NoSQL document-oriented

Matteo Francia, Ph.D.

m.francia@unibo.it

Who am I?

Matteo Francia, Ph.D.

- Adjunct professor & research fellow @ UniBO
- BIG: Business Intelligence Group (http://big.csr.unibo.it/)

Thanks to Enrico Gallinucci for these slides

Perché NoSQL?

Cosa significa NoSQL

Il termine viene usato per la prima volta nel '98 da Carlo Strozzi

- RDBMS open-source che usava un linguaggio diverso da SQL per le interrogazioni

Nel 2009 viene usato da un meetup di San Francisco

- Ospitavano discussioni di progetti open-source ispirati ai nuovi database di Google e Amazon
- Gruppi partecipanti: Voldemort, Cassandra, Dynamite, HBase, Hypertable, CouchDB, MongoDB

NoSQL indica dei DBMS (DataBase Management System) in cui il meccanismo di persistenza è diverso dal modello relazionale (RDBMS)

- NoSQL = Not Only SQL
- Secondo Strozzi, il termine NoREL sarebbe stato più consono

I punti forti degli RDBMS

Meccanismo delle transazioni

- Garanzia nella gestione della consistenza e degli accessi concorrenti

Integrazione

- Applicazioni diverse possono condividere e riutilizzare le stesse informazioni

Standard

- Il modello relazionale ed il linguaggio SQL sono standard affermati
- Un unico background teorico condiviso da diverse tecnologie

Solidità

- In uso da oltre 40 anni

I punti deboli degli RDBMS

Conflitto di impedenza

- La memorizzazione del dato si basa sul modello relazionale, ma la manipolazione del dato si basa tipicamente sul modello a oggetti
- Tante soluzioni proposte, nessuno standard
  - E.g.: Object Oriented DBMS (OODBMS), Object-Relational Mapping (ORM) frameworks

Difficile scalabilità orizzontale

- I Big Data sono una realtà; un unico server non può gestire tutto
- Distribuire un RDBMS non è una soluzione facile

Consistenza vs efficienza

- Garantire la consistenza dei dati è un must – anche a costo delle performance
- Le applicazioni odierne richiedono letture e scritture con grande frequenza e a bassa latenza

Rigidità dello schema

- Una modifica "a regime" può essere molto costosa

NoSQL: tanti modelli

Una delle principali difficoltà è capire quale modello adottare

Modello	Descrizione	Casi d'uso
Key-value	Associa un qualunque valore ad una stringa di testo	Dizionari, tabelle di lookup, cache, memorizzazione file e immagini
Document	Memorizza informazioni gerarchiche con una struttura ad albero	Documenti, qualunque dato idoneo ad una struttura gerarchica
Column family	Memorizza matrice sparse usando sia la riga che la colonna come chiave	Crawling, sistemi con elevata variabilità, matrici sparse
Graph	Memorizza nodi e archi	Query su reti sociali, inferenza, pattern matching

Il modello documentale

Document: modello

Un DB contiene una o più collezioni (corrispettive delle tabelle)

Ogni collezione contiene documenti (tipicamente JSON)

- Un documento ha una struttura ad albero auto-descrittiva

Ogni documento contiene un insieme di campi

- L'unico campo obbligatorio è l'ID

Ogni campo è strutturato come una coppia chiave-valore

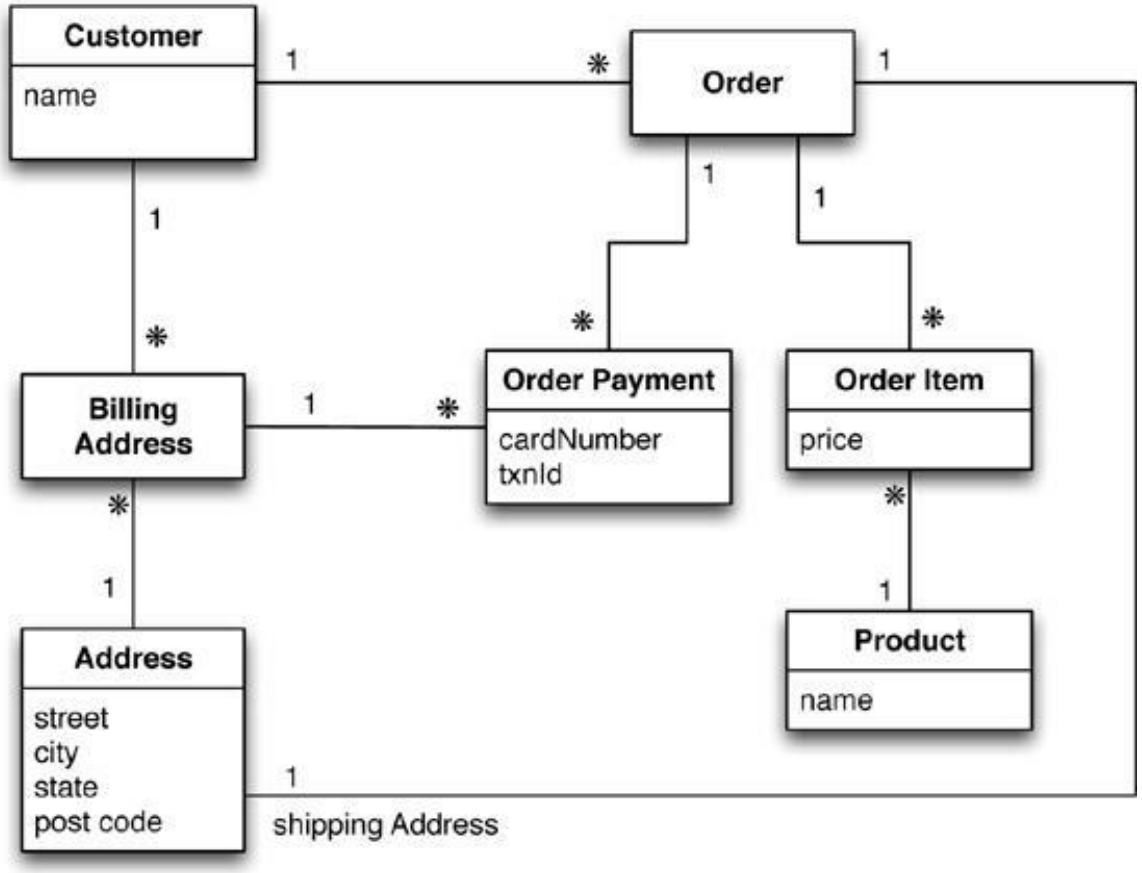
- Chiave: stringa di testo univoca all'interno del documento
- Valore: può essere semplice (stringa, numero, booleano) o complesso (oggetto, array, BLOB)
  - Un valore può contenere campi a sua volta

Livello di atomicità: il documento

```
{
    "_id": 1234,
    "name": "Enrico",
    "age": 32,
    "address": {
        "city": "Ravenna",
        "postalCode": 48124
    },
    "contacts": [ {
        "type": "office",
        "contact": "0547-338835"
    }, {
        "type": "skype",
        "contact": "egallinucci"
    } ]
}
```

Modellazione di documenti: un esempio

Tipico caso d'uso: clienti, ordini, prodotti



Modellazione di documenti: un esempio

Modellazione relazionale

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

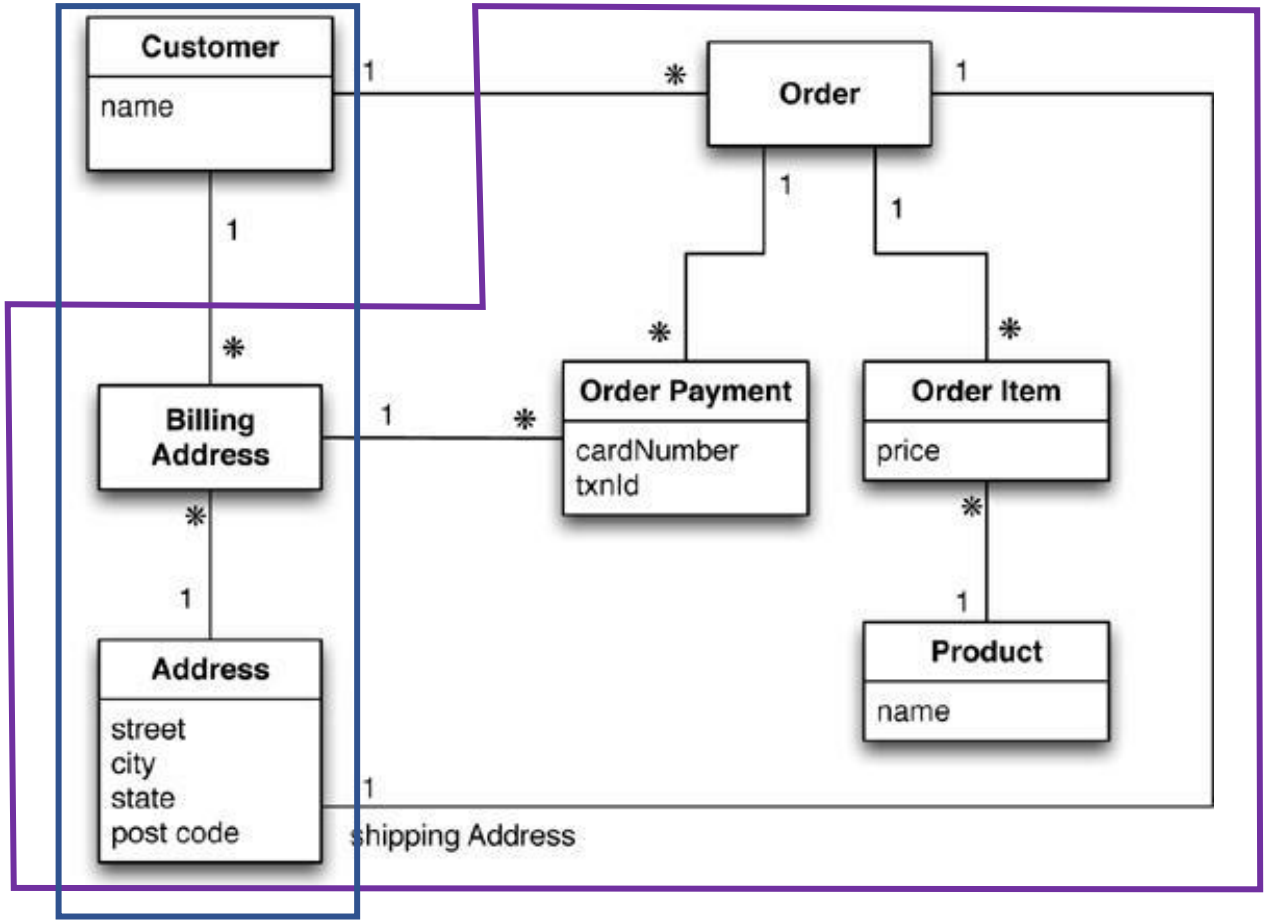
Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Modellazione di documenti: un esempio

Una possibile modellazione di documenti



Modellazione di documenti: un esempio

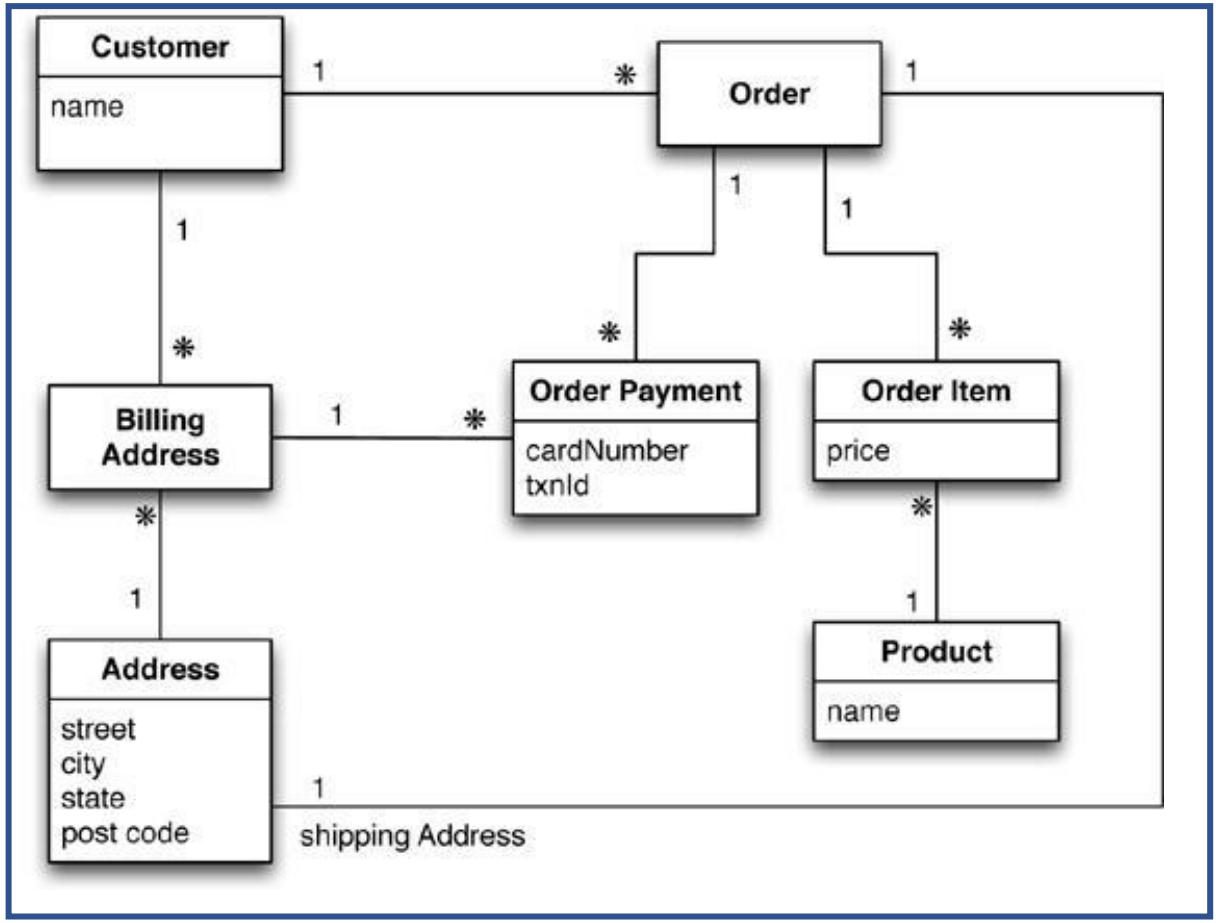
Risultato

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Modellazione di documenti: un esempio

Una modellazione alternativa



Modellazione di documenti: un esempio

Risultato

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

Modellazione di documenti: progettazione

Aggregate data modeling

- Un aggregato è un insieme di oggetti legati tra loro e che vengono trattati in blocco
- Un aggregato è un'unità per la manipolazione dei dati e la gestione della consistenza

Gli aggregati:

- + Facilitano il lavoro degli sviluppatori software, che spesso manipolano i dati attraverso strutture aggregate
- + Facili da gestire in un sistema distribuito
  - I dati che devono essere manipolati insieme (e.g., gli ordini ed i relativi dettagli) vengono modellati nello stesso aggregato – e quindi risiedono nello stesso nodo
- - Duplicazione dei dati memorizzati in livelli innestati (e rischio di inconsistenze)

Non esiste strategia universale per la definizione degli aggregati

- Dipende unicamente da come si intende manipolare i dati

Casi d’uso

Diversi contesti posso trarre beneficio dall’utilizzo di un database documentale

- **Log di eventi / web services**
  - Repository centrali per la memorizzazioni di log di eventi di diverse applicazioni
- **CMS, piattaforme di blogging**
  - Assenza di uno schema predefinito → adatti per CMS, gestione di siti web, commenti, profili utente
- **Web Analytics o Real-Time Analytics**
  - Indicizzazione di contenuti testuali → real-time sentiment analysis, social media monitoring
- **Applicazioni di e-commerce**
  - Flessibilità sullo schema per memorizzare prodotti e ordini
  - Evoluzione del modello dati senza incorrere in costi di refactory o di migrazione

Ma ci sono anche aspetti negativi da considerare

- Aggregate data modeling → fonte di duplicazione e inconsistenze
- Schemaless è un vantaggio in scrittura, non in lettura
- Non ideale per un carico di lavoro analitico (OLAP)

Casi d’uso reali

Servizi di advertising

- MongoDB nasce come sistema di gestione di banner pubblicitari
  - Il servizio deve essere disponibile 24/7 e molto efficiente
  - Necessarie regole complesse per trovare il banner giusto in base agli interessi della persona
  - Necessità di gestire tipologie diverse di ad e di avere una reportistica dettagliata

Internet of Things

- Gestione real-time dei dati generati da sensori
- Bosch utilizza MongoDB per catturare dati da automobili (sistema di frenata, servosterzo, tergicristalli, ecc.) e da strumenti di manutenzione di velivoli
  - Implementate regole di business che avvisano il pilota in caso di pressione dei freni calata sotto un livello critico, o avvisano l'operaio se uno strumento è utilizzato in maniera impropria
- Technogym utilizza MongoDB per catturare dati dagli attrezzi connessi

Aspetti che non approfondiremo

Sharding

- Criteri di distribuzione dei dati
- Replicazione dei dati
- Architettura master-slave di ReplicaSet
- Eventual consistency (teorema CAP)

Polyglot persistence

Amministrazione DB

GUI

- MongoDB Compass
- MongoDB Atlas
- Studio 3T

Integrazione con applicazioni

Getting started with MongoDB

Introduzione

I database document-oriented sostituiscono il concetto di *riga* con un modello più flessibile: il ***documento***

- Possibilità di rappresentare relazioni gerarchiche complesse in un unico documento
- Non esiste uno schema predefinito

Alcune delle caratteristiche principali:

- **Tanti indici a disposizione: composti, geo-spaziali, full-text**
- Un meccanismo di *aggregation pipeline* per costruire aggregazioni complesse attraverso la concatenazione di piccoli «pezzi»
- **Diversi tipi di collezione: time-to-live, fixed-size**
- Possibilità di usare script nel linguaggio Javascipt per manipolare i dati

Documenti

I documenti corrispondono sostanzialmente a oggetti JSON

- E' possibile utilizzare tipi di dato che il formalismo JSON non prevede

In generale, sono ricorsivamente definiti **come *oggetti* composti da coppie ordinate chiave-valore**, in cui:

- La *chiave* è una stringa case-sensitive
  - Non si possono usare i caratteri "." e "$"
  - Non possono esistere due chiavi identiche all'interno dello stesso oggetto
- Il *valore* può essere di diversi tipi:
  - Un *tipo semplice* (e.g., stringa, numero, data, ecc.)
  - Un altro *oggetto*
  - Un *array* di valore
- Generalmente, l'ordine delle chiavi non è importante
- In ogni documento viene automaticamente inserito un campo speciale, identificato dalla chiave ***_id***, il cui valore è unico all'interno della collezione (corrisponde alla chiave primaria)

Documenti

Un esempio

```
{
   "_id": ObjectId("5037ee4a1084eb3ffeef7228"),
   "info": {
      "nome": "Enrico",
      "dataNascita" : ISODate("1988-08-04T20:42:00.000Z")
   },
   "interessi": ["Calcetto", "Viaggi", "Serie TV"],
   "didattica": [{
      "corso": "Big Data",
      "datore": "Università di Bologna"
   }, {
      "corso": "Introduzione ai database NoSQL",
      "tipologia": "IFTS"
   }]
}
```

Collezioni

Una collezione è costituita da un insieme di documenti

- Non esiste uno schema di base (schemaless)
- Perché creare più collezioni invece che tenerne una sola?
  - Comodità
  - Performance
  - Data locality
  - Indici diversi in collezioni diverse

Una collezione è identificata da un nome

- Non si può usare il carattere “$”, ma si può usare il “.”, in particolare per organizzare concettualmente le collezioni in sotto-collezioni
  - E.g., blog.posts, blog.authors, ec..

Database

Un’istanza di MongoDB può contenere tanti database, ciascuno dei quali può ospitare tante collezioni

Ogni database ha il suo meccanismo di gestione dei permessi

- Di norma si utilizza un database per ogni applicazione

I database sono identificati da un nome

- Ci sono molte restrizioni sui caratteri (usare caratteri alfanumerici ASCII)

Database

Alcuni database sono riservati

admin

- E’ il database principale in termini di autenticazione; gli utenti assegnati a questo database possono accedere anche a tutti gli altri
- Alcuni comandi possono essere eseguiti solo da questo database (e.g., elencare tutti i database, spegnere il server)

local

- In un cluster, ne esiste uno per ogni macchina in cui è installato MongoDB
- Può essere usato per memorizzare dati che non devono essere distribuiti

config

- Memorizza informazioni utili per l’utilizzo in modalità distribuita

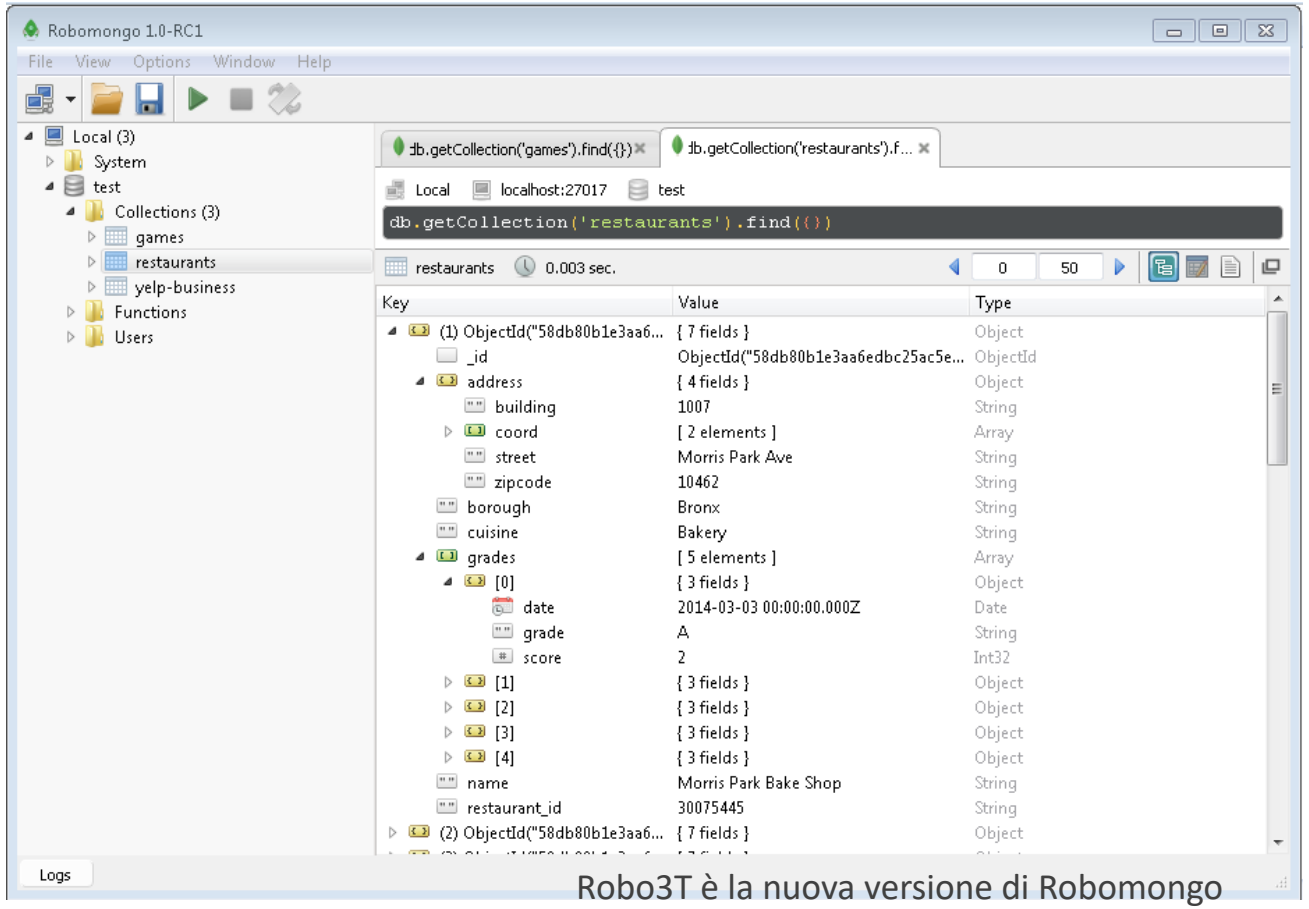
Connessione via Robo3T

Perché Robo3T?

- Semplifica la gestione e la navigazione del database
- Incorpora una shell di MongoDB

Parametri:

- Connessione a localhost
- Porta 27017



Robo3T è la nuova versione di Robomongo

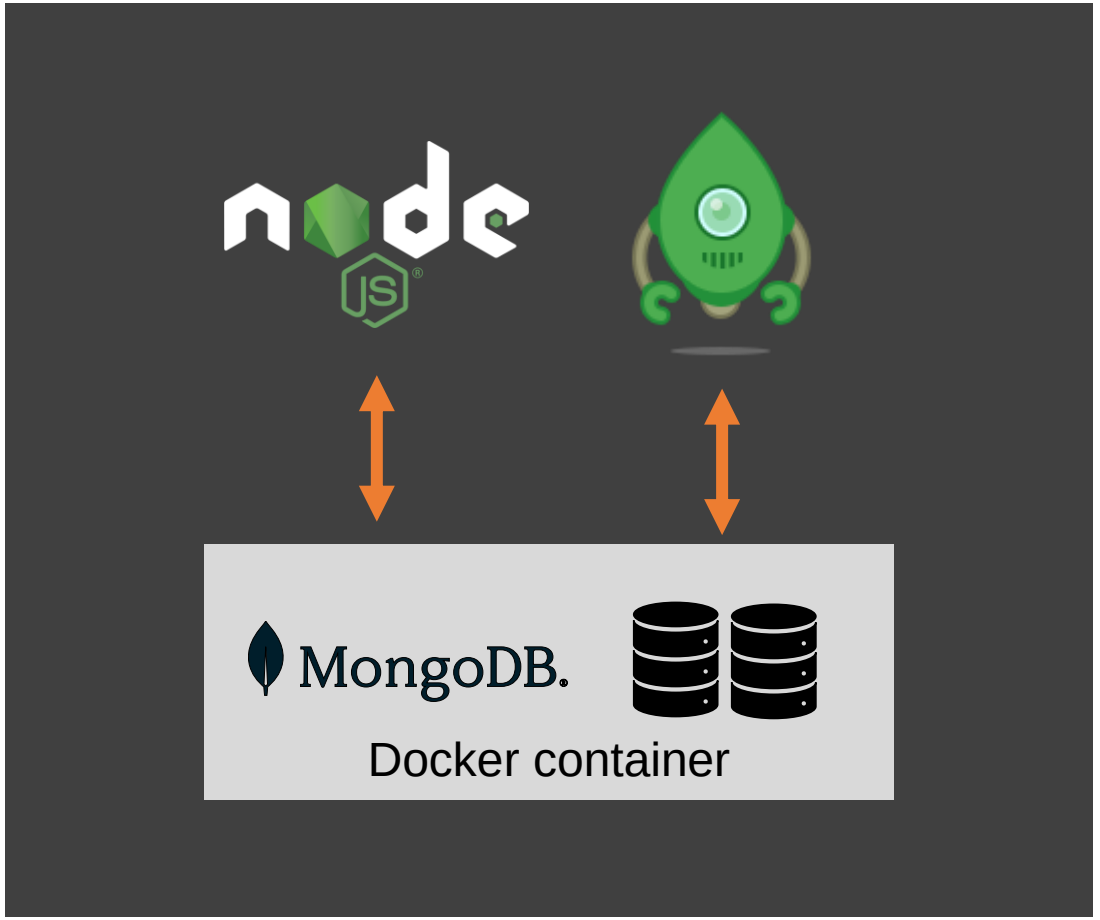
Software components

MongoDB

- Deployed on a docker container
- Already contains the data
- Check `docker-compose.yml`

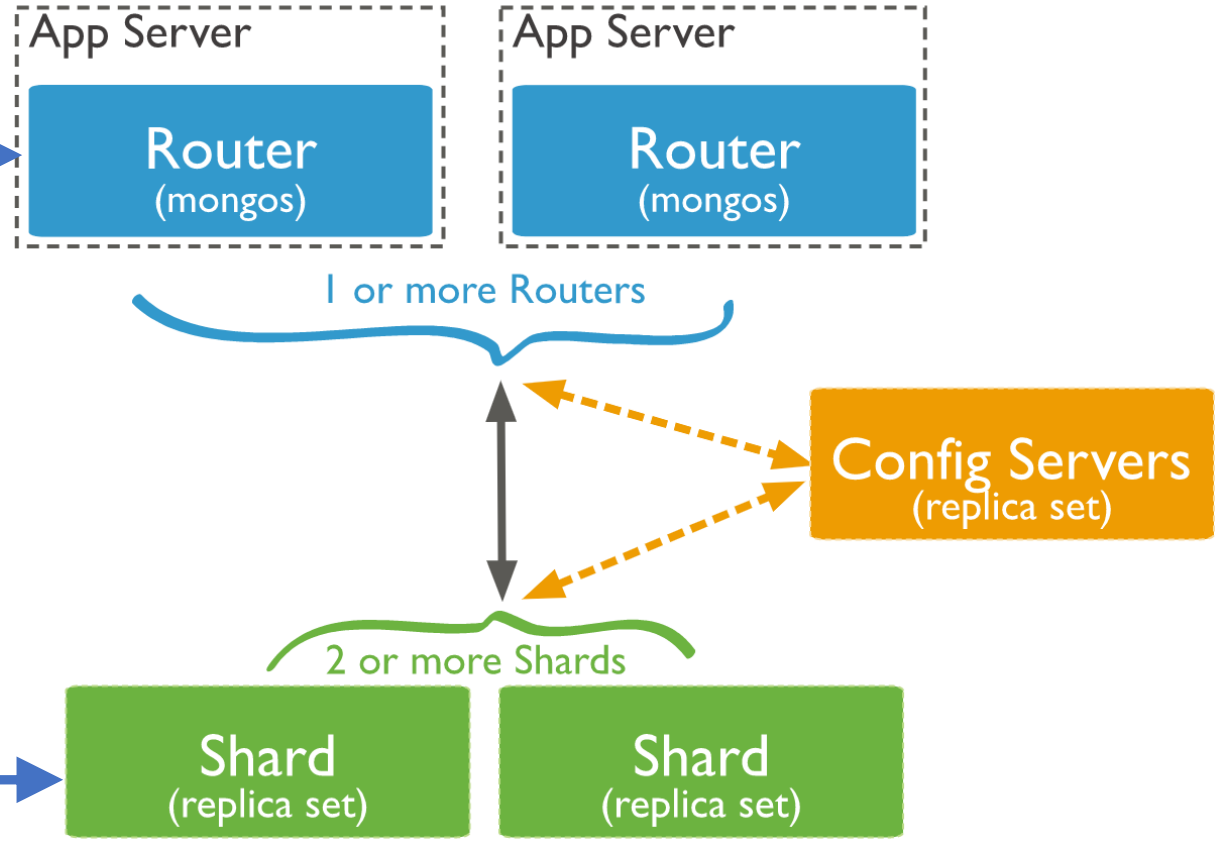
Querying the data, two alternatives:

- Robo3T (external program)
- Nodejs (check `src/`)



Connessione ad un cluster

Porta 27020
- Accesso a collezioni distribuite (*sharded*)



Porta 27017 o 27018
- Accesso a collezioni locali
- Accesso a porzioni locali delle collezioni distribuite

Collezioni per le esercitazioni (1)

Ristoranti

- https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json
- 25359 documenti relativi a ristoranti (nome, indirizzo, tipo di cucina, voti)

Partite NBA

- http://bit.ly/1gAatZK
- 31686 documenti relativi a 30 anni di partite dell'NBA (data, rose, statistiche)

Yelp

- https://www.yelp.com/dataset_challenge
- Dati reali messi a disposizione della ricerca scientifica
  - Più di 50.000$ distribuiti in competizioni, più di 100 paper accademici

```
mongorestore --collection games --db test C:\games.bson
```

Collezioni per le esercitazioni (2)

Statistiche NBA

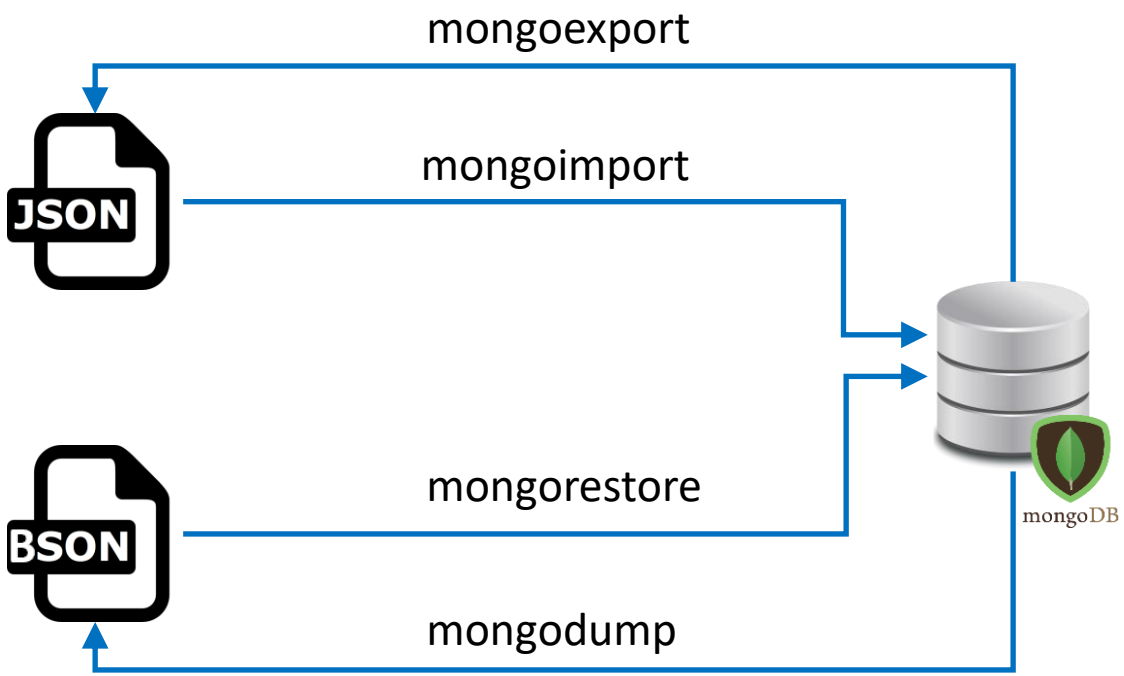
- http://www.mediafire.com/file/ju52cn1eadiydz6/NBA2016.json
- 1 documento contenente statistiche relative alla stagione 2016/17 per tutti i giocatori e tutte le squadre
- Modificato per separare le statistiche su giocatori e squadre in due file

Statistiche città USA

- https://gist.githubusercontent.com/Miserlou/c5cd8364bf9b2420bb29/raw/2bf258763cdddd704f8ffd3ea9a3e81d25e2c6f6/cities.json
- 1000 documenti con statistiche sulle città più popolose negli Stati Uniti d'America

```
mongoimport --collection nba2016players --db test C:\nba2016players.json --jsonArray
```

Strumenti di import/export



https://www.mongodb.com/basics/bson

Comandi di base

La maggior parte dei comandi di MongoDB sono metodi dell'oggetto db

Di base, la shell si collega al database vuoto *test*

- `db` – mostra il nome del database
- `db.` + `tab` – mostra i metodi richiamabili

Alcuni esempi

- `db.getMongo().getDBs()` – mostra i database presenti nell'istanza
- `db.getCollectionNames()` – mostra i nomi delle collezioni nel DB corrente
- `db.getSisterDB("foo")` – passa al database foo

Per lavorare su una collezione:

- `db.[collectionName].[method]([parameters])`

Comandi principali

Interrogazione dei dati

- Find, FindOne – modalità semplici per effettuare letture con proiezioni e selezioni
- Count, Distinct – modalità semplici per effettuare aggregazioni di dati
- Aggregate – modalità avanzata per effettuare aggregazioni di dati attraverso la concatenazioni di operazioni più semplici (match, unfold, group, ecc.)

Modifica dei dati

- Insert, Delete, Update

MongoDB query language

Il comando Find

È il comando che permette di eseguire interrogazioni (query) sul DB

La forma di base è:

```
db.nomeCollezione.find([[objSel],[objProj]])
```

Dove

- `nomeCollezione` va sostituito col nome della collezione da interrogare; corrispettivo SQL: FROM (ma limitato ad un'unica collezione)
- [`objSel`] è un (eventuale) oggetto che contiene i criteri di ricerca; corrispettivo SQL: WHERE
- [`objProj`] è un (eventuale) oggetto che contiene i criteri di ricerca; corrispettivo SQL: SELECT

Il comando Find

MongoDB

```
db.nomeCollezione
  .find({objSel}, {objProj})
  .sort({attrs})
```

Relational

```
select objProj
from nomeCollezione
where objSel
order by attrs
```

Il comando Find

Alcuni esempi

db.restaurant.find()

- Restituisce tutti i documenti

db.restaurant.findOne()

- Restituisce solo il primo documenti

db.restaurant.find({cuisine: "Hamburgers"})

- Restituisce i documenti in cui l'attributo cuisine (se presente) è valorizzato con la stringa "Hamburgers"

db.restaurant.find({}, {cuisine: 1})

- Restituisce tutti i documenti, ma proiettando solamente l'attributo cuisine (oltre all'_id, che viene restituito di default)

db.restaurant.find({cuisine: "Hamburgers"}, {cuisine: 1})

- La combinazione di selezione e proiezione

Find – proiezione

In caso di proiezione non specificata, vengono restituiti tutti gli attributi di tutti i documenti

Se si indica una proiezione, vengono mantenuti solo i campi indicati – ad eccezione del campo _id, che viene mantenuto ugualmente

- E' comunque possibile escludere il campo

Sintassi

- nomeChiave: [0, 1]

Dove

- nomeChiave è il nome di un attributo
- 1 va indicato se si vuole mantenere il campo
- 0 va indicato se, invece, si vuole escludere il campo (e.g., per l'_id)

Find – selezione semplice

Una prima modalità di selezione avviene attraverso il match esatto del valore dell'attributo con un valore specificato

Esempi:

- db.users.find({"age" : 27})
- db.users.find({"username" : "joe"})
- db.users.find({"username" : "joe", "age" : 27})

Come esprimere condizioni più complesse?

- ~~db.restaurant.find({restaurant_id > 40367790})~~
- Non è possibile, perché bisogna rispettare la sintassi JSON

Find – selezione complessa

L’espressione di condizioni di selezione complesse avviene attraverso l’incapsulamento di nuovi oggetti

Sintassi

- nomeChiave : { operatore : valore }

Dove

- operatore corrisponde ad un operatore di confronto secondo la sintassi di MongoDB (e.g., "$gte", acronimo di "Greather Than or Equal to")
- valore corrisponde ad un valore semplice (e.g., un numero o una stringa)
- Alcuni operatori richiedono che valore sia a sua volta un oggetto, composto da un’altra coppia operatore : valore

Find – operatori di confronto

Quali sono gli operatori

- $gte, $gt – corrispondono a ≥ e >
- $lte, $lt – corrispondono a ≤ e <
- $ne – corrisponde a ≠

Esempi

- db.users.find({"age" : {"$gte" : 18} })
- db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
- db.users.find({"registered" : {"$lt" : new Date("2007-01-01")} })
  - Il formato della data dipende dalla localizzazione
- db.users.find({"username" : {"$ne" : "joe"}})

Find – condizioni multiple

Quali sono gli operatori

- $in, $nin – equivalenti alle clausole IN e NOT IN di SQL
- $or, $nor, $and – equivalenti ai rispettivi operatori logici

Esempi

- db.users.find({"user_id": {"$in": [12345, "joe"]}})
- db.raffle.find({"ticket_no": {"$nin": [725, 542, 390]}})

- db.raffle.find({"$or": [{"ticket_no": 725}, {"winner": true}]})
- db.raffle.find({"$or": [{"ticket_no": {"$in": [725, 542, 390]}}, {"winner": true}]})

- db.raffle.find({"$nor": [{"ticket_no": 725}, {"winner": true}]})
- db.raffle.find({"$and": [{"ticket_no": 725}, {"winner": true}]})

Find – condizioni multiple

Possono esserci modi diversi per esprimere lo stesso criterio, più o meno ottimizzati

Esempi

- db.users.find({"$and" : [{"x" : {"$lt" : 5}}, {"x" : 1}]})
- db.users.find({"x" : {"$lt" : 5, "$in" : [1]}})

L'ottimizzatore fa più fatica in presenza di operatori $and e $or; se possibile, è meglio evitare di usarli

Find – negazione

Quali sono gli operatori

- $not – permette di negare un determinato criterio

Esempi

- db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})

Find – esistenza e campi nulli

Alcuni attributi possono avere `null` come valore.

Il comando `db.c.find({"y" : null})`

- restituisce sia i documenti in cui la chiave y esiste ed è valorizzata a null, sia i documenti in cui la chiave y non esiste.

Per avere solo i documenti in cui la chiave y esiste ed è valorizzata a null, bisogna verificare anche l'esistenza della chiave stessa:

- `db.c.find({"y" : {"$in" : [null], "$exists" : true}})`

Find – interrogare array

Contesto: collezione food con 3 documenti:

- {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
- {"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]}
- {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}

Comandi

- db.food.find({"fruit" : "banana"})
  match se l'array contiene banana (restituisce: 1 e 3)
- db.food.find({fruit : {$all : ["apple", "banana"]}})
  match se l'array contiene sia apple che banana (restituisce: 1 e 3)
- db.food.find({fruit : {$in : ["apple", "banana"]}})
  match se l'array contiene apple o banana (restituisce: 1, 2 e 3)

Find – interrogare array

Contesto: collezione food con 3 documenti:

- {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
- {"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]}
- {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}

Comandi

- db.food.find({"fruit" : ["banana", "apple", "peach"]})
  match se l'array corrisponde esattamente a quello indicato (restituisce: nulla)
- db.food.find({"fruit.2" : "peach"})
  match se l'array contiene peach in posizione 2 0-based (restituisce: 1)
- db.food.find({"fruit" : {"$size" : 3}})
  match se l'array contiene 3 elementi (restituisce: 1, 2 e 3)

Find – interrogare array

In fase di proiezione è possibile limitare il numero di elementi dell'array che vengono restituiti dalla query

Contesto: un doc che contiene il post di un blog ed i relativi commenti

- db.blog.posts.find(criteria, {"comments": {"$slice": 10}})
  restituisce i primi 10 commenti
- db.blog.posts.find(criteria, {"comments": {"$slice": -10}})
  restituisce gli ultimi 10 commenti
- db.blog.posts.find(criteria, {"comments": {"$slice": [23,10]}})
  salta i primi 23 documenti e restituisce i 10 successivi (dal 24° al 33°)
- db.blog.posts.find(criteria, {"comments.$": 1})
  restituisce i commenti che rispondo ai criteri di selezione indicati

Attenzione: se $slice è l'unico operatore utilizzato nella proiezione, tutti i campi dei documenti vengono restituiti

Find – interrogare array

Quando si pone una selezione con più criteri, questi sono valutati in AND a livello di documento

- db.test.find({"x": {"$gt":10, "$lt":20}})
- (∃ x > 10) ∧ (∃ x < 20)

Se x è un attributo semplice

- *Il documento contiene un x maggiore di 10 e minore di 20?*

Se x è un array

- *Il documento contiene un x maggiore di 10?*
- *E il documento contiene un x maggiore di 20?*
- Se il documento non è vuoto, **sarà sempre restituito**

Per imporre due vincoli in AND a livello di elemento di un array, bisogna utilizzare l'operatore **$elemMatch**

- db.test.find({"x": {"$elemMatch": {"$gt":10, "$lt":20}})

Find – interrogare oggetti

Contesto

- {"name": {"first":"Joe", "middle":"K" , "last":"Schmoe" }}

Esistono due modalità

- db.people.find({"name" : {"first":"Joe", last":"Schmoe"}})
  Match esatto: l'oggetto cercato deve essere uguale a quello specificato (in questo caso, non restituisce nulla)
- db.people.find({"name.first":"Joe", "name.last":"Schmoe"})
  In alternativa, si può usare la dot notation per referenziare i singoli campi (in questo caso, restituisce il documento)

Find – interrogare array di oggetti

Obiettivo: cercare i commenti di Joe con un punteggio di almeno 5

- db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}})
  Sbagliato: cerca il match esatto
- db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}})
  Sbagliato: restituisce entrambi i commenti, perché le condizioni sono valutate in OR
- db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe", "score" : {"$gte" : 5}}}})
  Corretto

Contesto:

```
{
 "content" : "...",
 "comments" : [{
  "author" : "joe",
  "score" : 3,
  "comment" : "nice post"
 },{
  "author" : "mary",
  "score" : 6,
  "comment" : "terrible post"
 }]
}
```

Find – Javascript scripts

L’espressività delle query tramite coppie chiave-valore è limitata

Per interrogazioni particolarmente complesse è possibile utilizzare l’operatore $where, che consente di eseguire uno script Javascript

- db.mycoll.find({$where : function() { return this.date.getMonth() == 11} })
- La complessità dello script è liberamente definita dall’utente

Tramite script è possibile fare praticamente qualunque tipo di operazione

- Per questioni di sicurezza, però, è fortemente sconsigliato l’utilizzo dell’operatore $where
- In generale, agli utenti finali non dovrebbe MAI essere concesso di eseguire questo tipo di interrogazioni

Limit, skip & sort

Al comando find possono essere applicati in cascata ulteriori comandi, al fine di applicare alcune trasformazioni al risultato ottenuto

Limit: restituisce solo i primi n documenti

- db.c.find().limit(3)

Skip: salta i primi n documenti e restituisci i successivi

- db.c.find().skip(3)

Sort: ordina i risultati sulla base di uno o più attributi

- db.c.find().sort({username : 1, age : -1})
- L'ordinamento può essere crescente (1) o decrescente (-1)

Count

Count è il comando per contare il numero di documenti restituiti da una query

- db.foo.count()
- db.foo.count({"x" : 1})
- Sostanzialmente simile al Find, con l'eccezione dell'assenza dell'oggetto di selezione

Distinct

Distinct è il comando per restituire i valori distinti di un campo a partire dai documenti che corrispondono ai criteri indicati

- db.inventory.distinct("item.sku", { dept: "A" })
- Resituisce i valori distinti del campo item.sku nei documenti in cui il dipartimento è A
- Se item.sku è un array, vengono restiuiti i valori distinti anche rispetto all'array di un singolo documento

Framework di aggregazione

Aggregate

Framework di aggregazione

Il framework di aggregazione permette di applicare **trasformazioni e aggregazioni** sui documenti di una collezione

E' costituito da una serie di **operatori di pipeline**, *mattoni* che possono essere liberamente combinati tra loro (anche più volte ed in qualunque ordine) per dar vita ad interrogazioni più o meno complesse

- Match, Project, Group, Unwind, Sort, Limit, Skip

Non solo aggregazioni: l'elevata espressività del framework consente di formulare **interrogazioni che non si potevano fare col Find**

- Applicare trasformazioni sulle date
- Concatenare due o più campi
- Confrontare i valori di due campi
- Restituire un singolo elemento di un array invece dell'array intero

Framework di aggregazione

Un esempio: in una collezione di riviste, voglio sapere quali sono gli autori che hanno venduto più di tutti

- Project: estraggo da ogni documento l'autore della rivisita
- Group: raggruppo per autore, contando il numero di occorrenze di ciascuno
- Sort: ordino in maniera decrescente sul numero di occorrenze
- Limit: mantengo solo i primi 5 risultati

La query:

- db.articles.aggregate([
  {"$project" : {"author" : 1}},
  {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}},
  {"$sort" : {"count" : -1}},
  {"$limit" : 5}
  ])

Operatore $match

L’operatore **$match** permette di **filtrare i documenti**

- Opera sostanzialmente come una query di Find
- Unica eccezione: non supporta operatori geospaziali

E’ buona norma utilizzare l’operatore il prima possibile

- Riduce il numero di documenti delle operazioni successive
- Può sfruttare gli indici (in fasi successive potrebbero non essere utilizzabili)

Un esempio

- db.restaurants.aggregate([{$match: {cuisine: "Hamburger"} }])
- db.restaurants.find({cuisine: "Hamburger"})

Operatore $project

L’operatore **$project** permette di effettuare una proiezione dei campi

- E’ **molto più potente** della proiezione nel comando Find
- Permette di estrarre campi da oggetti innestati e di applicare trasformazioni

db.articles.aggregate([{"$project" : {"author" : 1, "_id" : 0}}])

- Restituisce l’autore di un articolo ed esclude il campo _id

db.users.aggregate([{"$project" : {"userId" : "$_id", "_id" : 0}}])

- Rinomina il campo _id in userId
- In pratica, introduce un nuovo campo userId il cui valore corrisponde al valore di _id
- NB: l’utilizzo dell’operatore $ in "$_id" permette di indicare il riferimento ad un campo; altrimenti, "_id" verrebbe interpretato come un semplice valore

Operatore $project – espressioni matematiche

Espressioni su 1 o più valori: $add, $multiply

- "$add" : [expr1, expr2, ..., exprN]

Espressioni su 2 valori: $subtract, $divide, $mod

- "$subtract" : [expr1, expr2]
- "$divide": divide il primo valore per il secondo
- "$mod": divide il primo valore per il secondo e restituisce il resto

Un esempio

- db.employees.aggregate([{ "$project" : {
  "totalPay" : { "$subtract" : [{"$add" : ["$salary", "$bonus"]}, "$401k"] }
  } }])
- Restituisce un campo calcolato: totalPay = (salary + bonus) - 401k

Operatore $project – espressioni su date

Ci sono diverse espressioni che permettono di **estrarre una specifica informazione a partire da una data**

- "$year", "$month", "$week"
- "$dayOfYear", "$dayOfMonth", "$dayOfWeek"
- "$hour", "$minute", "$second"

Esempi

- db.employees.aggregate([{"$project": {"hiredIn" :{"$month" : "$hireDate"}} }])
- Restituisce il mese in cui gli impiegati sono stati assunti
- db.employees.aggregate([{"$project" : {"tenure" :
  { "$subtract" : [{"$year" : new Date()}, {"$year" : "$hireDate“}] }
  } }])
- Restituisce il numero di anni trascorsi dall’assunzione degli impiegati
- Un’operazione aritmetica tra due date restituisce un risultato in millisecondi

Operatore $project – espressioni su stringhe

"**$substr**" : [*expr*, *startOffset*, *numToReturn*]

- Restituisce una sottostringa della stringa passata come primo parametro; parte da *startOffset* e restituisce *numToReturn* byt
- Attenzione alla codifica: un byte potrebbe non corrispondere ad un carattere

"**$concat**" : [*expr1[, expr2, ..., exprN]*]

- Concatena le stringhe passate come parametri

"**$toLower**", "**$toUpper**"

- Restuiscono la stringa passata come parametro in tutte minuscole o maiuscolo.

Esempio

- db.employees.aggregate([{"$project" : {"email" : {"$concat" :
  [{"$substr" : ["$firstName", 0, 1]}, ".", "$lastName", "@example.com"]
  } } }])
- Restituisce una stringa come e.gallinucci@example.com

Operatore $project – espressioni logiche

Espressioni di confronto

- "**$cmp**" : [expr1, expr2]
  Confronta expr1 con expr2. Ritorna 0 se sono uguali, un numero negativo se expr1 < expr2, un numero positivo se expr1 > expr2.
- "**$strcasecmp**" : [string1, string2]
  Confronto case-insensitive tra due stringhe
- "**$eq**"/"**$ne**"/"**$gt**"/"**$gte**"/"**$lt**"/"**$lte**" : [expr1, expr2]
  Confronta expr1 con expr2 e ritorna true o false

Espressioni booleane

- "**$and**", "**$or**" : [*expr1[, expr2, ..., exprN]*]
  Ritorna vero se tutte ($and) o almeno una ($or) delle espressioni è vera
- "**$not**" : *expr*
  Ritorna il valore booleano opposto di *expr*

Operatore $project – espressioni logiche

Espressioni di controllo

- "**$cond**" : [*booleanExpr*, *trueExpr*, *falseExpr*]
  Se l'espressione *booleanExpr* è vera, ritorna *trueExpr*, altrimenti *falseExpr*
- "**$ifNull**" : [*expr*, *replacementExpr*]
  Se *expr* vale null, ritorna *replacementExpr*, altrimenti ritorna *expr*

Un esempio: gli studenti vengono valutati per il 10% sulla presenza, 30% sulle interrogazioni, 60% sulle verifiche; ma prendono 100 se sono "cocchi"

- db.students.aggregate([{"$project" : {"grade" : {"$cond" :
  ["$teachersPet", 100,
  {"$add" : [
  {"$multiply" : [.1, "$attendanceAvg"]},
  {"$multiply" : [.3, "$quizzAvg"]},
  {"$multiply" : [.6, "$testAvg"]}
  ]}
  ]}
  } }])

Operatore $group

L’operatore **$group** permette di raggruppare i documenti sulla base di determinate chiavi e di calcolare dei valori aggregati. Alcuni esempi:

- Contesto: misurazioni meteo minuto-per-minuto.
  Query: umidità media per giorno
- Contesto: collezione di studenti
  Query: raggruppare gli studenti per voto
- Contesto: collezione di utenti
  Query: raggruppare gli utenti per città e stato

I campi su cui si vuole raggruppare costituiscono le chiavi del gruppo

- {"$group" : {"_id" : "$day"}}
- {"$group" : {"_id" : "$grade"}}
- {"$group" : {"_id" : {"state" : "$state", "city" : "$city"}}}

Operatore $group ed operatori aritmetici

Oltre a specificare le chiavi su cui raggruppare è possibile indicare una o più operazioni per calcolare valori aggregati.

Gli operatori aritmetici sono due:

- "**$sum**" : value
  Produce la somma dei valori
- "**$avg**" : value
  Produce la media dei valori

Un esempio completo

- db.sales.aggregate([{"$group" : {
  "_id" : "$country",
  "totalRevenue" : {"$avg" : "$revenue"},
  "numSales" : {"$sum" : 1}
  } }])

Operatore $group ed operatori su estremi

Ci sono quattro operatori per ottenere gli "**estremi**" del dataset:

- "**$max**" : *expr ;* **"$min**" : *expr*
  Esaminano tutti i documenti e restituiscono rispettivamente il massimo ed il minimo valore trovato
- "**$first**" : *expr ;* "**$last**" : *expr*
  Esaminano solo il primo e l'ultimo documento per restituire il valore trovato

Due esempi

- ```
  db.scores.aggregate([{"$group" : {
     "_id" : "$grade",
     "lowestScore" : {"$min" : "$score"},
     "highestScore" : {"$max" : "$score"}
  } }])
  ```

```
db.scores.aggregate([
  {"$sort" : {"score" : 1} },
  {"$group" : {
    "_id" : "$grade",
    "lowestScore" : {"$first" : "$score"},
    "highestScore" : {"$last" : "$score"}
  }
}])
```
```

Operatore $group ed operatori di collezione

Ci sono due operatori che consentono di costruire un array con i valori riscontrati in ciascun gruppo

- "**$addToSet**" : *expr*
  Costruisce un array con tutti i valori distinti
- "**$push**": *expr*
  Costruisce un array con tutti i valori trovati, anche duplicati

Un esempio

- db.sales.aggregate([{"$group" : {
  "_id" : { day : { $dayOfYear: "$date"}, year: { $year: "$date" } },
  "itemsSold" : { $addToSet: "$item" }
  } }])
  Restituisce l'elenco distinto dei prodotti venduti in ciascun giorno

Operatore $unwind

L'operatore **$unwind** permette di *appiattire* un array, costruendo tanti documenti quanti sono gli elementi dell'array

- {
  _id: 1,
  categories: ['A','B','C']
  }

{ _id: 1, categories: 'A' },
{ _id: 1, categories: 'B' },
{ _id: 1, categories: 'C' }

Ciò torna utile per effettuare proiezioni e aggregazioni sugli elementi interni degli array

- db.blog.aggregate([
  {"$project" : {"comments" : "$comments"} },
  {"$unwind" : "$comments"},
  {"$match" : {"comments.author" : "Mark"} }
  ])
  Restituisce i commenti di Mark

Operatore $unwind

Un altro esempio

- { _id: 1, nome: "Enrico", città: "Cesena", voti: [1, 2, 3],
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: [4, 5, 6] },
  { _id: 3, nome: "Matteo", città: "Trieste", voti: [7, 8, 9] }
- db.col.aggregate([
  {"$unwind" : "$voti"},
  {"$group" : {"_id" : "$città", "mediaVoti": { "$avg" : "$voti" } } },
  ])
  Restituisce la media dei voti per città

Se l'array contiene un altro array, è possibile applicare l'operatore $unwind in cascata (prima sull'array esterno, poi su quello interno)

Operatore $unwind

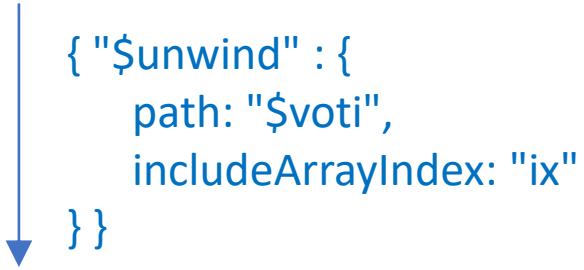
L’operatore $unwind può essere dichiarato anche come un oggetto, in cui indicare (oltre al campo da appiattire) alcuni parametri opzionali

- $unwind: {
  path: <field path>,
  includeArrayIndex: <string>,
  preserveNullAndEmptyArrays: <boolean>
  }
- **path** è il percorso dell’array (come nella versione semplice)
- **includeArrayIndex** è il nome di un nuovo campo in cui si vuole estrarre l’indice posizionale dell’array
- **preserveNullAndEmptyArrays**, se impostato a true, permette di restituire un documento anche se l’array indicato non esiste (oppure è null o vuoto)

Operatore $unwind

Un esempio con la versione estesa di $unwind

- { _id: 1, nome: "Enrico", città: "Cesena", voti: [1, 2, 3],
  { _id: 1, nome: "Lorenzo", città: "Cesena", voti: [4, 5, 4] }



- { _id: 1, nome: "Enrico", città: "Cesena", voti: 1, ix: 0 },
  { _id: 1, nome: "Enrico", città: "Cesena", voti: 2, ix: 1 },
  { _id: 1, nome: "Enrico", città: "Cesena", voti: 3, ix: 2 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 4, ix: 0 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 5, ix: 1 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 6, ix: 2 }

Operatori $sort, $limit e $skip

Gli operatori **$sort**, **$limit** e **$skip** funzionano come nella formulazione delle interrogazioni semplici

- Se si vuole ordinare un grande numero di documenti, è buona norma fare l'ordinamento il prima possibile lungo la pipeline e avere un indice sul campo

Un esempio

- ```
  db.employees.aggregate([
    { "$project" : {
      "compensation" : { "$add" : ["$salary", "$bonus"] },
      "name" : 1
    } },
    { "$sort" : {"compensation" : -1, "name" : 1} }
  ])
  ```
- E' possibile ordinare anche sui campi creati lungo la pipeline
  ```

Operatore $lookup

Introdotto a partire dalla versione 3.2

L'operatore **$lookup** permette di eseguire il left outer join tra collezioni residenti nello stesso database

- Nella collezione «primaria» viene creato un nuovo campo di tipo *array*, contenente gli eventuali documenti corrispondenti nella collezione «secondaria»

La sintassi:

- $lookup: {
  from: <collection to join>,
  localField: <field from the input documents>,
  foreignField: <field from the documents of the "from" collection>,
  as: <output array field>
  }

Operatore $lookup

Collezione **orders**

- { "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2 }
- { "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1 }
- { "_id" : 3 }

Collezione **inventory**

- { "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
- { "_id" : 2, "sku" : "def", description: "product 2", "instock" : 80 }
- { "_id" : 3, "sku" : "ghi", description: "product 3", "instock" : 60 }
- { "_id" : 4, "sku" : "jkl", description: "product 4", "instock" : 70 }
- { "_id" : 5, "sku": null, description: "Incomplete" }
- { "_id" : 6 }

Operatore $lookup

Esempio di lookup

- db.orders.aggregate([{
  $lookup: {
  from: "inventory",
  localField: "item",
  foreignField: "sku",
  as: "inventory_docs"
  }
  }])

```
{
 "_id" : 1,
  "item" : "abc",
 "price" : 12,
 "quantity" : 2,
 "inventory_docs" : [
  { "_id" : 1, "sku" : "abc",
    description: "product 1", "instock" : 120 }
 ]
}
..
{
 "_id" : 3,
 "inventory_docs" : [
  { "_id" : 5, "sku" : null, "description" : "Incomplete" },
  { "_id" : 6 }
 ]
}
```

Indici

Indici

La costruzione di indici consente di migliorare le performance in lettura

- ATTENZIONE: gli indici vengono aggiornati ad ogni modifica; il rischio è di peggiorare le performance in scrittura

Un esempio: db.products.createIndex({category: 1, price: -1})

- :1 indica un ordinamento crescente, -1 decrescente
- L'ordine dei campi è importante

Caratteristiche

- Si possono indicizzare gli array
- Esistono tipi speciali di indici: geospaziali, di testo, hash
- Indici parziali: indicizzano solo alcuni documenti sulla base di una query
- Indici sparsi: indicizzano solo i documenti per cui il campo non esiste
- Indici TTL: cancellano il documento se la data indicizzata è obsoleta

Indici di testo

Consentono di fare ricerche di testo all'interno di uno o più campi

- ATTENZIONE: può essere creato un solo indice di testo per collezione
- db.stores.createIndex({ name: "text", description: "text" })

L'indice esclude punteggiatura e *stop-words* e riduce i termini a radice

- Esempio: "Collezione con indice di testo." → ["collezion", "indic", "test"]

Caratteristiche

- Si possono indicizzare campi testuali e array di stringhe
- Si possono assegnare pesi diversi a campi diversi
- Tarato di default sull'inglese, ma si possono impostare lingue diverse

Operatore $text

```
db.stores.insert([
   { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
   { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
   { _id: 3, name: "Coffee Shop", description: "Just coffee" },
   { _id: 4, name: "Clothes Clothes", description: "Discount clothing" },
   { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
db.stores.createIndex( { name: "text", description: "text" } )
```

Operatore $text

```
db.stores.insert([
   { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
   { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
   { _id: 3, name: "Coffee Shop", description: "Just coffee" },
   { _id: 4, name: "Clothes Clothes", description: "Discount clothing" },
   { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
db.stores.find( { $text: { $search: "java coffee shop" } } )
```

- Le parole indicate sono considerate in OR

Operatore $text

```
db.stores.insert([
   { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
   { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
   { _id: 3, name: "Coffee Shop", description: "Just coffee" },
   { _id: 4, name: "Clothes Clothes", description: "Discount clothing" },
   { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
db.stores.find( { $text: { $search: "java \"coffee shop\"" } } )
```

- Le stringhe indicate tra virgolette (") vengono cercate esattamente

Operatore $text

```
db.stores.insert([
   { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
   { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
   { _id: 3, name: "Coffee Shop", description: "Just coffee" },
   { _id: 4, name: "Clothes Clothes", description: "Discount clothing" },
   { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
db.stores.find( { $text: { $search: "java shop -coffee" } } )
```

- Le parole precedute dal meno (-) causano l'esclusione del documento

Operatore $text

```
db.stores.insert([
   { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
   { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
   { _id: 3, name: "Coffee Shop", description: "Just coffee" },
   { _id: 4, name: "Clothes Clothes", description: "Discount clothing" },
   { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
db.stores.find(
   { $text: { $search: "java coffee shop" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

- E'possibile ordinare i risultati sulla base di un punteggio di similarità con la query di ricerca

Grazie per l’attenzione!