Sistemi NoSql
Document & Graph Model Databases

P. Rullo
Corso di Laurea in Informatica
Demacs - Unical

1 Dal relazionale al NoSql

Negli ultimi anni si è verificata una esplosione di nuove tipologie di applicazioni legate al web: reti sociali, dispositivi mobili, Internet of Things, ecc.. Aziende come Twitter, Facebook e Google raccolgono terabyte di dati utente ogni giorno, di tipologia variegata (testi, immagini, ecc.), di struttura variabile e generati ad alta velocità. In sintesi, queste le principali caratteristiche di tali dati:

1. Elevato volume (Big Data)
2. Varietà di tipi e formati (testi, immagini, suoni, ecc.)
3. Rapidità dei cambiamenti, anche strutturali.

Si tratta quindi di dati e applicazioni rispetto alle quali la tecnologia "tradizionale" delle basi di dati relazionali risulta inadeguata. E ciò per almeno due classi di motivi:

- limiti del potere espressivo del modello dei dati
- limiti architetturali legati alla natura centralizzata dei sistemi relazionali.

Proprietà tipiche del modello dei dati relazionale sono:

- *Semplicità*. Le relazioni sono insiemi di tuple di dati atomici. Non esistono costrutti idonei a rappresentare e manipolare testi, immagini, ecc.
- *Rigidità della struttura dei dati*. Ogni relazione ha uno schema predefinito, al quale si devono attenere tutte le tuple. Questa rigidità rende difficile uno sviluppo evolutivo della BD, cioè, l'adattamento della la BD ai cambiamenti e alle dinamiche che sono tipici di ambienti in evoluzione.
- *Normalizzazione*. Ogni relazione memorizza un singolo concetto. Ciò implica una parcellizzazione dell'informazione che limita il livello di ridondanza dei dati e, quindi, l'occupazione di memoria – fattore particolarmente importante quando la memoria era una risorsa poco disponibile. Vi è tuttavia un prezzo da pagare, cioè, l'inefficienza delle interrogazioni a causa delle numerose operazioni di join che sono necessarie per ricostruire informazioni complessive. Ciò è particolarmente grave in caso di grandi volumi di dati (Big Data).

Esempio. La metafora dell'automobile: una relazione per ogni componente – informazione parcellizzata.

Per ricostruire informazioni complessive sono necessarie molte operazioni di join per ricombinare opportunamente i singoli componenti

```
SELECT *
FROM Cars, Wheels, Seats, Brakes…
WHERE Cars.owner = "Paolo"
and Cars.id = Wheels.car_id  and Cars.id = Seats.car_id and Cars.id = Brakes.car_id  …
```
■

Nel caso di sistemi NoSql, non si può parlare di un unico modello dei dati. Esistono infatti quattro tipologie di sistemi, classificati in base al modello: sistemi a documenti, a grafi, basati su valori chiave, e sistemi a colonne. Tuttavia, a prescindere dallo specifico modello adottato, essi consentono di rappresentare oggetti

complessi e, in genere, sono privi di schemi fissi. La possibilità di rappresentare oggetti complessi, permettendo una organizzazione non parcellizzata dell'informazione, favorisce una esecuzione più efficiente delle interrogazioni. D'altra parte, l'assenza di schemi rigidi consente quella *flessibilità* necessaria per uno sviluppo veloce e iterativo.

Per quanto riguarda gli aspetti architetturali, notiamo che i sistemi relazionali sono in genere **centralizzati** (vengono cioè eseguiti su un singolo server), e ciò ha forti implicazioni su:

- Efficienza: il carico computazionale è concentrato su una singola macchina, e ciò non è in genere compatibile con i volumi dei Big Data
- Disponibilità dei dati: il database server costituisce un unico punto di attacco e, quindi, la disponibilità è ad alto rischio
- Costi di gestione: quando si superano le capacità di elaborazione/memorizzazione del server, è necessario procedere ad una costosa estensione verticale - CPU più potente, più RAM, ecc. (scalabilità verticale).

Di contro, i sistemi NoSQL sono in genere caratterizzati da una architettura distribuita, in quanto vengono eseguiti su macchine interconnesse, formanti un cluster. I dati vengono distribuiti tra le macchine all'interno del cluster. Ne conseguono

- Efficienza: le query possono essere esguite in parallelo
- Scalabilità orizzontale: quando si esaurisce la capacità di un cluster, è possibile semplicemente aggiungere una nuova macchina
- Disponibilità: grazie all'uso di cluster, molti sistemi NoSql supportano replicazione automatica dei dati, garantendo la disponibilità dei dati in caso di guasti o attacchi.

Per le suddette ragioni, i database NoSQL rappresentano la scelta migliore quando bisogna lavorare con un grande volume di dati senza una struttura impostata, e sono richieste elevate prestazioni e scalabilità orizzontale.

Di contro, i DB relazionali rimangono la soluzione valida laddove i dati sono strutturati e immutabili, e le proprietà ACID delle transazioni sono importanti. Si pensi al settore bancario, in cui le transazioni di denaro devono essere gestite correttamente, soprattutto in caso di trasferimento non riuscito: il fallimento di una transazione può costare infatti molto caro.

Nei prossimi paragrafi forniremo una panoramica dei Document Model Databases e dei Graph Model Databases (GMD).

2 Sistemi basati su "documenti"

Il "documento" è il costrutto di base dei document model DB (DMD). La struttura di un documento si basa sul principio che "pezzi" di informazione che concettualmente stanno assieme e vengono usati assieme, vanno memorizzati assieme. Mentre un sistema relazionale ottimizza i dati dal punto di vista dell'occupazione di memoria, un DMD ottimizza i dati rispetto al modo in cui vengono usati.

Un documento è un oggetto complesso che può incorporare altri oggetti. Nel raffronto con il modello relazionale, esso corrisponde ad una tupla relazionale. Un insieme di documenti forma una collezione (collection), che quindi corrisponde ad una relazione. Un insieme di collezioni forma un document DB.

NoSql	**Relazionale**
documento	tupla
Collezione: insieme di documenti	Relazione: insieme di tuple
BD: insieme di collezioni	BD: insieme di relazioni

I documenti non hanno uno schema fisso predefinito. Ogni oggetto può variare, in struttura, rispetto agli altri (vedi par. 2.3). Essi sono rappresentati utilizzando formati semi-strutturati come XML o JSON.

Come in ogni modello dei dati, un aspetto fondamentale è il modo in cui essi consentono di rappresentare le associazioni.

2.1 Rappresentazione di associazioni tramite embedding

Informazioni correlate vengono *incorporate* in una struttura documento-sottodocumento. Ad esempio, si considerino i seguenti dati relazionali:

- Person(pers_id, first_name, surname, city)
- Car(car_id, model, year, value, owner*)

Ogni persona può essere proprietaria di più automobili, ognuna delle quali ha un unico proprietario.

Usando il modello a documenti, possiamo rappresentare l'informazione della BD relazionale come segue:

Document persona {
_id : <personId1> // persId1 è una stringa univoca generata automaticamente
PersId: 1;
first_name: "Alvaro",
surname: "Ortega",
city: "Valencia",
cars: [
{ model: "Bentley",
year: 1973,
value: 100000},
{ model: "Rolls Royce",
year: 1965,
value: 330000},
]
}

Il suddetto documento rappresenta una persona unitamente alle auto di cui è proprietario. Come si può vedere, esso ha un identificatore unico "_id" che viene creato automaticamente.

In tale documento, l'associazione 1-n è rappresentata incorporando (*embedding*) nell'oggetto che sta dalla parte 1 (persona) tutti gli oggetti dalla parte n (auto). Infatti, il documento che rappresenta Alvaro incorpora come sotto-documenti tutte le auto di cui è proprietario. Si noti che l'attributo *cars* è di tipo array di oggetti $[c_1, \ldots, c_n]$, dove ogni oggetto c_i rappresenta una particolare auto.

Alternativamente, possiamo rappresentare l'associazione 1-n incorporando nell'oggetto dalla parte n (auto) il singolo oggetto (proprietario) dalla parte 1:

Document automobile {
_id: <carId1>
car_id: 101;
model: "Bentley"
year: 1973,
value: 100000,
owner:{
PersId: 1;
first_name: "Alvaro",
surname: "Ortega",
city: "Valencia"}
}

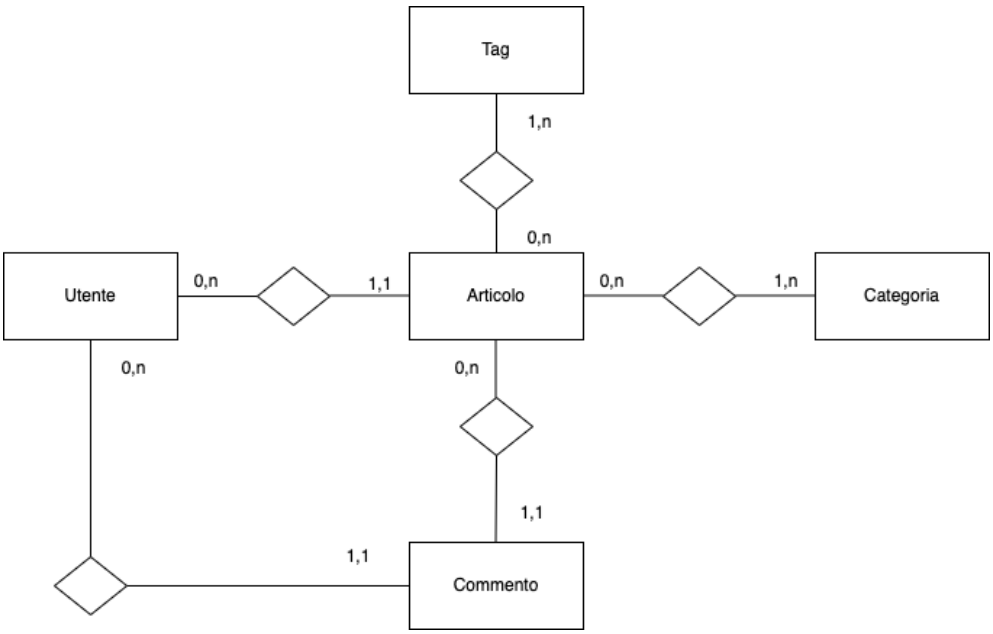
Si noti che, se Alvaro è proprietario di più automobili, i dati ad esso relativi sono duplicati tante volte per quante sono le auto di cui è proprietario.

In entrambi i suddetti esempi, gli oggetti tra loro associati sono incorporati (*embedded*) in un unico documento. La scelta di quale delle due rappresentazioni utilizzare dipende dal tipo di interrogazioni che vengono fatte sulla BD. Se l'accesso è per *persona*, si preferisce la prima rappresentazione, se è per *auto*, la seconda.

2.2 Rappresentazione di associazioni tramite referenziamento (referencing)

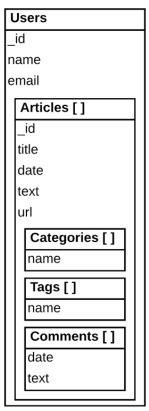
Oltre all'embedding, si possono anche usare meccanismi di *referencing* (simili alle chiavi secondarie del modello relazionale). In alcuni casi, infatti, ha senso memorizzare informazioni correlate in documenti separati, spesso in diverse collezioni o database.

Esempio 3. Il seguente schema modella un blog.

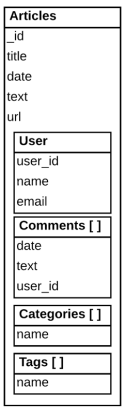


Come già visto, esistono più alternative nella costruzione di un modello a documenti.

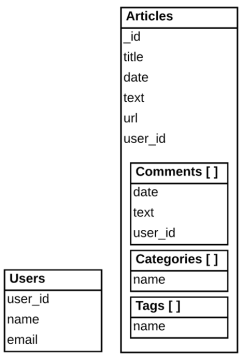
Solution A



Solution B



Solution C



Soluzione A. Rappresentazione per utente – conveniente quando si accede prevalentemente per utente

Document "utente" {
_id: <utenteId1>
codice: 101;
nome: "Paolo"
email: paolo@ppp.it
articoli: [
{ codice: xyz
titolo: "guerra in ucraina"
data: 3/3/2022
testo: "abc def …."
url: http ://myblog.com
categorie: [{politica,…}, {attualità, …}, ….]
tag: [{tagA, ….}, {tagB,…}, …]
commenti: [{date: 3/322021; text: "abc"; **utente: <utenteId2>**}, {date: 3/4/2022; text: "xyz", **utente: <utenteId5>**}]

{ codice: ppp
Titolo: …
…..
commenti: […] }
…
}

Si noti che nel suddetto documenti gli articoli di cui l'utente 101 è autore sono embedded. A sua volta, ogni articolo incorpora (1) un array di oggetti che rappresentano le categorie, (2) un array di oggetti che rappresentano i tag, e (3) un array di oggetti che rappresentano i commenti. Si noti che ad ogni commento è associato un utente per *referencing* (in grassetto).

Soluzione B. Rappresentazione per articolo – conveniente quando si accede prevalentemente per articolo

Document "articolo" {
_id: <articoloId5>
codice: xyz
titolo: "guerra in ucraina"
data: 3/3/2022
testo: "la guerra è un orrore"
url: http ://myblog.com
autore: {
identifier: 101
nome: "Paolo"
email: paolo@ppp.it
}
categorie: [{politica,…}, {attualità, …}, ….];
tag: [{tagA, ….}, {tagB,…}, …];
commenti: [{date: 3/3/2021; text: "abc", utente: <utenteId2>}, {date: 3/4/2022; text: "xyz", utente: <utenteId2>}]

Come nel caso precedente, la relazione articolo-autore è embedded, mentre quella commento-utente è referenced. Si noti che questa rappresentazione richiede la duplicazione delle informazioni sugli utenti.

Soluzione C. Rappresentazione per utente e per articolo

Document utente {

_id: <utenteId1>
codice: 101;
nome: “Paolo”
email: paolo@ppp.it
}

Document articolo {
_id: <articoloId500>
codice: xyz
titolo: “guerra in ucraina”
data: 3/3/2022
testo: “ppp … qqqq”
url: http ://myblog.com
autore: <utenteId1>
categorie: [{politica,…}, {attualità, …}, ….]
tag: [{tagA, ….}, {tagB,…}, …]
commenti: [{date: 3/3/2021; text: “abc”, utente: <utenteId2>}, {date: 3/4/2022; text: “lmn”, utente: <utenteId3>}]

In questo caso, tutte le relazioni sono di tipo *referencing*.

Una soluzione alternativa alla precedente è quella in cui ad ogni utente è associato un array di riferimenti agli articoli di cui è autore:

Document utente {
_id: <utenteId1>
codice: 101;
nome: “Paolo”
email: paolo@ppp.it
articoli: [articolo: <articoloId5>, articolo: <articoloId10>, ….]
}.

Si noti che potremmo definire un altro documento utente, ad esempio il documento con _id: <utenteId2>

Document utente {
_id: <utenteId2>
codice: 102;
nome: “Aldo”
email: aldo@ppp.it
articoli: [**articolo: <articoloId5>**, articolo: <articoloId99>, ….]
}

che contiene, tra i suoi articoli, un articolo di cui utenteId1 è autore, in tal modo alterando la natura dell’associazione (da uno-a-molti a molti-a-molti). Infatti, l’articolo <articoloId5> avrebbe due autori: <utenteId1> e <utenteId2>. Ciò è coerente col fatto che un DMD, a differenza di un DB relazionale, non deve essere conforme ad uno schema predefinito.

2.3 Embedding vs Referencing

Il vantaggio dell’embedding è quello di concentrare in un unico documento pezzi di informazione correlate che sono usate contestualmente. In tal modo è sufficiente un unico accesso per catturare tutta l’informazione cercata, piuttosto che fare più operazioni di join per ricostruirla. Ciò ha un notevole impatto positivo sull’efficienza delle interrogazioni.

Vi sono tuttavia anche svantaggi, legati al fatto che in genere l'embedding comporta duplicazione dell'informazione. È il caso, come abbiamo sopra osservato, del documento *automobile* che incorpora il proprietario – questo viene replicato tante volte per quante sono le auto di cui è proprietario. La ridondanza dei dati ha un impatto negativo sulle operazioni di aggiornamento – la modifica di un proprietario deve essere replicata più volte.

Il referencing evita i problemi di ridondanza e consistenza dei dati tipici dell'embedding ma, al contempo, ha un impatto negativo sulle prestazioni delle interrogazioni a causa della necessità di effettuare operazioni di join.

In genere, mentre per le associazioni uno-a-uno e uno-a-molti si usa l'embedding, le associazioni molti-a-molti vengono tipicamente rappresentate tramite referencing.

2.4 Flessibilità della struttura dei dati

A differenza dei database SQL, dove è necessario dichiarare lo schema di una tabella prima di inserire i dati, le collezioni non richiedono che i loro documenti debbano aderire ad un unico schema. Cioè, i documenti di una collezione non devono necessariamente avere lo stesso insieme di attributi. Ad esempio, il seguente utente ha l'attributo "città" non presente nel documento dell'utente _id: <utenteId2>

Document utente {
_id: <utenteId3>
codice: 103;
nome: "Maria"
email: maria@ppp.it;
città: "Roma";
articoli: [articolo_id: <artId5>, articolo_id: <artId99>, ….]
}

É possibile, tuttavia, forzare alcuni vincoli, per cui i documenti di una collezione devono condividere una struttura di base comune – un sottoinsieme di attributi (ad es., ogni persona deve avere un nome, un cognome, ecc.)

2.5 Creazione, interrogazione e modifica di documenti

Esistono vari sistemi NoSql document-oriented, tra i quali MongoDB e Couchebase Server. Questi non utilizzano uno standard comune, a differenza dei sistemi relazionali, che aderiscono allo standard SQL come linguaggio di definizione, interrogazione e manipolazione dei dati,.

Per questa ragione, nei seguenti esempi facciamo riferimento al linguaggio usato da MongDB.

Inserimento di documenti:

```
db.automobile.insertOne(
    {car_id: "101",
     model: "Bentley",
     year: "1973",
     value: "100.000",
     owner:{
            PersId: "1",
            first_name: "Alvaro",
            surname: "Ortega",
            city: "Valencia"}
     )
```

Interrogazioni di documenti:

1. identificativo delle auto di marca Bentely costruite prima del 1980

```
db.automobile.find(
{model: "Bentley", year:{$lt 1980}}       // selezione
{car_id:1}          // proiezione
)
```

equivalente a

```
Select car_id
From Automobile
Where model=Bentley and year < 1980
```

2. identificativo e modello delle auto di cui è proprietario Alvaro di Valencia

```
db.automobile.find(
{owner: {first_name: "Alvaro", city:"Valencia"}}          // selezione
{car_id:1, model:1}       // proiezione
)
```

equivalente a

```
Select car_id, model
From automobile as A, persona as P
Where A.owner=P.persId and P.first_name=alvaro and P.city=Valencia
```

3. Ricerca di tipo text:

- trova gli articoli che contengono la parola "guerra" oppure "orrore"

```
db.articoli.find({$text: {$search: guerra  orrore }})
```

- trova gli articoli che contengono la frase "la guerra è un orrore"

```
db.articoli.find({$text: {$search: "\" la guerra è un orrore \"" }})
```

- trova gli articoli che contengono "guerra" e non "orrore"

```
db.articoli.find({$text: {$search: guerra  -orrore }})
```

- trova gli articoli che contengono "war" e non "horror"

```
db.articoli.find({$text: {$search: war  -horror}, $language: english }})
```

la lingua determina il modo in cui il testo viene processato – eliminazione di stop words, lemmatizzazione,, ecc.

Modifica di documenti: incrementa del 10% il valore delle bentley

```
db.automobile.updateMany(
        {model: "bentley")              // filtro
        {$set: { value: value*1,1} }    //azione
)
```

Cancellazione di documenti: cancella tutte le auto costruite prima del 1980

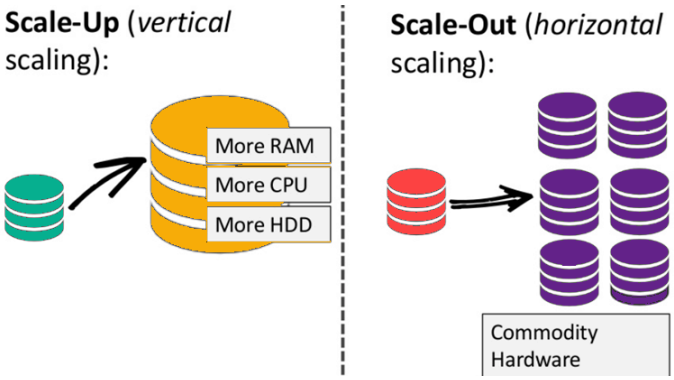
```
db.automobile.deleteMany(
        {year: {$lt: 1980}}             /filtro
)
```

2.6 Memorizzazione di documenti, sharding e scalabilità orizzontale

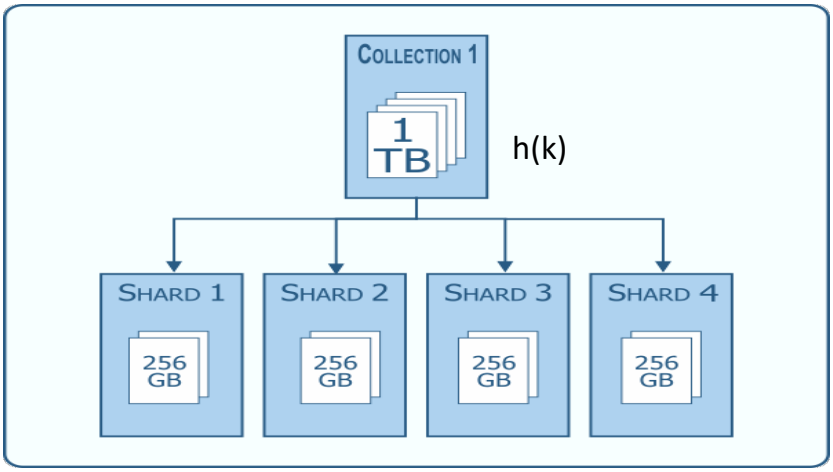
I database a documenti (e, in genere, i NoSQL) vengono eseguiti su un numero di macchine interconnesse, formanti un **cluster**. I documenti vengono distribuiti tra le macchine all'interno del cluster. I documenti memorizzati sulla stesso nodo del cluster formano un **shard**.

La memorizzazione distribuita dei documenti (**partizionamento orizzontale**) non penalizzia l'efficienza delle interrogazioni, essendo i documenti auto-consistenti grazie al meccanismo dell'embedding. Essi infatti contengono in genere tutti i dati necessari, limitando il numero di join da eseguire. In aggiunta, la distribuzione orizzontale incrementa il livello di parallelismo.

Grazie alla distribuzione orizzontale, i database NoSQL offrono quella che viene indicata come **scalabilità orizzontale**. Ciò significa che se si esaurisce la capacità del cluster attuale, è possibile semplicemente aggiungere una macchina al cluster. Queste macchine sono generalmente molto più economiche dei server usati per i DBMS relazionali.



Lo **sharding** (frammentazione) è il meccanismo di distribuzione dei documenti sulle diverse macchine del cluster. Una possibile tecnica di sharding è quella basata su funzioni hash, che utilizzano la chiave di un documento come argomento della funzione. Ovviamente esistono anche svantaggi derivanti dallo sharding, in particolare, quando è necessario eseguire join tra oggetti appartenenti a diversi shard.



L'architettura distribuita di un NoSQL DB ha un altro vantaggio: una parte dei dati può essere replicata e archiviata su più macchine, offrendo tolleranza ai guasti. Se una macchina si guasta, i dati su di essa sono presenti su un'altra macchina nel cluster e possono essere utilizzati senza che l'utente si accorga del guasto (ciò ovviamente non è possibile con i database SQL centralizzati).

2.7 Proprietà BASE

La replicazione dei dati in un database NoSql, cioè, la presenza di più copie degli stessi dati su macchine diverse, pone problemi di *consistenza* (o *coerenza*). Infatti, quando un aggiornamento viene effettuato su un dato, esso deve essere propagato a tutte le repliche, prima che altre transazioni possano leggerle. La *consistenza* in un sistema distribuito è la proprietà in base alla quale tutti gli utenti vedono gli stessi dati allo stesso tempo, indipendentemente dal nodo a cui si collegano.

Tuttavia, i sistemi NoSql in genere non garantiscono la piena coerenza, bensì la *coerenza finale*.

Le caratteristiche di un sistema NoSql sono riassunte dall'acronico BASE: **B**asically **A**vailable, **S**oft state, **E**ventual consistency:

- Basically Available (fondamentalmente disponibile). Nella eventualità che un guasto interrompa l'accesso a un nodo, il sistema continua comunque ad offrire il servizio.
- Soft state (stato leggero). lo stato del sistema può cambiare nel corso del tempo, anche senza intervento dell'utente. Cià a causa della eventual consistency
- Eventual consistency (coerenza finale). I NoSql non garantiscono la consistenza, ma solo la *enventual consistency*. Infatti, l'unico requisito che i database NoSQL hanno per quanto riguarda la coerenza è quello di richiedere che ad un certo punto in futuro i dati convergano verso uno stato coerente. Tuttavia, non viene fornita alcuna garanzia su quando ciò avverrà. Alla fine del processo di convergenza (che tutti i sistemi a consistenza finale devono implementare per assicurare l'evoluzione del sistema verso un medesimo stato), il sistema farà sì che tutti i nodi dispongano dell'ultima versione.

Si noti la differenza con i sistemi relazionali, dove vi è l'obbligo di coerenza immediata (proprietà ACID), che vieta l'esecuzione di un'operazione fino al completamento della precedente operazione e fino a quando la base di dati non abbia raggiunto uno stato coerente.

2.8 Sistemi commerciali

I seguenti sistemi si basano sul formato JSON per la rappresentazione dei documenti:

- MongoDB
- CouchDB
- OrientDB
- DocumentDB.

2.9 Conclusioni

I DMD sono caratterizzati dalle seguenti prorietà principali:

- Flessibilità: i database a documenti offrono schemi flessibili che consentono uno sviluppo più veloce e iterativo
- Potere espressivo: i database a documenti offrono un modello dei dati che consente di rappresentare in maniera semplice e naturale oggetti complessi, con una varietà di tipi di dati (non solo interi, ecc., ma anche testi, immagini, ecc.)
- Efficienza: mentre i DB relazionali sono ottimizzati nell'uso della memoria (normalizzazione dei dati), i database a documenti sono ottimizzati per consentire prestazioni elevate. Ciò grazie a meccanismi di rappresentazione compatta dell'informazione, in virtù della quale "pezzi" di informazione che vengono usati congiuntamente, vengono memorizzati insieme
- Scalabilità e disponibilità dei dati
- Modello BASE (contrapposto al modello ACID dei sistemi relazionali).

3 Sistemi basati su Grafi

I grafi sono strutture matematiche che consentono di rappresentare in maniera naturale ed afficace oggetti del mondo reale, e il modo in cui tali oggetti si relazionano.

Le basi di dati basate sul modello dei dati a grafo (Graph Database – GDB) sfruttano il potere espressivo dei grafi per rappresentare in maniera naturale il dominio applicativo. Infatti, oggi esistono molte applicazioni che si prestano ad essere realizzate su un GDB, quali reti sociali, sistemi di raccomandazione, applicazioni geospaziali, ecc.

Nel seguito, riportiamo una breve descrizione del modello dei dati e del linguaggio di interrogazione Cypher, implementato sul GDB system Neo4j.

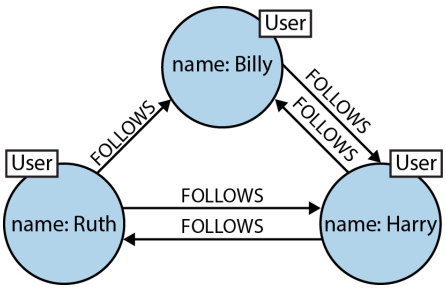
Gli esempi e le figure riportati nel prossimo paragrafo sono tratti dal testo Graph Databases (bit.ly/dl-neo4j).

3.1 Graph data model

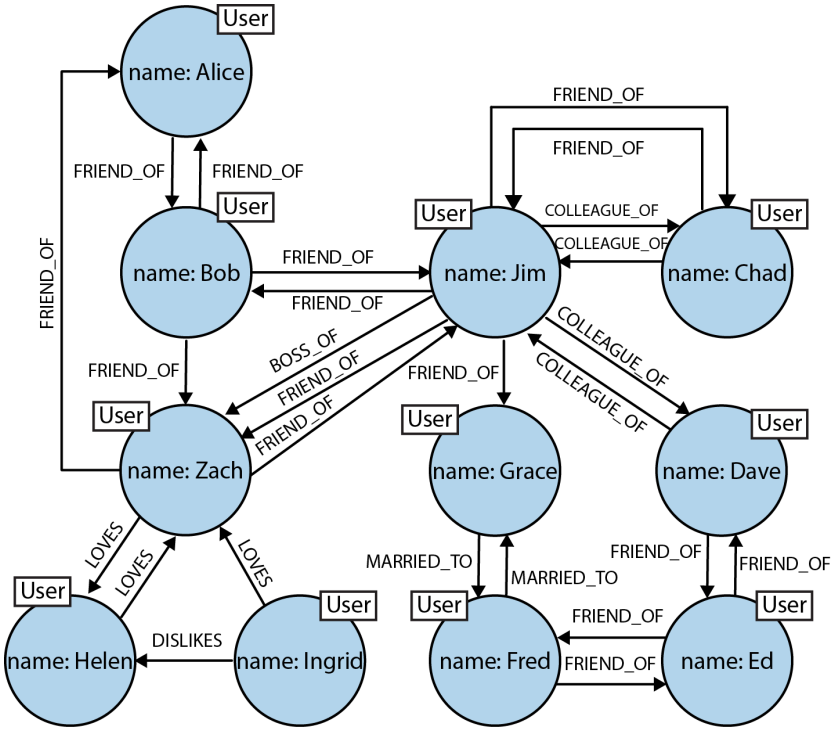
Un *graph DB* (GDB) è un insieme di dati organizzati secondo una struttura logica a grafo. Più precisamente, un GDB è un grafo orientato, cioè, una coppia <N,A>, dove N è un insieme di nodi (o vertici) e A è un insieme di archi orientati ed etichettati, cioè, triple del tipo <n,m,e>, dove n e m sono nodi ed e è una etichetta.

I nodi rappresentano oggetti del mondo reale, e gli archi relazione tra i nodi. Un grafo descrive quindi come gli oggetti si relazionano tra di loro.

Esempio 1. Il seguente grafo rappresenta un frammento di rete Twitter. Ogni nodo rappresenta uno specifico utente, e gli archi rappresentano la relazione “follows”. Quindi, ad esempio, Billy segue ed è seguito da Harry, ed è seguito, non ricambiato, da Ruth. ■



Esempio 2. Il seguente grafo descrive un insieme di utenti di una rete sociale ed i loro rapporti di amicizia, colleganza, amore, ecc.



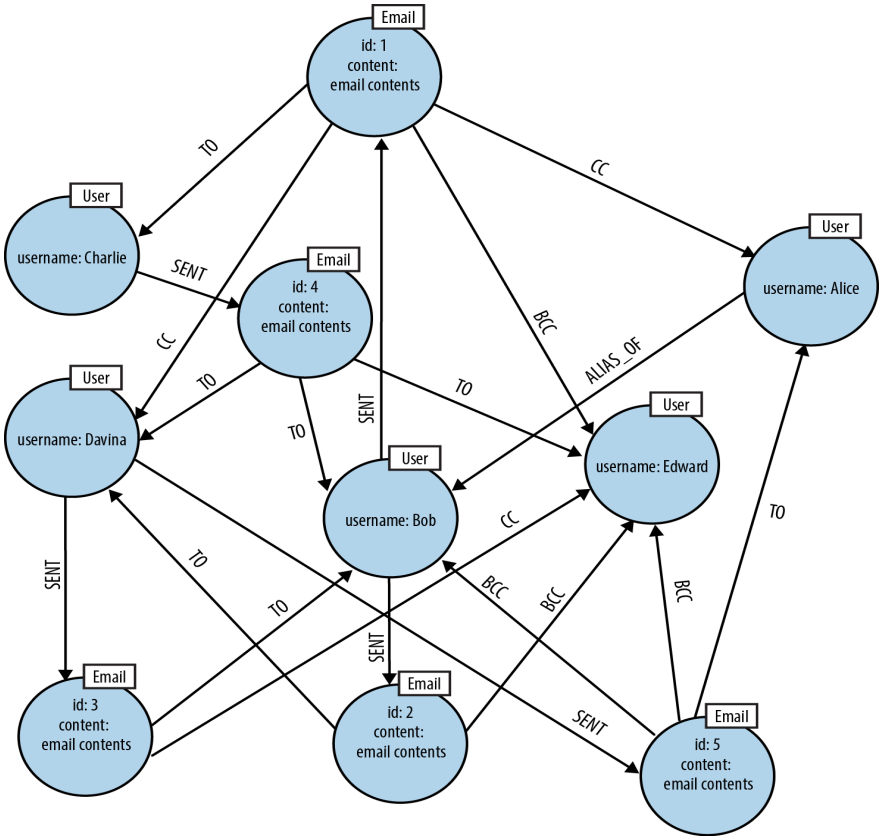
Come si può vedere, le relazioni tra le entità non presentano uniformità in tutto il dominio. In altri termini, la rete è strutturata in modo variabile, in quanto non tutti i nodi sono collegati allo stesso modo agli altri nodi. ■

Quindi non è possibile in linea di principio definire uno schema fisso che vale per tutti i "pezzi" della rete. La flessibilità del modello a grafo permette di aggiungere nuovi nodi e nuove relazioni senza compromettere la rete esistente.

Esempio 3. Il seguente grafo descrive un flusso di email. Ci sono due tipi di nodi, User (username) e Email, connessi attraverso le seguenti relazioni:

- user-email
  - SENT: per specificare il mittente di una email
  - TO: per specificare i destinatari diretti di una email
  - CC e BCC: per specificare i destinatari in copia carbone e copia carbone nascosta
- user-user: ALIAS-OF per specificare che un certo indirizzo utente è un alias di un altro ■

.



3.2 Creazione e interrogazione di un GDB: il linguaggio Cypher

Cypher è il linguaggio di Neo4j per la creazione, la ricerca e l'aggiornamento di un GDB. Nel seguito di questo paragrafo riportiamo alcuni esempi di creazione ed interogazione di un GDB.

3.2.1 Create

Con riferimento al grafo delle email (vedi Esempio 3), creiamo alcuni nod1:

```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'}),
       (alice)-[:ALIAS_OF]->(bob)

CREATE (email_1: email {id:'1', content:'email1 contents'}),
       (bob)-[:SENT]->(email_1),
       (email_1)-[:TO]->(charlie),
       (email_1)-[:CC]->(davina),
       (email_1)-[:CC]->(alice),
       (email_1)-[:BCC]->(edward);

CREATE (email_2: Email {id:'2', content:'email2 contents'}),

.....
```

Come si può vedere, ad ogni nodo N si associa un identificativo; ad es., al nodo User con username Bob viene associato l'identificativo 'bob', che viene usato nel nodo 'email_1' per specificare l'arco entrante

SENT – ‘bob’ ha inviato l’email con identificativo ‘email_1’. Si noti inoltre che un arco viene descritto in una sola CREATE – ad es., l’arco (bob)-[:SENT]->(email_1) è rappresentato solo nella CREATE di ‘email_1’ – e non anche nella CREATE di Bob.

3.2.2 Match

Le interrogazioni si basano sul costrutto MATCH. Essenzialmente, questo consente di descrivere un percorso sul grafo che da un nodo iniziale raggiunge altri nodi collegati attraverso relazioni. Di seguito riportiamo alcuni semplici esempi di interrogazioni.

1. Le seguenti interrogazioni fanno riferimento al grafo delle email (Esempio 3)

1.1. Email ricevute da bob come diretto destinatario

```
MATCH (x: email) - [:TO] ->(a: User)
WHERE a.username = Bob
RETURN x.id, x.content
```

In questo esempio abbiamo usato 3 clausole fondamentali di Cypher:

- **Match**: consente di definire un percorso sul grafo. In particolare, (email) - [:TO] -> (a: User) descrive il percorso che dalla generaica email “email” porta all’utente *a* con username “Bob” attraverso la relazione TO
- **Where**: specifica che il nodo utente è quello con username “Bob”
- **Return**: specifica i dati restituiti dalla query - equivale alla Select di SQL.

La suddetta interrogazione può anche essere riformulata eliminando la clausola where

```
MATCH (x: Email) - [:TO] - > (:User{name:{Bob}})
RETURN x.id, x.content
```

1.2. Mittenti delle email ricevute da bob

```
MATCH (a: User) <- [:TO] - (:email)  <- [:SENT] – (x:User)
WHERE a.username = Bob
RETURN x.username
```

1.3. Email inviate da Bob in cui Edward appare in BBC

```
MATCH (a: user{username: “Bob”})-[:SENT]-> (x:email)-[:BBC]->( b: User{username: “edward”})
RETURN x.id, x.content
```

1.4. Utenti che hanno ricevuto l’email con id = 1 come destinatari diretti, o in cc on in bcc

```
MATCH (a: email) - [:TO] -> (x:user) , (a) - [:CC] –> (y:user), (a) - [:BCC] –> (z:user)
WHERE a.id = “1”
RETURN x.username, y.username, z.username
```

1.5. destinatari diretti delle email inviate da Charlie

MATCH (a: user) - [:SENT] -> (:email) - [:TO] –> (b:user)
WHERE a.username = “Charlie”
RETURN b.username

1.6. Numero di destinatari diretti della email con id=1

MATCH (a: email) - [:TO] -> (x:user)
WHERE a.id = “1”
RETURN count(x)

Si noti che in questa query viene usata la funzione *count*, che conta il numero di istanze di x di utenti che ricevono l’email *a*.

1.7. Numero complessivo di destinatari della email id=1

MATCH (a: email) - [:TO] -> (d1: user) , (a) - [:CC] –> (d2: user), (a) - [:BCC] –> (d3:user)
WHERE a.id = “1”
RETURN count(d1)+count(d2)+count(d3)

1.8. Utenti che hanno spedito una sola email

MATCH (a: user) -> [x:SENT]
WHERE count(x) = 1
RETURN a.username

In tal caso, la funzione *count* conteggia il numero di archi di tipo SENT uscenti dal nodo a.

2. Le seguenti query fanno riferimento al grafo della rete sociale (Esempio 2)

2.1. Amici di livello 3 di alice

MATCH (a: user)-[:FRIEND-OF]-> (b:user)-[: FRIEND-OF]->(c:user))-[: FRIEND-OF]->(d:user)
WHERE a.username = “Alice”
RETURN d.name

oppure
MATCH (a: user)-[:FRIEND-OF*1..3]-> (b:user)
WHERE a.username = “Alice”
RETURN b.name

Si noti la sintassi [:FRIEND-OF*1..3] con cui si specifica la lunghezza del percorso, definito dalla relazione FRIEND-OF, che parte dal nodo user *Alice*.

2.2. Amici di Bob (di primo livello) che ricambiano l’amicizia

MATCH (a: user)-[:FRIEND-OF]-> (b:user)-[: FRIEND-OF]->(a)
WHERE a.username = “Bob”

RETURN b.name

2.3. Amici di bob che NON ricambiano l'amicizia

MATCH (a: user)-[:FRIEND-OF]-> (b:user)
WHERE a.username = "Bob" and NOT ((b) -[: FRIEND-OF]->(a))
RETURN b.name

3. Le seguenti query fanno riferimento al grafo di Twitter (Esempio 1)

3.1. Utenti che non hanno follower

MATCH (a: user)
WHERE NOT ((a) <- [:FOLLOWS])
RETURN a.username

oppure

MATCH (a: user) <- [x:FOLLOWS]
WHERE count(x)=0
RETURN a.username

3.2. Utenti che hanno almeno 2 follower e non sono follower di nessuno

MATCH (a: user) <- [x:FOLLOWS], (a) -> [y:FOLLOWS]
WHERE count(x) $\geq$ 2 and count(y)=0
RETURN a.username

3.3 Memorizzazione di grafi ed efficienza di un GDB

Alcuni GDB usano strutture di memeorizzazione native, cioè, ottimizzate per memorizzare e manipolare grafi. Neo4J è uno di questi. Altri GDB, invece, mappano i grafi nel modello relazionale. Ad esempio, usando il modello relazionale, la relazione FRIEND_OF (di tipo molti-a-molti) dell'esempio 2, viene rappresentata come segue:

User(<u>username</u>, …)
FriendOf(user1, user2)

dove user1 e user 2 sono chiavi secondarie. E' evidente che una tale rappresentazione penalizza molte interrogazioni dal punto di vista dell'efficienza. Ad esempio, per calcolare gli amici di livello n di un dato utente (vedi interrogazione 2.1, con n=3), con una rappresentazione relazionale è necessario eseguire n-1 join della relazione FriendOf con sé stessa. Considerato che la cardinalità di tale relazione è, in un caso reale, molto elevata (molti milioni di tuple), è facile immaginare che l'interrogazione avrà tempi di risposta molto elevati.

La seguente tabella confronta i risultati della ricerca, fino ad una profondità massima di cinque, ottenuti da un GDB con *native graph storage* (in particolare, Neo4j) e un DB relazionale, per un social network contenente 1.000.000 di persone, ciascuno con circa 50 amici.

Profondità	Tempo RDB	Tempo GDB	Nodi restituiti
2	0.016	0.01	2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Ricerca non terminata	2.132	~800,000

Come si può vedere, man mano che aumenta la profondità della ricerca, i tempi di un RDB crescono esponenzialmente (causa operazioni di join), a differenza dei tempi del GDB, che mostrano una crescita molto contenuta.

3.4 Sistemi commerciali

- Neo4J
- Giraph.

3.5 Conclusioni

In molti domini applicativi la struttura dei dati è naturalmente rappresentata attraverso un grafo. Ad esempio, il DB di una rete sociale è intrinsecamente una grafo.

I GDB che supportano nativamente meccanismi (strutture dati) idonei alla memorizzazione di grafi forniscono prestazioni molto elevate, come dismostrato nel pragrafo precedente.

I GDB (e, più in generale, i No Sql DB) presentano anche il vantaggio della *flessibilità*: non avendo infatti uno schema predefinito, è possibile aggiungere dati di tipo e struttura diversi, senza compromettere quelli già memorizzati. Ciò rende i GDB molto convenienti per quelle tipologie di applicazioni, oggi sempre più diffuse, in cui i dati e/o i requisiti delle applicazioni sono mutevoli. Con i database a grafo, infatti, non è necessario modellare in modo esaustivo il dominio in anticipo, e poi via via rimodellarlo per tenere conto della evoluzione dei dati; al contrario, è possibile estendere la struttura esistente senza inficiare l'esistente.