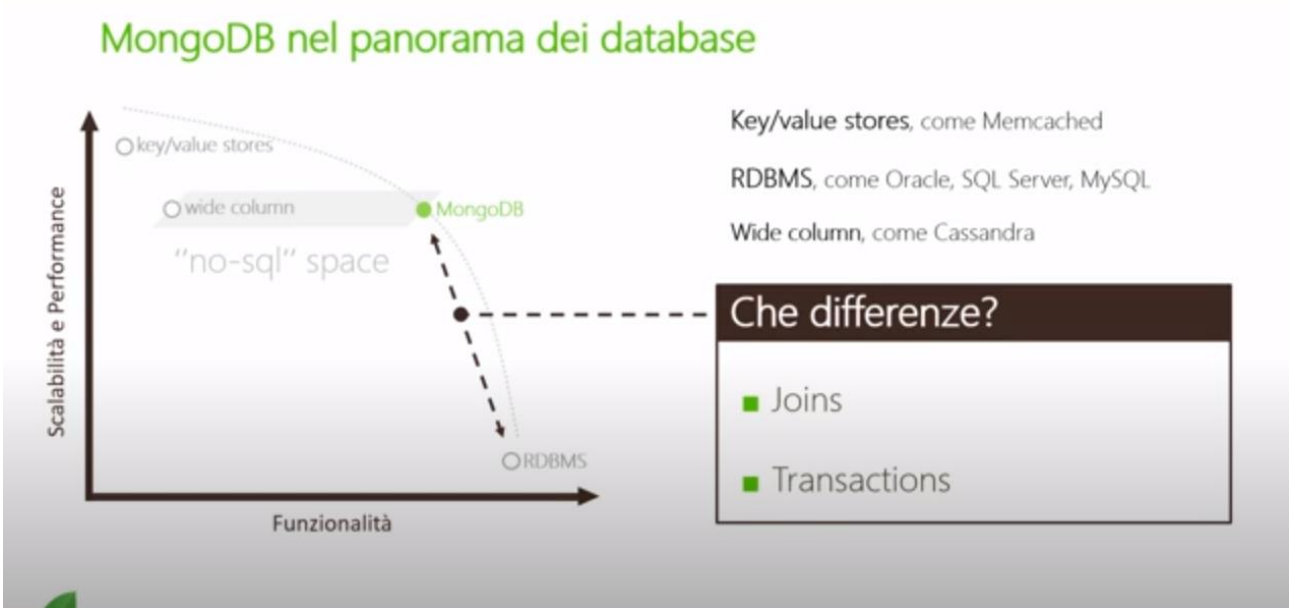
Mongo DB

Corso youtube mongodb:

https://www.youtube.com/watch?v=9MGN9A7LZG0&list=PL8hgLnRSqwynQrmdrGwROlw56zrikXy4B

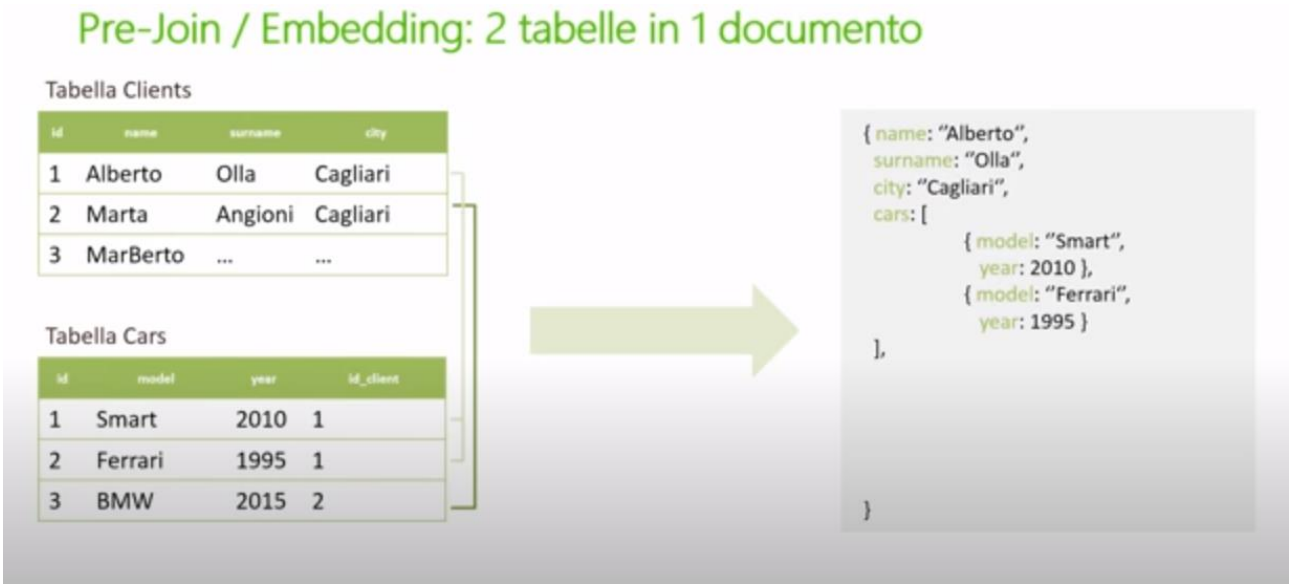
Nato per gestire enorme quantità di dati.

Non va bene ne la scalabilità verticale ne orizzontale.



Mongo si pone a metà strada tra i database relazionali e le memorizzazioni chiavi valore, rinunciando per necessità alle join e transaction (altrimenti non avrebbe potuto essere performante).

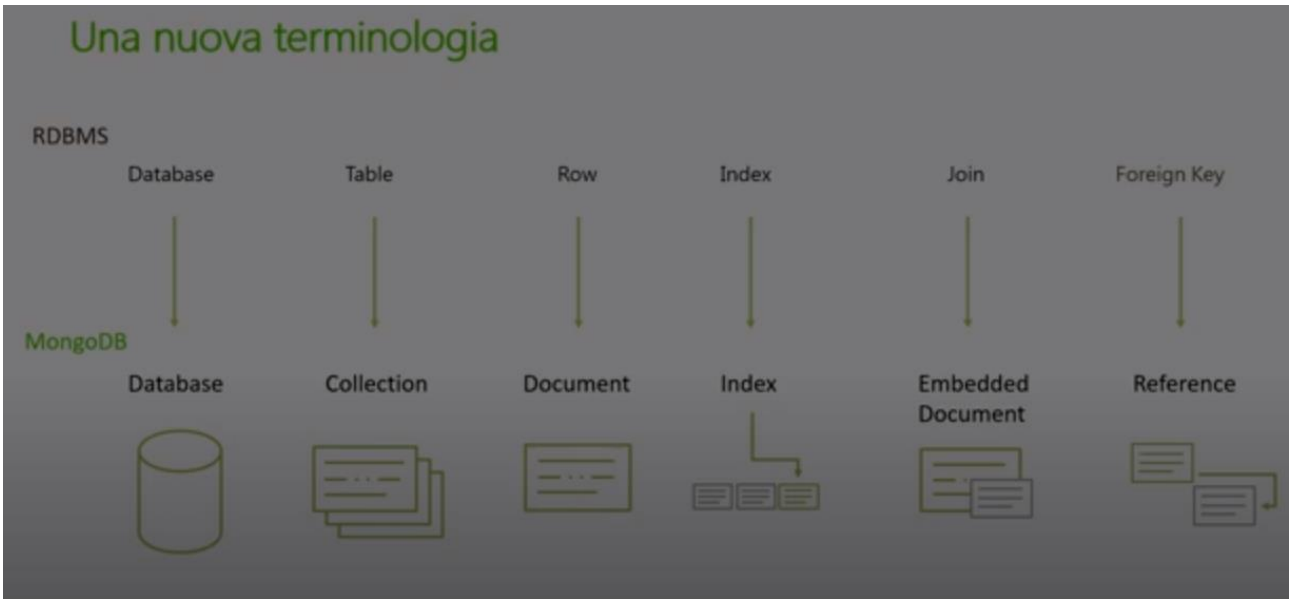
Al posto dei join si usa il ragionamento inverso anzichè estrarre con i join dei dati da più tabelle, includiamo già nel salvataggio del dato altri dati che sarebbero stati ottenuti mediante join in un dbrms. Questo processo si chiama Incapsulazione, oppure Denormalizzazione. Vediamo con un esempio:



Estremamente leggibile intuitivo.

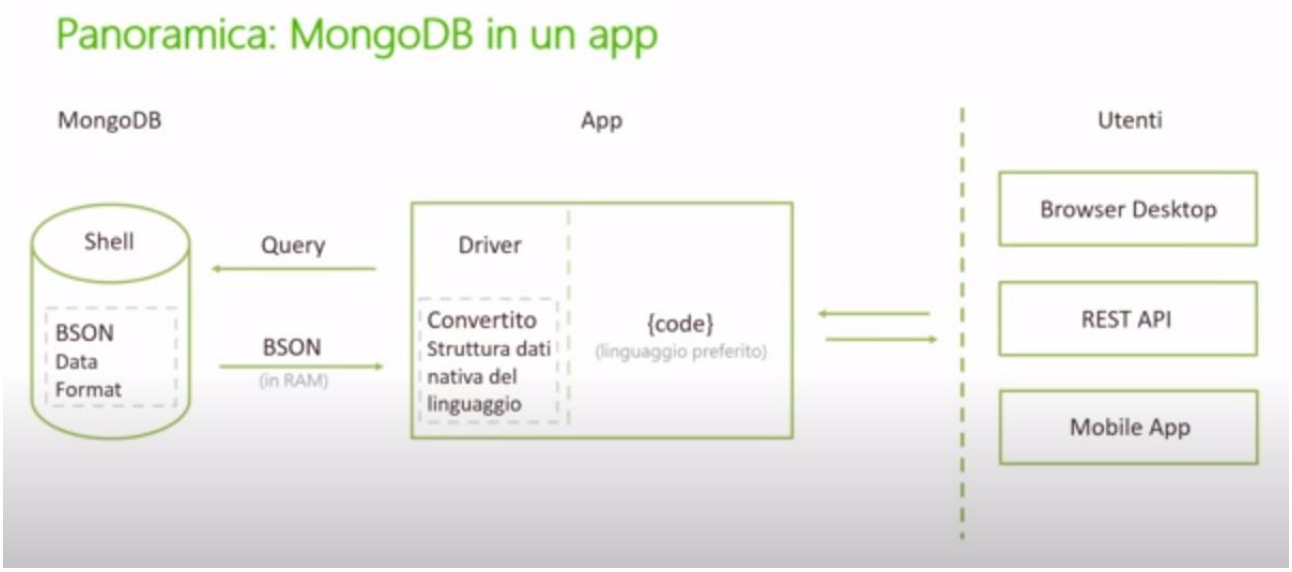
Mongodb e schemaless (possiamo aggiungere liberamente informazioni senza prima creare un struttura atta a raccoglierle):

Mongo poi si occupa della distribuzione dei dati fra server e la sincronizzazione dei dati fra i server.



Mongodb ci mostra i dati in json ma li salva in realtà in binario, per una questione di performance, ha una shell dove è possibile lanciare query da riga di comando.

Questi dati convertiti da json in forma binaria sono detti bson.



MongoDb.

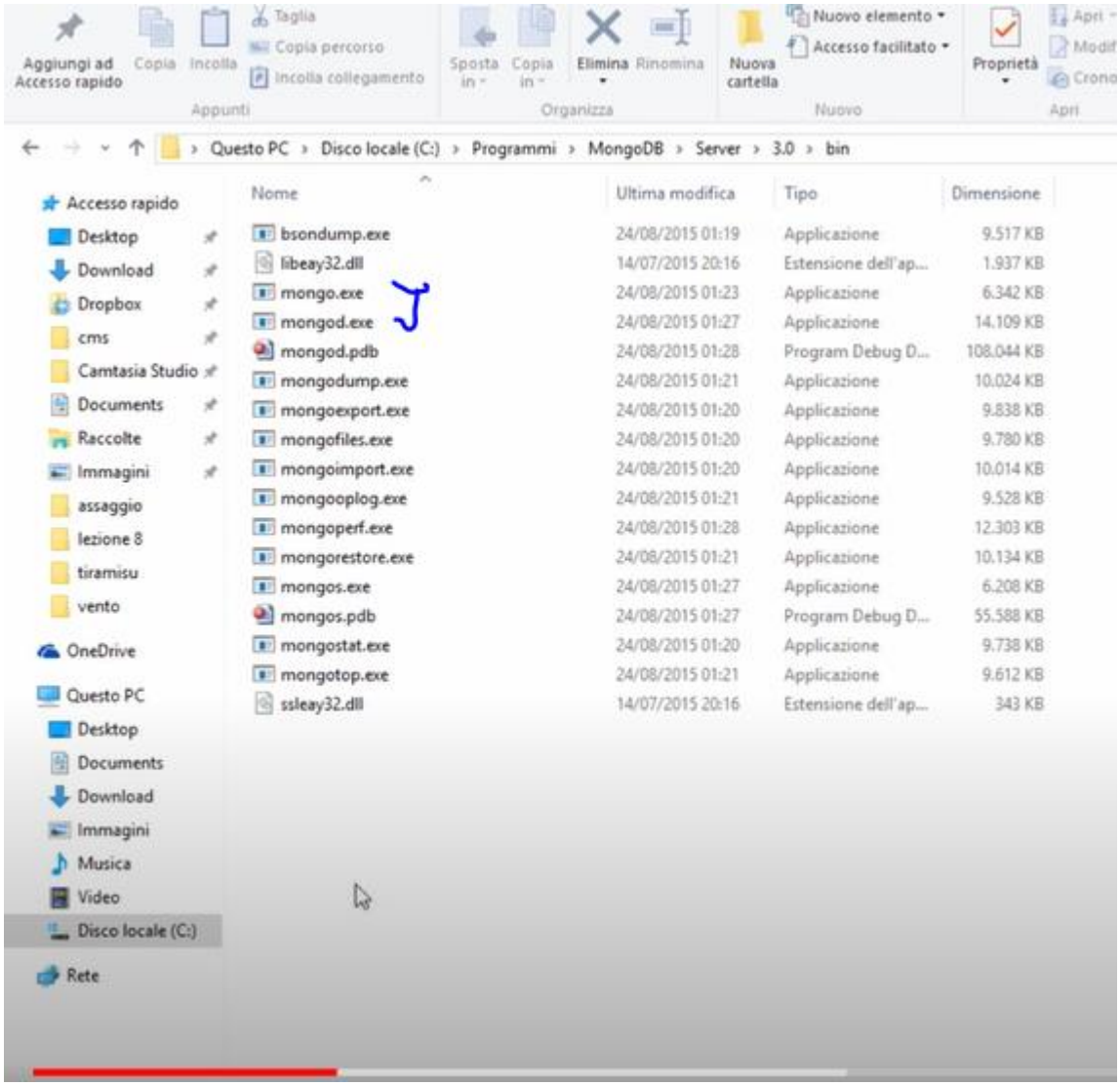
Installazione e uso strumneti (shell e server)

Dopo aver installato Mongodb (meglio versione 64 bit) (https://downloads.mongodb.com/win32/mongodb-win32-x86_64-enterprise-windows-64-4.2.7-signed.msi)

Mongo (shell amministrativa)

Mangod è per il server

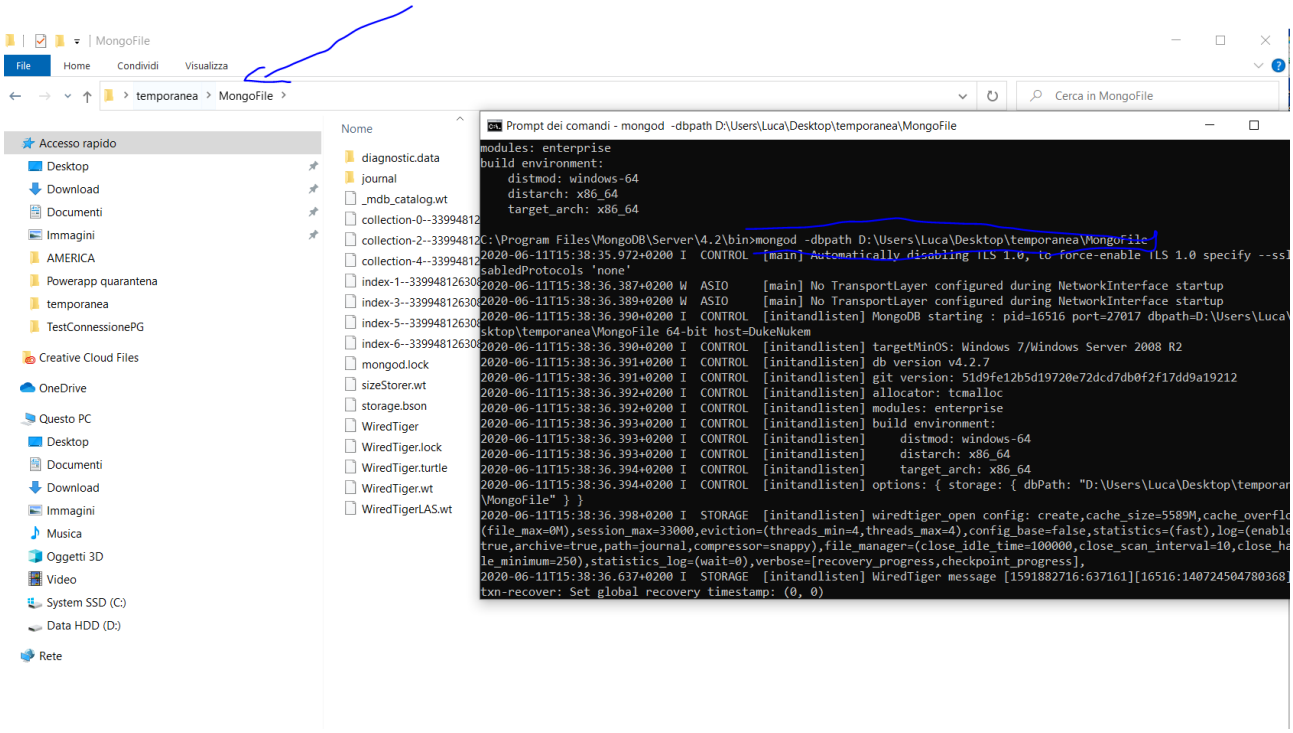
Copiamo il percorso della cartella bin

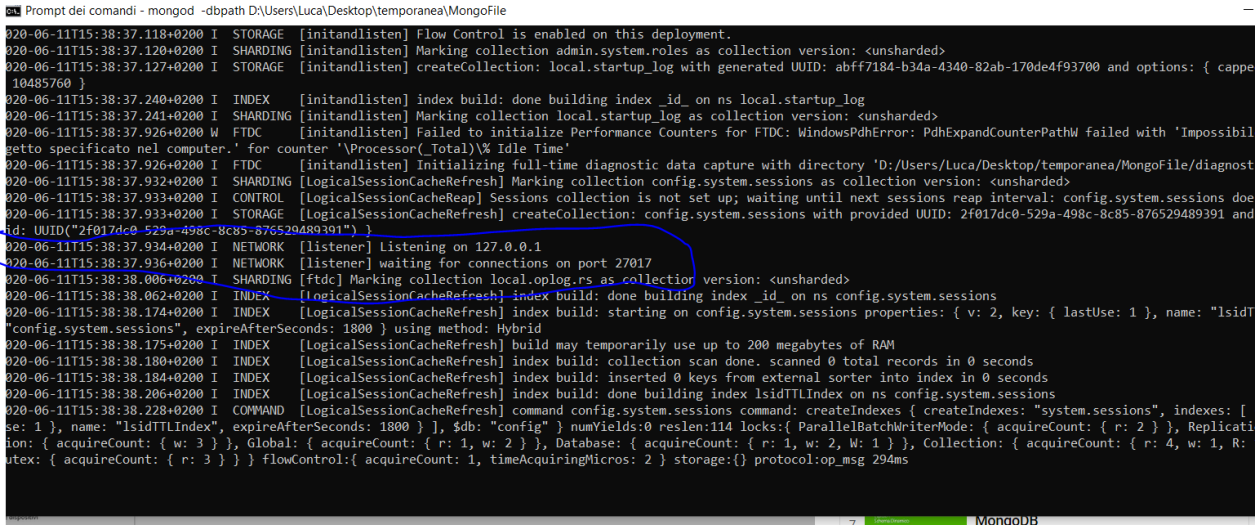


Per vedere se mongo è installato andare nella cartella di mongo dove è installato mongo.exe da prompt e digitare:

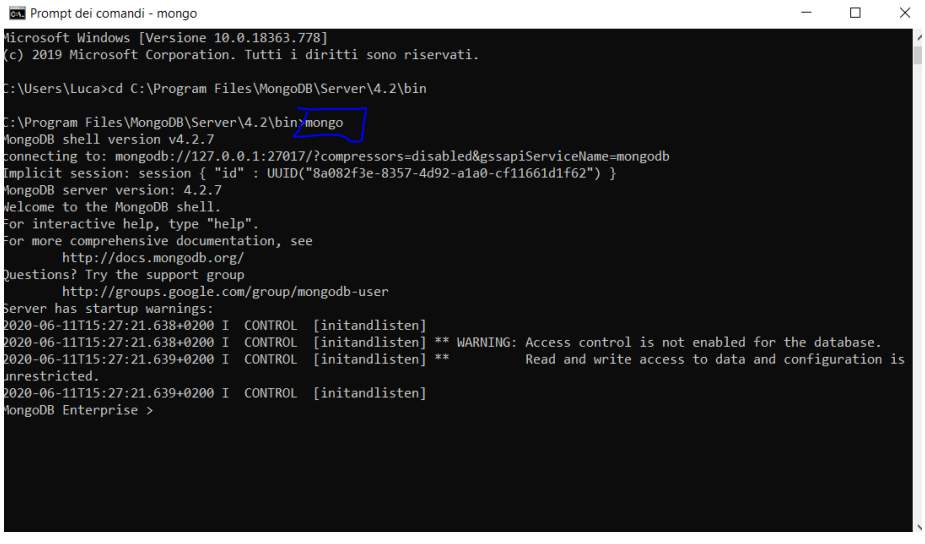
mongo --version

per installare un server creare una cartella e lanciare il comando: mongod –path (percorso cartella creata)



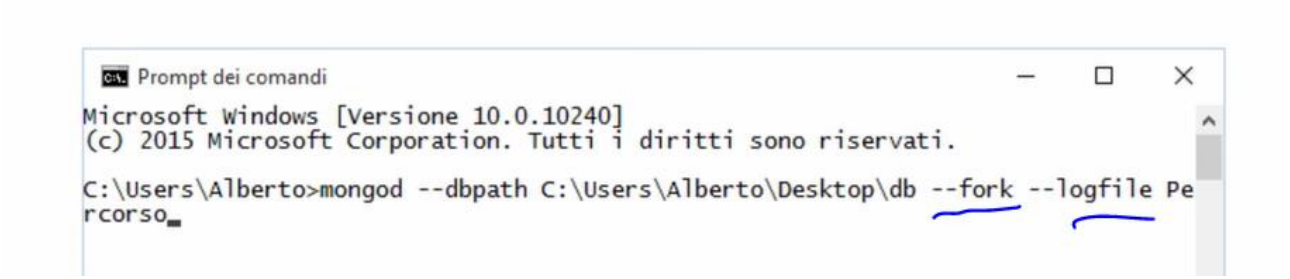


Apriamo un’altra shell e lanciamo mongo



Come è possibile vedere su abbiamo aperto una shell in connessione con il server creato prima ed ora possiamo lanciare comandi.

In produzione per lanciare il server occorrerà anche aggiungere fork per mantenere il server attivo anche dopo chiusura della console e logfile per definire un percorso di log.



Nella console mongo possiamo lanciare dei comandi come:

db (visualizza db attuale)

show dbs (visualizza la lista db)

use demo (switchia al db demo, anche se non esiste….)

il db demo verrà effettivamente creato quando lanciamo una prima istruzione di insert es:

```
switched to db demo
MongoDB Enterprise > db.animals.find()
MongoDB Enterprise > db.animals.insert({name:"delfino",color:"blue",oceans["atlantico","pacifico"]})
2020-06-11T16:09:31.038+0200 E  QUERY    [js] uncaught exception: SyntaxError: missing : after property id :
@(shell):1:53
MongoDB Enterprise > db.animals.insert({name:"delfino",color:"blue",oceans:["atlantico","pacifico"]})
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise >
```

Ad esempio la prima isrtuzione cerca degli animali, non c'è nulla ancora, inseriamo un animale, vi è errore di sintassi, correggiamo e reinseriamo, ora il dato è inserito, rilanciamo la query per ottenere gli animali:

```
MongoDB Enterprise > db.animals.find()
{ "_id" : ObjectId("5ee23b432065ef52a9152556"), "name" : "delfino", "color" : "blue", "oceans" : [ "atlantico", "pacific
o" ] }
MongoDB Enterprise > db.animals.find().pretty
function() {
    this._prettyShell = true;
    return this;
}
MongoDB Enterprise > db.animals.find().pretty()
{
        "_id" : ObjectId("5ee23b432065ef52a9152556"),
        "name" : "delfino",
        "color" : "blue",
        "oceans" : [
                "atlantico",
                "pacifico"
        ]
}
```

Il metodo pretty ritorna la select formattata

Si possono inserire dei where:

```
> db.animals.find( { name:"pantera" } )
```

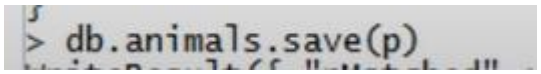
Solo gli animali il cui name=pantera…

Siamo in javascript:

quindi se mettiamo in una variabile il risultato:

```
> var p = db.animals.findOne( { name:"pantera" } )
> p
{
        "_id" : ObjectId("55f6f9ae1870eb1e7e898000"),
        "name" : "pantera",
        "color" : "nero",
        "zoos" : [
                {
                        "name" : "Zoo di Dublino",
                        "address" : {
                                "street" : "Phoenix park, Dublino 8",
                                "city" : "Dublino",
                                "country" : "Irlanda"
                        }
                }
        ]
}
> p.lengs = 4
```

Salviamo l'oggetto nel db:



```
> db.animals.save(p)
```

Se riselezioniamo gli animals vediamo che l'oggetto è stato salvato:

```
        "name" : "delfino",
        "color" : "grigio",
        "oceans" : [
                "atlantico",
                "pacifico",
                "indiano"
        ]
}
{
        "_id" : ObjectId("55f6f9ae1870eb1e7e898000"),
        "name" : "pantera",
        "color" : "nero",
        "zoos" : [
                {
                        "name" : "Zoo di Dublino",
                        "address" : {
                                "street" : "Phoenix park, Dublino 8",
                                "city" : "Dublino",
                                "country" : "Irlanda"
                        }
                }
        ],
        "lengs" : 4
}
```

La funzione save corrisponde o a un insert o un update

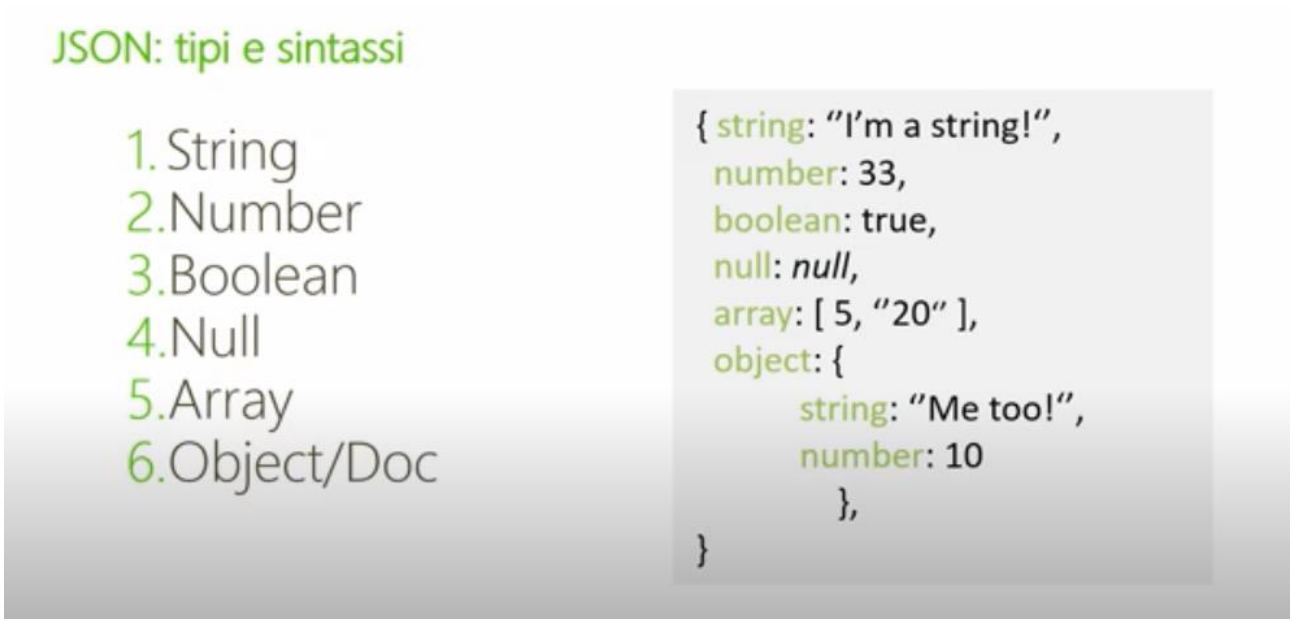
Db.animals.save (senza parentesi tode in fondo mostra il codice della funzione:

```
MongoDB Enterprise > db.animals.save
function(obj, opts) {
    if (obj == null)
        throw Error("can't save a null");

    if (typeof (obj) == "number" || typeof (obj) == "string")
        throw Error("can't save a number or string");

    if (typeof (obj._id) == "undefined") {
        obj._id = new ObjectId();
        return this.insert(obj, opts);
    } else {
        return this.update({_id: obj._id}, obj, Object.merge({upsert: true}, opts));
    }
}
```

Per gli aiuti possiamo lanciare:

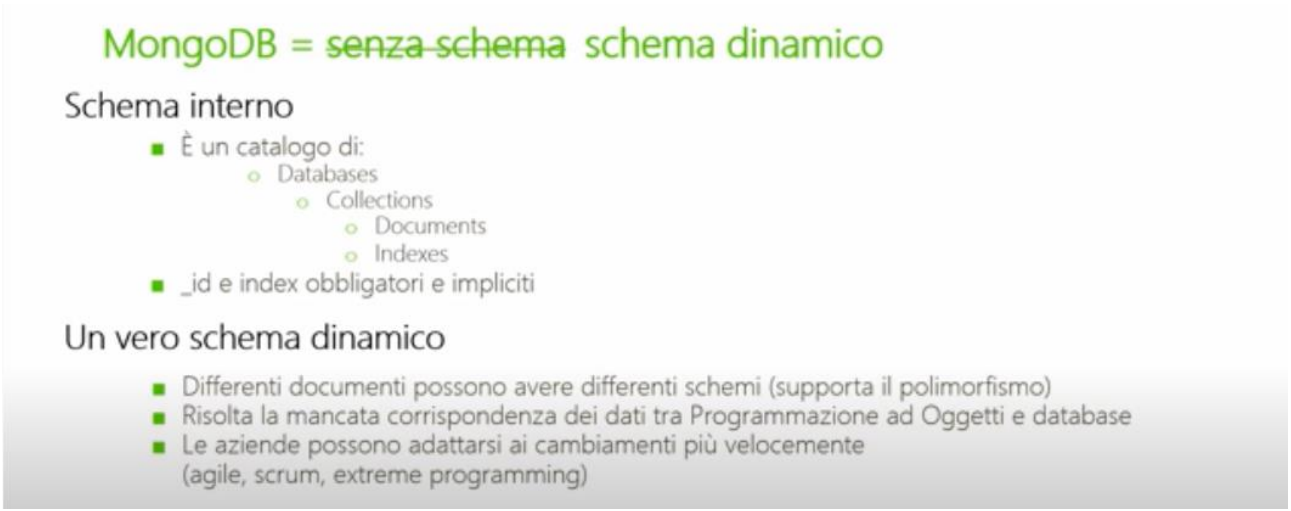
help, oppure db.help(), oppure db.animals.help() ecc…

altro comando utile e show: show, show dbs, show collections

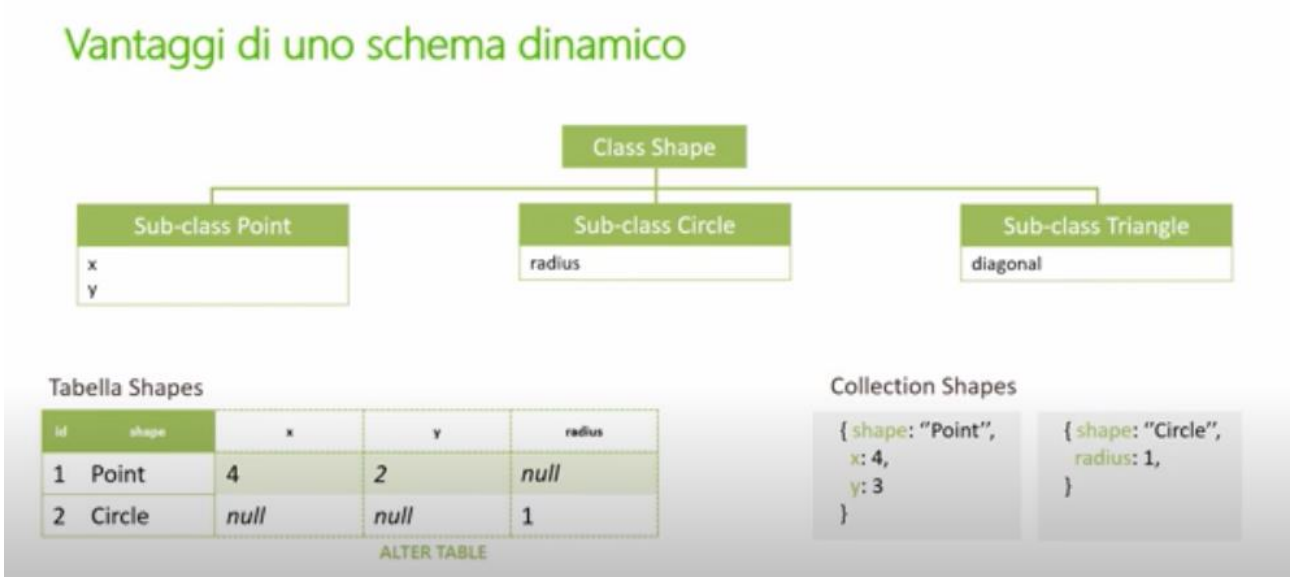


Il BSON è la rappresentazione binaria del json per un accesso più veloce ed efficiente ai dati. Il BSON aggiunge nuovi tipi di dati al json oltre a quelli appena visti, si veda schemino sopra.

Mongodb si dice che e schemaless ma questo non è del tutto vero, esso ha infatti un suo schema ossia un db che contiene collection che contengono documenti, si possono poi creare degli indici che velocizzano la ricerca.

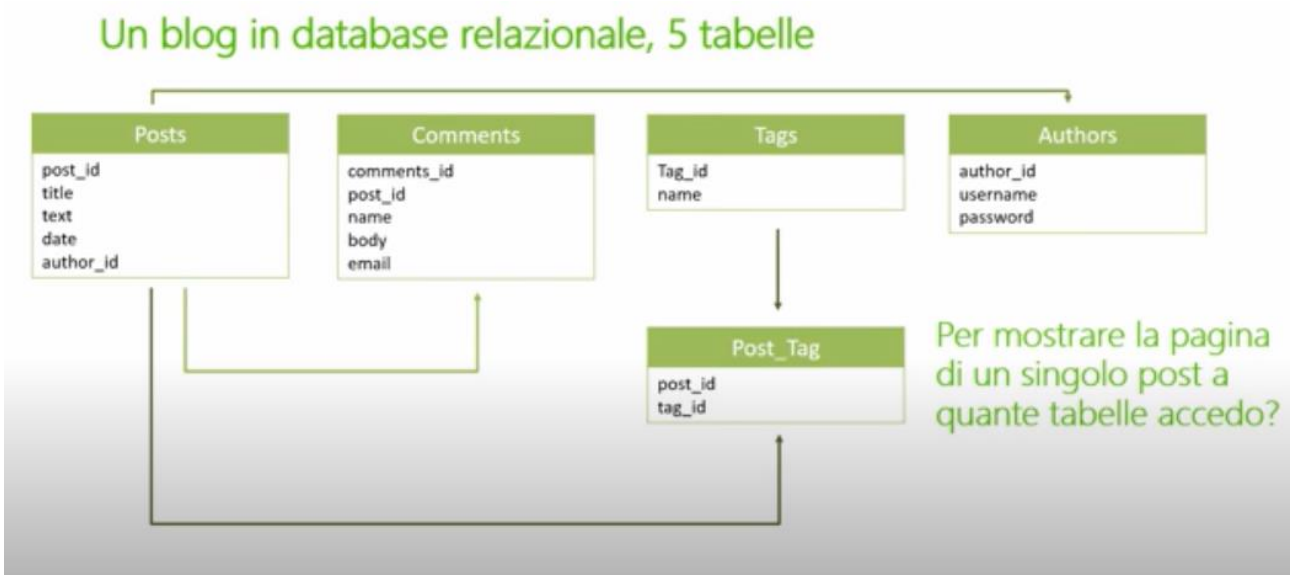


Per rappresentare diverse forme (shapes) in un db relazionale o dobbiamo creare dei campi in più in una unica tabella di nome shape che in base al tipo di forma specifica conterrà dei dati o su dei campi o su degli altri campi, oppure creare una tabella specifica per ogni tipo di forma, entrambe non sono soluzioni ottimali. Risolviamo in questo modo con mongodb:



Una collection shape che può contenere più documenti differenti.

Facciamo un esempio pratico, supponiamo di voler realizzare un blog semplice, se come base relazionale abbiamo un db relazionale le tabelle che dovremo creare saranno queste:

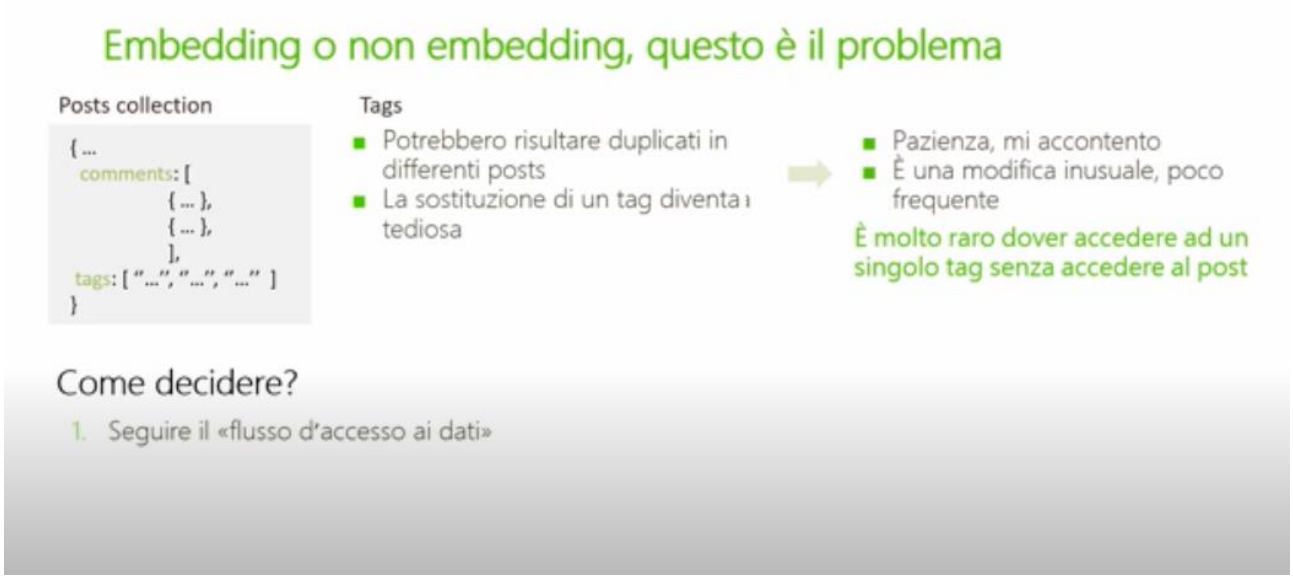


Se volessimo mostrare un articolo (post) dovremo accedere **a tutte le tabelle** qui sopra rappresentate, infatti un articolo conterrà oltre titolo e descrizione anche il nome dell'autore, tags a cui è stato associato, i commenti.

Se dovessimo invece realizzare con mongo il db avremo questa situazione:

Con la tecnica delle prejoin possiamo creare i commenti, i tags saranno un array. Nella collection autori inseriamo il nome dell'autore come id in questo modo per recuperare l'autore non dovremo ricercarlo all'interno della collection.

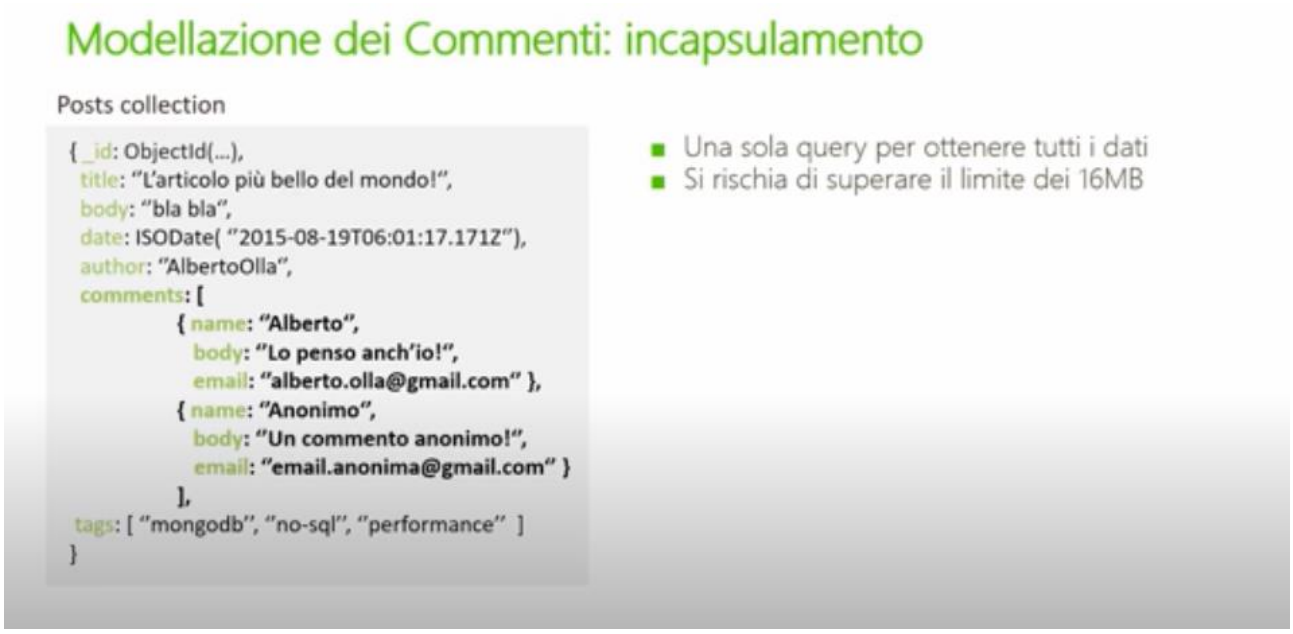
In generale sempre meglio rindondare i dati (con l'embedding) che cercare di applicare le regole di normalizzazione dei db relazionali, ad esempio se un tags è molto lungo si sarebbe tentati a creare una collection tags e sostituire il contenuto lungo con l'id del tag (Non farlo!!, non è un db relazionale).



Si può seguire la regola del saggio che è : "se il flusso dati seguirai l'embedding non sbaglierai"

Esiste un limite di 16mb per singolo documento (da tenere presente, ma si supera il limite solo in casi rari di siti molto grandi).

La prima tecnica di memorizzazione l'abbiamo vista è l'incapsulamento, ma occorre stare attenti a non superare i 16 mb se per esempio vengono inseriti tanti commenti….:

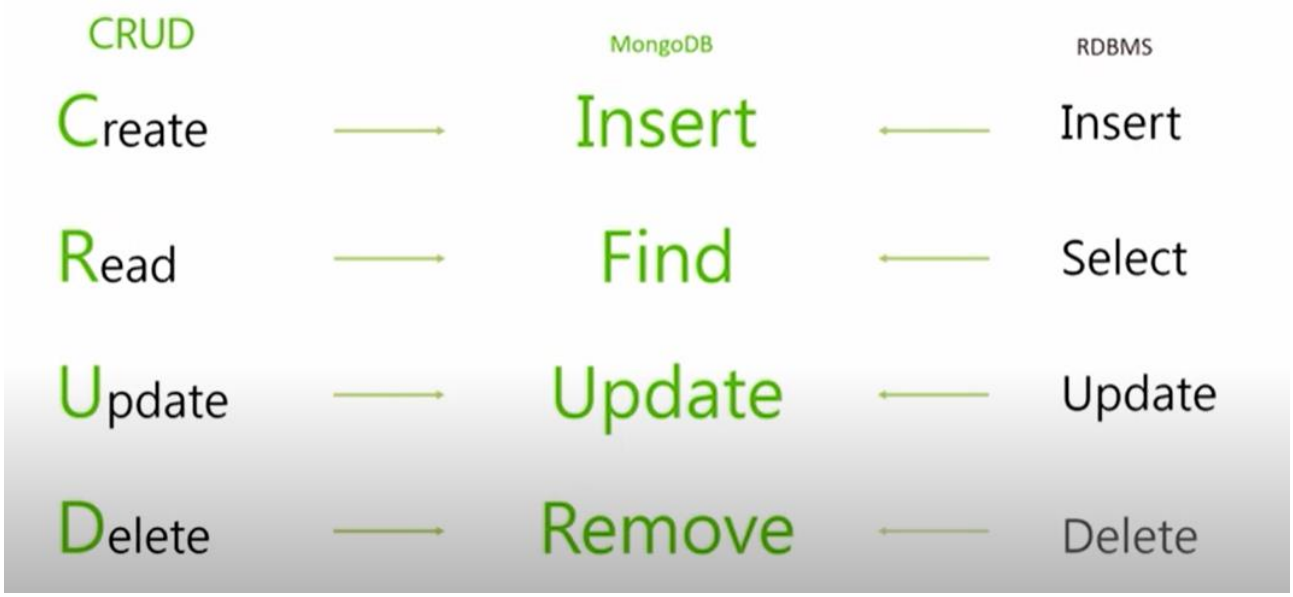


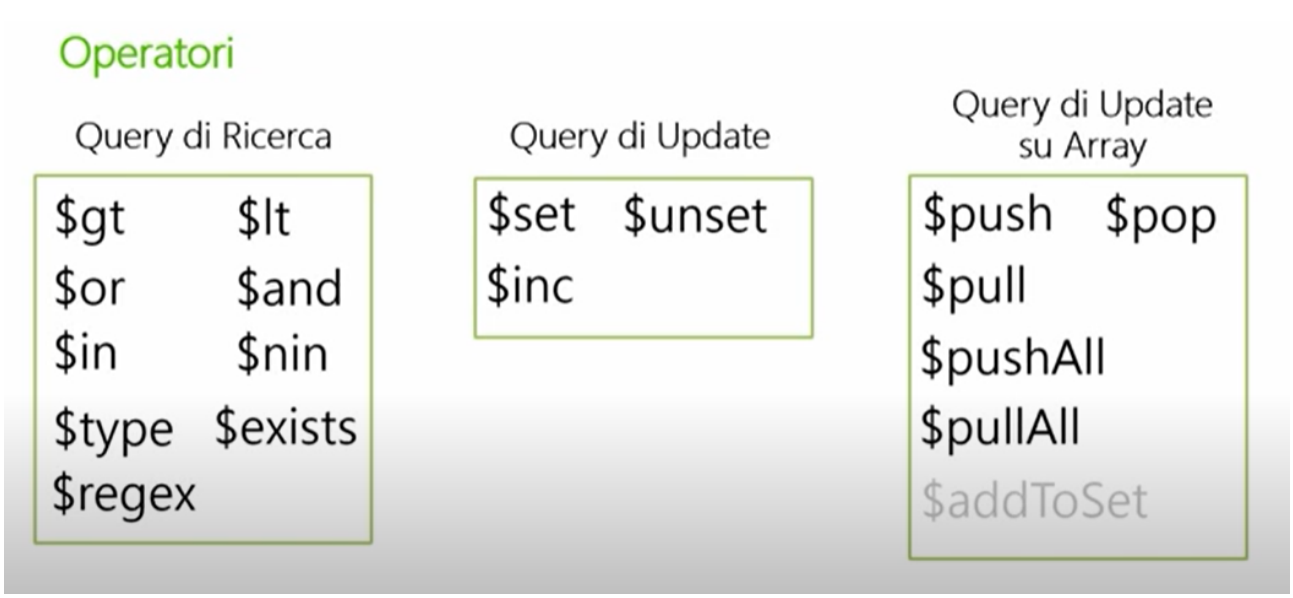
Un approccio alternativo potrebbe essere quello basato sulle reference ma si perde in performance (il commento viene sostituito con objectid e viene creata una nuova collection documents)

non buono… passiamo al prossimo approccio…

il migliore approccio consiste nel salvare i commenti come incapsulati ma in un array a dimensione fissa (20) che salva gli ultimi 20 commenti mentre i rimanenti vengono salvati in una collection a parte. Ci preoccupiamo di salvare però dei dati in più come il numero complessivo di commenti ed il numero di pagine.

Avremo quindi la prima pagina con gli ultimi commenti come qui sopra e poi se l'utente vuole vedere oltre i 20 commenti gli caricheremo i successivi che vengono salvati in altre pagine:





Insert:

```
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Alberto>mongo
MongoDB shell version: 3.0.6
connecting to: test
> use demo
switched to db demo
> var doc = { "name":"Alberto", "surname":"Olla", "city":"Cagliari" }
> doc
{ "name" : "Alberto", "surname" : "Olla", "city" : "Cagliari" }
> db.users.insert( doc )
WriteResult({ "nInserted" : 1 })
> db.users.find()
{ "_id" : ObjectId("55f6e444536437072c7201c6"), "name" : "Alberto", "surname" :
"Olla", "city" : "Cagliari" }
>
```

Per cercare documenti si usa find e find one, quest'ultimo serve per cercare un singolo documento, partiamo da questo:

findone e basta restituisce un unico documento random, l'ultimo esempio per farci restituire solo l'attributo nome.

```
> db.users.findOne()
{
        "_id" : ObjectId("55f6e444536437072c7201c6"),
        "name" : "Alberto",
        "surname" : "Olla",
        "city" : "Cagliari"
}
> db.users.findOne( { name:"Alberto"  } )
{
        "_id" : ObjectId("55f6e444536437072c7201c6"),
        "name" : "Alberto",
        "surname" : "Olla",
        "city" : "Cagliari"
}
> db.users.findOne( { name:"Marta"  } )
{
        "_id" : ObjectId("55f6e47a536437072c7201c7"),
        "name" : "Marta",
        "surname" : "Angioni",
        "city" : "Cagliari"
}
> db.users.findOne( { name:"Marta"  }, { name:true } )
```

```
> db.scores.find( { student:19, type:"prova"  } )
```

Qui sopra stiamo usando find con due condizioni dove la virgola funge da and.

Anche qui è possibile filtrare gli attributi

```
> db.scores.find( { student:19, type:"prova"  }, { score:true } )
{ "_id" : ObjectId("55f6e8aa536437072c721972"), "score" : 95 }
```

CLS per pulire la console di comandi.

Gli operatori gt ed lt:

```
> db.scores.findOne()
{
        "_id" : ObjectId("55f6e8aa536437072c721938"),
        "student" : 0,
        "type" : "esame",
        "score" : 36
}
> db.scores.find( { score: { $gt : 95  }  } )
```

Qui sopra cerchiamo tutti gli studenti con punteggio maggiore di 95, abbiamo dovuto aprire con le graffe un nuovo documento dove specificare la condizione.

Possiamo aggiungere nuove condizioni:

```
> db.scores.find( { score: { $gt : 95  },type:"prova" } )
```

Per impostare un filtro di intervallo:

```
> db.scores.find( { score: { $gt : 95, $lt: 98  },type:"prova"  } )
```

Compreso tra 95 e 98.

Se vogliamo considerare gli estremi dell'intervallo dovremo aggiungere la e di equal:

```
> db.scores.find( { score: { $gt : 95, $lte: 98  },type:"prova"  } )
```

L'operatore exist serve per cercare documenti che hanno un determinato attributo es:

```
> db.users.find( { city: { $exists:true }  } )
{ "_id" : ObjectId("55f6e444536437072c7201c6"), "name" : "Alberto", "surname" :
"Olla", "city" : "Cagliari" }
{ "_id" : ObjectId("55f6e47a536437072c7201c7"), "name" : "Marta", "surname" : "
ngioni", "city" : "Cagliari" }
>
```

Per chiedere tutti i documenti che hanno un certo tipo usiamo type specificando un numero di tipo secondo le specifiche beason:

```
> db.users.find( { name: { $type: 2  } } )
{ "_id" : ObjectId("55f6e444536437072c7201c6"), "name" : "Alberto", "surna
"Olla", "city" : "Cagliari" }
{ "_id" : ObjectId("55f6e47a536437072c7201c7"), "name" : "Marta", "surname
ngioni", "city" : "Cagliari" }
```

Regex: operatore che utilizza molte risorse da tenere presente:

```
> db.users.find( { name: { $regex : "a"  } } )
{ "_id" : ObjectId("55f6e47a536437072c7201c7"), "name"
ngioni", "city" : "Cagliari" }
{ "_id" : ObjectId("55f6ea7b536437072c7224f1"), "name"
{ "_id" : ObjectId("55f6ea81536437072c7224f2"), "name"
> db.users.find( { name: { $regex : "e$"  } } )
```

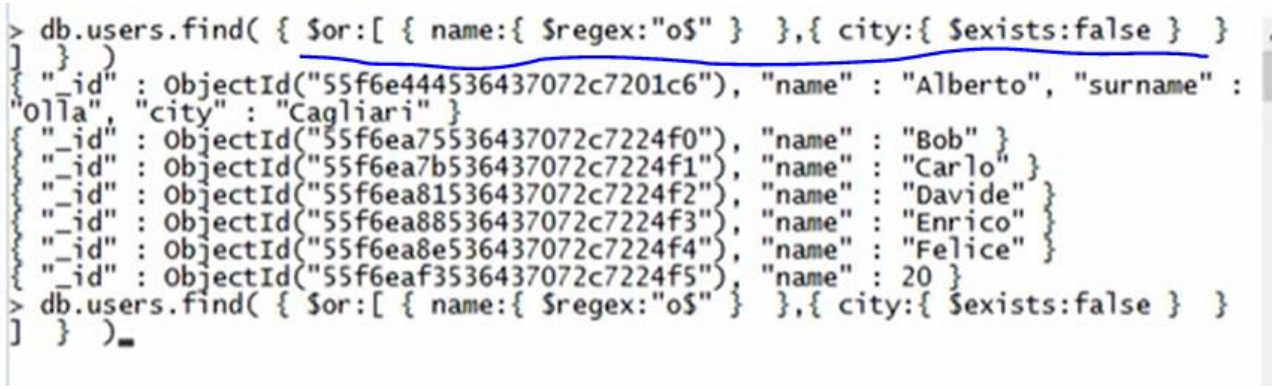
```
> db.users.find( { name: { $regex : "^A"  } } )
```

Qui sopra cerca i nomi che contengono la parola a oppure i nomi che finiscono con la e o ancora che finiscono con la a

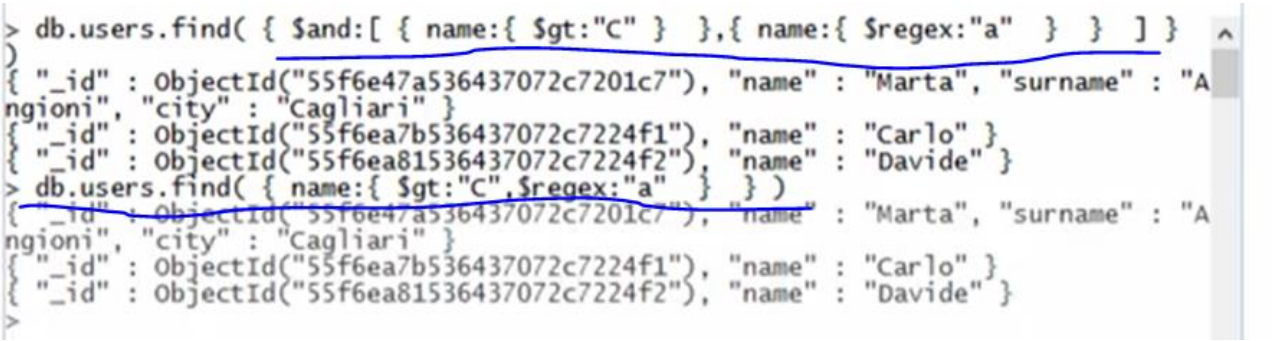
Per fare questo occorrerebbe usare textindex e l'operatore text che non è argomento del corso.

AND e OR

Non si scrivono come sottodocumenti dell'attributo, ad es qu sotto viene definito un array dove ogni elemento dell'array è la condizione che verrà concatenata con gli altri elementi dell'array come or.



Ora vediamo in maniera analoga come impostare and:



Qui sopra esistono 2 modi di fare la stessa cosa, il primo con l'operatore and ed il secondo già visto prima. Per questioni di performance si usa la seconda soluzione.

Ricerche all'interno di array:

se scriviamo come sotto nell'esempio sottolineato, anche se stiamo ricercando nell'array, siccome mongo è intelligente riesce a trovare con la sintassi fino ad ora utilizzata per cercare attributi, ma la ricerca non è ricorsiva nel senso che se vi sono array annidati si ferma al livello più esterno.

Operatori all in nin

Hanno la caratteristica, a differenza degli operatori visti fino ad ora, di ricevere un array in entrata.

Per cercare tutti i documenti che contengono diversi tag usiamo l'operatore all:

```
> db.articles.find( { tags: { $all: [ "mongodb","array"  ]  } } )
```

Con in basta la presenza anche di un solo elemento cercato:

```
> db.articles.find( { tags: { $in: [ "mongodb","query"  ]  } } )
```

Ricerca nei sottodocumenti:

nell'esempio sotto se ricerchiamo con find passando il sottodocumento socials così come è stato memorizzato lo troveremo, ma se invertiamo gli attributi al suo interno non lo troviamo più, perché? Perché mongo effettua una comparazione in binario dei dati ed invertendo l'ordine degli attributi è cambiato il numero binario che lo rappresenta. Quindi come facciamo a ricercare impostando un criterio che coinvolge attributi di sottodocumenti?

Usando il punto, ultimo esempio.

```
> db.contacts.insert( { name:"Alberto Olla",socials:{ twitter:"yourAlbertoOlla",
email:"info@mongodbitalia.it"  } } )
WriteResult({ "nInserted" : 1 })
> db.contacts.find().pretty()
{
        "_id" : ObjectId("55f8325fa784c9c01114d0f0"),
        "name" : "Alberto Olla",
        "socials" : {
                "twitter" : "yourAlbertoOlla",
                "email" : "info@mongodbitalia.it"
        }
}
> db.contacts.find( { socials: { twitter : "yourAlbertoOlla",email : "info@mongo
dbitalia.it"  } } )
{ "_id" : ObjectId("55f8325fa784c9c01114d0f0"), "name" : "Alberto Olla", "social
s" : { "twitter" : "yourAlbertoOlla", "email" : "info@mongodbitalia.it" } }
> db.contacts.find( { socials: { email : "info@mongodbitalia.it",twitter : "your
AlbertoOlla"  } } )
> db.contacts.find( { socials: { twitter : "yourAlbertoOlla"  } } )
> db.contacts.find( { "socials.twitter":"yourAlbertoOlla"  } )
```

Anche qui possiamo usare le combinazioni di condizioni viste precedentemente

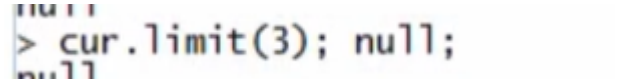
Cursori:

Quando noi lanciamo un find viene costruito un cursore su una lista di documenti

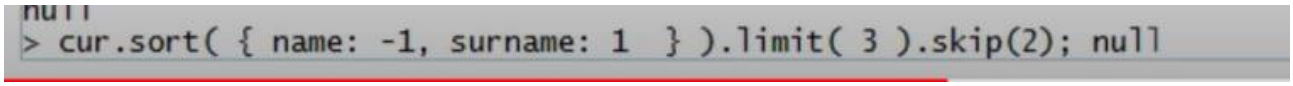
Qui sopra aggiungiamo null alla fine dell'istruzione per evitare che la shell lanci l'esecuzione della riga.

Recuperiamo un cursore e quindi utilizziamo i metodi del cursore anche per scorrere con un ciclo i dati.

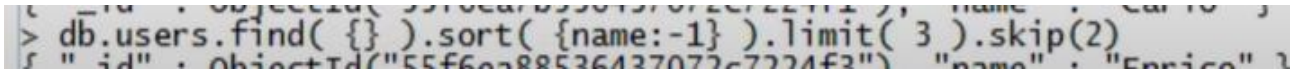
Possiamo limitare i dati del cursore:



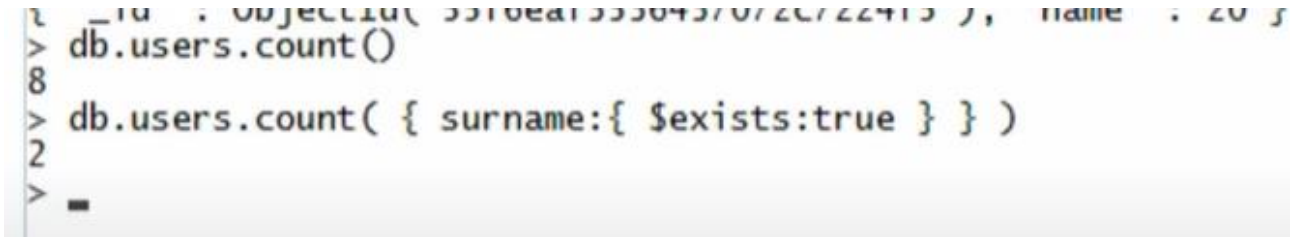
In quest'esempio ordiniamo in senso decrescente name e poi surname in senso crescente, limitando a 3 i risultati saltando i primi 2.



Funziona anche così:



Esiste anche la funzione count:



I quattro approcci dell'update:

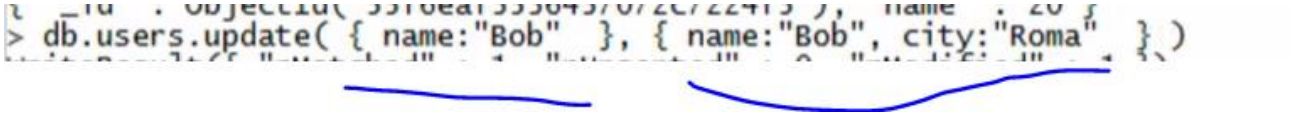
sostituzione completa del documento

aggiunta o modifica attributo documento

la modifica del documento se esiste e la creazione del documento se non esiste

il multy update

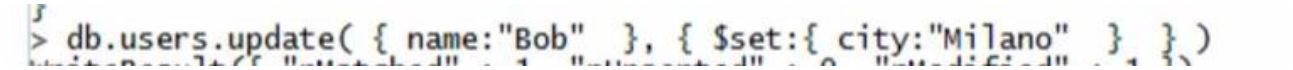
primo approccio:



La prima è la condizione e la seconda è il documento da sostituire interamente dove si verifica la condizione.

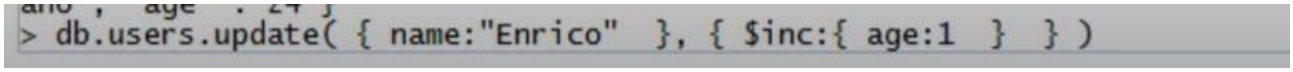
Il consiglio è di non usare quest'approccio.

Meglio usare il secondo approccio ossia la modifica di uno specifico attributo ottenuta mediante l'operatore $set:

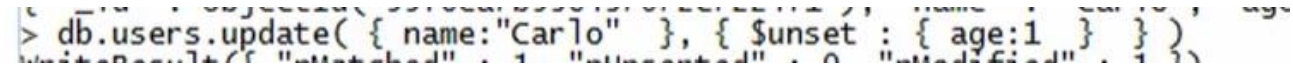


Dove il nome è bob aggiorna city a milano e se non c'è la city la crea.

L'operatore $inc ci permette di incrementare un valore esistente nel documento specificando lo step di incremento come nell'esempio sotto:



Per eliminare uno specifico attributo si usa $unset dove l'1 è da intendersi come tru e non come valore dell'attributo:



Modifiche sugli array:

```
> db.arrays.insert( { _id: 0, a: [ 1,2,3,4  ] } )
WriteResult({ "nInserted" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 1, 2, 3, 4 ] }
> db.arrays.update( { _id:0  },{ $set: { "a.2" : 5  }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 1, 2, 5, 4 ] }
> db.arrays.update( { _id:0  },{ $push:{ a: 6 }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 1, 2, 5, 4, 6 ] }
> db.arrays.update( { _id:0  },{ $pull:{ a: 5 }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 1, 2, 4, 6 ] }
> db.arrays.update( { _id:0  },{ $pop:{ a:1 }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 1, 2, 4 ] }
> db.arrays.update( { _id:0  },{ $pop:{ a:-1 }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 2, 4 ] }
> db.arrays.update( { _id:0  },{ $pushAll : { a: [  ]  }  }  )
```

Quello qui sopra fa quello descritto qui:

Inserisce alla posizione 2 il valore 5

Aggiunge il valore 6

$Pull Elimina il valore 5 scorrendo l'array quando lo trova

$pop elimina il primo valore

Elimina l'ultimo valore

Inoltre abbiamo anche questi metodi:

```
> db.arrays.update( { _id:0  },{ $pushAll : { a: [ 5,6,7,8 ]  }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 2, 4, 5, 6, 7, 8 ] }
> db.arrays.update( { _id:0  },{ $pullAll : { a: [ 7,8 ]  }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.arrays.findOne()
{ "_id" : 0, "a" : [ 2, 4, 5, 6 ] }
> db.arrays.update( { _id:0  },{ $addToSet: { a:7  }  }  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Ultimi due approcci dell'update:

terzo approccio:

qui viene modificato mario aggiungendo la city di Venezia, ma poiché mario non esiste possiamo dire che se non esiste venga creato, con upsert:

```
> db.users.update( { name:"Mario" },{ $set:{ city:"Venezia" }  }, { upsert:true
 } )
```

Ora vediamo multyupdate:

di default l'update, a differenza dei database relazionali, modifica un solo documento, se vogliamo che ne modifichi più di uno dobbiamo specificarlo noi (multi=true):

```
> db.users.update( { city:{ $exists:true }  },{ $set:{ city:"Roma" }  }, { multi
:true } )
WriteResult({ "nMatched" : 6, "nUpserted" : 0, "nModified" : 5 })
> db.users.find()
{ "_id" : ObjectId("55f6e444536437072c7201c6"), "name" : "Alberto", "surname" :
"Olla", "city" : "Roma" }
{ "_id" : ObjectId("55f6e47a536437072c7201c7"), "name" : "Marta", "surname" : "A
ngioni", "city" : "Roma" }
{ "_id" : ObjectId("55f6ea8e536437072c7224f4"), "name" : "Felice" }
{ "_id" : ObjectId("55f6eaf3536437072c7224f5"), "name" : 20 }
{ "_id" : ObjectId("55f6ea75536437072c7224f0"), "name" : "Bob", "city" : "Roma"
}
{ "_id" : ObjectId("55f6ea81536437072c7224f2"), "name" : "Davide", "city" : "Rom
a" }
{ "_id" : ObjectId("55f6ea88536437072c7224f3"), "name" : "Enrico", "city" : "Rom
a", "age" : 25 }
{ "_id" : ObjectId("55f6ea7b536437072c7224f1"), "name" : "Carlo" }
{ "_id" : ObjectId("55f84358a9a8ad6f4a8d5858"), "name" : "Mario", "city" : "Roma
" }
```

Ad esempio qui sopra modifica l'attributo city a roma dove l'attributo city esiste.

Qui di seguito invece la condizione viene bypassata con un documento vuoto:

```
> db.users.update( { },{ $set:{ city:"Roma" }  }, { multi:true } )
```

Eliminare documenti e collection:

per eliminare tutti i documenti basta usare remove e un documento vuoto

```
{ "_id" : ObjectId("55f6eaf3536437072c7224f5"), "name" : 20, "city" : "Roma" }
> db.users.remove( { name:{ $gt:"B" } } )
WriteResult({ "nRemoved" : 7 })
> db.users.find()
{ "_id" : ObjectId("55f6e444536437072c7201c6"), "name" : "Alberto", "surname" :
"Olla", "city" : "Roma" }
{ "_id" : ObjectId("55f6eaf3536437072c7224f5"), "name" : 20, "city" : "Roma" }
> db.users.remove( {  } )
WriteResult({ "nRemoved" : 2 })
> db.users.remove()
2015-09-15T18:19:36.616+0200 E QUERY    Error: remove needs a query
    at Error (<anonymous>)
    at DBCollection._parseRemove (src/mongo/shell/collection.js:305:32)
    at DBCollection.remove (src/mongo/shell/collection.js:328:23)
    at (shell):1:10 at src/mongo/shell/collection.js:305
>
```

Per eliminare in maniera più performante una intera collezione meglio usare drop:

```
> db.users.drop()
true
```

Poi occorre approfondire gli indice, sharding, replicaset ecc…