



# Basi di dati

**Maurizio Lenzerini**

***Dipartimento di Informatica e Sistemistica “Antonio Ruberti”  
Università di Roma “La Sapienza”***

Anno Accademico 2023/2024

<http://www.dis.uniroma1.it/~lenzerin/home/?q=node/44>



## 7. Modelli e sistemi NoSQL

1. Introduzione ai sistemi di basi di dati NoSQL
2. Document-oriented databases
3. Graph-databases

*Una parte di queste slides è basata sul materiale didattico preparato dal Prof. Marco Di Felice dell'Università di Bologna*



## 7. Modelli e sistemi NoSQL

1. Introduzione ai sistemi di basi di dati NoSQL
2. Document-oriented databases
3. Graph-databases



# Introduzione ai sistemi di basi di dati NoSQL

**NoSQL** → Movimento che promuove l'adozione di DMBS non basati sul modello relazionale

- ✧ Il termine NOSQL appare per la prima volta in una pubblicazione del 1998
- ✧ Oggi, il termine NOSQL viene usato per lo più nell'accezione **Not Only SQL**

*“Next generation databases mostly addressing some of the points: being non-relational, distributed, open source and horizontally scalable”*

(definizione da <http://nosql-database.org/>)



# Introduzione ai sistemi di basi di dati NoSQL

## PROPRIETÀ dei SISTEMI NO-SQL

- ✧ Database **distribuiti**
- ✧ Sono **scalabili** orizzontalmente
- ✧ Sono in grado di gestire **enormi moli di dati**
- ✧ Supportano le **repliche** dei dati
- ✧ Strumenti generalmente **open-source**
- ✧ **NON dispongono necessariamente di schema**
- ✧ Tipicamente NON supportano operazioni di **join**
- ✧ Tipicamente NON supportano le proprietà **ACID** delle transazioni



# Introduzione ai sistemi di basi di dati NoSQL

Il termine NoSQL identifica **una moltitudine di DBMS**, basati su **modelli logici differenti**:

- **chiave/valore**
- **document-oriented**
- **column-oriented**
- **graph-oriented**



# Introduzione ai sistemi di basi di dati NoSQL

Il termine NoSQL identifica **una moltitudine di DBMS**, basati su **modelli logici differenti**:

- **chiave/valore**
- **document-oriented**
- **column-oriented**
- **graph-oriented**



## 7. Modelli e sistemi NoSQL

1. Introduzione ai sistemi di basi di dati NoSQL
2. Document-oriented databases
3. Graph-databases





# Un sistema NoSQL document-oriented: MongoDB

- ✧ Gestione di dati **eterogenei** e **complessi** (semi-strutturati)
- ✧ Scalabili **orizzontalmente**, supporto per partizionamento (*sharding*) dei dati in sistemi distribuiti
- ✧ **Documento** → insieme di coppie chiave/valore (JSON)
- ✧ Forniscono **funzionalità** per aggregazione/analisi dei dati (MapReduce)



# Basi di dati MongoDB

- ✧ **MongoDB** (<https://www.mongodb.org>)
- ✧ Database organizzato in **collezioni**; una collezione contiene una lista di **documenti**. Ogni documento è un insieme non rigido di **campi** con relativi valori.
- ✧ Analogia:

MongoDB	Modello Relazionale
<i>Collezione</i>	Tabella
<i>Documento</i>	Riga
<i>Campo</i>	Colonna di una riga



## Esempio di collezione in MongoDB

```
{  "company": "DeLorean",  
  "vehicle": "DMC-12",  
  "year": 1981,  
  "max-speed": "88 mph",  
  "notes": ["Può viaggiare nel tempo", "Contiene flusso canalizzatore"]  
}
```

campo/valore

```
{  "company": "Fiat",  
  "vehicle": "Panda 4x4",  
  "year": 1986,  
  "max-speed": "70 km/h (forse)",  
  "colour": "verde acqua"  
}
```

documento

```
{  "vehicle": "Carro armato",  
  "year": 1942,  
  "max-speed": "10 km/h (in discesa)",  
  "gun": "57 mm"  
}
```

collezione



# JSON

## MongoDB è basato su documenti JSON

- ✧ **JSON** è un formato per lo scambio di dati tra applicazioni.
- ✧ Documenti JSON facilmente interpretabili da macchine; molti parser disponibili.
- ✧ I dati di un documento sono racchiusi tra { }.
- ✧ Ogni dato assume la forma → **nomeCampo: valore**

*Esempi di documento:*

- { nome: "mario" }
- { nome: "mario", cognome: "rossi" }

## I valori in JSON

✧ Valore → **Numero**, intero o reale

- { nome: "mario", eta: 15, punti: 13.45 }

✧ Valore → **Stringa**, tra apici

- { nome: "mario", cognome: "rossi" }

✧ Valore → **Booleano**, true o false

- { nome: "mario", impiegato: true }

✧ Valore → **Array**, tra parentesi quadre

- { nome: "mario", cap: [ "134", "042" ] }

✧ Valore → **a sua volta documento**, tra graffe

- { nome: "mario", indirizzo: {citta: "bologna", via: "po", numero: 3} }

# Documenti in JSON

Esempio  
di documento

```
{  nome: "Mario",
  cognome: "Rossi",
  eta: 45,
  impiegato: false,
  salario: 1205.50,
  telefono: ["0243434", "064334343"],
  ufficio: [ {nome: "A", via: Zamboni, numero: 7},
              {nome: "B", via: Irnerio, numero: 49}
            ]
}
```

NomeCampo

Valore



# Server MongoDB

## ✧ Avvio del **server** e della **shell**

- `mongod` // demone in ascolto sulla porta
- `mongo` // shell client

## ✧ Comandi della shell di **MongoDB**

Comando	Azione
<code>show dbs</code>	Mostra DB disponibili
<code>show collections</code>	Mostra le collezioni del db
<code>show users</code>	Mostra gli utenti del sistema
<code>show rules</code>	Mostra il sistema di accessi
<code>show logs</code>	Mostra i log disponibili



# Un sistema NoSQL: MongoDB

## ✧ **Utilizzo/Creazione** di un DB

- use `provaDB`

## ✧ Creazione di una **collezione** (vuota)

- `db.createCollection("circoli")`





# Un sistema NoSQL: MongoDB

✧ **Documento** in MongoDB → oggetto **JSON** con **\_id**

```
{ _id: val, campo1: valore1, campo2: valore2,  
  campo3:valore3, ....}
```

## Esempi

```
{ _id: 100, nome: "Marco", cognome: "Rossi",  
  eta: 22, data: new Date(1997,6,2,12,30) }
```

```
{ _id: 200, nome: "Gianni", cognome: "Bosi",  
  eta: 30, data: new Date(1999,7,12,10,00) }
```



## Un sistema NoSQL: MongoDB

- ✧ Nella stessa collezione, è possibile inserire documenti con **strutture campo/valore** differenti.

### COLLEZIONE ANAGRAFICA

DOC1	Marco		22	
DOC2	Massimo	Rossi		
DOC3	Maria	Bianchi	24	1/5/1990

- ✧ È molto più difficile e laborioso rappresentare strutture simili nel modello relazionale: richiede di prevedere a priori le varie (molte?) possibilità e si basa sull'uso massiccio di valori nulli



# Un sistema NoSQL: MongoDB

✧ Inserimento di un documento in una collezione:

```
db.NOMECOLLEZIONE.insert(DOCUMENTO)
```

```
db.anagrafica.insert({nome: "Marco", cognome:  
"Rossi", eta: 22})
```

```
db.anagrafica.insert({cognome: "Rossi", eta: 22,  
domicilio:["Roma", "Bologna"]})
```

```
db.anagrafica.insert({nome: "Maria", eta: 25})
```

# Un sistema NoSQL: MongoDB

- ✧ Ogni documento contiene un campo `_id`, che corrisponde alla **chiave primaria** nella collezione.
- ✧ Il campo `_id` può essere definito **esplicitamente**, o viene aggiunto in maniera implicita da MongoDB.

```
db.anagrafica.insert({_id: 1, name: "Marco",  
                      cognome: "Rossi", eta: 22})
```

ID ESPLICITO

```
db.anagrafica.insert({name: "Marco", cognome:  
"Rossi", ruolo: 'Manager'})
```

ID IMPLICITO

Il campo `_id` ed il suo valore vengono aggiunti dal sistema



## Un sistema NoSQL: MongoDB

✧ Rimozione di documenti da una collezione

`db.NOMECOLLEZIONE.delete({})`

→ Svuota la collezione, eliminando tutti gli elementi. Ad esempio:

`db.anagrafica.delete({})`

→ svuota la collezione anagrafica

`db.NOMECOLLEZIONE.delete(SELETTORE)`

→ Elimina dalla collezione tutti i documenti che soddisfano il selettore (vedi dopo)



## Un sistema NoSQL: MongoDB

✧ **SELETTORE** → Documento JSON che specifica le caratteristiche dei documenti che vogliamo selezionare

and

{campo1: valore1, campo2:valore2, ... }

Esempio: {name: "Marco", cognome: "Rossi"}

{**\$or** [{campo1: valore1}, {campo2:valore2}]}

Esempio: {**\$or**: [{name: "Marco"}, {cognome: "Rossi"}]}

\$gt, \$lt, \$gte, \$lte, \$ne, \$in

{ campo1: valore1, campo2:{**\$OP** : VALORE o VALORI} }

Esempio: { name: "Marco", eta: {\$gt:30} }



# Un sistema NoSQL: MongoDB

✧ Aggiornamento di documento in una collezione:

```
db.NOMECOLLEZIONE.update(SELETTORE,DOCUMENTO)
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $SET: CAMPI })
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $PUSH: CAMPI })
```

Esempio:

```
db.anagrafica.update({nome: "Mario"},{eta:45})
```

→ Nel primo documento D in cui c'è la coppia nome: "Mario", si sostituisce D con il documento {eta:45}.



# Un sistema NoSQL: MongoDB

✧ Aggiornamento di documento in una collezione:

```
db.NOMECOLLEZIONE.update(SELETTORE,DOCUMENTO)
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $SET: CAMPI })
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $PUSH: CAMPI })
```

Esempio:

```
db.anagrafica.update({nome: "Mario"},{$set:{eta:45}})
```

→ Nel primo documento in cui c'è la coppia nome: "Mario", si pone l'età pari a 45 o si aggiunge il campo età con valore 45.





# Un sistema NoSQL: MongoDB

✧ Aggiornamento di documento in una collezione:

```
db.NOMECOLLEZIONE.update(SELETTORE,CAMPI)
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $SET: CAMPI })
```

```
db.NOMECOLLEZIONE.update(SELETTORE,{ $PUSH: CAMPI })
```

appende il valore alla fine di un array

Esempio:

```
db.anagrafica.update({nome: "Mario"},{$push:{eta:45}})
```

→ Nel primo documento in cui c'è la coppia nome: "Mario", si appende il valore 45 all'array che è il valore di eta o si aggiunge il campo eta ed il valore [45] (di tipo array).

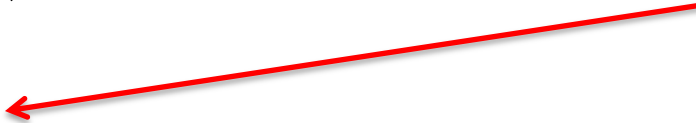


# Un sistema NoSQL: MongoDB

✧ Aggiornamento di documento in una collezione:

```
db.NOMECOLLEZIONE.update(SELETTORE,CAMPI,OPZIONI)
```

**{upsert:<boolean>, multi:<boolean>}**



Esempio:

```
db.anagrafica.update({nome: "Mario"},{$set:{eta:45}},{multi:true})
```

→ In tutti i documenti in cui c'è la coppia nome: "Mario", si modifica il valore del campo età.



# Un sistema NoSQL: MongoDB

- ✧ Il costrutto di `find` consente di definire delle operazioni di ricerca (**query**) su una collezione.
- ✧ `db.nomeCollezione.find()` → restituisce tutti i documenti presenti nella collezione.
- ✧ `db.nomeCollezione.find(SELETTORE)` → restituisce tutti i documenti, i cui campi rispettano la condizione espressa nella query.
- ✧ `db.nomeCollezione.find(SELETTORE, PROJECTION)` → restituisce tutti i campi `projection` dei documenti i cui campi rispettano la condizione espressa nella query



# Un sistema NoSQL: MongoDB

✧ Esempi del costrutto di find

✧ `db.anagrafica.find()`

→ `SELECT * FROM anagrafica`

✧ `db.anagrafica.find({nome: "Mario", eta:30})`

→ `SELECT * FROM anagrafica WHERE  
(nome= "Mario") AND (ETA=30)`

**AND**



✧ `db.anagrafica.find({nome: "Mario"},{eta: 1})`

→ `SELECT _ID,ETA FROM anagrafica  
WHERE nome= "Mario"`

**Campo selezionato**





# Un sistema NoSQL: MongoDB

## ✧ Esempi del costrutto di find

```
db.anagrafica.find({$or:[{nome: "Mario"},{eta:56}]}  
                  {eta:1}))
```

```
→ SELECT _ID, ETA  
   FROM anagrafica  
   WHERE ((nome= "Mario") OR (ETA=56))
```

```
db.anagrafica.find({eta:{$gte:60}})
```

```
→ SELECT *  
   FROM ANAGRAFICA  
   WHERE (eta >= 60)
```



# Un sistema NoSQL: MongoDB

✧ Esempio del costrutto di find

```
db.anagrafica.find({eta: {$exists : true}})
```

→ SELECT \*

FROM anagrafica

WHERE eta is not null



# Un sistema NoSQL: MongoDB

✧ Operatori di **ordinamento** sulla find

✧ `db.nomeCollezione.find(...).sort(CAMPO/CAMPI)`

1=Ordinamento crescente, -1=Ordinamento decrescente

Esempio:

✧ `db.anagrafica.find({nome:'Mario'}).sort({eta:1})`

→ `SELECT *`  
`FROM anagrafica`  
`WHERE (nome= "Mario")`  
`ORDER BY ETA;`



# Un sistema NoSQL: MongoDB

✧ Operatori di **conteggio** sulla `find`

✧ `db.nomeCollezione.find(...).count()`

Esempio:

✧ `db.anagrafica.find({nome: "Mario"}).count()`

```
SELECT COUNT(*)  
FROM anagrafica  
WHERE (nome= "Mario")
```





## Un sistema NoSQL: MongoDB

✧ Operatori di **filtro duplicati** sulla find

✧ `db.nomeCollezione.distinct([CAMPO],SELETTORE)`

✧ `db.anagrafica.distinct("eta",{nome: "Mario"})`

```
SELECT DISTINCT(eta)
FROM anagrafica
WHERE nome= "Mario"
```



## Un sistema NoSQL: MongoDB

- ✧ E' possibile raccogliere i comandi **mongoDB** in uno script (**linguaggio JavaScript**)

```
mongodb myfile.js
```

- ✧ Le prime istruzioni dello script contengono la connessione al server MongoDB e al database su cui si vuole operare:

```
conn = new Mongo();  
db = conn.getDB("tennis");
```



## Un sistema NoSQL: MongoDB

- ✧ Il file di script può contenere **costrutti iterativi e/o di selezione**:

```
while (condizione) { LISTACOMANDI}  
if (condizione) { LISTACOMANDI }  
else { LISTACOMANDI }
```

- ✧ I **cursori** vengono usati per scorrere il risultato di una query.

```
cursor = db.collection.find(...);  
while (cursor.hasNext()) {  
    printjson( cursor.next() );  
}
```



## Un sistema NoSQL: MongoDB

- ✧ Supponiamo di dover rappresentare **correlazioni tra collezioni** in MongoDB.

*Es. Circoli Tennis e Soci dei Circoli*

- ✧ MongoDB non mette a disposizione i costrutti di vincoli di integrità referenziale tra collezioni/tabelle.
- ✧ In MongoDB il join tra collezioni è molto difficoltoso!




## Un sistema NoSQL: MongoDB

✧ Le **correlazioni** possono essere espresse mediante **campi** **\_id** replicati tra più collezioni...

```
db.circoli.insert({_id: "120"},  
                  nome: "Nettuno"})
```

```
db.soci.insert({nome: "Mario", cognome: "Rossi"  
               circolo: "120"})
```





## Un sistema NoSQL: MongoDB

- ✧ Le associazioni *uno-a-molti*, o *molti-a-molti*, tra documenti di diverse collezioni possono essere rappresentate sfruttando il fatto che in MongoDB il valore di un campo può essere anche un **array**, o una struttura complessa (es. documento annidato).

```
db.soci.insert({nome: "Mario", cognome: "Rossi",  
               circolo: ["120", "150", "200"]})
```



## Un sistema NoSQL: MongoDB

✧ Problema: come scrivere la query che restituisce nome e cognome dei soci che partecipano a circoli situati a Bologna, **senza usare il JOIN?**

✧ **Soluzione 1: usare 2 query nello stesso script!**

*Esempio: nome e cognome dei soci dei circoli di Bologna:*

```
db.circoli.find({luogo: "Bologna"},{})
```

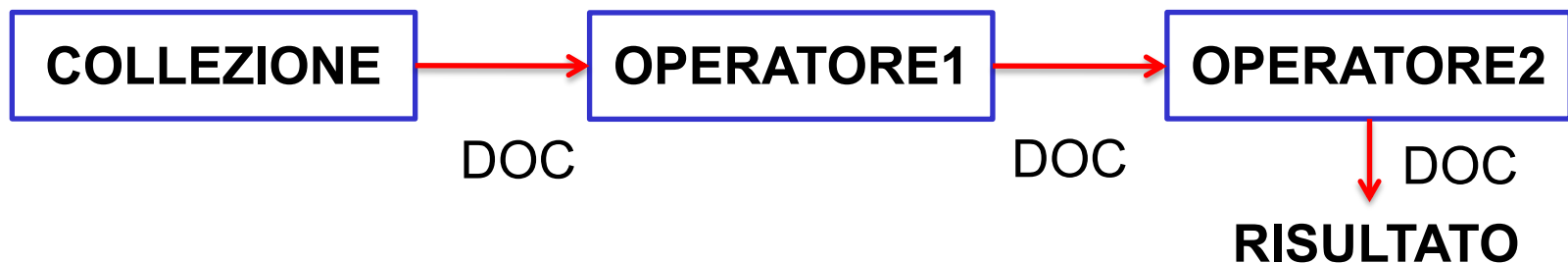
```
>> {_id:"432"}
```

```
db.soci.find({circolo:"432"},{nome:1,  
                                cognome:1})
```



## Un sistema NoSQL: MongoDB

- ✧ Soluzione 2: usare operatori di **aggregazione** - **funzione «aggregate»**
- ✧ Aggregate consente di implementare una **pipeline** di operazioni da eseguire sulla base di dati.
- ✧ Ad ogni passo della pipeline, vengono eseguite operazioni che prendono in input dei documenti JSON e producono in output documenti JSON.







# Un sistema NoSQL: MongoDB

## Orders

```
{"_id" : 1.0, "item" : 1.0, "price" : 49.99, "qty" : 1.0}  
{"_id" : 2.0, "item" : 1.0, "price" : 49.99, "qty" : 3.0}  
{"_id" : 3.0, "item" : 2.0, "price" : 99.99, "qty" : 1.0}
```

## Products

```
{"_id" : 1.0, "name" : "Kick Scooter", "price" : 49.99}  
{"_id" : 2.0, "name" : "Power Jeep", "price" : 99.99}  
{"_id" : 3.0, "name" : "Power Wheel Lightning McQueen", "price" : 199.9}
```

Vogliamo il **join** tra Orders e Products sulla condizione Orders.item = Products:\_id, per combinare

- il primo documento di Orders con il primo di Products
- il secondo documento di Orders con il primo di Products
- il terzo documento di Orders con il secondo di Products



# Un sistema NoSQL: MongoDB

```
db.orders.aggregate(  
  { $lookup:  
    {  
      from: "products",  
      localField: "item",  
      foreignField: "_id",  
      as: "ordered_product"  
    }  
  }  
)
```

JOIN

uguaglianza

nome nuovo  
campo



# Un sistema NoSQL: MongoDB

## Orders

```
{"_id" : 1.0, "item" : 1.0, "price" : 49.99, "qty" : 1.0}  
{"_id" : 2.0, "item" : 1.0, "price" : 49.99, "qty" : 3.0}  
{"_id" : 3.0, "item" : 2.0, "price" : 99.99, "qty" : 1.0}
```

## Products

```
{"_id" : 1.0, "name" : "Kick Scooter", "price" : 49.99}  
{"_id" : 2.0, "name" : "Power Jeep", "price" : 99.99}  
{"_id" : 3.0, "name" : "Power Wheel Lightning McQueen", "price" : 199.9}
```

## Risultato del join

```
{"_id" : 1.0, "item" : 1.0, "price" : 49.99, "qty" : 1.0,  
  "ordered_product" : [ {"_id" : 1.0, "name" : "Kick Scooter", "price" : 49.99} ]}  
{"_id" : 2.0, "item" : 1.0, "price" : 49.99, "qty" : 3.0,  
  "ordered_product" : [ {"_id" : 1.0, "name" : "Kick Scooter", "price" : 49.99} ]}  
{"_id" : 3.0, "item" : 2.0, "price" : 99.99, "qty" : 1.0,  
  "ordered_product" : [ {"_id" : 2.0, "name" : "Power Jeep", "price" : 99.99} ]}
```



## 7. Modelli e sistemi NoSQL

1. Introduzione ai sistemi di basi di dati NoSQL
2. Document-oriented databases
3. Graph-databases



# NoSQL databases: the case of Graph databases

---

- A graph database is a database that uses **graph structure** with nodes, edges, and properties to represent and store data.
  - A management systems for graph databases offers Create, Read, Update, and Delete (CRUD) methods to access and manipulate data.
  - Differently from other NoSQL management systems, Graph database systems (e.g., Neo4j) are generally optimized for *transactional performance*, and tend to guarantee ACID properties.
-



# Graph databases

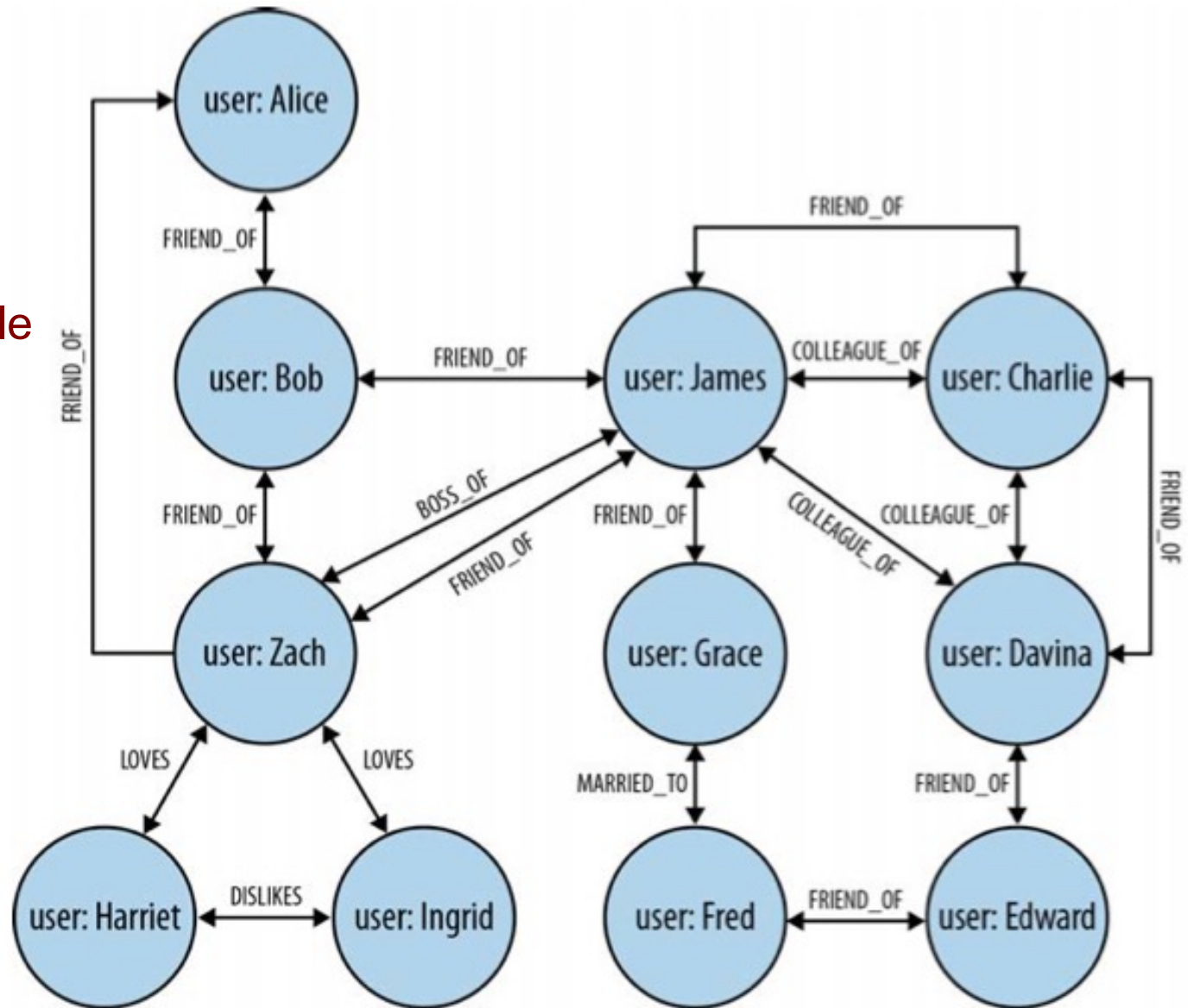
---

- Graph databases are **schemaless**:
    - Thus they will behave in response to the dynamics of big data: you can accumulate data incrementally, without the need of a predefined, rigid schema
    - This does not mean that intensional aspects cannot be represented into a graph, but they are not pre-defined and are normally managed as data are managed (as, e.g., for RDF)
    - They provide flexibility in assigning different pieces of information with different properties, at any granularity
    - They are very good in managing sparse data
  - Graph databases can be queried through declarative languages (some of them standardized): they can provide very good performances on certain queries, because essentially they know how to deal with specific types of join (*but, then, performances depend on the kind of queries*).
-



# Flexibility in graph databases

Incorporating  
dynamic  
information is  
natural and simple





# Graph Databases Embrace Relationships

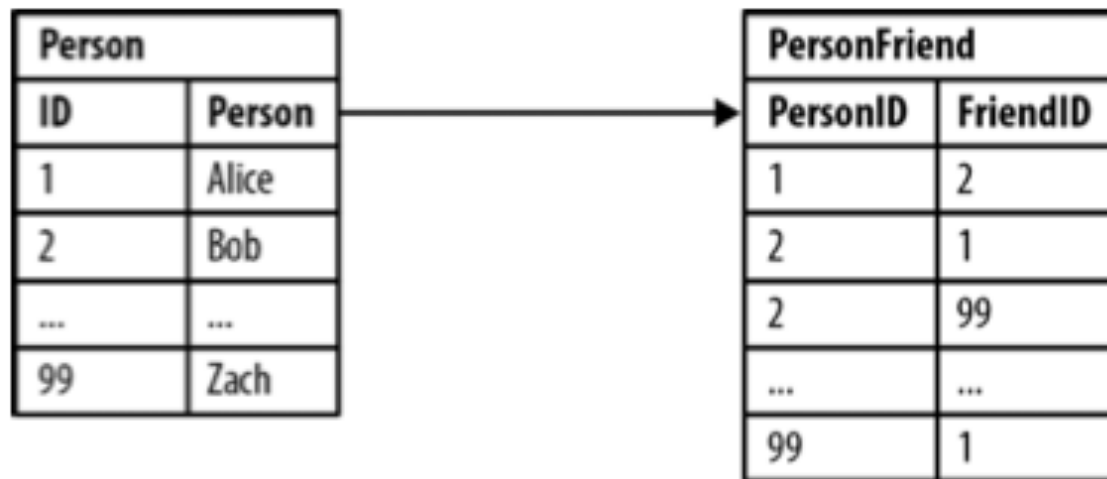
---

- Obviously, graph databases are particularly suited to model situations in which the information is somehow “**natively**” **in the form of a graph**.
  - The real world provides us with a lot of application domains: social networks, recommendation systems, geospatial applications, computer network, authorization and access control systems, to mention a few.
  - The success key of graph databases in these contexts is the fact that they provide **native means to use links to explicitly represent relationships**.
  - Relational databases instead lack explicit relationships: they have to be simulated through the help of foreign keys, thus adding additional development and maintenance overhead, and “navigating” them require costly join operations.
-



# Graph DBs vs Relational DBs- Example

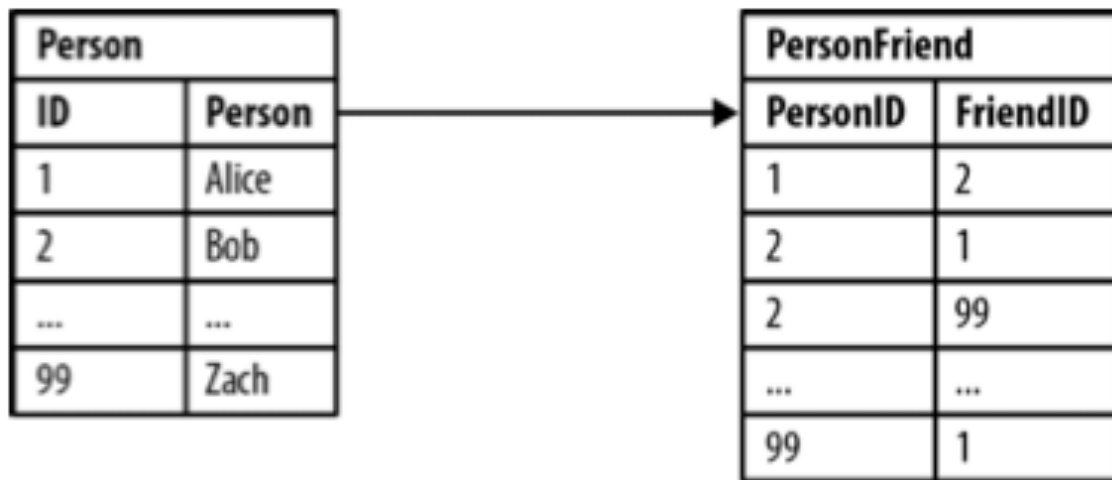
Modeling friends and friends-of-friends in a relational database



Notice that in this example, PersonFriend is not symmetric: Bob may consider Zach as friend, but the converse does not necessarily hold.

# Graph DBs vs Relational DBs- Example

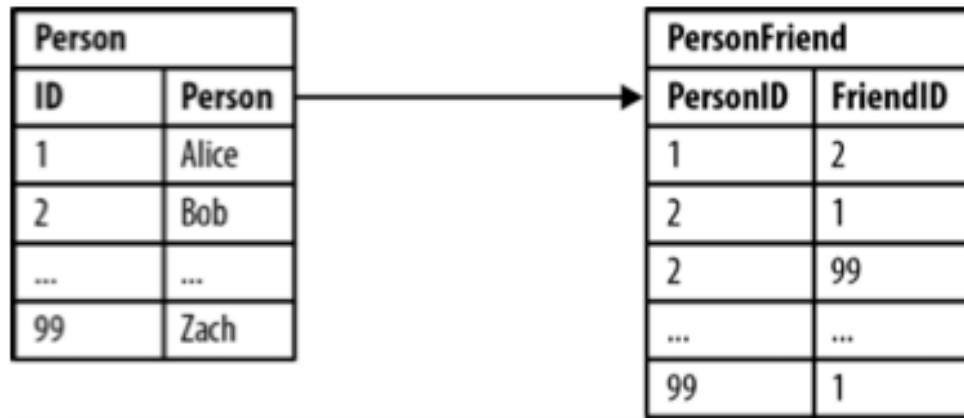
Asking “which are the names of Alice’s friends?” (i.e., those that Alice considers as friend) is easy



```
SELECT p2.Person AS ALICE_FRIEND
FROM Person p1 JOIN PersonFriend pf ON
    p1.ID = pf.PersonID JOIN Person p2 ON
    pf.FriendID = p2.ID
WHERE p1.Person = 'Alice'
```

# Graph DBs vs Relational DBs- Example

Things become more problematic when we ask, “which are the names of *Alice*’s friends-of-friends?”



```
SELECT p2.Person AS ALICE_FRIEND_OF_FRIEND
FROM Person p1 JOIN PersonFriend pf1 ON
    p1.ID = pf1.PersonID JOIN PersonFriend pf2 ON
    pf1.FriendID = pf2.PersonID JOIN Person p2 ON
    pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

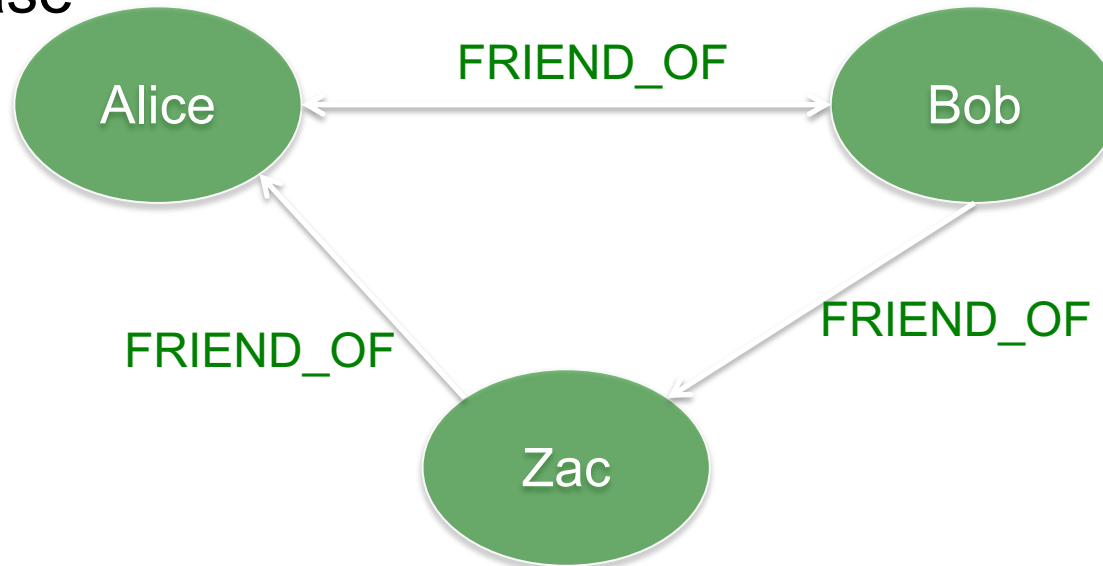
to exclude  
'Alice'  
from her  
FOFs

*Performances highly deteriorate when we go more in depth into the network of friends*



# Graph DBs vs Relational DBs- Example

Modeling friends and friends-of-friends in a graph database



Relationships in a graph naturally form paths. Querying means actually traversing the graph, i.e., following paths. Because of the fundamentally path-oriented nature of the data model, the majority of path-based graph database operations are extremely efficient.



# Graph DBs vs Relational DBs - Queries\*

---

## Comparison in modeling graph-like structures

- **Relational Databases** (querying is through joins)
    - The join operation forms a graph that is dynamically constructed as one table is linked to another table. The limitation is that this graph is not explicit in the relational structure, but instead must be inferred through a series of index-intensive operations.
    - Moreover, while only a particular subset of the data in the database may be desired (e.g., only Alice's friends-of-friends), all data in all queried tables must be considered in order to extract the desired subset.
  - **Graph Databases** (querying is through traversal paths)
    - There is no explicit join operation because vertices maintain direct references to their adjacent edges. In many ways, the edges of the graph serve as explicit, "hard-wired" join structures (i.e., structures that are not computed at query time as in a relational database).
    - What makes this more efficient in a graph database is that traversing from one vertex to another is a (quasi) constant time operation.
-



# Querying Graph DBs

- A traversal refers to visiting elements (i.e. vertices and edges) in a graph in some algorithmic fashion. Query languages for graph databases allow **for recursively traversing the labeled edges while checking for the existence of a path whose label satisfies a particular regular condition** (i.e., expressed in a regular language).
- Basically, a *graph database*  $G = (V, E)$  over a finite alphabet  $\Sigma$  consists of a finite set  $V$  of nodes and a set of **labeled edges**  $E \subseteq V \times \Sigma \times V$ .
- a path  $\pi$  in  $G$  from node  $v_0$  to node  $v_m$  is a sequence of the form

$$(v_0, a_1, v_1)(v_1, a_2, v_2) \dots (v_{m-1}, a_m, v_m)$$

where  $(v_{i-1}, a_i, v_i)$  is an edge in  $E$ , for each  $1 \leq i \leq m$ . The *label* of  $\pi$ , denoted  $\lambda(\pi)$ , is the string  $a_1 a_2 \dots a_m \in \Sigma^*$ .

- A *Regular path query* is a **regular expression**  $L$  over  $\Sigma$ . The evaluation  $L(G)$  of  $L$  over  $G$  is the set of pairs  $(u, v)$  of nodes in  $V$  for which there is a path  $\pi$  in  $G$  from  $u$  to  $v$  such that  $\lambda(\pi)$  satisfies  $L$ .



# Regular Expressions

Syntax of regular expressions:

---


$$L ::= s \mid L \cdot L \mid L \mid L \mid L^* \mid L^+ \mid L? \mid (L)$$

where

- $s$  is an element of the alphabet  $\Sigma$
- $\cdot$  denotes string concatenation (it can be omitted, i.e.,  $LL = L \cdot L$ ),
- $|$  denotes an OR, i.e.,  $L_1 \mid L_2$  in an expression matching with  $L_1$  or  $L_2$
- $*$  is the kleen operator, denoting concatenation of 0 or any number of string matching the expression  $L$
- $+$  is similar to  $*$  but there must be at least one occurrence of a string matching the expression  $L$
- $?$  denotes 0 or 1 occurrences of the string matching the  $L$  expression.

Examples:

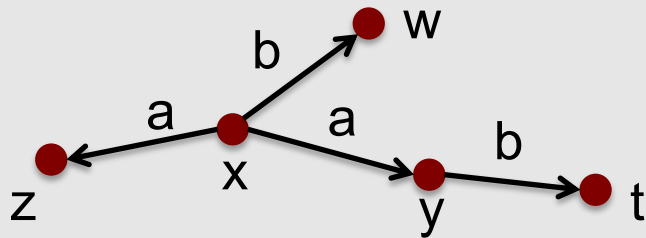
Has ancestors:

`isChildOf+`

Are cousins (for simplicity, an individual can be cousin of hersef):

`isChildOf · isChildOf · hasChild · hasChild`

# Example



Graph  $G (\Sigma=\{a,b\})$

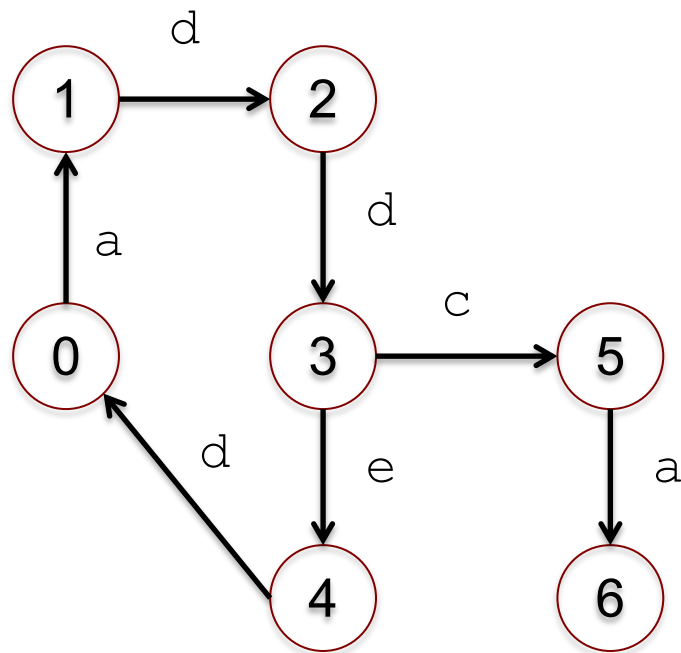
regular expression  $L = ab^*$

$$L(G) = \{(x,y); (x,z); (x,t)\}$$

*Query languages for graph databases typically extend this class of queries*

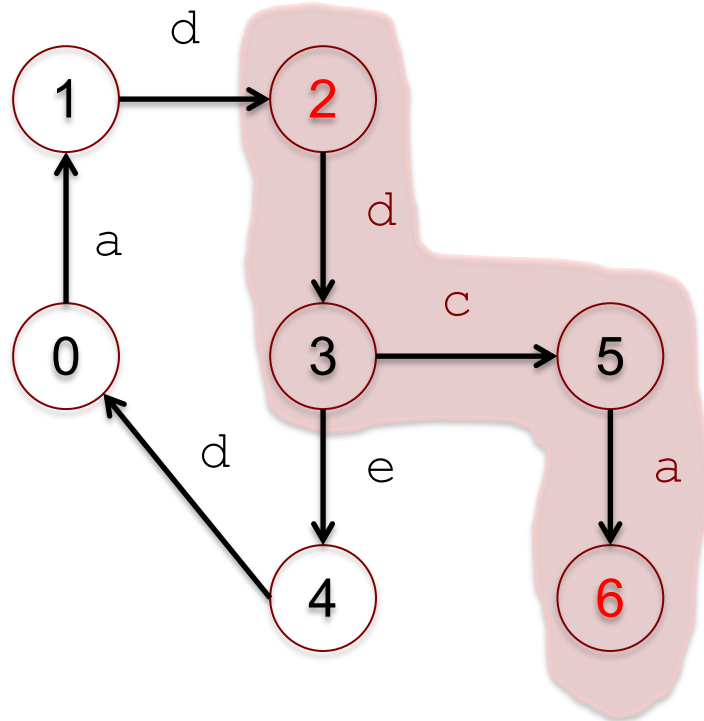


# Example



regular expression:  $d^+ (c | e) a$

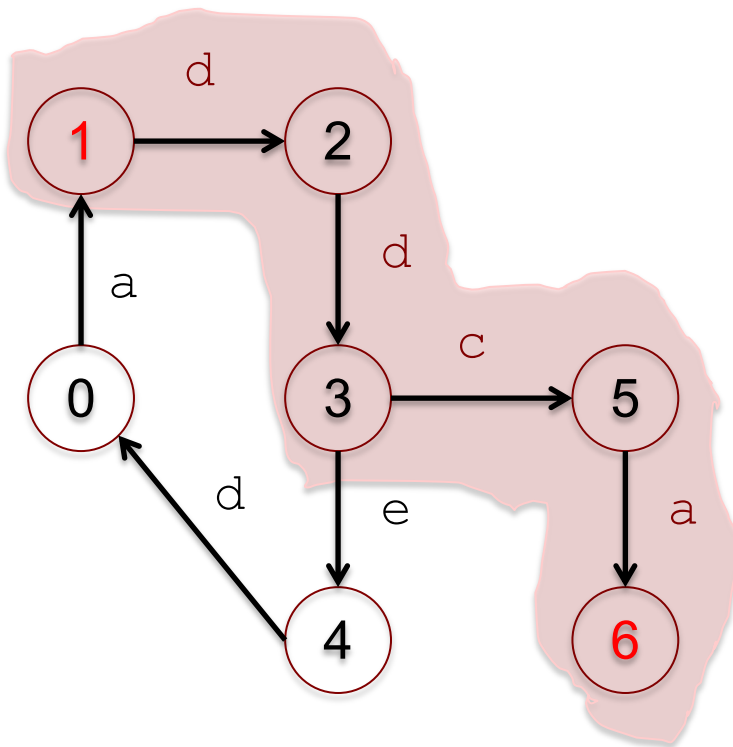
# Example



regular expression:  $d^+(c|e)a$

matching path:  $dca$

# Example



regular expression:  $d^+ (c | e) a$

matching path:  $ddca$