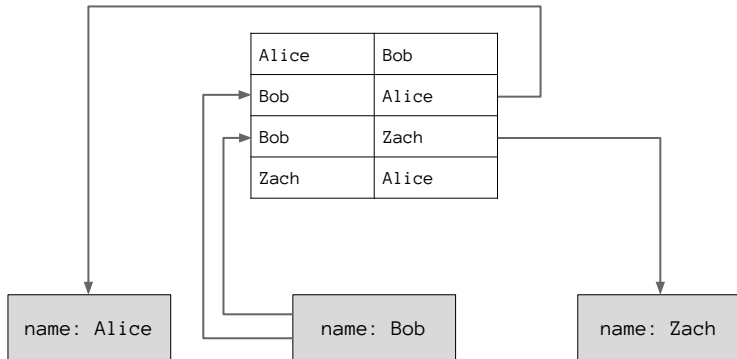
Con i database NoSQL Aggregate-Oriented



Modellare le relazioni

Con i database NoSQL Aggregate-Oriented

"Chi sono gli amici di Bob?"



Modellare le relazioni

Con i database NoSQL

Non soddisfacente:

- ▶ uso di “chiavi esterne”
- ▶ gestione relazioni a carico del software che usa il DB
- ▶ necessità di una struttura navigabile → indice globale → look-up
- ▶ query reciproche → scansione brute-force del DB

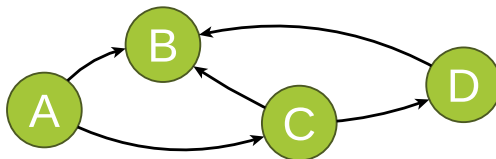


Cosa sono? Cos'è un grafo?

Si basano sulla teoria dei grafi

Grafo

struttura matematica formata da un insieme di nodi connessi da un insieme di archi



I **nodi** rappresentano le entità che si vogliono modellare
Gli **archi** rappresentano le relazioni tra di esse (*join precalcolati*)



II Labeled Property Graph Model

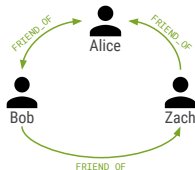
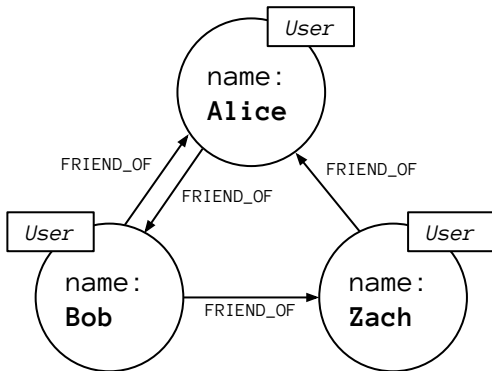
Definizione

- ▶ il grafo contiene nodi e archi (relazioni)
- ▶ I nodi posseggono delle proprietà (coppie chiave-valore)
- ▶ i nodi possono essere etichettati con una o più label
- ▶ le relazioni hanno un nome (un tipo), un verso ed hanno sempre un nodo di partenza ed uno di arrivo
- ▶ anche le relazioni possono avere delle proprietà (coppie chiave-valore)



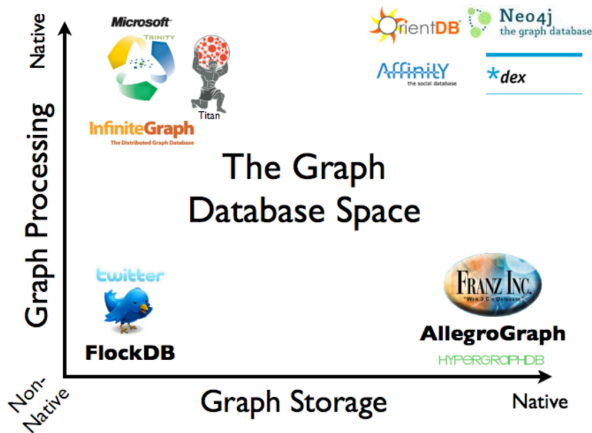
II Labeled Property Graph Model

Modellare una rete con il Labeled Property Graph Model



I Graph-DBMS

Panoramica dei Graph-DBMS



I Graph-DBMS

I Graph-DBMS "nativi"

- ▶ **Storage interno:** memorizzano i dati come grafo, senza usare DB relazionali o NoSQL di altro tipo
- ▶ **Process Engine:** usa la "*index-free-adjacency*", cioè sfrutta i nodi come indici locali piuttosto che usare un indice globale

Nei Graph Database *nativi* le query hanno un costo **proporzionale alla porzione di grafo esplorata**





- ▶ Graph-DBMS nativo
- ▶ sviluppato in Java dalla NeoTechnology
- ▶ implementa il Labeled Property Graph Model
- ▶ Open Source: Community (GPLv3), Enterprise (AGPLv3 per progetti Open Source)
- ▶ ultima release stabile: ~~2.2.6~~ 2.3



Neo4j Internals

Indici e Schema

Label ai nodi (anche a runtime)

Indici sulle proprietà dei nodi (eventually available)

È possibile definire dei **vincoli** (e.g. vincolo di unicità)

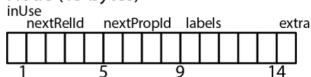
Label e vincoli costituiscono congiuntamente lo **schema** del grafo, che tuttavia un database Neo4j non è obbligato ad avere



Neo4j Internals

Memorizzazione dei dati

Node (15 bytes)



Relationship (34 bytes)



Record di dimensione fissata

Accesso ai record in $O(1)$: **ID * RecordSize**

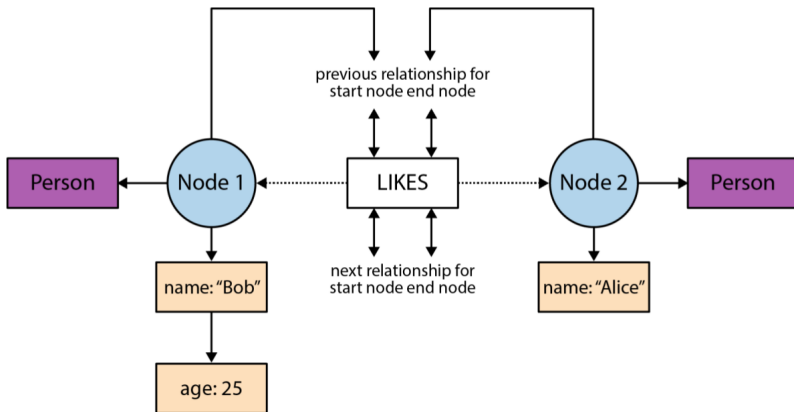
Relazioni come liste doppiamente concatenate (*relationship chain*)

Attraversa un arco **in entrambi i versi** in $O(1)$



Neo4j Internals

Memorizzazione dei dati



Neo4j Internals

Transazioni

Neo4j è **transazionale**, con rispetto delle proprietà **ACID**

Le transazioni in Neo4j sono semanticamente identiche a quelle dei database relazionali

Neo4j scrive i dati secondo la regola *Write Ahead Log*

Il livello di isolamento usato è *read committed*



Neo4j Internals

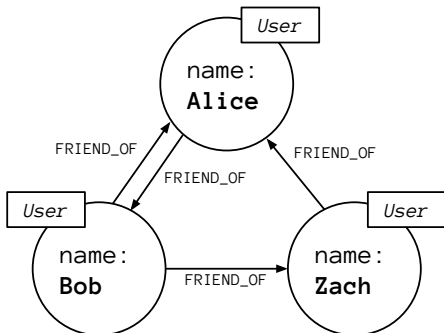
Cache

- ▶ **File Buffer Cache:** si occupa di memorizzare i file di storage nello stesso formato in cui sono salvati su memoria permanente; le scritture su memoria permanente vengono procrastinate
- ▶ **Object Cache** (*presente fino alla release 2.2.6*): memorizza nodi, relazioni e proprietà in un formato ottimizzato per navigare velocemente il grafo
 - ▶ **Reference Cache:** sfrutta il più possibile l'heap della JVM per memorizzare nodi e relazioni
 - ▶ **High-Performance Cache** (solo Enterprise): assegna una massima quantità di spazio nell'heap della JVM e rimuove gli oggetti che crescono oltre questo limite

Nella release 2.3 è stata introdotta una cache che opera fuori dall'heap.



Cypher

Pattern

```
(Zach) <-[:FRIEND_OF]-(Alice) <-[:FRIEND_OF]->
(Bob) -[:FRIEND_OF]->(Zach)
```



Cypher

Le clausole indispensabili

- ▶ MATCH specifica il pattern della rete che si sta cercando
- ▶ RETURN specifica quali dati devono essere restituiti dalla query
- ▶ WHERE applica un filtro alla ricerca



Cypher

Chi sono gli amici (degli amici) di Bob?

- ▶ *"Chi sono gli amici di Bob?"*

```
MATCH (n) -[r:FRIEND_OF] -> (m)
WHERE n.name = 'Bob'
RETURN n, r, m
```

- ▶ *"Chi sono gli amici **degli amici** di Bob?"*

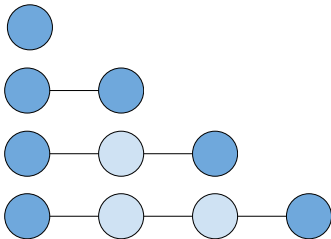
```
MATCH (n) -[r:FRIEND_OF*2] -> (m)
WHERE n.name = 'Bob'
RETURN n, r, m
```



Cypher

Andiamo in profondità!

Profondità di una ricerca: lunghezza di un percorso tra due nodi, calcolata in termini di numero di archi che li separano



Profondità zero

Profondità uno

Profondità due

Profondità tre



Cypher

Andiamo in profondità (variabile)!

- ▶ intervallo (da 1 a 3)

```
MATCH (n) - [r*1..3] - (m)
RETURN n, r, m
```

- ▶ intervallo (minimo 2)

```
MATCH (n) - [r*2..] - (m)
RETURN n, r, m
```

- ▶ intervallo (massimo 4)

```
MATCH (n) - [r*..4] - (m)
RETURN n, r, m
```



Cypher

Andiamo in profondità (variabile)!

- ▶ qualunque profondità

```
MATCH (n) -[r*] - (m)
RETURN n, r, m
```



Cypher

Ancora sui pattern

- ▶ Nei pattern ciò che non si vuole specificare può essere omissso:

`() -- ()`

`() --> ()`

`(n) -- ()`

`() --> (m)`

- ▶ Nella MATCH si possono specificare proprietà direttamente nella definizione del pattern:

```
MATCH (n {name: 'Alice'})
```

```
RETURN n
```



Cypher

Ancora sui pattern

- Possiamo specificare più pattern

```
MATCH (n {name: 'Bob'}) -[r1:FRIEND_OF] - (m)
MATCH (m) -[r2:FRIEND_OF] - (fof)
RETURN fof
```

- Possiamo specificare più path (e un pattern)

```
MATCH (n {name: 'Bob'}) -[r1:FRIEND_OF] - (m) ,
      (m) -[r2:FRIEND_OF] - (fof)
RETURN fof
```

equivalente a:

```
MATCH (n {name: 'Bob'}) -[r1:FRIEND_OF] - (m) -[
      r2:FRIEND_OF] - (fof)
RETURN fof
```



Cypher

Ancora sulla WHERE

- ▶ Espressioni regolari

```
MATCH (n)
WHERE n.name =~ 'Al.*'
RETURN n
```

- ▶ Filtrare in base a pattern

```
MATCH (n), (m)
WHERE n.name = "Zach" AND (n)-->(m)
RETURN n, m
```



Cypher

Ancora sulla WHERE

- ▶ Se un elemento esiste in una lista (Collection)

```
MATCH (n)
WHERE n.name IN ['Alice', 'Bob', 'Zach']
RETURN n
```



Cypher

Piping delle query: la clausola WITH

Con **WITH** si concatenano due query

Ad esempio, se vogliamo sapere qual è il nodo che ha più di due relazioni uscenti:

```
MATCH (n) -[r] -> (m)
WITH n, COUNT(r) AS NumberOfFriends
WHERE NumberOfFriends > 2
RETURN n, NumberOfFriends
```



Cypher

"Chiedo l'aiuto da casa" (cit.)

Con **USING** È possibile suggerire l'uso di indici o label

► Label hint

```
MATCH (n:User)
USING SCAN n:User
WHERE n.surname = 'Rossi'
RETURN n
```

► Index hint

```
MATCH (n:User)
USING INDEX n:User(surname)
WHERE n.surname IN ['Rossi']
RETURN n
```



Cypher

Creare un database Neo4j

Per creare nodi e relazioni si usa la clausola **CREATE**

Ad es:

- ▶ creare un nodo

```
CREATE (n)
```

- ▶ creare un nodo e assegnargli una label

```
CREATE (n:Person)
```

- ▶ creare un nodo e assegnargli molteplici label

```
CREATE (n:Person:Italian)
```



Cypher

Creare un database Neo4j

- ▶ creare un nodo, assegnargli una label e definire le proprietà

```
CREATE (n:Person { name: 'Donald',  
surname: 'Duck' })
```



Cypher

Creare un database Neo4j

- creare una relazione tra due nodi

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'NodeA' AND b.name = 'NodeB'
CREATE (a)-[r:RELTYPE]->(b)
RETURN r
```

- creare una relazione tra due nodi e definire le proprietà (ad es. un peso)

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'NodeA' AND b.name = 'NodeB'
CREATE (a)-[r:RELTYPE { weight: 0.5 }]->(b)
RETURN r
```



Cypher

Creare un database Neo4j

- ▶ creare un path (nodi e relazioni)

```
CREATE (a:User {name:'Qui'})-[r1:FRIEND_OF  
]->(b:User {name:'Quo'})<-[r2:FRIEND_OF]-  
(c:User {name:'Qua'})
```

Abbiamo creato una rete composta da 3 nodi e due relazioni (e definito le proprietà dei nodi)



Cypher

E se non volessimo creare duplicati?

La clausola **MERGE** è una sorta di combinazione tra **MATCH** e **CREATE**: crea un path solo se non esiste già

Es:

- ▶ Creare il nodo "Bob" solo se non presente

```
MERGE (n:User { name:'Bob' })  
RETURN n
```

- ▶ Creare relazione solo se non presente (**i nodi devono esistere**)

```
MATCH (n:User {name:'Zach'}) ,  
      (m:User {name:'Alice'})  
MERGE (n) <- [r:FRIEND_OF] - (m)  
RETURN r
```



Cypher

E se non volessimo creare duplicati?

- Creare relazione solo se non presente (**i nodi non presenti vengono creati**)

```
MERGE (n:User {name:'Zach'})  
MERGE (m:User {name:'Alice'})  
MERGE (n) <- [r:FRIEND_OF] - (m)  
RETURN n,r,m
```



Cypher

E se non volessimo creare duplicati?

- ▶ con **ON CREATE** si definiscono le proprietà *se il nodo deve essere creato*

```
MERGE (n:User { name:'Donald' })  
ON CREATE SET n.surname = 'Duck'  
RETURN n
```

- ▶ con **ON MATCH** si definiscono le proprietà *se il nodo è stato trovato*

```
MERGE (n:User { name:'Donald' })  
ON MATCH SET n.surname = 'Duck'  
RETURN n
```



Interagire col DB

Neo4j Tools

- ▶ Web Interface
- ▶ Neo4j Shell



Profiling delle query

Profiling delle query

- ▶ **EXPLAIN**: mostra il piano della query ma non la esegue
- ▶ **PROFILE**: mostra il piano della query, la esegue e tiene traccia del numero della quantità di dati coinvolta e delle interazioni col database



Bibliografia

- ▶ Robinson I., Webber J., Eifrem E., *Graph Databases*, O'Reilly, 2015
- ▶ neo4j.com
- ▶ Documentazione di Neo4j (neo4j.com/docs/stable/)



FINE