I database noSql : mongoDB

Rev 5.4 del 14/01/2021

Introduzione ai DB noSQL

NO_SQL significa **Not Only SQL**
Si parla anche di database non relazionali, cioè **NR-DBMS** (not relational DBMS)

I database concentrati su un singolo PC

Un tempo il database server era costituito da una una singola macchina. Per aumentarne le prestazioni si ricorreva alla cosiddetta **scalabilità verticale** cioè l'aumento di potenza nell'ambito dello stesso server, quindi l'aumento del numero di processori, RAM, dischi, etc. In questa soluzione esiste chiaramente un limite dettato dalla potenza massima che un singolo computer può raggiungere.
La scalabilità verticale con un unico server che gestisce anche migliaia di utenti contemporanei, è andata bene fino a quando le connessioni sono rimaste chiuse nell'ambito dalla rete locale dell'organizzazione.

I tre step che hanno portato verso il No-SQL

1) L'avvento di internet e la nascita dei database distribuiti

Con l'avvento di internet si è assistito ad un aumento esponenziale del volume dei dati da gestire ed è cresciuto esponenzialmente anche il numero di richieste che un server deve gestire, con picchi che possono verificarsi in modo imprevedibile e che il server deve riuscire a soddisfare in tempi accettabili.

In questo nuovo scenario è diventato indispensabile il passaggio a **sistemi distribuiti** e si è iniziato a parlare di **scalabilità orizzontale** cioè l'aggiunta di più server paralleli e la distribuzione dei dati su di essi. Nei sistemi distribuiti in linea di principio si possono aggiungere server all'infinito.

2) La pesantezza dei Vincoli ACID

Con l'aumento del volume di dati ed il passaggio ai sistema distribuiti è emerso un altro problema, cioè che i **tradizionali database SQL hanno incominciato a manifestare grossi limiti di performance** dovuti soprattutto al soddisfacimento dei cosiddetti Vincoli Acid che impongono una certa lentezza nella risposta.

I vincoli **ACID** (**Atomicity, Consistency, Isolation, Durability)** sono vincoli molto stringenti mirati a garantire il buon esito di una **transazione** (insieme di operazioni inscindibili).

Per poter operare correttamente, una transazione deve essere **atomica, consistente, isolata e duratura**. Implementare tutte queste caratteristiche comporta una serie di controlli che penalizzano le prestazioni.

- **atomicità**: la transazione deve essere indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla, non sono ammesse esecuzioni parziali;
- **consistenza**: quando inizia una transazione il database si trova in uno stato coerente e quando la transazione termina il database deve di nuovo essere in uno stato coerente, ovvero non deve violare i vincoli di integrità, cioè non devono verificarsi contraddizioni (*inconsistenza*) nei dati archiviati nel DB
- **isolamento**: ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre. L'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione;
- **durabilità**: detta anche **persistenza**, si riferisce al fatto che una volta che una transazione abbia richiesto un ***commit*** *work*, i cambiamenti apportati non dovranno essere più persi. Per evitare che nel lasso di tempo fra il momento in cui la base di dati si impegna a scrivere le modifiche e quello in cui li scrive effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul DB.

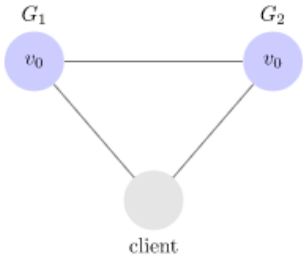
3) CAP Theorem

Riguardo ai **database distribuiti**, nel 2000 Eric Brewer ha dimostrato un teorema fondamentale relativamente ai **vincoli di utilizzo dei database in ambito distribuito**. Ogni lettera di CAP sta a rappresentare una caratteristica che un sistema distribuito (SQL o noSQL) deve avere :

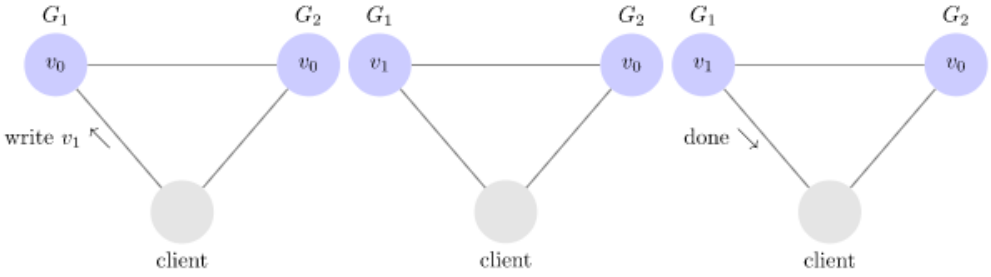
- **Consistency** (Consistenza): dopo qualsiasi modifica, **tutti i nodi del sistema distribuito devono riflettere le modifiche allo stesso modo**. Ogni server deve restituire lo stesso dato più recente, ovvero proveniente dall'ultima scrittura dello stesso, oppure un messaggio di errore. Ogni client che interroga un qualsiasi cluster del sistema distribuito ha la stessa "vista" dei dati.
- **Availability** (Disponibilità): Un server **deve sempre dare una risposta ad ogni richiesta** ricevut da un client. In altre parole il server deve essere sempre disponibile e non andare in stand-by perché sta facendo altro. Una risposta troppo ritardata è altrettanto negativa quanto una risposta non data.
- **Partition Tollerance** (Tolleranza al Partizionamento): **I dati devono essere sufficientemente replicati tra i nodi in modo tale da garantire una risposta anche in caso di interruzione di una connessione tra i server.** Un sistema tollerante alle partizioni deve poter sopportare qualsiasi problema di rete tra i vari server. Nei sistemi distribuiti moderni, il Partition Tolerance non è un'opzione, ma una necessità.

Il CAP theorem afferma che questi tre vincoli, in un database distribuito, NON possono essere realizzati contemporaneamente. Secondo il teorema, se ne possono realizzare contemporaneamente soltanto 2, potendo scegliere quali prediligere in base alle necessità

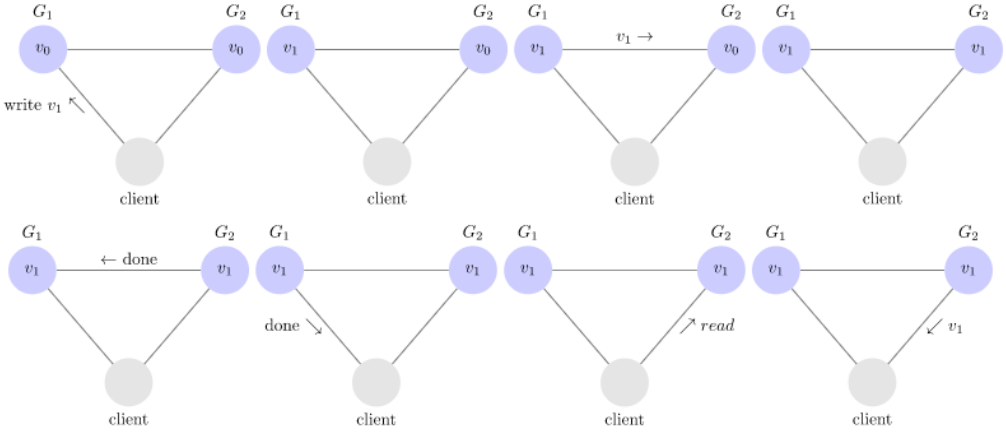
Dimostrazione: *https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/*
Si ipotizzino 2 server distribuiti **G1** e **G2** interconnessi tra loro



Se il client modifica la variabile v sul primo server da 0 a 1, il server G1 deve inviare l'informazione al server G2 e **solo dopo** inviare l'ok al client. Il sistema indicato in figura (con il done immediato) è **inconsistente**, in quanto se il client legge l'informazione v sul server G2 legge un valore non valido (cioè 0).



Di seguito è invece riportata la soluzione corretta che soddisfa ai criteri di consistenza:



Si suponga ora che il collegamento tra G1 e G2 si interrompa.
Un sistema *partition tolerance* dovrebbe essere in grado di funzionare anche in corrispondenza di eventuali interruzioni del collegamento fra server.

Se però si interrompe il collegamento tra i due server ci sono due possibilità

- G1 non invia l'ok al client fino a quando non sarà ripristinato il collegamento con G2.
  In questo caso viene meno **l'availability:** il sistema non risponde più al client.
- G1 invia subito l'ok al client e invia la variazione a G2 solo quando sarà ripristinato il collegamento.
  In questo caso viene meno la **consistency**. G1 e G2 hanno dati non consistenti.

Avvento dei database noSQL

I database noSQL nascono soprattutto per sopperire alla lentezza causata dai **vincoli ACID**.
L'idea di base è quella di rinunciare a qualche vincolo **ACID** per ottenere prestazioni migliori e maggiore scalabilità in ambito distribuito velocizzando l'accesso ai dati e riducendo il costo di gestione delle join.

Riducendo però il numero di vincoli (si pensi alle imposizioni delle forme normali) potrebbero crearsi problemi di inconsistenza inammissibili in un DB tradizionale.

La dimostrazione del CAP Theorem ha fornito una grande spinta ai database noSQL che lo hanno subito abbracciato puntando sulla velocità di funzionamento e scegliendo di prediligere i fattori **AP** a scapito della Consistenza. Viene introdotto il concetto di DB "**eventualmente consistente**", ovvero la propagazione ritardata delle informazioni a tutti i nodi. I dati vengono replicati sui server eventualmente anche con un certo ritardo, per cui può capitare che i messaggi cancellati vengano temporaneamente riproposti, etc.
Tutti i controlli necessari al mantenimento della consistenza sono spostati al livello applicativo trasformando di fatto l'NRDMBS in uno **storage altamente performante ma "poco intelligente".**

La prima grande società ad utilizzare in DB noSQL in ambito distribuito è stata **Amazon** nel 2002.

Caratteristiche di un DB noSQL

- La prima cosa importante è che **cade il concetto di schema**.
  Nei database **NoSQL** non c'è più la strutturazione rigida tipica dei database relazionali.
  Le informazioni non sono più memorizzate in righe strutturate, ma in oggetti che possono essere anche completamente diversi fra loro.
- Con gli NR-DBMS si tende a memorizzare ogni elemento in modo **atomico**, con tutte le informazioni rilevanti concentrate in un unico record.
- Non esiste più il concetto di **relazione** con i relativi vincoli di integrità. Decade anche il concetto di **join**, operazione estremamente onerosa che consiste in un incrocio di tabelle in modo da restituire all'interno di un recordset unico tutte le informazioni necessarie.
  Il join comporta spreco di tempo di esecuzione e anche di spazio occupato dal recordset a causa della presenza di campi non significativi.
- Il modello noSQL basato sugli oggetti, è molto più vicino alla programmazione **OOP** di quanto non lo sia il modello relazionale che richiede continue ed onerose conversioni tra record e oggetti

Svantaggi dei database noSQL

Come tutte le cose, anche nei database NoSQL non esistono solo vantaggi.
Se questa tecnologia risolve il problema di performance, introduce comunque alcune criticità:

- il limite principale è la **mancanza dell'integrità** dei dati. Non esistendoci relazioni (in senso stretto), il sistema non può garantire che le operazioni effettuate mantengano il DB in uno stato consistente.
  Poco adatto per applicazioni di tipo bancario dove l'integrità dei dati è vitale, anche a discapito della **Availability** e della **Partition Tollerance.** E' meglio una risposta consistente anche se ritardata.

- rende molto più complessa la lettura dei dati per **un'analisi statistica**.
  Talvolta i dati vengono convertiti in db relazionali per poterli analizzare con più semplicità.
- il processo di **migrazione dei dati** da un database all'altro è molto più complesso rispetto alle soluzioni relazionali.

Quando conviene utilizzare un database noSQL

- **Quando si hanno grandi quantità di dati e si ha necessità di prestazioni elevate**, cioè laddove i vincoli ACID diventano troppo stringenti
- Quando è necessario interagire molto frequentemente con il database, nel qual caso occorrerebbero continue conversioni oggetto / relazione
- Quando i dati sono molto collegati tra loro e il JOIN rischia di essere eccessivamente pesante.
- Quando la struttura dei dati non è definibile a priori. Si pensi ad esempio alla variegata gestione dei contenuti che impongono oggi i social network o gli applicativi web per la gestione dei contenuti.

Le diverse famiglie di database noSQL

con il termine"NoSQL" si identificano tre famiglie di storage abbastanza differenti tra loro:

- **Document Oriented** I dati sono suddivisi in **collezioni** (che sostituiscono le vecchie tabelle) e **documenti**. Un documento presenta la struttura tipica di un **Object** che possiede un certo numero di proprietà che rappresentano le informazioni (i campi del documento). Esempio :
  `{"nome":"Mario", "cognome":"Rossi", "eta":25, "mail":"mario.rossi@gmail.com"}`
  Per i documenti non è obbligatorio avere una struttura fissa, anche se alcuni DBMS permettono di imporne una, ma utilizzano una struttura dinamica.
  I documenti possono essere messi in relazione tra loro con dei riferimenti gerarchici, e ciò rende questa tipologia di database NoSQL particolarmente adatta a rappresentare delle gerarchie.
  Sono in pratica una **via di mezzo tra i Key-Value ed i Graph**
  Tra i database più utilizzati in questa categoria vi sono **MongoDB** e **CouchDB**.

- **Key-Value** Hanno come obiettivo principale la **velocità**. I dati vengono memorizzati come Dictionary caratterizzati da una chiave ed un valore ad essa associato. Quest'ultimo è un oggetto contenente l'informazione vera e propria, mentre il primo è l'identificatore che permette l'estrazione rapida del valore dal database mediante tecniche di hash. Sono utilizzati in tutti quei sistemi in cui si trattano valori ai quali è facilmente associabile una chiave univoca, come ad esempio un sistema di messaggistica: ogni **chiave** può rappresentare un "account", ed il suo **valore** sarà costituito dalla ista dei messaggi ricevuti. Visto che puntano tutto sulla velocità, sono spesso ottimizzati per conservare i dati in memoria dinamica e salvarli periodicamente su disco in modo da garantire, sia un certo livello di efficienza delle risposta, sia la persistenza dei dati. Questa rapidità di interazione, oltre alla struttura a dizionario, li rende ideali per salvare, ad esempio, dati di sessione in ambiente client/server, dove la chiave rappresente l'ID di sessione, mentre i dati sono memorizzati come valori.
  Tra i database più utilizzati in questa categoria vi sono **Apache Cassandra** (utilizzato da Facebook e Twitter), **Microsoft Azure** e **Redis**

- **Graph** memorizzano le informazioni con strutture a grafi che rappresentano una rete di nodi collegati tra loro da archi. Le informazioni possono essere custodite sia nei nodi sia negli archi. La forza di questa tipologia è tutto l'insieme del valore informativo che si può estrapolare ricostruendo percorsi attraverso il grafo. Gestire i grafi non è mai impresa troppo semplice, proprio per la loro struttura intricata, ma il loro contributo è decisivo in casi d'uso in cui si hanno **relazioni fortemente correlate** tra loro. Si pensi, ad esempio, ad una rete di trasporti: se ogni nodo rappresenta una città italiana, il collegamento tra due nodi contiene come informazione la distanza chilometrica che tra di esse intercorre. Con un database a grafo si può trovare un percorso tra due città lontane – quindi non collegate direttamente – e, sommando la distanza di ogni tratto, si potrebbe calcolare la distanza totale dalla partenza alla destinazione, più ogni altra grandezza derivata (come tempi, costi, eccetera).
  Uno dei database più utilizzati in questa categoria è **OrientDB**

Linee Guida per la progettazione di un database noSQL

Come detto nei DBMS NoSQL a documenti scompare il concetto di **relazione**. Ogni collezione contiene tutti i dati necessari. I meccanismi con cui vengono collegate le informazioni possono essere due:

- **Embedding**: significa annidare un **oggetto** all'interno di un altro. Questa tecnica sostituisce molto spesso le relazioni 1:1 e 1:N. È tuttavia sconsigliabile utilizzarla quando i documenti (quello annidato e quello che lo contiene) crescono di dimensione in maniera sproporzionata tra loro, oppure se la frequenza di accesso ad uno dei due è molto minore rispetto all'altro;
- **Referencing**: più simile alle relazioni dei RDBMS, e consiste nel fare in modo che un documento contenga, tra i suoi dati, un **vettore enumerativo** con gli ID di tutti i documenti collegati.
  Utile per realizzare strutture complesse, relazioni N:M oppure casistiche non rientranti tra quelle consigliate per l'embedding.

Ciò non toglie che il database possa essere costruito con stile 'relazionale' con informazioni differenti memorizzate in collezioni differenti e indicazione della chiave esterna
Si ricade però nel concetto di join che dovrà essere realizzato tramite due query, una per il recupero del codice in una prima tabella, ed una seconda query che utilizzerà il codice per accedere alla seconda tabella

Esempio 1: Embedding

```
{    "_id" :  ObjectId("57bdd4a1eb426816499fb1be"),
     "Titolo" : "Promessi Sposi",
     "Autore" : "Alessandro Manzoni",
     "Pubblicazione" : {
         "Editore" : "PremiateEdizioni",
         "Anno_pubblicazione" : 2012,
         "Numero_pagine" : 732
     }
}
```

All'interno del documento **`libro`**, sono inseirte tutte le informazioni relative alla sua pubblicazione fisica, raccolte all'interno di un oggetto annidato. In questo modo, con il recupero del documento-libro, avremo implicitamente anche il recupero delle informazioni relative alla pubblicazione. Tuttavia, ciò potrebbe comportare la presenza di oggetti annidati simili in documenti diversi, con conseguente ridondanza dei dati, il male da cui i database relazionali hanno sempre tentato di sfuggire. Velocità contro consistenza.

La tecnica dell'embedding consente di inserire oggetti annidati multipli, raggruppandoli ad esempio in un array. Ad esempio i piloti di una scuderia di F1 possono essere inseriti direttamente all'interno dell'oggetto relativo alla scuderia:

```
{
     "_id" : ObjectId("57bdd826eb426816499fb1bf"),
     "Nome" : "Ferrari",
     "Motore" : "Ferrari",
     "Piloti" : [
          {  "Nome" : "Sebastian Vettel",
             "DataNascita" : 3/7/1987,
             "Nazione" : Germania
        },
        {    "Nome" : "Charles Leclerc",
             "DataNascita " : 16/10/1997,
             "Nazione" : Principato di Monaco
        }
     ]
}
```

Esempio 2: Embedding and Referencing

Si vuole gestire il parco collaboratori di un'azienda informatica che archivia, per ognuno di essi, un insieme di competenze settoriali che viene valutato con un voto compreso tra 4 e 10. Oltre ai collaboratori, l'azienda tiene nota dei progetti avviati, e ad ognuno associa un elenco di collaboratori che ne compongono il team.

Soluzione SQL : 3 Entità

- I collaboratori,
- Le competenze
- I progetti.

Due Relazioni (entrambe di tipo N:M) :
collaboratori - competenze
collaboratori - progetti

Soluzione noSQL

Nel caso di un database NoSQL vengono create solo due *collection*: i **Progetti** ed i **Collaboratori**.

Le competenze, pur essendo in relazione N:M con i collaboratori, risultano perfettamente idonee ad essere annidate all'interno del collaboratore (in modo simile agli attributi vettoriali di un sistema relazionale), ripetendo eventualmente le stesse competenze all'interno di tutti i collaboratori che dispongono di quella competenza. La relazione viene pertanto risolta tramite l'utilizzo di un oggetto embedded.

La seconda relazione viene invece risolta creando all'interno della collection Progetti un array di ObjectId relativi agli ID dei collaboratori che partecipano al progetto.
Si potrebbe ancora valutare se si vogliono memorizzare all'interno del documento Collaboratore gli ObjectId dei progetti ai quali il collaboratore partecipa. Dipende se interessa eseguire questa ricerca oppure no.

struttura interna dei documenti

Per il collaboratore si utilizzano 4 campi: nome, cognome, luogo e data di nascita.
Per le competenze si utilizzano delle proprietà variabili tante quante sono le competenze, ognuna delle quali valorizzata con un numero intero compreso tra 4 e 10.
Per ogni progetto si utilizzano i campi: nome e data di inizio

Collaboratore (RoboMongo)

```
(1) ObjectId("57bad4515df5d42060857de3")   { 6 fields }
    _id                                    ObjectId("57bad4515df5d42060857de3")
    nome                                   Luca
    cognome                                Rossi
    data_di_nascita                        1986-10-13 23:00:00.000Z
    luogo_di_nascita                       Milano
    competenze                             { 3 fields }
        Java                               4
        Javascript                         6      // livello della competenza
        PHP                                9
```

Progetto (RoboMongo)

```
(1) ObjectId("57badc945df58b9c5c91edb2")   { 3 fields }
    _id                                    ObjectId("57badc945df58b9c5c91edb2")
    nome                                   Aurora
    membri                                 [ 4 elements ]
        [0]                                ObjectId("57badc945df58b9c5c91eda8")
        [1]                                ObjectId("57badc945df58b9c5c91eda9")
        [2]                                ObjectId("57badc945df58b9c5c91edaa")
        [3]                                ObjectId("57badc945df58b9c5c91edab")
```

Esempio 3 Embedding and Referencing

Si consideri il caso di un blog in cui vengono raccolti dei post basati sulle seguenti regole:

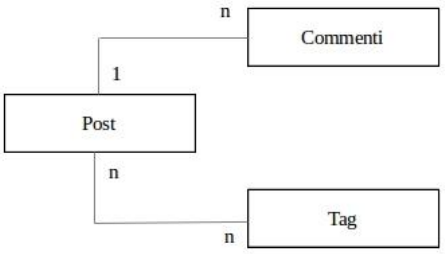
- ogni post può essere commentato più volte
- ogni post viene classificato con dei tag, delle etichette che specificano gli argomenti salienti. (ogni post può essere associato a più tag e, viceversa, ogni tag può essere usato per più post)

Si assume inoltre che:

- i commenti verranno letti e modificati solo in congiunzione con un post;
- quando si carica un post dal database questo venga raccolto con tutta la sua lista di commenti;
- la ricerca dei post e dei tag deve essere indipendente. Può capitare di leggere un tag e recuperare i post associati, oppure può essere necessario leggere un post e recuperare tutti i tag relativi.

Soluzione SQL

In un database relazionale, una classica rappresentazione del problema potrebbe essere la seguente:



Soluzione noSQL

- La collection dei ***Post*** sarà la collection principale del sistema;
- la relazione con i commenti può essere resa con **l'embedding.** Si tratta di una relazione 1:N, inoltre si è deciso che i commenti siano sempre abbinati ad un post.
  Tutti i commenti ad un post verranno recuperati al momento del caricamento del post;
- la relazione N:M tra ***Post*** e ***Tag*** viene risolta tramite tramite referencing. Ogni ***Tag*** conterrà l'elenco degli id dei post relativi, mentre ogni ***Post*** conterrà al suo interno l'elenco dei tag associati.

```
db.post.insert( {
   "titolo": "benvenuto in mongoDB",
   "autore": "Matteo Bianchi",
   "contenuto": "this is the content",
   "commenti": [ { },  { }. { },  etc],
   "elenco_tag": [ _id, _id, _id, etc]
} );
```

- Il campo ***commenti*** contiene oggetti JSON embedded, riportanti ad es testo ed autore del commento.
- Il campo ***elenco_tag*** potrebbe essere costituito da un vettore di _id relativi ai tag associati

Esempio 4

Si supponga di voler memorizzare le informazioni relative ad infrastruttura e dipendenti di una certa azienda che presenta sedi in tutte le regioni italiane. Si potrebbe ipotizzare di:

- Definire una **collezione** per ogni regione
- All'interno di ogni collezione un **documento** per ogni sede dell'azienda in quella regione.
  Ogni sede presenta alcuni campi relativi all'infrastruttura di quella sede
- All'interno di ogni documento relativo all'azienda un vettore di **documenti annidati,** uno per ogni dipendente delle varie sedi
- Ogni collezione può essere memorizzata sul dbms server di quella regione

Mongo DB

MongoDB (da **humongous** = gigantesto) è un database noSQL, **document oriented**, schema-free. Secondo recenti statistiche, è il DBMS noSQL più utilizzato al mondo (lo utilizzano tra gli altri eBay, Foursquare, SourceForgee ed il New York Times).

MongoDB è stato sviluppato a partire dal 2007 in C++ dalla piccola società americana 10gen (oggi diventata MongoDB Inc.) e rilasciato sotto una combinazione della GNU Affero General Public License e dell'Apache License. La versione base è libera e open source.

Caratteristiche Fondamentali

Supporta la replicazione **multi-server**, in modo da garantire la persistenza dei dati.
Dalla versione 2.0 il **journaling** è attivo per default, il che garantisce il recupero rapido dei dati di un server che sia andato in crash o che abbia subito una perdita di alimentazione.

MongoDB può gestire più database ognuno dei quali può contenere più **collezioni**.
Una collezione è una raccolta di **documenti**. Ogni documento può avere un suo set specifico di **campi.**

Volendo fare una analogia con gli R-DBMS si potrebbe dire che :

collezione -> tabella
documento -> riga (**record** della tabella)
campo -> campo del record

- Il valore di un campo può essere a sua volta un altro JSON Object
- Più documenti JSON racchiusi tra parentesi quadre costituiscono un vettore di oggetti JSON
- Ogni documento deve essere identificato da un valore univoco, detto **ObjectID** e denominato **_id**
  Il campo **_id** può essere definito esplicitamente in fase di inserimento oppure si può lasciare che sia MongoDB a generare un valore. Può essere indifferentemente numerico (1, 2, 3, etc) oppure stringa ("a1", "2", "a3", etc). Quello generato da mongo è di tipo **ObjectID**. Per impostazione predefinita è indicizzato. Se si imposta un _id già esistente, viene generato un errore di chiave duplicata.
- Su ogni collezione è possibile definire degli **indici** analoghi agli indici degli R-DBMS.
- Quando si chiedono dati a MongoDB questi restituisce un puntatore al set dei risultati chiamato **cursore** che è analogo al puntatore a recordset restituito da un R-DBMS. Con il cursore si possono compiere operazioni come contare i documenti o spostarsi in avanti, prima ancora di scaricare i dati.

Download e installazione

Scegliere **Community Server** (v **3.6.3** al 10/4/2018. Viene richiesto di installare insieme anche Compass; occorre però deselezionare Compass ed eventualmente scaricarlo ed installarlo separatamente).

Prima di iniziare occorre creare nella cartella bin un nuovo file di testo chiamato **mongodb.config** e costituito dalla seguente riga indicante il il percorso dove salvare i databases:

```
dbpath= C:\mongoDB
```

La cartella di installazione Programmi / mongoDB è protetta, quindi conviene scegliere un altro path. Assicurarsi che il dbpath scelto esista.

Dalla versione 4.2 (ottobre 2019) in poi :

- In fase di installazione viene richiesto dove posizionare il database (Data Direcotry).
  La cartella proposta è **`C:\Program Files\MongoDB\Server\4.4\data`**
  che può essere modificata in fase di installazione oppure anche successivamente

- Sulla base di questa impostazione l'installer crea automaticamente un file **mongod.cfg** che, attraverso la variabile **dbPath**, indica al server dove sono posizionati i databases.
  dbPath= C:\Program Files\MongoDB\Server\4.4\data *(scritta senza virgolette)*
- Oltre a **dbPath**, viene creata anche una variabile analoga per la gestione dei files di **log**.
  Queste variabili possono eventualmente essere modficate anche successivamente
- Viene infine chiesto se installare mongo come **servizio** (non sarà più necessario avviare ogni volta manualmente il server). Per vedere i servizi CTRL+ALT+CANC / **Gestione Attività** (Task Manager)
- In compenso non vengono più installate le **MongoDB Database Tools** (ad es mongoimport) che devono essere installate a parte, sempre dal medesimo sito (terzo pulsante : ***Tools***)

MongoDB è costituito da due applicazioni principali che sono:
mongod che è il processo server, **mongo** che è la shell (il client).

Avvio manuale del Server

c:\mongodb\bin > **mongod --config c:\programmi\mongodb\server\3.6\bin\mongodb.config**
Occorre specificare, tramite l'opzione –config, il path assoluto del file di configurazione mongodb.config.
Il server avviato risponde con il messaggio "Waiting for connections on port **27017**"
c:\mongodb\bin > **mongod --help** // Finestra di aiuto

Utilizzo del Client a linea di comando

c:\mongodb\bin > **mongo** *(creare un file **mongo.bat** oppure aggiornare il **path**)*

Viene aperta una **shell** (**interprete dei comandi**) connessa al server. Si tratta di una shell java script in cui si possono richiamare i metodi disponibili lato client. Trattandosi di metodi devono sempre avere le parentesi **tonde finali**. Omettendo le tonde si vedrà il contenuto del metodo invece che la sua l'esecuzione.

db.version() Restituisce il numero della versione installata
db.help() Elenco dei comandi utilizzabili sul client
exit Chiude il client

show dbs oppure **db.adminCommand({ listDatabases: 1 })**
Motrano l'elenco dei database gestiti dal dbms. Il secondo comando mostra più informazioni

use Studenti
consente di cambiare il database attivo. Non importa se il database non esiste ancora. Quando creeremo la prima collezione, allora verrà creato anche il database. Idem per le tabelle. Quando verrà eseguito il primo insert verrà creata anche la tabella. I nomi dei database e delle tabelle sono trattati come variabili, per cui sono **Case Sensitive**. Quando si ha un database attivo tutti i comandi sul database iniziano con **db.---**

db.getName() oppure semplicemente **db**
Visualizza il nome del database corrente

db.getCollectionNames();
restituisce l'elenco delle collezioni (tabelle) presenti nel database corrente. Poiché le collezioni sono schema-less non occorre crearle esplicitamente. Verranno create automaticamente al momento dell'inserimento del primo documento.

db.dropDatabase();
cancella il database corrente

db.users.drop();
cancella la tabella users

db.logout()
Chiude la connessione al database corrente

L'utility mongoimport.exe (da lanciare esternamente rispetto a client mongo)

```
mongoimport --db <dbName>   --collection <collName>  --file <filename.json>
mongoimport --db 5b_unicorns --collection unicorns    --file c:\mongodb\5b_unicorns.txt
```

Il file json deve avere il seguente formato:

- deve contenere un solo oggetto per riga
- gli oggetti non devono essere separati tramite virgola
- dalla versione 4.2 il json DEVE essere well formed, cioè le chiavi devono avere gli apici doppi e per i valori non numerici occorre necessariamente utilizzare gli apici doppi

```
{ "name": "Widget 1", "desc": "This is Widget 1" }
{ "name": "Widget 2", "desc": "This is Widget 2" }
```

L'opzione **--type json** è il default per cui può essere omessa
L'opzione **--jsonArray** consente di importare un file json contenente un vettore di oggetti in formato 'classico', con la virgola come separatore:

```
[  { name: "Widget 1", desc: "This is Widget 1" },
   { name: "Widget 2", desc: "This is Widget 2" }  ]
```

In alternativa è possibile eseguire l'import tramite COMPASS creando prima Database e Collection e poi, all'interno della Collection, utilizzando il comando **ADD DATA.** Sono ammessi SOLO vettori di oggetti.

L'insermento dei dati ed il formato BSON

I documenti importabili in MongoDB possono essere scritti come **sequenze di JSON scritti uno dopo l'altro senza nessun separatore**. In realtà il formato utilizzabile per la scrittura di un documento importabile è il cosiddetto formato **BSON** (**B**inary J**SON**), che è una estensione del JSON con l'aggiunta di alcuni operatori che consentono di tipizzare maggiormente i dati. Esempio:

```
{  "_id" : {"$oid": "5f8627503e5c67374ef76400"},
   "volume" : {"$numberLong" : "50" },
   "data"  : { "$date":"2001-09-15T00:00:00Z" } }
```

BSON, come JSON consente, di assegnare come valore ad una proprietà un altro oggetto BSON o un array di oggetti BSON. Ciò conferisce "profondità" al documento e pone la necessità di affrontare un nuovo step nella progettazione: l'individuazione dei documenti da inserire all'interno di altri.

Accesso ai dati

Per accedere a proprietà e metodi di una tabella del database corrente si utilizza la sintassi
db.nomeTabella.metodo()
db.studenti.count(); restituisce il numero di documenti presenti nella collezione
Il punto e virgola finale è facoltativo. Serve eventualmente per accodare più comandi in cascata.

Il metodo insert()

Si aspetta come parametro un semplice object :

```
db.studenti.insert({nome:'Aurora', eta:16, residenza:'Fossano'});
db.studenti.insert({_id:1, nome:'Leto', spec:'informatica', promosso:true});
```

Campi Vettoriali: `db.studenti.insert({nome:'Pippo', hobbies:["nuoto", "tennis"] });`

Campi Annidati:
```
db.studenti.insert({nome:'Pippo',
            scuola:{denominazione:"vallauri", citta:"fossano"} });
```

Le query di selezione

db.studenti.find().**toArray();** oppure .pretty();
restituisce un recordeset (cursore) con tutti i documenti presenti nella collezione.
Questo recordset può essere visualizzato cos'ì come è, oppure si può applicare il metodo **.toArray()** che restituisce i documenti all'interno di un **vettore enumerativo di object**. Il metodo **.pretty()** è analogo.

La condizione **where** viene impostata tramite un Object del tipo **{campo: valore}**
Il valore della stringa da assegnare al campo è di tipo case sensitive

Si usano gli operatori **$lt**, **$lte**, **$gt**, **$gte** **$ne** per indicare le condizioni di minore (less than), minore o uguale (less than or equal), maggiore (greater than), maggiore o uguale e diverso (not equal).

Gli operatori $ vengono trattati come se fossero degli pseudo-oggetti annidati, per cui devono sempre essere racchiusi tra parentesi graffe { }. Esempi:

```
db.unicorns.find( {weight:700}} )
db.unicorns.find( {weight:{$eq: 700}} )   // uguale al precedente
db.unicorns.find( {weight:{$gte:701}} )
db.unicorns.find( {weight:{$gte:700, $lte: 800}} )
```

L'operatore $not

```
price : {$not:{$gt:2}}
```

Gli operatori $and e $or

Si aspettano come parametro un vettore enumerativo di object:

```
{ $and: [ {<expression1>},  {<expression2>}, …… , {<expressionN>} ] }
db.unicorns.find({$and: [{gender:'m'},{loves:'apple'},{weight:{$lt:500}}] })

{ $or: [ {<expression1>},  {<expression2>}, …… , {<expressionN>} ] }
db.unicorns.find({$or: [{gender:'m'},{loves:'apple'},{weight:{$lt:500}}] })
```

Condizioni Annidate. Le **$and** e **$or** possono ovviamente essere annidate.
Esempio: unicorni femmina *che amano le mele* oppure *che pesano meno di 500 libbre*

```
db.unicorns.find({
      $and: [ {gender: 'f'}, {$or:[{loves: 'apple'}, {weight: {$lt: 500 }}]} ]
})
```

Forma and abbreviata

Soltanto per la AND (non per la OR) è disponibile anche una forma **abbreviata** costituita da un unico oggetto con più campi separati da virgola, come per la insert.

```
db.unicorns.find({ gender: 'f',  weight: {$lt: 500} })
```

Notare che eventuali **condizioni multiple su uno stesso campo** devono essere raggruppate insieme e NON scritte separatamente, altrimenti la seconda maschera lla prima:

```
db.unicorns.find({weight:{$gte:700, $lte: 800}} )  // ok
db.unicorns.find({weight:{$gte:700}, weight:{$lte: 800}} )  // nok
```

Campi vettoriali

- E' possibile passare come parametro un vettore. In tal caso ricerca gli unicorni il cui vettore loves coincide con il vettore ricevuto, cioè contiene le stesse voci nella stessa identica sequenza.

```
db.unicorns.find( { loves: ['apple','orange']} )
```

- passare un **singolo valore**. In tal caso ricerca gli unicorni il cui vettore loves contiene il valore passato come parametro:
  `db.unicorns.find({ gender:'f', loves:'apple'})`
- utilizzare l'operatore **$in:** seguito da un vettore di voci. La query verifica se uno dei valori del vettore soddisfa alla condizione. Per esempio : unicorni che amano le mele **oppure** le arance :
  `db.unicorns.find( { loves: {$in:['apple', 'orange']}} )`
  oppure
  `var parametro="apple, orange";`
  `db.unicorns.find( { loves: {$in: parametro.split(",") }} )`
- utilizzare l'operatore **$nin:** seguito da un vettore di voci.
  Per esempio : unicorni che non amano **né** le mele **nè** le arance :
  `db.unicorns.find( { loves: {$nin:['apple', 'orange']}} )`
- utilizzare l'operatore **$all:** seguito da un vettore di voci.
  Per esempio : unicorni che amano **sia** le mele **sia** le arance :
  `db.unicorns.find( { loves: {$all:['apple', 'orange']}} )`
  che è equivalente a: `{"$and":[{"loves":"apple"},{"loves":"orange"}]}`
  In questo caso **NON** è consentita la forma breve, perché conterrebbe 2 campi con lo stesso nome
- utilizzare l'operatore **$pull** che elimina da un campo vettoriale la / le voci indicate
  `db.unicorns.update( {}, { $pull: { loves: {$in:["apple","orange"]} } } ))`
  Elimina le voci `:"apple" e "orange"` da tutti i vettori di tutti gli unicorni

$in e $nin: possono essere utilizzati anche su campi scalari. $all su campi scalari ha poco senso
Per esempio : unicorni che hanno il pelo marrone oppure grigio :
`db.unicorns.find( { hair: {$in:['brown', 'grey']}} )`

L'operatore $exists

Può essere usato per verificare la presenza o l'assenza di un campo.
Ad esempio Unicorni privi del campo vampires
`db.unicorns.find({ vampires: {"$exists": false}}).`

L'operatore $regex

Può essere usato sulle stringhe per filtrare una porzione di stringa sulla base di una regular expression, implementando in pratica quello che è il **LIKE** nell'SQL tradizionale. Tenere presente che il **find** è SEMPRE case sensitive; per renderlo case insensitive si può usare una regex con l'opzione ***i***

Esempio: nomi che iniziano con A oppure B

```
db.unicorns.find({"nome":/^[AB]/i})     // oppure
db.unicorns.find({"nome":"^[AB]"})      // oppure
var regex = new RegExp("^[AB]", "i");
db.unicorns.find({"nome": regex}})      // oppure
db.unicorns.find({"nome":{"$regex" : regex}})
```

Esempio: nomi delle persone che contengono la parola Maria : Maria Grazia, Anna Maria,

```
db.unicorns.find( {"name":{"$regex" : "\\b[Mm]aria\\b"}} )
```

Tenere presente che la sintassi breve non consente l'utilizzo delle variabili.

Le regex possono essere anche utilizzate direttamente anche all'interno degli operatori $in, $nin, $all, $pull, sia in forma breve che in forma estesa, nel qual caso occorre passare un vettore di new RegExp :

```
{ $in: [new RegExp("R1"),  new RegExp("R1"),  new RegExp("R1"),] }
```

Selezione di un record sulla base dell'_id :

```
db.unicorns.find({_id: 2})
db.unicorns.find({_id: "a13"})
db.unicorns.find({_id: ObjectId("5e28c5a6912053b07bf092dc")})
```

Per eseguire invece una ricerca sull'ID tramite il driver mongoDB di nodejs occorre utilizzare la sintassi:

```
var ObjectId = require('mongodb').ObjectId;
var oid = new ObjectId("5e28c5a6912053b07bf092dc"); // id del record da cercare
db.unicorns.find({_id:oid})
```

Utilizzo di variabili e istruzioni javascript all'interno di mongoclient

```
var vet = [];
vet.push({gender:'m'});    vet.push({loves:'apple'});
print(vet[1].loves);     // oppure semplicemente nome della variabile / INVIO
var filter={$and: vet};
db.unicorns.find(filter);
```

per rimuovere una variabile dalla shell: **`delete varName`**
(che però in javascript agisce soltanto su variabili dichiarate senza VAR e senza LET)

Selettori di campo (projection)

Per default il server restituisce il documento completo. Il metodo **`find()`** di mongoClient accetta un **secondo parametro** chiamato "projection" che consente di specificare l'elenco dei campi da visualizzare.

Nelle utlime versioni del driver **mongoDB embedded** di nodejs**,** il metodo project deve essere inserito NON come secondo parametro, ma applicato in cascata a find() :

```
db.unicorns.find({}).project({name:1, _id:0})    // return name only
```

Impostando alcuni campi ad 1 vengono visualizzati SOLTANTO i campi impostati a 1
Impostando alcuni campi ad 0 vengono visualizzati SOLTANTO i campi rimanenti
Non è possibile impostare alcuni campi a 1 ed altri a 0 (Projection cannot have a mix of inclusion and exlusion). Fa eccezione l'ID che viene mostrato in entrambi i casi e può essere nascosto solo impostando 0

Ordinamenti: il metodo .sort()

Può essere applicato in cascata a find()

```
db.unicorns.find({}).project({name:1, _id:0}).sort({name: 1, vampires: -1})
```

1 = crescente, **-**1 = decrescente

A differenza di aggregate() in cui ogni metodo opera sul recordset restituito dal metodo precedente, nel caso di find():

- L'ordine con cui vengono scritti i metodi è **irrilevante** .
- sort(), anche se scritto dopo rispetto a project(), vede comunque tutti i campi

I metodi .limit() e .skip()

```
db.unicorns.find().sort({weight: -1}).limit(2).skip(1)
```

Vengono individuati il 2° e 3° unicorno con maggior peso (due record visualizzati).

Anche in questo caso, a differenza di aggregate() l'ordine è irrilevante; viene eseguito sempre ***prima*** lo skip e solo ***dopo*** il limit. In aggregate invece vengono eseguiti nell'ordine in cui sono scritti.

L'impostazione **`limt(0)`** ha come significato la visualizzazione di tutti i record

Conteggi: il metodo .count()

In alternatica a .toArray(), in coda a find, si puè utilizzare il metodo .count()

```
db.unicorns.find({vampires: {$gt: 50}}).count()
```

Numero di unicorni che hanno ucciso più di 40 vampiri.

Il metodo `findOne()`

In alternativa a find() si può utilizzare il metodo findOne() che ottimizza l'esecuzione di query che restitusicono un solo record. In questo caso non è ammesso il metodo finale .toArray().

Il metodo `distinct()`

```
db.unicorns.distinct("loves");
```

Restituisce l'elenco di tutti i frutti amati dagli unicorni, con ogni frutto espresso una sola volta.
Eseguibile da mongo client anche con la seguente sintassi:

```
db.runCommand({ distinct: "unicorns", key: "loves" });
```

- Consente la restituzione di un **unico campo**
- Restituisce un **vettore di stringhe** (anche senza l'aggiunta del metodo .toArray() che non è supportato). La funzione di callback viene passata direttamente come ultimo parametro.
- Accetta come secondo parametro un object che consente di specificare una **query di ricerca**

```
db.unicorns.distinct("loves", {gender: 'f'});
```

- Utilizzando mongoDB Driver non è consentito abbinare il metodo .**sort**()

Per implementare il SORT o eseguire query più complesse occorre utilizzare il metodo **aggregate()**

Il metodo remove()

Utilizza la stessa identica sintassi di find(). Rimuove tutti i record individuati dal filtro

```
db.unicorns.remove({_id: "a13"})
```

Il metodo update()

Utilizza due parametri:

- Il **primo** parametro rappresenta il filtro di selezione con la stessa sintassi del find
- Il **secondo** parametro rappresenta i campi da aggiornare
- È disponibile un **terzo** parametro di opzioni (come ad esempio l'opzione **multi**)

```
db.unicorns.update({name: 'Roodles'}, {$set: {weight: 590}})
```

mongodb per default **aggiorna un solo documento alla volta**. Nel caso precedente il record da aggiornare era univoco, quindi non c'erano problemi. Altrimenti verrebbe aggiornato soltanto il PRIMO record incontrato. Per aggiornare contemporaneamente più record occorre utilizzare l'opzione **multi**.

Il primo parametro può anche essere un **json vuoto**, nel qual caso l'aggiornamento viene tipicamente applicato soltanto al primo record trovato (o a tutti i record nel caso si utilizzi l'opzione **multi**)

Se uno dei campi da aggiornare non esiste, **viene creato automaticamente**:

Oltre all'operatore **$set** sono disponibili altri operatori di update quali:

$unset{ } rimuove il campo indicato.

```
db.unicorns.update({name: 'Pilot'}, {$unset: {vampires: 1}})
```

$inc{ } incrementa / decrementa campi di tipo numerico (accetta sia interi che float).

```
db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})
```

$mul{ } moltiplica campi di tipo numerico (accetta sia interi che float).

```
db.unicorns.update({name: 'Pilot'}, {$mul: {vampires: 3}})
```

$rename{ } rinomina il campo indicato.

```
db.unicorns.update({name: 'Pilot'}, {$rename: {'vampires':'nVampiri', ……}})
```

$push{ } aggiunge nuovi elementi in campi di tipo vettoriale

```
db.unicorns.update({name: 'Aurora'},{$push: {loves: 'sugar'}})
db.unicorns.update({name: 'Aurora'},{$push: {loves: {$each:["a1", "a2"]}}})
```

$addToSet{ } aggiunge nuovi elementi in campi di tipo vettoriale solo se non sono già esistenti

```
db.unicorns.update({name: 'Aurora'},{$addToSet: {loves: 'sugar'}})
```

In tutti questi casi se il campo NON esiste viene automaticamente creato. E' anche possibile :

- **utilizzare più operatori contemporaneamente**:

```
db.unicorns.update({name:'Aurora'},{$set:{weight:590}, $inc:{vampires:2}})
```

- **settare più campi all'interno dello stesso operatore**:

```
db.unicorns.update({name:'Aurora'},{$set:{weight:590, vaccinated:true}})
```

oppure

```
db.unicorns.update({name:'Aurora'},{$set:{weight:99},$set:{vacinated:true}})
```

Opzioni terzo parametro: multi

Come detto all'inizio l'update aggiorna un solo record. Se vogliamo aggiornare record multipli occorre specificare nel terzo parametro l'opzione **{multi:true}**

```
db.unicorns.update( {}, {$set: {vaccinated: true }}, {multi:true});
```

Opzioni terzo parametro: upsert

Aggiungendo nel terzo parametro l'opzione **{upsert:true},** se il record da aggiornare non viene trovato viene automaticamente creato un nuovo documento con un suo _id:

```
db.unicorns.update({name: 'pippo'},{$inc: {vampires: 1}}, {upsert:true});
```

Se il documento con nome 'pippo' non esiste, viene automaticamente creato con `vampires=1`

Le opzioni **upsert e multi** possono essere specificate contemporaneamente:

```
db.unicorns.update({name:'pippo'},{$inc:{vampires:1}},{upsert:true, multi:true})
```

I metodi del driver mongodb 3.1

Il metodo **insert** è stato deprecato a favore dei metodi
insertOne() che equivale al metodo insert() tradizionale
insertMany() che si aspetta come parametro un vettore di oggetti

Il metodo **remove** è stato deprecato a favore dei metodi
deleteOne() che rimuove un solo record (il primo che soddisfa la condizione di filtro)
deleteMany() che rimuove tutti i record che soddisfano la condizione di filtro

Il metodo update è stato deprecato a favore dei metodi
updateOne() che equivale al metodo update() senza l'opzione {multi:true}
updateMany() che equivale al metdo update() con settata l'opzione {multi:true}

Entrambi questi metodi non supportano un'azione diretta del tipo {"`field`":"`value`"} che nel vecchio metodo update provocava la sostituzione del documento corrente a favore dei nuovi campi

Per la sostituzione di un documento è stato aggiunto il metodo **replaceOne()** che consente la totale sostituzione dei campi con i valori dei nuovi campi passati come parametro
Il record interessato dalla sostituzione è il primo fra quelli individuati dalla condizione di filtro.

Il secondo parametro di replaceOne è un semplice elenco di campi da aggiornare.
Al suo interno **NON** è consentito l'utilizzo degli Update Operators ($set, $inc, etc.).
Tutti i campi non espressamente indicati nel secondo parametro vengono rimossi dal record, compreso il campo indicato nel filtro, che dovrà pertanto essere **replicato** all'interno del secondo parametro, e con esclusione soltanto dell'ID di cui viene mantenuto il vecchio valore.

E' possibile comunque passare anche il campo **_id** che in tal caso sostituirà il valore precedente.
Attenzione però a non impostare un valore già esistente, perchè altrimenti il metodo va in errore.

Come **terzo parametro** si può passare `{upsert:true}` nel qual caso se l'oggetto non esiste viene creato

```
db.restaurant.replaceOne(
      { "name" : "Pizza Rat's Pizzaria" },
      { "name" : "Pizza Rat's Pizzaria", "_id":4, "Borough":"Manhattan"},
      { upsert: true }
);
```

Un'operazione simile potrebbe essere fatta con **updateOne** utilizzando il **$set** davanti al secondo oggetto:

```
db.restaurant.replaceOne(
      { "name" : "Pizza Rat's Pizzaria" },
      {$set:{ "name" : "Pizza Rat's Pizzaria", "Borough":"Manhattan"}},
      { upsert: true }  );
```

Però nel caso del **replaceOne** i campi non settati vengono rimossi,
mentre nel caso di **updateOne** i campi non settati mantengono il loro vecchio valore.

Non esiste un metodo **replaceMany**

Il metodo aggregate()

Mentre **find()** esegue un singolo comando (costituito eventualmente da una AND di tante condizioni), il metodo **aggregate()** si aspetta come parametro un **vettore enumerativo contenente un elenco ordinato di operatori di aggregazione** (pipeline) che:

- **vengono eseguito sequenzialmente uno dopo l'altro**
- **ognuno opera a catena sul recordset restituito dall'operatore precedente**

E' simile al group by degli R-DBMS ma con notevoli potenzialità aggiuntive. Gli operatori principali sono:

`$match`	applica un filtro sul recordset ricevuto ed utilizza la stessa sintassi del find(). E' possibile utilizzare più macth successvi per concatenare condizioni multiple.
`$group`	raggruppa i record sulla base di uno o più campi indicati.
`$project`	consente di realizzare una proiezione sui campi che dovranno essere utilizzati dall'operatore successivo. Accetta al suo interno l'utilizzo delle funzioni di aggregazione (come ad esempio **`$avg()`**) in modo da poter eseguire calcoli che coinvolgono più colonne oppure su campi singoli costituiti da vettori enumerativi
`$unwind`	consente di 'srotolare' un campo costituito da un vettore enumerativo producendo un elenco di **N documenti** ciascuno contenente una sola delle voci presenti all'interno del vettore enumerativo.
`$sort`	esegue un ordinamento sul recordset finale. Accetta come parametro un JSON avente come chiavi una o più delle chiavi restituite dall'operatore **`$group`**

`$limit` e **`$skip`** A differenza del find(), in cui viene eseguito SEMPRE prima lo SKIP, questa volta i due operatori vengono eseguiti nell'ordine in cui sono scritti:

`{$limit:100}, {$skip:10}` Ne prende 100 e poi elimina i primi 10 [10-100]
`{$skip:10}, {$limit:100}` Ne esclude 10 poi prende i primi 100 [10-110]

Le opzioni **`$match e $project`** possono essere utilizzate anche **DOPO** l'opzione **`$group`**, nel qual caso agiscono sui gruppi restituiti da **`$group`**, consentendo di fatto di realizzare il tipico costrutto having ($match) oppure di rielaborare ulteriormente le colonne restituite da $group ($project). L'opzione **`$group`** a sua volta può essere richiamata una seconda volta per raggruppare ulteriormente i gruppi già creati dal primo $group ed eventualmente filtrati tramite $match e $project.

Esempio 1

Sia data la seguente collezione **`orders`** in cui **`nDettagli`** è un vettore contenente :
la quantità di dettagli di **tipo A**, la quantità di dettagli di **tipo B**, la quantità di dettagli di **tipo C**

```
{ _id: 1,  cust_id: "abc1",  status: "A",  amount: 50,  qta:30,  nDettagli : [7, 12, 11]  }
{ _id: 2,  cust_id: "xyz1",  status: "A",  amount: 100, qta:46,  nDettagli : [10, 16, 20]  }
{ _id: 3,  cust_id: "xyz1",  status: "D",  amount: 25,  qta:70,  nDettagli : [30, 20, 20]  }
{ _id: 4,  cust_id: "xyz1",  status: "D",  amount: 125, qta:20,  nDettagli  : [5, 10, 5]  }
{ _id: 5,  cust_id: "abc1",  status: "A",  amount: 25,  qta:32,  nDettagli : [10, 10, 12]  }
```

Si vogliono selezionare i record che hanno status A, raggrupparli secondo il campo **cust_id** e, per ogni gruppo, eseguire la somma del campo **`amount`** (importo unitario)

```
db.orders.aggregate([
        { $match: { status: "A" } },
        { $group: { _id:"$cust_id", amount: { $sum:"$amount" } } },
        { $sort: { amount: -1 } }   ]);
```

La query produrrà un recordset costituito da 2 (o più) campi:
`_id` che è obbligatorio e indica il campo da utilizzare per il raggruppamento;
`amount` che rappresenta il nome del secondo campo che conterrà la somma di ttui i valori del campo `"$amount"` di ogni record appartenente al gruppo.

Note:

- Quando un campo viene utilizzato per operazioni di aggregazione / conteggio **deve** essere preceduto dal $
- I nomi dei campi **$cast_id** e **$amount** devono essere racchiusi tra virgolette.

Esempio 2 : Utilizzo della funzione di aggregazione $avg

Con riferimento alla collezione iniziale si può impostare la seguente query

```
db.orders.aggregate([{
        $group:  {
            _id: "$cust_id",
            avgAmount: { $avg: "$amount" },
            avgTotal: { $avg: { $multiply: ["$qta", "$amount"] } }
        }
}]);
```

Esempio 3

Conteggio degli unicorni maschi e degli unicorni femmina

```
db.unicorns.aggregate([{$group:{_id: '$gender', totale: {$sum:1}}}])
   {_id : "m",  totale: 7 }
   {_id : "f",  totale: 6 }
```

// oppure, volendo cambiare il nome della colonna usata per il raggruppamento:

```
db.unicorns.aggregate([{$group:{_id:{gender:'$gender'}, totale:{$sum:1}}}])
   {_id : {gender : "m"}, total:7 }
   {_id : {gender : "f"}, total:6 }
```

Esempio 4

Calcolare il numero medio di vampiri uccisi dagli unicorni femmina e dagli unicorni maschi

```
db.unicorns.aggregate([{$group:{_id:{gender:'$gender'},
                                mediaVampiri:{$avg:"$vampires"}}}])
```

Esempio 5 - Aggregazione su chiavi multiple

La sintassi dell'esempio precedente consente di raggruppare i record utilizzando chiavi multiple.
Esempio: raggruppare gli unicorni per genere e tipo di pelo

```
db.unicorns.aggregate([
      { "$group": {"_id":{"gender":"$gender", "hair":"$hair"},
                   "nEsemplari":{"$sum":1}} },
      { "$sort" : {"nEsemplari":-1, "_id.gender":-1} }     ])
```

Esempio 6 - Aggregazione sull'intero recordset

Se non si intende utilizzare una chiave di raggruppamento ma si vuole agire su tutti i documenti della collezione, occorre specificare una chiave vuota **_id:{}**, oppure **_id:null**
Es numero medio di vampiri uccisi dagli unicorni complessivamente presenti nella collezione

```
db.unicorns.aggregate([
     {"$group": {"_id":{},  "media":{$avg:"$vampires"}} },
     {"$project":{"_id":0, "media":1}}   ])
```

Esempio 7a Utilizzo delle funzioni di aggregazione all'interno della funzione $project

All'interno del `$project`, **al posto dell'1**, si può anche utilizzare una **funzione di aggregazione** che può eseguire una elaborazione numerica:

- **su un singolo campo** (funzioni di proiezione)
- **su un campo vettoriale,**
- **su colonne numeriche differenti** appartenenti al medesimo record.

Ad esempio, dati i seguenti documenti, in cui per ogni studente sono riportate 7 valutazioni

```
{"_id": 1, "quizzes": [10,6,7], "labs": [5,8], "midterm": 75, "final": 80 }
{"_id": 2, "quizzes": [9,10 ],  "labs": [8,8], "midterm": 80, "final": 95 }
{"_id": 3, "quizzes": [4,5,5],  "labs": [6,5], "midterm": 70, "final": 78 }

db.documents.aggregate([{
     $project: {
       quizAvg: { $avg: "$quizzes"},
       labAvg:  { $avg: "$labs" },
       examAvg: { $avg: ["$midterm","$final"] } }
}])
```

pruduce il seguente risultato:

```
{ "_id" : 1, "quizAvg": 7.66,  "labAvg": 6.5,  "examAvg": 77.5 }
{ "_id" : 2, "quizAvg": 9.50,  "labAvg": 8.0,  "examAvg": 87.5 }
{ "_id" : 3, "quizAvg": 4.66,  "labAvg": 5.5,  "examAvg": 74.0 }
```

Esempio 7b Media delle medie

L'eserczio precedente potrebbe essere completato eseguendo, per ogni tipo di prova, la media complessiva sull'intera classe:

```
db.documents.aggregate([
 {$project: {quizAvg:{$avg:"$quizzes"}, labAvg:{$avg:"$labs"},
                                        examAvg:{$avg:["$final","$midterm"]} }},
 {$group:{_id:{}, mediaQuiz:{$avg:"$quizAvg"}, mediaLab:{$avg:"$labAvg"},
                                               mediaExam:{$avg:"$examAvg"} }}
])
```

NOTA

- Per eseguire una aggregazione in orizzontale si usa **$PROJECT**
- Per eseguire una aggregazione in verticale si usa **$GROUP**

Esempio 6b Utilizzo delle funzioni di proiezione all'interno di $project

Ritornando all'esercizio 6, all'interno del `$project` finale al posto di `"media":1`, si può utilizzare la **funzione di proiezione** **`$round`** sul contenuto del singolo campo in modo da arrotondare il valore:

```
{"$project":{"_id":0, "mediaArrotondata":{"$round":"$media"}}} // oppure
{"$project":{"_id":0, "mediaArrotondata":{"$round":["$media", 2]}}}
```

La prima riga arrotonda alla cifra intera, la seconda riga arrotonda a 2 cifre dopo la virgola
Nota: sul sito uffciale le "funzioni di proiezione" sono classificate come **funzioni di aggregazione.**

Esempio 8 operatori in cascata

Supponendo di avere una collezione di studenti dove, per ogni studente è riportato un array enumerativo con l'elenco dei suoi voti, individuare nome e codice del secondo studente femmina con la media più alta.

```
db.studenti.aggregate([
    { $match: { genere: "f" } },
    { $project: {nome:1, classe:1, media:{$avg: "$voti"}} },
    { $sort: {media: -1 } },
    { $limit:2 },
    { $skip:1 }   ])
```

Notare che, a differenza di quanto avviene con find(), LIMIT e SKIP vengono eseguiti nell'ordine in cui sono scritti, cioè prima il LIMIT (prende i primi due) e dopo lo SKIP (che esclude il primo).

La funzione di aggregazione $first

Quando si creano dei gruppi, tutti i valori di dettaglio che NON fanno parte della chiave vengono persi.
La funzione **$first** utilizzata all'interno di un **$group**, restituisce il valore del primo elemento di quel gruppo. Ha senso quando tutti gli elementi di un gruppo hanno lo stesso valore in quel campo oppure per elenchi ordinati. Supponendo che gli unicorni siano ordinati per numero decrescenti di vampiri uccisi:

```
{ "$group": {"_id":"$hair", "vampires":{"$first":"$vampires"}}
```

Ragruppa gli unicorni sulla base del pelo e, per ogni gruppo, restituisce il numero maggiore di vampiri uccisi da un elemento del gruppo.

Esempio 9: L' operatore $unwind

Con riferimento alla collezione iniziale degli **orders**, si può impostare la seguente query:

```
db.orders.aggregate([
    { $project: { status: 1, nDettagli: 1 } },
    { $unwind: "$nDettagli" },
    { $group: { _id: "$status",  sommaDettagli: { $sum: "$nDettagli" } } }
])
```

Il $**project** seleziona i campi **status** e **nDettagli**
Il $**unwind**, dato il documento `{ status:"A", nDettagli:[7,12,11] }` lo trasforma nei seguenti documenti (temporanei al servizio della query. Privi di _id:

```
{ status:"A", nDettagli:7 }
{ status:"A", nDettagli:12 }
{ status:"A", nDettagli:11 }
```

Il $**group** crea i gruppi sulla base dello status e somma i dettagli relativi ai i documenti con un certo status

Notare che lo stesso risultato poteva essere ottenuto applicando la funzione di aggregazione $sum direttamente all'interno di $project

Esempio 10: L'operatore $expr

L'operatore consente di fare dei confronti tramite $gt, $lt, $eq con l'aggiunta di diverse potenzialità.
Si aspetta come primo parametro uno degli operatori precedenti, e come secondo parametro un vettore di due elementi in cui confronterà la prima voce con la seconda.

- Consente di fare il confronto fra diversi campi di uno stesso record
- Consente di usare al suo interno gran parte delle funzioni di aggregazione

```
collection.find({"$expr": {"$gt": ["$spesa", "$budget"]}})     //spesa>budget
collection.find({"$expr": {"$gt": ["$avg":["$midterm","$final"], 100]}})
collection.find({"$expr": {"$eq": [{"$year":"$nato"}, 1999]}}) //nato nel 1999
```

MongoDB Driver

```
npm install mongodb
var mongo = require('mongodb');
var mongoClient = mongo.MongoClient;
```

Il metodo .connect()

Nella versione **2.x.x** del driver mongodb era possibile passare al metodo connect() il nome del database a cui connettersi ed il metodo connect() restituiva come parametro della funzione di callback un riferimento al database richiesto. Esempio:

```
mongoClient.connect("mongodb://127.0.0.1:27017/unicorns",function(err, db){
```

Nella versione **3.x.x** questa sintassi è stata deprecata ed il metodo connect() esegue semplicemente la connessione al dbms e, come parametro iniettato alla funzione di callback, restituisce un riferimento al client connesso (sintassi peraltro già supportata anche nella versione 2.x.x)

```
mongoClient.connect("mongodb://127.0.0.1:27017", function(err, client){
```

Occorre pertanto aggiungere una seconda riga in cui si specifica il database a cui accedere:

```
    if(!err){ var db = client.db('unicorns');
```

Questo ultimo metodo non può mai andare in errore in quanto, se il database non esiste, viene creato.

Nella versione **4.x.x** occorre aggiungere alla `connect` un secondo parametro relativo alle options:

```
mongoClient.connect("mongodb://127.0.0.1:27017",{useNewUrlParser:true},function (err,c)
```

Il metodo .close()

Dalla versione 3.x.x in avanti il metodo close() non è più un metodo dell'oggetto **db**, ma è diventato un metodo dell'oggetto **client**. Es `client.close();`

Il metodo collection()

L'oggetto db disponde di un metodo collection che consente di accedere ad una singola collezione del database selezionato. Il metodo .collection() è utilizzabile in modalità asincrona oppure sincrona:

```
1)  db.collection("studenti", function(err, collection) { } )
```

Cioè la collezione richiesta viene iniettata come secondo parametro alla funzione di callback insieme all'oggetto **err**. Nel caso però che la collezione non esista all'interno del database, questa viene automaticamente creata, per cui l'errore non può mai verificarsi.

Per cui, a ragione di quanto sopra, è dispobile e preferibile anche la seguente versione sincrona:

```
2)  var collection = db.collection("unicorns");
```

Il metodo count()

```
unicorns.countDocuments(function(err, data) {
      response.writeHead(200, header );
      response.end(JSON.stringify(data));     // N. documenti presenti nella collezione
});
```

NB: Le variabili con contenuto numerico, prima di essere 'passate' ai vari metodi, devono essere convertite in numero tramite `parseInt()`. Vale per tutti i metodi (insert, find, update e delete).

Il metodo **find()**

```
mongoClient.connect("mongodb://127.0.0.1:27017",function(err,client){
  if(!err) {
    var db = client.db('5b_studenti');
    var collection = db.collection("studenti"); // sincrono
    collection.find().sort({_id:1}).toArray(function(err,data){
        if(!err){
          res.writeHead(200,headerText);
          res.end(JSON.stringify(data));
        }
        else
          console.log("Errore esecuzione query\n" + err.message);
        client.close();
    });
  }
  else console.log("Errore connessione al server");
});
```

Nel caso del metodo **find()** l'oggetto **data** è un vettore enumerativo contenente tutti gli oggetti restituiti.
Il metodo **findOne()** ottimizza l'esecuzione di query che restitusicono un solo record. In questo caso la callback va inserita direttamente al termine di find(), senza richiamare il metodo .toArray().

Notare il ***client.close()*** che DEVE essere eseguito all'interno della funzione di callback relativa alla query. Se si imposta un unico client.close() alla fine di tutto, questo viene eseguito PRIMA che la funzione asicrona di acceso ai dati sia potuta terminare.

Note sull'errore: Il metodo **find()** retituisce errore solo in caso di stretto errore sintattico.
Se la tabella richiesta non esiste oppure non esite uno dei campi, restituisce semplicemente oggetto vuoto.
Va invece in errore, ad esempio, se si tenta di replicare un ID già esistente.

Il metodo **distinct()**

```
  collection.distinct("loves", {gender: 'f'},
       function(err, data) {
           if (err) {
               console.log("Errore esecuzione query: " + err.message);
           } else {
               console.log(JSON.stringify(data));
           }
           client.close();
       }
  );
```

Il metodo **insert()**

```
  var collection = db.collection("unicorns");
    collection.insert( {name:'pippo', gender:'m', weight:parseInt(myVar),
      loves: ['grape', 'watermelon'], vampires:0 },  function(err, data){
        if (!err) {
          res.writeHead(200, header );
          res.end(JSON.stringify(data));
        }
        else error(res, "Errore esecuzione query");
        client.close();
    });
```

Nel caso del metodo **insert()** l'oggetto **data** contiene i campi:

- **ops** che contiene l'intero oggetto inserito.
- **insertedCount** che indica il numero di record inseriti

Il metodo **update()**

```
var collection = db.collection("unicorns");
  collection.update({ "name": name }, {$push: {"loves":loves}},
                    {multi:true}, function (err, data) {
      if (!err) {
        res.writeHead(200, header);
        res.end(JSON.stringify(data));
      }
      else error(res, "Errore esecuzione query");
      client.close();
  });
```

Nel caso di **update()** l'oggetto **data** contiene il campo **nModified** che indica il numero di record modificati

Il metodo **remove()**

```
var collection = db.collection("unicorns");
  collection.remove({$and:[{"loves":s1},{"loves":s2}]}, function(err, data){
      if (!err) {
        res.writeHead(200, header);
        res.end(JSON.stringify(data));
      }
      else error(res, "Errore esecuzione query");
      client.close();
  });
```

Nel caso di **remove()** l'oggetto **data** contiene il campo **n** che indica il numero di record eliminati

Comandi annidati

Occorre fare attenzione che, quando occorre eseguire più comandi SQL in successione, tutti i vari metodi ritornano subito, per cui sostanzialmente vengono lanciati in parallelo. Per cui se un metodo necessita dei risultati del metodo precedente, deve necessariamente essere eseguito all'interno della funzione di callback del metodo precedente. **response.end()** e **db.close()** **devono** essere eseguiti soltanto all'interno dell'ultimo metodo, quello più annidato.

Restituzione di una promise

Tutti i metodi del mongoDB driver, se si omette il parametro callback, restituiscono una promise, la quale non supporta gli eventi .done() e .fail() tipici di jQuery, ma supporta invece .then() e .catch()

```
let rq = collection.find({ "gender": "m" }).toArray();
rq.then(function(data){
     console.log(data);
})
rq.catch(function(err) {
     log("Errore esecuzione query: " + err.message);
})
client.close();
```

In questo modo si possono evitare pesanti livelli di annidamento con un codice più leggibile ad estensione verticale.

Le date ed il formato ISODate

Per poter eseguire delle elaborazioni, le date devono essere oggetti di tipo **ISODate**
Per quanto riguarda la memorizzazione su DB, si può procedere in due modi:

- salvare le date come semplici stringhe (dove però tutte le date inserite devono avere la medesima formattazione) e poi convertirla in ISODate al momento della necessità.
  Esempi: 31-10-2020 oppure 10/31/2020 oppure 2020_10_31
- salvare le date direttamente come oggetto ISODate strutturato nel modo seguente:

  `YYYY-MM-DDTHH:MM:SS.mmm±HH:MM` oppure
  `YYYY-MM-DDTHH:MM:SS.mmmZ`

  dove i millisecondi sono sempre facoltativi
  L'ultima parte indica il cosiddetto **TZD** (Time Zone Designator) cioè il fuso orario rispetto al meridiano di Greenwich (L'italia è situata nel fuso + 0100 rispetto a Greenwich). Il simbolo **Z** indica che la data è riferita al meridiano di Greenwich. I valori ammessi sono (**Z** or **+hh:mm** or **-hh:mm**)
  Ad esempio i seguenti valori indicano tutti la stessa ora:
  `"18:30:00Z", "22:30:00+04:00", "11:30:-07:00"`

Utilizzo delle estensioni BSON all'interno del file json per l'import delle date

Nel momento in cui si crea un file di documenti json da importare in mondoDB, sono disponibili diversi operatori **BSON** (**B**inary J**SON**), che consentono di tipizzare maggiormente i dati da inserire.

$oid	definisce un campo di tipo object ID
$date	definisce un campo di tipo ISODate
$double	definisce un campo di tipo double
$numberLong	definisce un campo di tipo long
$regularExpression	definisce una regex : `{"pattern":"^A", "options":"ig"}`

esempio:

```
{  "_id" : {"$oid": "5f8627503e5c67374ef76400"},
   "volume" : {"$numberLong" : "50" },
   "data"  : { "$date":"2001-09-15T00:00:00Z" } }
```

Lo stesso record inserito tramite il driver mongoDB di nodejs può essere scritto nel seguente modo:

```
var ObjectId = require("mongodb").ObjectId;
db.myCollection.insert(
{  _id: new ObjectId("idDaAssegnareAlNuovoRecord"),
   volume: parseLong("50"),
   data: new Date("2001-09-15T00:00:00Z")  // oppure ISODate()
} );
```

Per inserire la data corrente si può scrivere semplicemente:
`db.myCollection.insert( {_id: 1, data:new Date()} )`

Gestione delle date in formato ISODate

```
let data = new Date("2002-12-31") // T00:00:00Z
```
nota: se si desiderano ore/minuti/secondi != 00 occorre specificalrli all'interno del new Date()
```
studenti.find({"dob": data );
studenti.find({"dob": {$lte : data}} );  // studenti nati prima del 31/12/2002
```

Le funzioni di aggegazione sulle date (funzioni di proiezione)

Con le ISODate sono disponibili diverse funzioni di aggregazione utilizzabili soltanto con aggregate/project

Esempio:

```
{  "_id":1,   "data" : ISODate("2014-01-01T08:15:39.736Z") }
```

```
collection.aggregate([{
    $project:  {
        year: { $year: "$data" },
        month: { $month: "$data" },
        day: { $dayOfMonth: "$data" },
        hour: { $hour: "$data" },
        minutes: { $minute: "$data" },
        seconds: { $second: "$data" },
        milliseconds: { $millisecond: "$data" },
        dayOfYear: { $dayOfYear: "$data" },
        dayOfWeek: { $dayOfWeek: "$data" },
        week: { $week: "$data" }
    }
}])
```

```
"year" : 2014,
"month" : 1,
"day" : 1,
"hour" : 8,
"minutes" : 15,
"seconds" : 39,
"milliseconds" : 736,
"dayOfYear" : 1,
"dayOfWeek" : 4,
"week" : 0
```

Gestione delle date come stringa

Se le date sono memorizzate nel DB come stringa, per poter fare una elaborazione occorre convertirle in ISODate mediante la seguente funzione di aggragazione :

```
$dateFromString: {
      "dateString":"$campo",  // nome del campo da convertire
      "format": "%d/%m/%Y",   // formato d memorizzazione (vedi sito ufficiale)
      "timezone": "+00"}      // offset da aggiungere alla data convertita (00)
```

Tenendo presente che le funzioni di aggragazione possono essere utilizzate con l'operatore **$expr,**
la query iniziale relativa agli studenti maggiorenni può essere riscritta nel modo seguente:

```
collection.find({"$expr":
    {"$gte":
        [{ $dateFromString: {"dateString":"$nato",  "format": "%d/%m/%Y",
                             "timezone": "+00"}}, dob]
    }
})
```

Altri Esempi

1) Trovare tutti i record aventi come data la data odierna

```
var todayStart = new Date();
todayStart.setHours(0,0,0,0);
var todayEnd = new Date();
todayEnd.setHours(23,59,59,999);
var query = { "date": {$gte: todayStart, $lte: todayEnd } }
```

2) Studenti che ad oggi hanno compiuto 18 anni

```
var dob = new Date(moment().subtract(18, 'years').format());
var query = { "dob": { $lte: dob  }}
```

Accesso agli oggetti annidati

Si consideri il DB di una biblioteca costituito da un elenco di libri aventi la seguente struttura, in cui ogni libro può avere più pubblicazioni:

```
{   "titolo" : "Promessi Sposi",
    "autore" : "Alessandro Manzoni",
    "posizione" : {"stanza":2, "scaffale":3, "piano":4}
    "pubblicazioni" : [
        {   "editore" : "PremiateEdizioni",
            "anno_pubblicazione" : 2012,  "numero_pagine" : 732
        },
        {   "editore" : "EdizioniItalianeLibere",
            "anno_pubblicazione" : 1999,  "numero_pagine" : 678
        },
        {   "editore" : "MieEdizioni",
            "anno_pubblicazione" : 1996,  "numero_pagine" : 312
        }
    ]
}
```

Per accedere agli oggetti interni si utilizza l'operatore **puntino** ed **occorre racchiudere la chiave composita tra virgolette.**

Esempio

```
collection.find({"posizione.scaffale":3})
```

Restituisce tutti i libri che si trovano nello scaffale n. 3. Per ogni libro restituisce **l'intero documento** con tutti i campi, compresa la lista completa delle pubblicazioni. Volendo è possibile applicare una proiezione per visualizzare soltanto il titolo del libro ed il solo nome di tutti i vari editori :

```
.project({titolo:1, 'pubblicazioni.editore':1})
```

La seguente restituisce invece l'elenco di tutti gli editori presenti nei vari libri del database

```
collection.distinct('pubblicazione.editore');
```

Spesso però sarebbe comodo visualizzare soltanto le informazioni relative ad una singola pubblicazione. A tal fine si possono seguire 2 approcci:

1) Trasferire tutte le varie pubblicazioni al primo livello tramite l'operatore **$unwind**.

```
let filter = {"titolo":"Promessi Sposi", "pubblicazioni.editore":"MieEdizioni"}
collection.aggregate([{"$unwind":"$pubblicazioni"}, {"$match":filter},
                      {"$project":{"pubblicazioni":1}} ])
```

   Operazione concettualmente semplice ma molto onerosa dal punto di vista computazionale

2) Utilizzare l'operatore **$elemMatch** che consente di ricercare una chiave all'interno di un vettore annidato e di gestire un segnaposto (unico) tale da poter restituire quel singolo elemento

```
collection.find({"titolo": "Promessi Sposi",
                 "pubblicazioni":{"$elemMatch":{"editore":"MieEdizioni"}}})
.project({"titolo":1, "pubblicazioni.$":1})
```

   Il **.$** in coda a pubblicazioni è un segnaposto posizionale e sta ad indicare che non si desiderano tutte le pubblicazioni di quel libro, ma soltanto quelle individuate da **$elemMatch**

   La query precedente avrebbe anche potuto essere scritta tramite l'operatore puntino nel mdo seguente:

```
collection.find({"titolo":"Promessi Sposi","pubblicazioni.editore":"MieEdizioni"})
.project({"titolo":1, "pubblicazioni.$":1})
```

A differenza di quanto potrebbe sembrare l'operatore puntino e **$elemMatch** non sono del tutto equivalenti. Per le sottili differenze si rimanda alla documentazione ufficiale:
https://docs.mongodb.com/manual/reference/operator/query/elemMatch/

3) Una terza soluzione al problema precedente potrebbe essere quella di utilizzare **$elemMatch** direttamente all'interno di project:

```
collection.find({"titolo": "Promessi Sposi"})
.project({"pubblicazioni":{"$elemMatch":{"editore":"MieEdizioni"}}})
```

Update di un singolo elemento interno ad un vettore annidato

L'operatore posizionale **$** consente di identificare un elemento da modificare all'interno di un array senza dover specificare esplicitamente la sua posizione.
A tal fine occorre **identificare** all'interno del filtro l'elemento dell'array che si intende modficare.
Utilizzando poi l'operatore $ all'interno di $set, si può modificare quel singolo elemento

```
collection.update({Autore:"Oscar Wilde", "pubblicazione.editore":"Mondadori"},
                  {$inc:{"pubblicazione.$.numero_pagine":2}})
```

Attenzione che, all'interno di un singolo record, se ci sono più pubblicazioni "Mondadori", ne aggiorna soltanto una, la prima che incontra

L'opzione `{multi:true}` consente di aggiornare più record, però continua a valere la stessa limitazione di prima:

```
collection.update({"pubblicazione.editore":"Newton"},
                  {$set:{"pubblicazione.$.editore":"Newton2"}}, {multi:true})
```

Modifica tutti gli editori dal valore "newton" al valore "newton2". Grazie all'opzione `{multi:true}` vengono aggiornati tutti i documenti presenti nella collezione. Se però all'interno di un libro ci sono 2 edizioni con editore "newton", ne viene aggiornata SOLO una, la prima che incontra, esattamente come prima.
Questo nonostante l'opzione finale `{multi:true}` che è riferita soltanto ai documenti di livello più alto.

L'utilizzo di **$elemMatch** è del tutto equivalente alla sintassi precedente. Viene comuqneu sempre solo aggiornato il primo elemento del vettore che va a buon fine.

L'operatore **$elemMatch** consente di ricercare una singola chiave all'interno di vettore annidato e di gestire un corrispondente segnaposto:

```
collection.update({'pubblicazioni':{$elemMatch:{'editore':'Newton'}}},
                  {$set: {"pubblicazioni.$.editore": "Newton2"}}, {multi:true})
```

$[<identifier>]

Per poter modificare tutte le occorrenze di "editore" : "newton", anche quelle eventualmente ripetute all'interno di uno stesso libro, si può utilizzare la seguente sintassi:

```
collection.update( { },
   {$set:{"Pubblicazione.$[elemento].Editore":"Newton2"}},
   { multi: true,  arrayFilters: [ {"elemento.Editore": "Newton" } ]
   }
)
```

Cioè, per ogni elemento dell'Array che presenta il campo Editore="Newton", viene assegnato al campo Editore medesimo il valore "Newton2".

$[ ]

L'operatore $[ ] utilizzato senza un identificatore interno, modificherebbe TUTTE le voci del vettore, cioè tutte le edizioni di tutti i documenti individuati.

Compass

https://docs.mongodb.com/compass/current/collections/

Consente di :

- Modificare un record mediante apposita interfaccia visuale senza utilizzo del metodo update
- Inserire nuovi record mediante apposita interfaccia visuale senza utilizzo del metodo insert
- Eliminare un record senza utilizzo del metodo remove
- impostare il Filtro di una query così come verrebbe passato al metodo find()
  con possibilità di impostare le opzioni **sort, projection, skip** e **limit**.

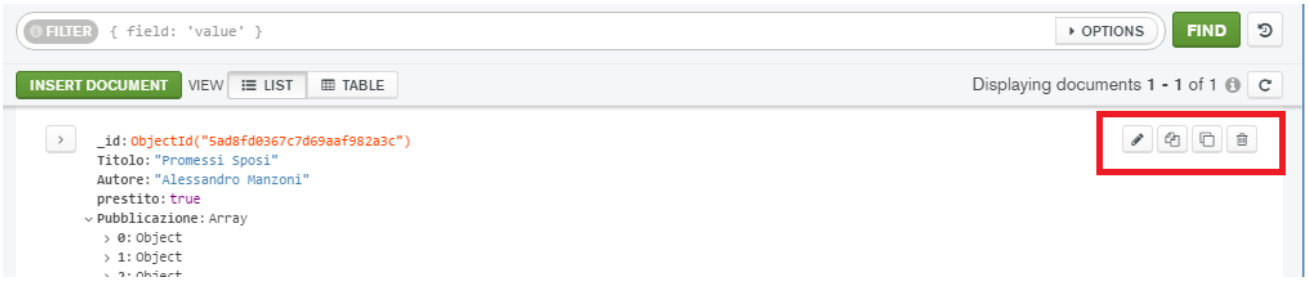
  Esempio `{'citta': 'Alba'}`
  Sul nome dei campi si possono utilizzare indifferentemente apici doppi, apici singoli, senza apici

Allo stato attuale (ver 1.12) non consente l'utilizzo dei comandi **aggregate** e **distinct**

Modifica e Cancellazione di un documento

Quando un documento è visualizzato, sul lato destro compaiono dei pulsantini che consentono di:

- modificare il documento,
- copiare il documento (per incollarlo ad esempio su un'altra applicazione
- clonare il documento (apre una nuova maschera di inserimento già contenente i dati correnti)
- eliminare il documento



Inserimento di un nuovo documento

Una apposita interfaccia consente di inserire le varie chiavi definendo anche il tipo di ciascun dato.
All'apertura viene proposto in automatico un ObjectId univoco

Impostazione di una query

L'indicatore **Filter** assume un colore rosso ed il pulsantino di esecuzione viene disabilitato se l'oggetto json impostato non è formalmente corretto.

La finestra Schema

Visualizza un report statistico sui dati contenuti all'interno della collezione corrente.
Al momento dell'**apertura** esegue un campionamento su 1000 documenti scelti casualmente.
Se il numero di documenti presenti è inferiore a 1000 vengono presi in considerazione tutti.

Se all'interno della **query bar** viene specificato un filtro, il report verrà eseguito soltanto sui dati restituiti dalla query. Il filtro può essere impostato anche solo selezionando una delle opzioni (limit, skip, etc.).

Per ogni campo viene visualizzato il **tipo del campo** ed i **valori** contenuti nei vari documenti.

Nella parte di sinistra viene visualizzata una barra che, dato il nome di un campo, indica la % di documenti in cui quel campo è presente assumendo il tipo indicato. Ad esempio, dati 100 documenti, il campo DataDiNascita potrebbe essere

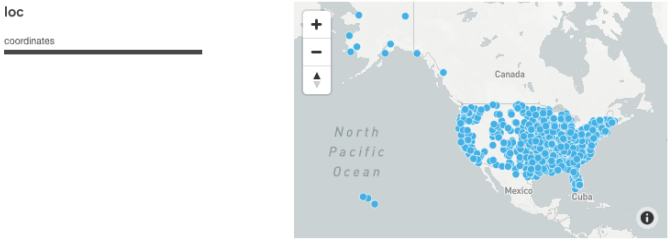
- nel 30% dei casi di tipo Date
- nel 50% dei casi di tipo String
- nel 20% dei casi undefined (cioè non presente all'interno del documento)

Nella parte di destra viene visualizzato un campione ridotto fra i 1000 valori estratti. Cliccando il pulsantino di refresh viene aggiornato il set dei valori visualizzati (sempre all'interno dei 1000 campioni estratti inizialmente). Se i valori all'interno del campo si ripetono (cioè sono quasi sempre i medesimi), viene visualizzata una barra con la percentuale di ciascun valore:

Se I valori del campo si ripetono ma il numero di valori presenti è elevato viene visualizzato un diagramma a barre. Spostando il mouse sulle barre viene visualizzato un piccolo tooltip

I campi di tipo Object o Array possono essere 'espansi' cliccandoci sopra. Ad esempio per quanto riguarda I frutti amati dagli unicorni:

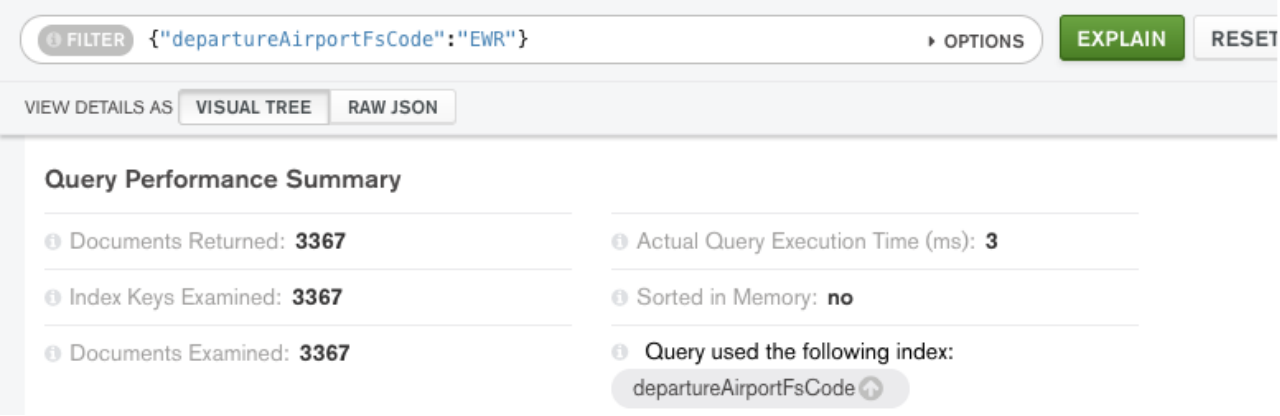
I dati che contengono un GeoJSon oppure delle coordinate [latitudine, logitudine] vengono visualizzati all'interno di una mapa interattiva.



La finestra Explain Plan

Fornisce informazioni sulla query come ad esempio

- tempo di esecuzione
- numero di record restituiti



Gli indici

Gestione degli indici tramite il Client di mongoDB

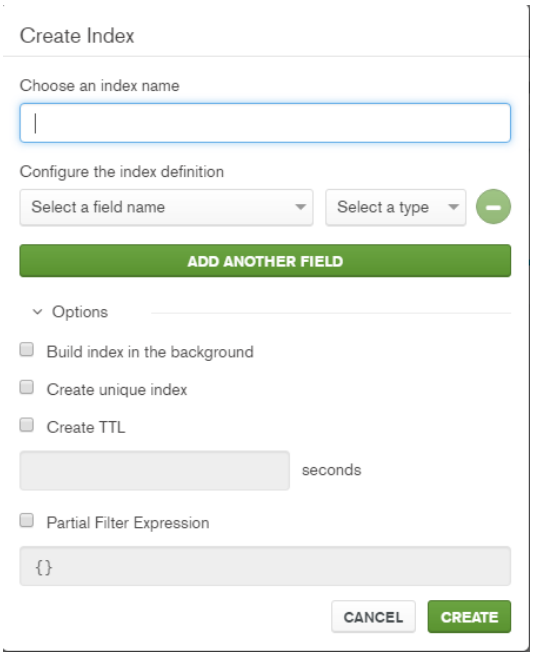
```
db.unicorns.ensureIndex({name: 1});
```
Indicizza il campo name (1 = crescente, -1 = decrescente)

```
db.unicorns.ensureIndex({name: 1}, {unique: true});
```
Crea un indice univoco.

```
db.unicorns.dropIndex({name: 1});
```
Rimuove l'indice indicato.

```
db.system.indexes.find()
```
visualizza l'elenco degli indici presenti nel database in uso.

Gestione degli indici tramite Compass



Nome dell'indice. Può essere lasciato vuoto, nel qual caso verrà deciso automaticamente da mongo

Nome del campo da indicizzare. Crescente/Decrescente

Creazione in background senza bloccare Compass

Indice univoco

L'indice **TimeToLive** è applicabile soltanto ai campi di tipo date (negli altri casi non ha effetto) e provoca la cancelazione dell'intero documento al raggiungimento della data indicata più un ulteriore offset in secondi eventualmente esprimibile. Uitle nel caso di log o **Session Object**

Il **Partial Filter** consente di indicizzare soltanto i record che prsentano un certo valore su un certo campo. Ad esempio : **`{gender: { $eq: "f" }}`.** Le query che riguardano SOLO gli unicorni di genere 'f' saranno ancora più veloci.

La finestra Validation

Consente di definire delle **Regole** di Validazione sull'inserimento di un nuovo Documento all'interno di una collezione. Possono essere definite al momento della creazione della collezione:

```
db.createCollection("students", {  validator: { }  });
```

Dato un certo campo, le Regole che possono essere specificate sono:

- **Exists** – Viene generato un errore o un warning se il campo non esiste
- **Must non Exist** – Viene generato un errore o un warning se il campo esiste
- **Type** – Consente di specificare un BSON Type per il campo
- **Range**– Consente di specificare un RANGE per I campi numerici
- **Regex**– Consente di specificare una espressione regolare per il controllo dell'input

Il **Validation Level** indica il comportamento da tenere quando verranno inseriti nuovi documenti e quando verranno aggiornati i record già esistenti.

- **strict** la regola corrente verrà applicata in corrispondenza di ogni insert/update
- **moderate** la regola corrente verrà applicata in corrispondenza di ogni insert. Nel caso dell'update la regola viene applicata **solo s**e il vecchio valore era già conforme alla regola medesima.

Il **Validation Action** = **Warning**, fa sì che un nuovo record non conforme venga accettato ma con la registrazione di un warning nel file di log.

I metodi map() e mapReduce

map()

In mongoDB non è consentito l'annidamento di query. Occorre eseguire una prima query che restituisca un vettore di dati, e poi una seconda query che utilizzi quel vettore. **Oppure meglio utilizzare Aggregate**

Il metodo **`map()`** consente di estrarre dei dati da un recordset e restituirlo sotto forma d vettore enumerativo Si aspetta come parametro una funzione alla quale viene iniettato il recordset restituito dalla funzione a monte. Mediante **`return`** è possibile ritornare sotto forma di vettore i valori di un singolo campo. Esempi:

1) Trovare tutti gli unicorni che amano le mele
2) Fra questi selezionare gli uncorni femmina
3) Fra questi selezionare quello che ha ucciso più vampiri

```
1) var vet1 = db.unicorns.find({loves:"apple"}).map(function(rs) {return
   rs._id})
2) var vet2 = db.unicorns.find({$and:[{gender:"f"},
                    {_id:{$in:vet1}}]}).map(function(rs) {return rs._id})
3) db.unicorns.find({_id:{$in:vet2}}).sort({vampires:-1}).limit(1)
```

mapReduce()

E' un altro vecchio metodo utilizzato in mongoDB per raggruppamenti e annidamenti prima della comparsa del metodo aggregate(). Più lento e più complesso dell'aggregate(). Da utilizzarsi per query complesse laddove il metodo aggregate() non dovesse essere sufficiente.

Hosting di un database mongoDB Atlas

`https://www.mongodb.com/cloud/atlas`

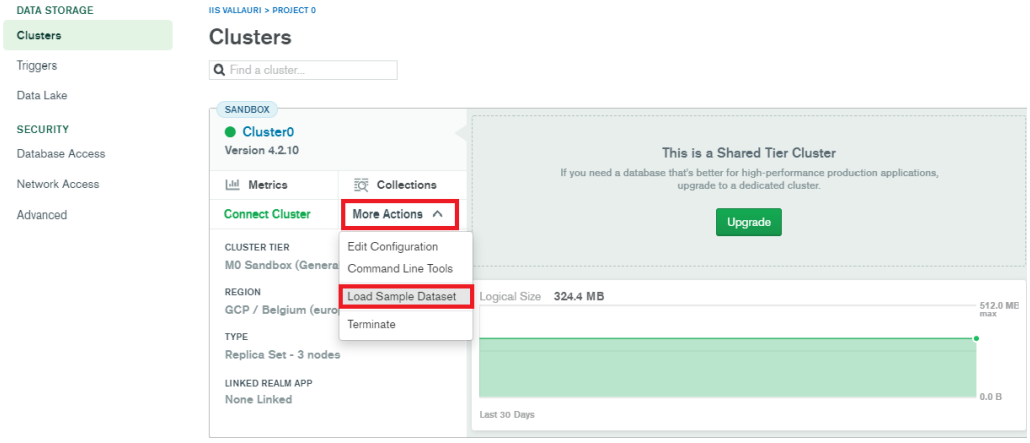
In quasi tutti gli step di registrazione è sufficiente accettare le impostazioni proposte.

1) Definire un cluster (google Europa)

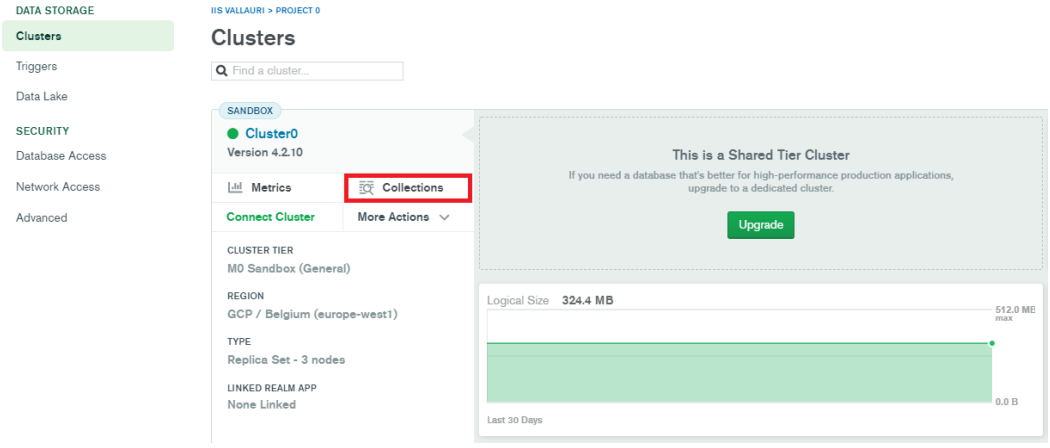
2) DATABASE ACCESS: Creare un utente di accesso al DB con username e password
Questo utente è un utente di accesso al DB che non c'entra nulla con l'utente usato in fase di registrazione per l'accesso al sito mongodb/atlas

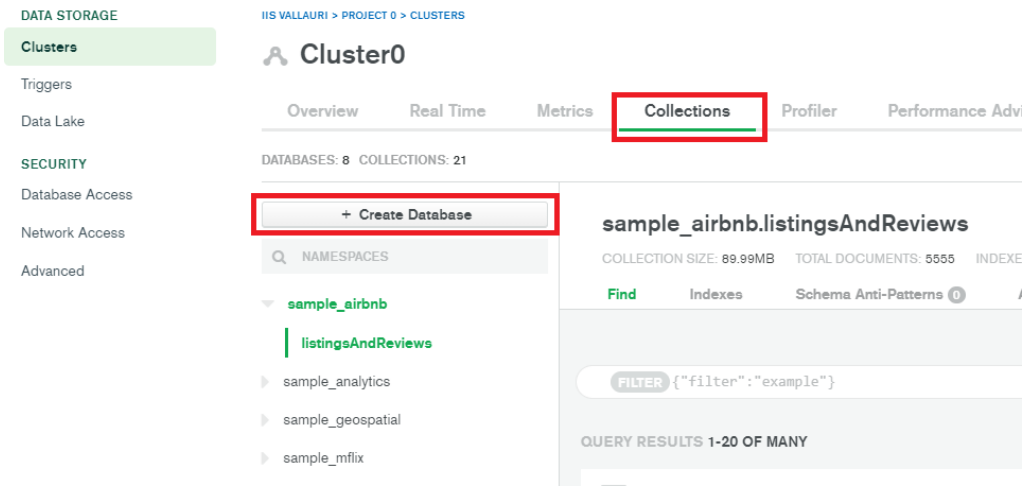
3) NETWORK ACCESS: Creare una whitelist di indirizzi IP (any address)

4a) Caricare eventualmente **un SAMPLE DATABASE:**

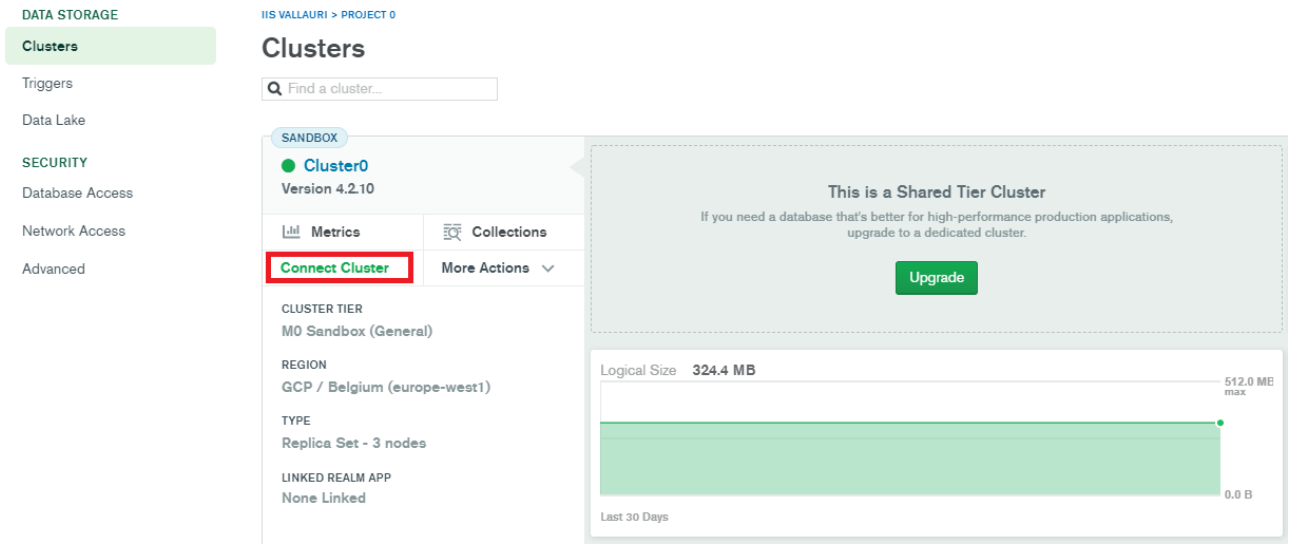


4b) CREARE un NUOVO DATABASE:



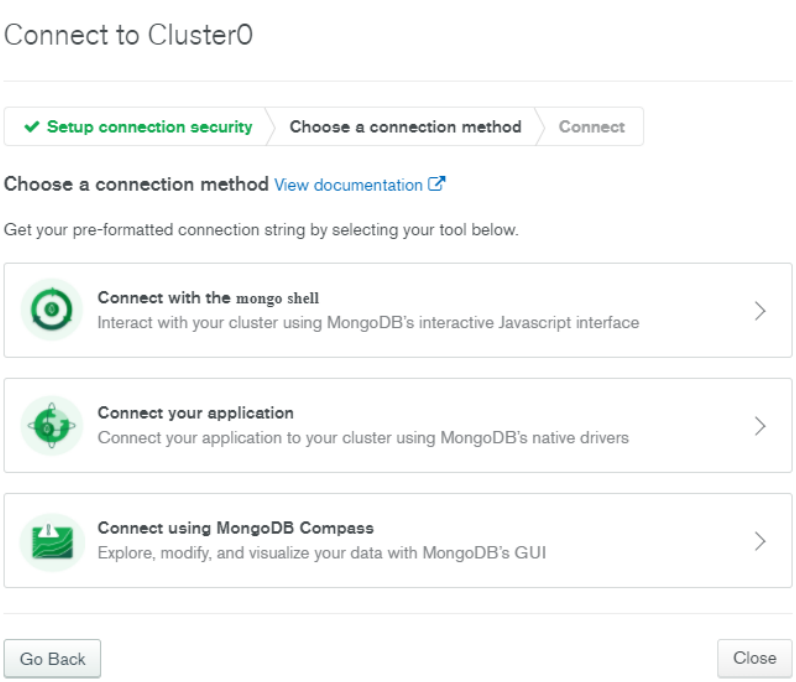


Stringhe di connessione al Database



In corrispondenza del click si apre una finestra che consente di copiare le stringhe di connessione per connettersi al database atlas

- tramite mongo shell
- tramite applicazione node.js e mongoDB Driver
- tramite Compass



Accesso ad un database remoto tramite Compass

L'installazione locale di **Compass** consente di collegarsi ai database creati su Atlas impostando nella finestra di apertura la stringa di connessione copiata al punto precedente.

L'utilizzo di Compass in locale con connessione ad un database di rete consente alcune operazioni aggiuntive rispetto all'interfaccia onLine, come ad esempio la creazione di una nuova collezione all'interno del database (possibile anche online) mettendo però a disposizione un comodissimo pulsante **ADD DATA** che consente di importare un elenco di dati da un solito file testuale con estensione **.json** che deve però contenere **sempre** un vettore di json e non una semplice sequenza.