BigData

I database NoSQL: Mongo DB

MongoDB

Negli RDBMS le informazioni vengono memorizzate in strutture basate su tabelle e relazioni.

Le caratteristiche principapi di un RDBMS sono:

- I dati sono memorizzati in **tabelle e campi**;
- Le tabelle hanno uno **schema fisso**. Esse possono avere dei vincoli cioè delle condizioni che devono essere soddisfatte dai dati contenuti;
- Utilizzano le **relazioni**, vengono sfruttati dei campi per relazionare i valori di due o più tabelle. La validità del legame è garantito dai **vincoli di integrità referenziale;**
- l'accesso ai dati viene garantito con le proprietà **ACID** (Atomicità, Consistenza, Isolamento e Durabilità)

I database NoSQL possono avere le caratteristiche più disparate: alcuni non utilizzano il modello relazionale, altri usano tabelle e campi ma senza schemi fissi, alcuni non permettono vincoli di integrità referenziale, e altri ancora non garantiscono transazioni ACID. E naturalmente ci possono essere varianti che combinano le precedenti.

Essi possono essere catalogati in:

- **Database orientati ai documenti** memorizzano i dati in documenti, ossia in oggetti complessi codificati in qualche modo e senza uno schema rigido;
- **Database a grafo** usano strutture a grafo con relazioni libere (non prefissate come nel caso dei database relazioni) tra nodi del grafo;
- **Database chiave-valore** utilizzano il modello dell'array associativo;
- **Database a colonne**.

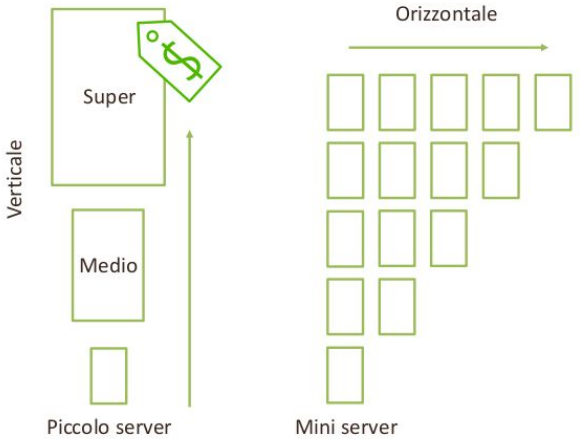
Non è facile valutare vantaggi e svantaggi dei database NoSQL rispetto ai RDBMS, appunto perché ogni database NoSQL merita un discorso a parte. Tuttavia si possono individuare alcune caratteristiche comuni.

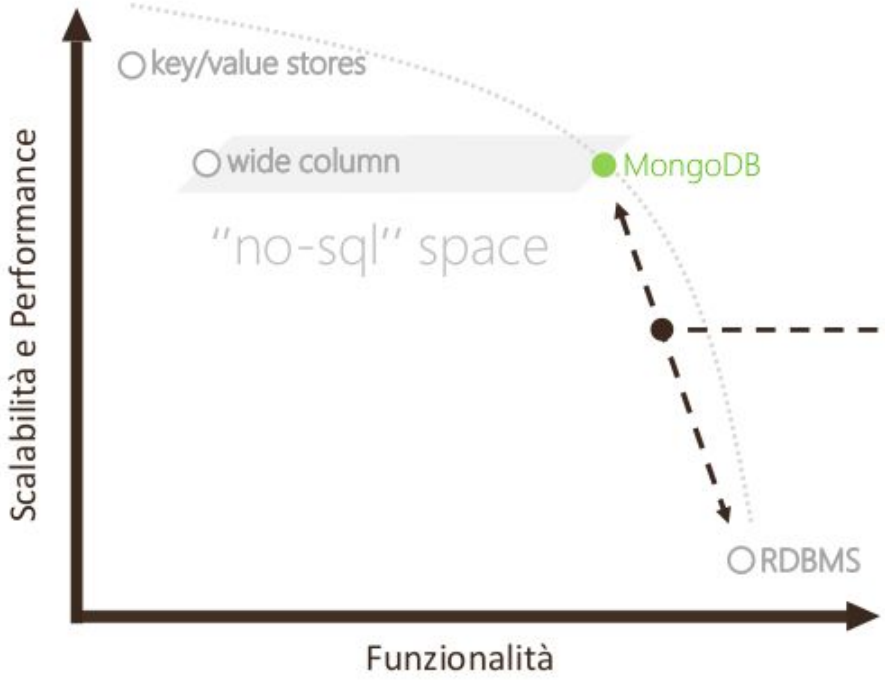
In primis, poiché il database viene scelto in base al contesto dell'applicazione che implementiamo, generalmente esso è più vicino al modello dei dati dell'applicazione.

In generale non esiste una soluzione perfetta per ogni evenienza, la valutazione di un prodotto dovrebbe essere fatta sempre in base alle esigenze che emergono in fase di analisi dell'applicazione che intendiamo sviluppare.

MongoDB è stato progettato e sviluppato soprattutto per venire in contro alle seguenti esigenze:

- Scalabilità
- Facilitare lo sviluppo
- Mettere a disposizione una rappresentazione dei dati intuitiva, dando la possibilità di rendere persistenti differenti tipologie di dati (semi-strutturati, non strutturati e strutture complesse) consentendo il Polimorfismo tra i dati.





Key/value stores, come Memcached

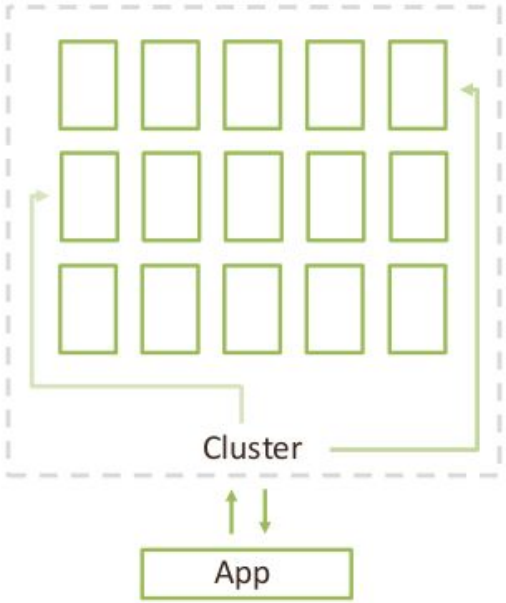
RDBMS, come Oracle, SQL Server, MySQL

Wide column, come Cassandra

Che differenze?

- Joins
- Transactions

MongoDB non supporta Join e Transaction. Il motivo che ha portato a questa scelta è relativo alla scalabilità del sistema, come si dovrebbero distribuire gli oggetti in cluster di macchine per poter utilizzare Join e Transaction?



In un database **document-oriented** come MondoDB i records sono memorizzati come “documenti” in formato JSON/BSON.

```
{
  nome: "Danilo",
  cognome: "Di Nuzzo",
  azienda: “R-DeV”,
  ruolo: ["CTO", "CoFounder" ],
  progetti: [
      { nome: “Progetto1”, tecnologie: [“Java, Hibernate”] }
  ]
}
```

I documenti sono in un formato molto vicino agli “oggetti” della programmazione OOP.

MongoDB sopperisce alla mancanza delle Join con le “**Pre-Join / Embedding**”, che consiste nel inglobare in un unico documento le informazioni dell’entità principale e delle eventuali entità relazionate.

MongoDB ha uno schema dinamico, questo ci permette di inserire delle informazioni aggiuntive ad un oggetto senza dover modificare la struttura del database.

Considerando il seguente un documento per la gestione dei contatti:

```
{
  nome: “Danilo”,
  cognome: “Di Nuzzo”,
  telefono: “333 0000000”
}
```

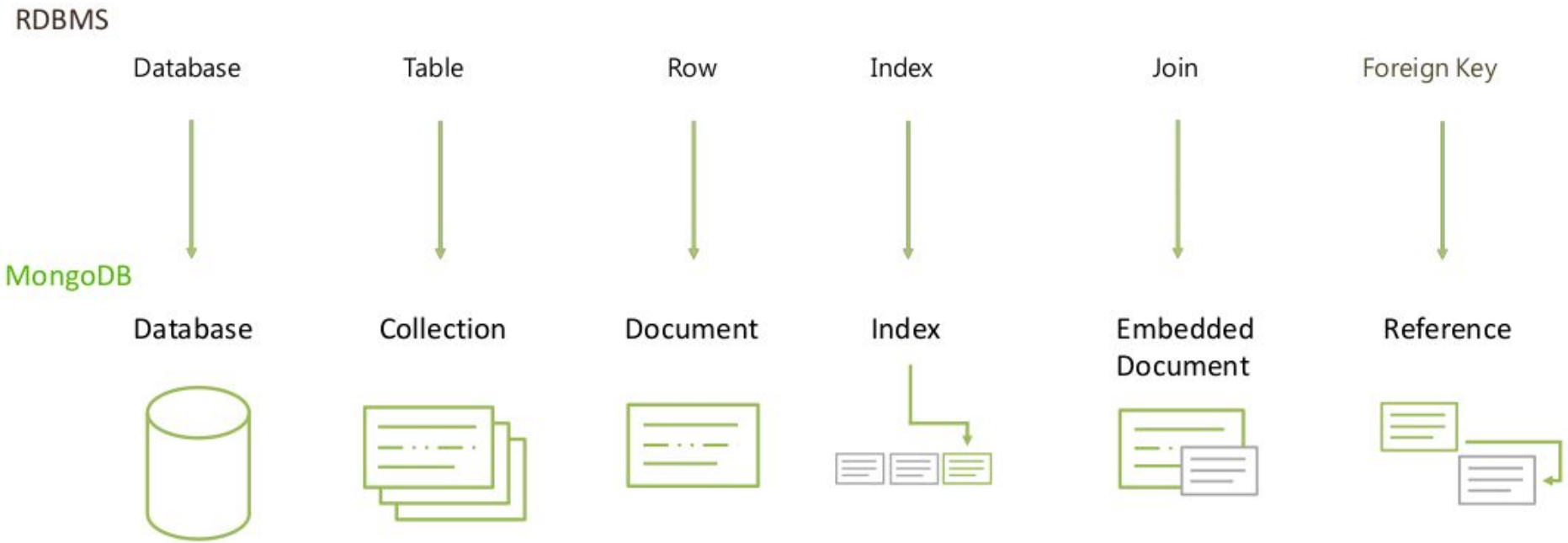
Potremmo aggiungere la lista di indirizzi email di un contatto senza dover modificare la struttura dati

```
{
  nome: “Danilo”,
  cognome: “Di Nuzzo”,
  telefono: “333 0000000”,
  emails: [“danilo.dinuzzo@r-dev.it”, “danilo.dinuzzo@hotmail.it”]
}
```

Per gestire la scalabilità MongoDB mette a disposizione i seguenti strumenti:

- **Replication**: replica dei dati tra un server primario ed i server secondari. Sarà cura di Mongo tenere i nodi sincronizzati e nel caso in cui il server principale non sia raggiungibile il sistema elegge automaticamente uno dei nodi secondari per prendere il suo posto;
- **Auto-sharding**: consiste nel separare le informazioni del nostro database su più server. Anche in questo caso il processo è completamente gestito dal cluster che si occuperà anche di bilanciare il carico di dati.

Di seguito è riportata la struttura di un database Mongo:



Una volta installato ed eseguito il nostro server Mongo abbiamo le seguenti possibilità per accedervi:

- Dalla riga di comando tramite la shell dedicata;
- tramite un client visuale come Robomongo;
- programmaticamente dalla nostra applicazione effettuando la connessione con gli appositi driver.

Per lanciare la shell basta eseguire il comando:

$ mongo

esso effettuerà direttamente la connessione all'istanza locale del server sulla porta 27017.

Una volta collegati alla shell possiamo provare i comandi che seguono per visualizzare il contenuto del nostro server.

Visualizzare il database attualmente selezionato:
> db
test

Visualizzare i database presenti:
> show dbs
local 0.000GB

Spostarsi ad un altro database presente all'interno del server:
> use rubrica
switched to db rubrica
anche se il database non esiste non otterremo un errore. Il database verrà creato automaticamente quando tenteremo di effettuare delle operazioni su di esso.

Proviamo ad effettuare un'operazione di ricerca all'interno del database in uso:
> db.contatti.find()
>

in db.contatti.find():

- **db** è un oggetto messo a disposizione dalla nostra shell che identifica il database in uso;
- **contatti** è la collection (che in questo momento non è ancora presente);
- **find()** è il comando che dice a Mongo di cercare tutti documenti nella collection (l'equivalente di *select * from contatti* in un ambiente relazionale).

Proviamo ad inserire all'interno della nostra collection un contatto con il seguente comando:

db.contatti.insert({ first_name: "Danilo", last_name: "Di Nuzzo", phone: "33300000" })

se ora riprovassimo a lanciare il comando di find su questa collection vedremo che effettivamente è presente il documento appena inserito.

Come noterete nel documento appena inserito è presente un ulteriore attributo **_id** di tipo **ObjectId** che viene sempre inserito se non specificato in fase di inserimento.

La funzione **find()** accetta, come parametro di input, un oggetto JSON che ci consente di filtrare il risultato di ricerca. Ad esempio, volendo recuperare solo i contatti che hanno come valore del campo first_name “Danilo” potremmo utilizzare il seguente comando:

> db.contatti.find({ first_name: “Danilo” }).pretty()

L’aggiunta di **pretty()** ci consente di formattare l’output della find().

I comandi che abbiamo utilizzato fino ad ora sono delle invocazioni di metodi di un oggetto. La nostra shell, in effetti, sta lavorando interpretando del codice Javascript. Per questo motivo è del tutto lecito fare delle operazioni del genere:

> var contatto = db.contatti.findOne({ first_name: “Danilo” })
> contatto.emails = [“danilo.dinuzzo@r-dev.it”, “danilo.dinuzzo@hotmail.it”]

con le righe di codice precedenti abbiamo recuperato un contatto assegnandolo ad una variabile e successivamente abbiamo aggiunto una lista di indirizzi email.

A questo punto se volessimo salvare le modifiche nel nostro documento non dovremmo fare altro che invocare il seguente comando:

> db.contatti.save(contatto)

Save è un metodo che effettua l'insert se l'oggetto non contiene l'attributo _id e un update altrimenti.

Come abbiamo visto i dati da inserire e quelli restituiti all'interno della shell amministrativa di MongoDB sono in formato JSON (JavaScript Object Notation).

JSON supporta le seguenti tipologie di dato:

- booleani (true e false);

- interi, numero in virgola mobile;
- stringhe racchiuse da doppi apici (");
- array: sequenze ordinate di valori, separati da virgole e racchiusi in parentesi quadre [];
- obj / Doc: sequenze coppie chiave-valore separate da virgole racchiuse in parentesi graffe;

Mentre per i dati possiamo avere diverse tipologie le chiavi utilizzate per definire un oggetto sono sempre delle stringhe. Lo standard consiglia di racchiudere gli identificatori degli attributi tra doppi apici. All'interno della console di mongo, fino ad ora, li abbiamo omessi ma ciò è stato possibile solo perché gli identificatori iniziavano sempre con una lettera. Nel caso di identificatori che iniziano con un numero, ad esempio, è necessario utilizzarli anche all'interno della shell.

In mongo vengono messi a disposizione altre 3 tipologie di dato: Date per gestire le date, BinData per file binari e ObjectID usato per identificare un documento.

Schema Interno di un database Mongo:

Lo possiamo vedere come un catalogo di:

- Databases
  - Collections (ha implicitamente un indice su _id dei documenti)
    - Documents
    - Indexes

Volendo possiamo inserire altri indici sui campi dei nostri documenti con il seguente comando:
> db.contatti.ensureIndex({ first_name: "text" })

Embedding

La soluzione delle PreJoin o Embending, in alcune circostanze può risultare inutilizzabile.

Consideriamo che MongoDB ha un limite di 16Mb per il salvataggio di ogni documento

Immaginiamo di dover creare una struttura per un blog dove, per ogni articolo, è presente una lista di commenti. Incapsulando all'interno dell'articolo gli eventuali commenti avremmo sicuramente il vantaggio di accedere direttamente ad un unico documento per recuperare tutte le informazioni da mostrare.

Questo approccio però potrebbe portarci, in strutture più complesse, ad avere un numero molto alto di elementi embeddati che potrebbero potenzialmente farci sforare il limite di storage del documento.

Per ovviare a questo problema potremmo strutturare i dati in modo da avere una lista di id invece di incapsulare direttamente la lista di commenti ma, naturalmente, questa soluzione ci porta a dover effettuare una serie di estrazioni per mostrare i vari commenti.

Una soluzione più performante potrebbe essere quella di paginare la lista di commenti, incapsulare la prima pagina all'interno dell'articolo e salvare le altre pagine in documenti separati.

CRUD ed operatori

Le operazioni di crud (Create, Read, Update e Delete). Nella shell di Mongo, come visto, non viene utilizzata una sintassi SQL ma sono utilizzare delle funzioni JavaScript.

INSERT

```
> var doc = { “first_name”: “Danilo”, “last_name”: “Di Nuzzo”, “email”: “danilo.dinuzzo@r-dev.it” }
> db.contatti.insert( doc )
```

Se provassimo a recuperare il documento appena inserito vedremo che è stato inserito l’attributo **_id** questo campo è una sorta di PK, è unico ed immutabile. Naturalmente è possibile inserire un valore arbitrario in fase di insert del documento.

FIND e FINDONE

> db.contatti.findOne({ “first_name” : “Danilo” })
findOne restituisce un solo elemento dalla collection su cui viene invocato. Passando un parametro come nell’esempio precedente viene recuperato il primo documento che ha come first_name il valore Danilo.

E’ possibile specificare in findOne anche la lista di attributi che vogliamo vedere in output:
> db.contatti.findOne({ “first_name” : “Danilo” }, { “first_name”: true })

per recuperare più di un documento dalla collection dobbiamo utilizzare il metodo **find()** al posto di **findOne()**

OPERATORI $gt e $lt (> e <)

Per applicare un attributo si deve utilizzare la seguente sintassi:
> db.contatti.find({ “age”: { $gt: 30 } })

Nell’ipotesi di avere una lista di documenti con il campo age che indica l’età, con l’istruzione precedente stiamo recuperando tutti i contatti che hanno un valore maggiore di 30 per il campo age.

Se volessimo recuperare i contatti con un’età compresa nell’intervallo 30 < eta < 40 dovremmo utilizzare il seguente comando:
> db.contatti.find({ “age”: { $gt: 30, $lt: 40 } })

Questi operatori non tengono in considerazione gli estremi. Gli operatori **<=** e **>=** sono rispettivamente **$gte** e **$lte**

Tutti questi operatori possono essere utilizzati anche con le stringhe

OPERATORI $exists, $type, $regex

$exists può essere utilizzato per recuperare solo i documenti che hanno uno specifico attributo:
> db.contatti.find({ emails: { $exists: true } })

$type ci permette di recuperare solo i documenti che, per un dato attributo, hanno un valore di un determinato tipo
> db.contatti.find({ phone: { $type: 2 } })

I tipi utilizzati per il confronto possono essere recuperati su **http://bsonspec.org/**

> db.contatti.find({ name: { $regex: 'REGEX' } })

OPERATORI $and e $or

> db.contatti.find({ $or: [{ first_name: “Danilo” }, { first_name: “Raffaele” }] })

L’operatore $and prevede la stessa sintassi

FIND e GLI ARRAY

> db.contatti.find({ emails: “danilo.dinuzzo@hotmail.it” })
La ricerca si adatta alla struttura dinamica dei documenti. Se, in fase di ricerca, viene trovato un documento che ha il valore email come stringa viene confrontata con il valore passato. Nel caso dovesse trovare un documento che ha un array nell’attributo emails controllerebbe se tra i suoi valori è presente la stringa passata in input.

OPERATORI $all, $in, $nin

> db.articoli.find({ tags: { $all: [“tag1”, “tag2”] } })
Recupera tutti gli articoli che hanno nell’array tags i valori “tag1” e “tag2”, non tiene in considerazione la posizione.

> db.articoli.find({ tags: { $in: [“tag1”, “tag2”] } })
Recupera tutti gli articoli che hanno nell’array tags almeno uno dei due valori “tag1” e “tag2” passati in input.

FILTRARE I SOTTO-DOCUMENTI

> db.contatti.find({ socials.skype: “danilo.di.nuzzo” })

ORDINARE I RISULTATI DELLE QUERY

> db.contatti.find().sort({ first_name: 1 })

Il valore 1 indica un ordinamento crescente, -1 decrescente.

altre 2 funzioni che possiamo invocare sono skip(n) e limit(n), entrambe prendono un intero in input. La prima consente di saltare i primi n risultati la seconda di restituire solo n risultati.

COUNT

> db.contatti.count({ first_name: “Danilo”})

UPDATE

Ci sono diverse operazioni di update.

SOSTITUZIONE INTERO ELEMENTO:

> db.contatto.update({ first_name: “Danilo” }, { … })
In questo caso verrebbe sostituito completamente il primo documento che rispetta la condizione fornita dal primo parametro con l’oggetto passato come secondo parametro.

UPDATE DI UNO SPECIFICO ATTRIBUTO

> db.contatto.update({ first_name: “Danilo” }, { $set: { phone: 331900000 } })
Se l’attributo phone non esiste viene creato altrimenti viene sostituito solo il valore di phone.

> db.contatto.update({ first_name: “Danilo” }, { $inc: { age: 1 } })
Consente di incrementare uno specifico attributo numerico, se l’attributo non esiste viene creato con lo step di incremento specificato.

> db.contatto.update({ first_name: “Danilo” }, { $unset: { phone: true } })
Elimina un attributo da un documento

> db.contatto.update({ first_name: “Danilo” }, { $set: { “emails.1”: “dnl.dinuzzo@gmail.com” } })
Sostituisce un determinato elemento dell’array emails

> db.contatto.update({ first_name: “Danilo” }, { $push: { “emails”: “dnl.dinuzzo@gmail.com” } })
Aggiunge un elemento in coda all’array

> db.contatto.update({ first_name: “Danilo” }, { $pull: { “emails”: “dnl.dinuzzo@gmail.com” } })
Se trova il corrispondente elemento lo elimina dall’array

$pop, $pushAll, $pullAll, $addToSet

> db.contatto.update({ $exists: { first_name: true } }, { $set { param: value } })
Con il comando precedente verrebbe modificato il valore del primo elemento che rispetta la condizione ma tutti i documenti hanno l’attributo first_name. Per far si che l’operazione di Update venga effettuata su tutti i documenti dobbiamo aggiungere come terzo parametro della funzione update il valore { multi: true }

ELIMINAZIONE

> db.contatti.remove({ first_name: “Danilo” })

Elimina tutti i documenti che rispettano la condizione (1 alla volta)

> db.contatti.drop()

Rimuove tutti i documenti della collection