https://github.com/learning-zone/mongodb-interview-questions

MongoDB Commands

Show All Databases

```
show dbs
```

Show Current Database

```
db
```

Create Or Switch Database

```
use acme
```

Drop

```
db.dropDatabase()
```

Create Collection

```
db.createCollection('posts')
```

Show Collections

```
show collections
```

Insert Row

```
db.posts.insert({
  title: 'Post One',
  body: 'Body of post one',
  category: 'News',
  tags: ['news', 'events'],
  user: {
    name: 'John Doe',
    status: 'author'
  },
  date: Date()
})
```

Insert Multiple Rows

```
db.posts.insertMany([
  {
    title: 'Post Two',
    body: 'Body of post two',
    category: 'Technology',
    date: Date()
  },
  {
    title: 'Post Three',
    body: 'Body of post three',
```

```
    category: 'News',
    date: Date()
  },
  {
    title: 'Post Four',
    body: 'Body of post three',
    category: 'Entertainment',
    date: Date()
  }
])
```

Get All Rows

```
db.posts.find()
```

Get All Rows Formatted

```
db.find().pretty()
```

Find Rows

```
db.posts.find({ category: 'News' })
```

Sort Rows

```
# asc
db.posts.find().sort({ title: 1 }).pretty()
# desc
db.posts.find().sort({ title: -1 }).pretty()
```

Count Rows

```
db.posts.find().count()
db.posts.find({ category: 'news' }).count()
```

Limit Rows

```
db.posts.find().limit(2).pretty()
```

Chaining

```
db.posts.find().limit(2).sort({ title: 1 }).pretty()
```

Foreach

```
db.posts.find().forEach(function(doc) {
  print("Blog Post: " + doc.title)
})
```

Find One Row

```
db.posts.findOne({ category: 'News' })
```

Find Specific Fields

```
db.posts.find({ title: 'Post One' }, {
  title: 1,
```

```
  author: 1
})
```

Update Row

```
db.posts.update({ title: 'Post Two' },
{
  title: 'Post Two',
  body: 'New body for post 2',
  date: Date()
},
{
  upsert: true
})
```

Update Specific Field

```
db.posts.update({ title: 'Post Two' },
{
  $set: {
    body: 'Body for post 2',
    category: 'Technology'
  }
})
```

Increment Field ($inc)

```
db.posts.update({ title: 'Post Two' },
{
  $inc: {
    likes: 5
  }
})
```

Rename Field

```
db.posts.update({ title: 'Post Two' },
{
  $rename: {
    likes: 'views'
  }
})
```

Delete Row

```
db.posts.remove({ title: 'Post Four' })
```

Sub-Documents

```
db.posts.update({ title: 'Post One' },
{
  $set: {
    comments: [
      {
        body: 'Comment One',
        user: 'Mary Williams',
        date: Date()
      },
      {
        body: 'Comment Two',
        user: 'Harry White',
        date: Date()
      }
    ]
```

```
  }
})
```

Find By Element in Array ($elemMatch)

```
db.posts.find({
  comments: {
     $elemMatch: {
       user: 'Mary Williams'
       }
    }
  }
)
```

Add Index

```
db.posts.createIndex({ title: 'text' })
```

Text Search

```
db.posts.find({
  $text: {
    $search: "\"Post O\""
    }
})
```

Greater & Less Than

```
db.posts.find({ views: { $gt: 2 } })
db.posts.find({ views: { $gte: 7 } })
db.posts.find({ views: { $lt: 7 } })
db.posts.find({ views: { $lte: 7 } })
```

List of the indexes in a collection

```
db.restaurants.getIndexes()
[
          {
                    "v" : 2,
                    "key" : {
                              "_id" : 1
                    },
                    "name" : "_id_",
                    "ns" : "test.restaurants"
          },
          {
                    "v" : 2,
                    "key" : {
                              "borough" : 1
                    },
                    "name" : "borough_1",
                    "ns" : "test.restaurants"
          }
]
```

Drop an existing index

```
db.restaurants.dropIndex("cuisine_1_grades.score_1")
{ "nIndexesWas" : 4, "ok" : 1 }
```

Mik azok a NoSQL adatbázisok? Melyek a különböző típusú NoSQL adatbázisok?

A NoSQL egy nem relációs DBMS, amely nem igényel fix sémát, elkerüli a join-okat és könnyen skálázható. A NoSQL adatbázis használatának célja az elosztott adattárolók, amelyeknek nagy az adattárolási igénye. A NoSQL-t nagy adatokhoz és valós idejű webalkalmazásokhoz használják.

A NoSQL adatbázisok típusai

- Document databases
- Key-value stores
- Column-oriented databases
- Graph databases

1. Dokumentum adatbázisok

Egy dokumentum adatbázis JSON, BSON vagy XML dokumentumokban tárolja az adatokat. A dokumentum adatbázisban a dokumentumok beágyazhatók. Az egyes elemek indexelhetők a gyorsabb lekérdezés érdekében.

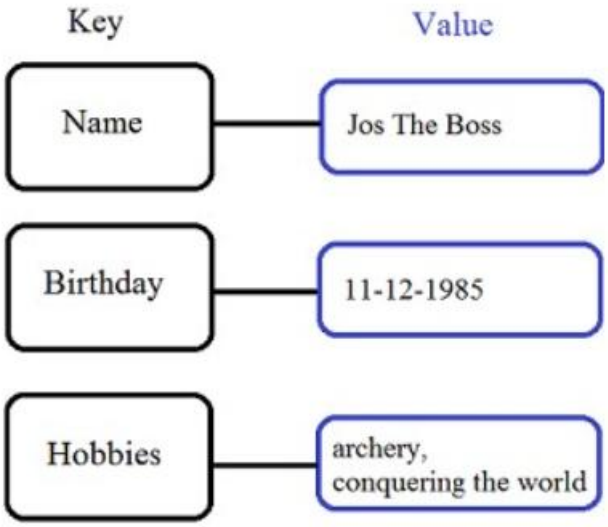
A dokumentumokat olyan formában lehet tárolni és visszakeresni, amely sokkal közelebb van az alkalmazásokban használt adatobjektumokhoz, ami azt jelenti, hogy kevesebb fordítás szükséges az adatok alkalmazásban történő felhasználásához.

Példa: Az Amazon SimpleDB, a CouchDB, a MongoDB, a Riak, a Lotus Notes népszerű dokumentumokból származó DBMS rendszerek.

2. Key-value Stores

Az adatokat kulcs / érték párokban tároljuk. Úgy tervezték, hogy rengeteg adatot és nagy terhelést kezeljen. A kulcs-érték páros tároló adatbázisok hash-táblákként tárolják az adatokat, ahol minden kulcs egyedi, és az érték lehet JSON, BLOB (bináris nagy objektumok), karakterlánc stb.

Példa: Redis, Voldemort, Riak és az Amazon DynamoDB.

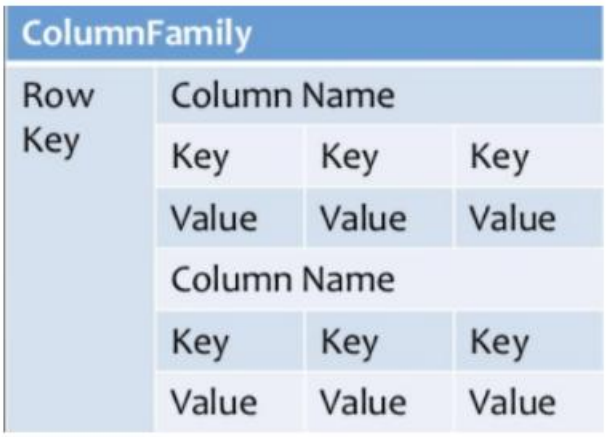


3. Oszloporientált adatbázisok (Column-Oriented Databases)

Az oszloporientált adatbázisok oszlopokon működnek, és a Google BigTable alapulnak. Minden oszlopot külön kezelünk. Az egy oszlopos adatbázisok értékei egymás mellett vannak tárolva.

Nagy teljesítményt nyújtanak olyan összesítési lekérdezéseknél, mint a SUM, COUNT, AVG, MIN stb., Mivel az adatok könnyen elérhetők egy oszlopban.

Példa: Az oszlopalapú NoSQL adatbázisokat széles körben használják az adattárházak kezelésére, az üzleti intelligencia, a CRM, a könyvtárkatalógusok, a HBase, a Cassandra, a HBase, a Hypertable.



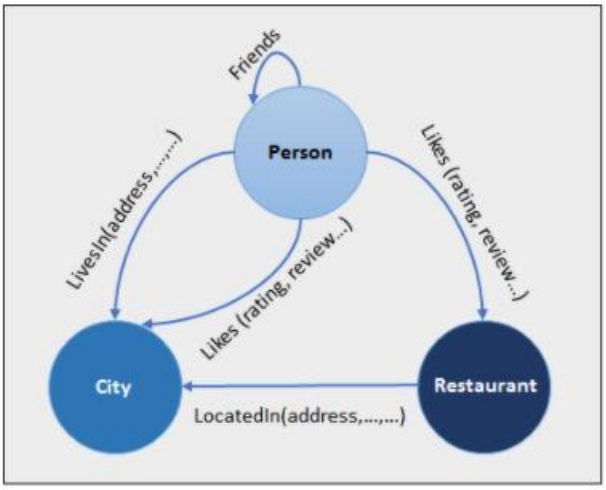
4. Gráf adatbázisok

A gráf típusú adatbázis tárolja az entitásokat, valamint az ezen entitások közötti kapcsolatokat. Az entitást csomópontként tároljuk, a kapcsolatot élként. Egy él kapcsolatot ad a csomópontok között. Minden csomópontnak és élnek egyedi azonosítója van.

Ahhoz a relációs adatbázishoz képest, ahol a táblák lazán vannak összekapcsolva, a Gráf adatbázis egy multi-relációs jellegű.

A gráf adatbázisok többnyire a szociális hálózatokhoz, a logisztikához, a téradatokhoz használatosak.

Példa: A Neo4J, az Infinite Graph, az OrientDB, a FlockDB.



Mi az a MongoDB?

A MongoDB egy dokumentum-orientált NoSQL adatbázis, amelyet nagy mennyiségű adattárolásra használnak. Ahelyett, hogy táblázatokat és sorokat használna, mint a hagyományos relációs adatbázisokban, a MongoDB gyűjteményeket és dokumentumokat használ. A dokumentumok kulcs-érték párokból állnak, amelyek a MongoDB alapvető adategységei. A gyűjtemények dokumentumok és függvények készletét tartalmazzák, amelyek megegyeznek a relációs adatbázis táblákkal.

Kulcs összetevők

1. _id: A _id mező egyedi értéket képvisel a MongoDB dokumentumban. A _id mező olyan, mint a dokumentum elsődleges kulcsa. Ha új dokumentumot hoz létre _id mező nélkül, a MongoDB automatikusan létrehozza a mezőt.

2. Gyűjtemény (Collection): Ez a MongoDB dokumentumok csoportosítása. A gyűjtemény egyenértékű egy olyan táblával, amelyet bármely más RDMS-ben, például az Oracle-ben létrehoznak.

3. Kurzor (Cursor): Ez egy mutató a lekérdezés eredményhalmazára. Az ügyfelek a kurzoron keresztül iterálhatnak az eredmények lekérése érdekében.

4. Adatbázis (Database): Ez az RDMS-hez hasonló tároló. Minden adatbázis saját fájlkészletet kap a fájlrendszeren. A MongoDB szerver több adatbázist is tárolhat.

5. Dokumentum (Document): A MongoDB gyűjteményben lévő rekordot alapvetően dokumentumnak nevezzük. A dokumentum viszont mezőnévből és értékekből áll.

6. Mező (Field): Név-érték pár a dokumentumban. A dokumentumnak nulla vagy több mezője van. A mezők analógak a relációs adatbázisok oszlopaira.

Mik az indexek a MongoDB-ben?

Az indexek támogatják a lekérdezések hatékony végrehajtását a MongoDB-ben. Index-ek nélkül a MongoDB-nek gyűjtemény-vizsgálatot kell végrehajtania, vagyis a gyűjtemény minden dokumentumát be kell szkennelnie, hogy kiválassza azokat a dokumentumokat, amelyek megfelelnek a lekérdezési utasításnak. Ha létezik megfelelő index egy lekérdezéshez, a MongoDB az index segítségével korlátozhatja az általa ellenőrizendő dokumentumok számát.

Az indexek olyan speciális adatstruktúrák, amelyek a gyűjtemény adatsorának kis részét könnyen átjárható formában tárolják. Az index egy adott mező vagy mezőkészlet értékét tárolja, a mező értéke szerint rendezve. Az indexbejegyzések sorrendje támogatja a hatékony egyenlőség-egyezéseket és tartomány-alapú lekérdezési műveleteket. Ezenkívül a MongoDB rendezett eredményeket adhat vissza az indexben lévő sorrend használatával.

Példa

A createIndex () metódus csak akkor hoz létre indexet, ha még nem létezik azonos specifikációjú index. A következő példa (a Node.js használatával) egyetlen kulcs csökkenő indexet hoz létre a névmezőben:

```
collection.createIndex( { name : -1 }, function(err, result) {
   console.log(result);
   callback(result);
}
```

Milyen típusú indexek érhetők el a MongoDB-ben?

A MongoDB a következő típusú indexeket támogatja egy lekérdezés futtatásához.

1. Single Field Index

A MongoDB támogatja a felhasználó által definiált indexeket, mint például az egy mezős indexeket. Az egyetlen mező index segítségével indexet lehet létrehozni a dokumentum egyetlen mezőjében. Egyetlen mező index esetén a MongoDB növekvő és csökkenő sorrendben haladhat. Alapértelmezés szerint minden gyűjteménynek egyetlen mező indexe van, amely automatikusan létrejön a _id mezőben, az elsődleges kulcsban.

```
{
  "_id": 1,
  "person": { name: "Alex", surname: "K" },
  "age": 29,
  "city": "New York"
}
```

Meghatározhatunk egyetlen mező indexet az életkor mezőn.

```
db.people.createIndex( {age : 1} ) // creates an ascending index

db.people.createIndex( {age : -1} ) // creates a descending index
```

Egy ilyen index segítségével javíthatunk minden olyan lekérdezést, amely feltételekkel és életkorral rendelkező dokumentumokat talál, például a következők:

```
db.people.find( { age : 20 } )
db.people.find( { name : "Alex", age : 30 } )
db.people.find( { age : { $gt : 25} } )
```

2. Összetett index (Compound Index)

Az összetett index több mező indexe. Ugyanazon emberek gyűjteményével létrehozhatunk egy összetett indexet, amely egyesíti a város és az életkor mezőt.

```
db.people.createIndex( {city: 1, age: 1, person.surname: 1  } )
```

Ebben az esetben létrehoztunk egy összetett indexet, ahol az első bejegyzés a városi mező értéke, a második a kor mező értéke, a harmadik pedig a person.name. Az összes mező itt növekvő sorrendben van meghatározva.

Az alábbi lekérdezések profitálhatnak az indexből:

```
db.people.find( { city: "Miami", age: { $gt: 50 } } )
db.people.find( { city: "Boston" } )
db.people.find( { city: "Atlanta", age: {$lt: 25}, "person.surname": "Green" } )
```

3. Multikey Index

Ez a tömbök index típusa. Ha egy tömbön indexet hoz létre, a MongoDB minden elemhez létrehoz egy index bejegyzést.

```
{
   "_id": 1,
   "person": { name: "John", surname: "Brown" },
   "age": 34,
   "city": "New York",
   "hobbies": [ "music", "gardening", "skiing" ]
 }
```

A multikey index a következőképpen hozható létre:

```
db.people.createIndex( { hobbies: 1} )
```

A következő példákhoz hasonló lekérdezések az indexet használják:

```
db.people.find( { hobbies: "music" } )
db.people.find( { hobbies: "music", hobbies: "gardening" } )
```

4. Geospatial Index

A GeoIndexes egy speciális indextípus, amely lehetővé teszi a keresést a hely, a ponttól való távolság és sok más különféle jellemző alapján. A térinformatikai adatok lekérdezéséhez a MongoDB kétféle indexet támogat - 2d indexes és 2d sphere indexeket. A 2d indexes sík geometriát használnak az eredmények visszaadásakor, a 2dsphere indexek pedig gömb geometriát használnak az eredmények visszaadásához.

5. Text Index

Ez egy másik típusú index, amelyet a MongoDB támogat. A szöveges index támogatja a string karakterláncok keresését a gyűjteményben. Ezek az indextípusok nem tárolják a nyelvspecifikus megállító szavakat (pl. "A", "a" vagy "). A szöveges indexek korlátozzák a gyűjtemény szavait, hogy csak gyökér szavakat tároljanak.

Example

Let's insert some sample documents.

```
var entries = db.people("blogs").entries;
entries.insert( {
  title : "my blog post",
  text : "i am writing a blog. yay",
  site: "home",
  language: "english" });
entries.insert( {
  title : "my 2nd post",
  text : "this is a new blog i am typing. yay",
  site: "work",
  language: "english" });
entries.insert( {
  title : "knives are Fun",
  text : "this is a new blog i am writing. yay",
  site: "home",
  language: "english" });
```

Let's define create the text index.

```
var entries = db.people("blogs").entries;
entries.ensureIndex({title: "text", text: "text"}, { weights: {
    title: 10,
    text: 5
  },
  name: "TextIndex",
  default_language: "english",
  language_override: "language" });
```

Queries such as these next examples will use the index:

```
var entries = db.people("blogs").entries;
entries.find({$text: {$search: "blog"}, site: "home"})
```

6. Hashed Index

A MongoDB támogatja a hash-alapú indexeket.

Magyarázza az index property-ket a MongoDB-ben?

1. TTL indexek

A TTL (Time To Live) egy speciális lehetőség, amelyet csak egyetlen mező indexre alkalmazhatunk, hogy lehetővé tegyük a dokumentumok automatikus törlését egy bizonyos idő után.

Az index létrehozása során meghatározhatunk egy lejárati időt. Ezt követően minden, a lejárati időnél régebbi dokumentum eltávolításra kerül a gyűjteményből. Ez a fajta szolgáltatás nagyon hasznos, ha olyan adatokkal foglalkozunk, amelyeknek nem kell fennmaradniuk az adatbázisban (pl. session data).

```
db.sessionlog.createIndex( { "lastUpdateTime": 1 }, { expireAfterSeconds: 1800 } )
```

Ebben az esetben a MongoDB automatikusan eldobja a dokumentumokat a gyűjteményből, ha fél óra (1800 másodperc) eltelt a lastUpdateTime mező értéke óta.

Korlátozások

- Csak egyetlen mező indexein lehet TTL opció
- az _id mező index nem tudja támogatni a TTL opciót
- az indexelt mezőnek dátum típusúnak kell lennie

2. Részleges indexek (Partial indexes)

A részleges index olyan index, amely csak az értékek egy részhalmazát tartalmazza egy szűrőszabály alapján. Hasznosak olyan esetekben, amikor:

- Az index mérete csökkenthető
- Indexelni akarjuk a legrelevánsabb és használt értékeket a lekérdezés feltételeiben
- Indexelni szeretnénk egy mező legszelektívebb értékeit

```
db.people.createIndex(
   { "city": 1, "person.surname": 1 },
   { partialFilterExpression: { age : { $lt: 30 } } }
)
```

Összetett indexet hoztunk létre a város és személy.név, de csak a 30 évnél fiatalabb dokumentumok esetében. A részleges index használatához a lekérdezéseknek tartalmazniuk kell egy feltételt a kor mezőben.

```
db.people.find( { city: "New Tork", age: { $eq: 20} } )
```

3. Ritka indexek (Sparse indexes)

A ritka indexek a részleges indexek részhalmaza. A ritka index csak azokat a dokumentumokat tartalmazza, amelyek indexelt mezővel rendelkeznek, még akkor is, ha az null.

Mivel a MongoDB egy sémátlan adatbázis, a gyűjtemény dokumentumai különféle mezőkkel rendelkezhetnek, így előfordulhat, hogy egyesekben nem szerepel indexelt mező.

Ilyen index létrehozásához használja a ritka opciót:

```
db.people.createIndex( { city: 1 }, { sparse: true } )
```

Ebben az esetben feltételezzük, hogy lehetnek olyan dokumentumok a gyűjteményben, amelyekből hiányzik a város mező. A ritka indexek egy mező létezésén alapulnak a dokumentumokban, és hasznosak az index méretének csökkentésére.

4. Egyedi indexek

A MongoDB egyedi indexet hozhat létre. Az így definiált index nem tartalmazhat ismétlődő bejegyzéseket.

```
db.people.createIndex( { city: 1 }, { unique: true } )
```

Az egyediség meghatározható az összetett indexek esetében is.

```
db.people.createIndex( { city: 1, person.surname: 1}, { unique: true } )
```

Alapértelmezés szerint a _id indexe automatikusan egyedi lesz.

Hány indexet hoz létre a MongoDB alapértelmezés szerint egy új gyűjteményhez?

Alapértelmezés szerint a MongoDB egy egyedi indexet hoz létre a _id mezőben a gyűjtemény létrehozása során. A _id index megakadályozza, hogy két azonos értékű dokumentumot illesszenek be a _id mezőbe.

Hozhat létre indexet a MongoDB tömbmezőjén?

Igen. Egy tömb mező indexelhető a MongoDB-ben. Ebben az esetben a MongoDB indexeli a tömb minden értékét.

Miért használja a Profiler a MongoDB-ben?

Az adatbázis-profiler rögzíti az olvasási és írási műveletekről, a kurzorműveletekről és az adatbázis-parancsokról szóló információkat. Az adatbázis-profiler adatokat ír a system.profile gyűjteménybe, amely egy korlátozott gyűjtemény.

Az adatbázis-profiler részletes információkat gyűjt a futó mongod-példány ellen végrehajtott adatbázis-parancsokról. Ez magában foglalja a CRUD műveleteket, valamint a konfigurációs és adminisztrációs parancsokat.

A Profiler 3 profilozási szinttel rendelkezik.

- szint - A Profiler nem naplóz adatokat
- szint - A Profiler csak lassú műveleteket vesz fel bizonyos küszöbérték felett
- szint - A Profiler naplózza az összes műveletet

1. To get current profiling level.

```
db.getProfilingLevel()

// Output
0
```

2. To check current profiling status

```
db.getProfilingStatus()


// Output
{ "was" : 0, "slowms" : 100 }
```

3. To set profiling level

```
db.setProfilingLevel(1, 40)

// Output
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

Hogyan lehet eltávolítani az attribútumot a MongoDB Object-ből?

$ unset

A $ unset operátor töröl egy adott mezőt. Ha a mező nem létezik, akkor a $ unset nem csinál semmit. Ha a $ -val együtt használjuk egy tömb elemhez, akkor a $ unset az egyező elemet nullával helyettesíti, ahelyett, hogy eltávolítaná a megfelelő elemet a tömbből. Ez a viselkedés megtartja a tömb méretét és az elem pozícióit.

```
{ $unset: { <field1>: "", ... } }
```

Example:

delete the `properties.service` attribute from all records on this collection.

```
db.collection.update(
    {},
    {
        $unset : {
            "properties.service" : 1
        }
    },
    {
        multi: true
    }
);
```

To verify they have been deleted you can use:

```
db.collection.find(
    {
        "properties.service" : {
           $exists : true
         }
    }
).count(true);
```

Mi a "Namespace" a MongoDB-ben?

A MongoDB BSON (Binary Interchange and Structure Object Notation) objektumokat tárol a gyűjteményben. A gyűjtemény és az adatbázis nevének összefűzését névtérnek nevezzük

Említse meg a parancsot egy dokumentum beillesztésére az iskola és az úgynevezett személyek nevű adatbázisba?

```
use school;
db.persons.insert( { name: "Alex", age: "28" } )
```

Mi a replikáció a Mongodb-ban?

A replikáció elsősorban az adatok redundanciájának és magas rendelkezésre állásának biztosítása érdekében létezik. Megőrzi az adatok tartósságát azáltal, hogy az adatok több másolatát vagy másolatát fizikailag elszigetelt szervereken tartja. A replikáció lehetővé teszi az adatok elérhetőségének növelését azáltal, hogy több másolatot készít az adatokról a szervereken. Ez különösen akkor hasznos, ha egy szerver összeomlik vagy hardverhiba merül fel.

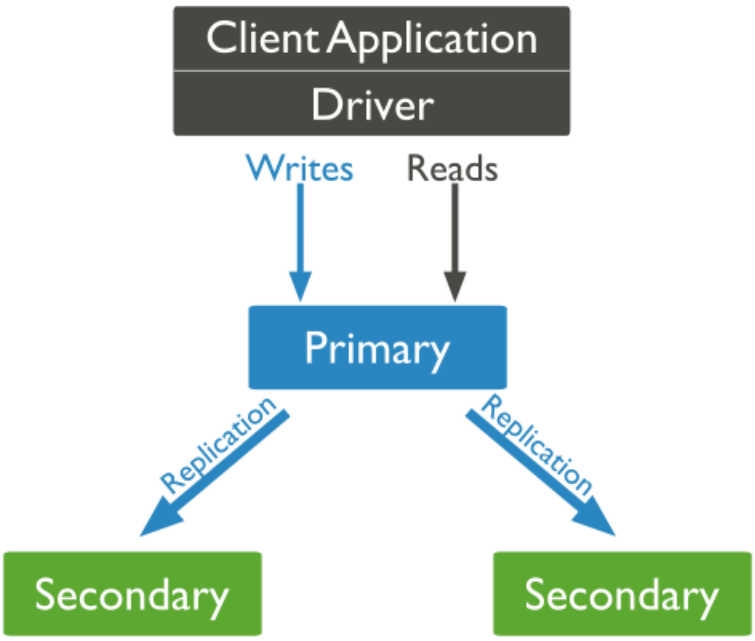
A MongoDB segítségével a replikációt egy Replika Készleten (Replica Set) keresztül érik el. Az írói műveleteket az elsődleges kiszolgálóra (csomópontra) küldik, amely a műveleteket másodlagos kiszolgálókon alkalmazza, az adatok replikálásával. Ha az elsődleges szerver meghibásodik (összeomlás vagy rendszerhiba miatt), akkor az egyik másodlagos kiszolgáló átveszi a választást, és az új elsődleges csomópont lesz. Ha ez a szerver online állapotba kerül, akkor a teljes helyreállítás után másodlagos lesz, és ez segíti az új elsődleges csomópontot.

Mi az a replika készlet (Replica Set) a MongoDB-ben?

Ez egy olyan mongo folyamatcsoport, amely ugyanazt az adatsort tartja fenn. A replikakészletek redundanciát és magas rendelkezésre állást biztosítanak, és ezek az összes production deployments alapját képezik. A replikakészlet tartalmaz egy elsődleges csomópontot és több másodlagos csomópontot.

Az elsődleges csomópont az összes írási műveletet megkapja. A replikakészletnek csak egy elsődleges lehet, amely képes megerősíteni az írásokat {w: "majority"} írási aggodalommal; bár bizonyos körülmények között egy másik mongod példány átmenetileg azt is hiheti, hogy ő is elsődleges.

A másodlagosok megismétlik az elsődleges oplog-ot, és a műveleteket úgy alkalmazzák az adathalmazaikra, hogy a másodlagos adatkészletek tükrözzék az elsődleges adatsort. Ha az elsődleges nem érhető el, a jogosult másodlagos új elsődlegessé választja magát.



Hogyan biztosítja a MongoDB a magas rendelkezésre állást?

A magas rendelkezésre állás (High Availability-HA) a rendszer és az alkalmazás elérhetőségének javítására utal a rutin karbantartási műveletek (tervezett) és a rendszer hirtelen összeomlása (nem tervezett) által okozott leállások minimalizálásával.

Replika szett (Replica Set)

A MongoDB replika készlet mechanizmusának két fő célja van:

- Az egyik az adatok redundanciája a hiba helyreállítása érdekében. Ha a hardver meghibásodik, vagy ha a csomópont egyéb okokból nem működik, használhat egy másolatot a helyreállításhoz.
- A másik cél az írás-olvasás felosztás. Átirányítja az olvasási kéréseket a replikához, hogy csökkentse az elsődleges csomópont olvasási nyomását.

A MongoDB automatikusan fenntartja a replikakészleteket, az adatok több példányát, amelyeket szerverek, között osztanak szét. A replikakészletek segítenek megakadályozni az adatbázis leállását a natív replikáció és az automatikus feladatátvétel segítségével.

A replikakészlet több replikakészlet tagból áll. Bármikor az egyik tag elsődleges, a többi tag másodlagosként jár el. Ha az elsődleges tag valamilyen okból meghiúsul (pl. Hardverhiba), akkor az

egyik másodlagos tagot automatikusan megválasztják az elsődleges taggá, és elkezdi feldolgozni az összes olvasást és írást.

Mi az a beágyazott MongoDB dokumentum (Embedded MongoDB Document)?

A beágyazott MongoDB dokumentum egy normál dokumentum, amelyet a MongoDB gyűjtemény egy másik dokumentumába ágyaznak be. A csatlakoztatott adatok egyetlen dokumentumba ágyazása csökkentheti az adatok megszerzéséhez szükséges olvasási műveletek számát. Általában úgy kell strukturálnunk a sémánkat, hogy az alkalmazás minden szükséges információt egyetlen olvasási művelet során megkapjon.

Példa:

A normalizált adatmodellben az address dokumentumok hivatkozást tartalmaznak a patron dokumentumra.

```
// patron document
{
   _id: "joe",
   name: "Joe Bookreader"
}

// address documents
{
   patron_id: "joe", // reference to patron document
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: "12345"
}
```

A beágyazott dokumentumok különösen akkor hasznosak, ha a dokumentumok között egy az egyhez viszony áll fenn. A fenti példában azt látjuk, hogy egyetlen ügyfélnek több címe van társítva. A beágyazott dokumentumstruktúra megkönnyíti az ügyfél teljes címadatainak egyetlen lekérdezéssel történő lekérését.

Hogyan érheti el az elsődleges kulcs - az idegen kulcs kapcsolatait a MongoDB-ben?

Az elsődleges kulcs-idegen kulcs kapcsolat úgy érhető el, hogy egyik dokumentumot beágyazzák a másikba. Például egy részlegdokumentum rendelkezhet munkavállalói dokumentumaival.

Mikor kell beágyaznunk egy dokumentumot a másikba a MongoDB-be?

- entitások közötti kapcsolatokat tartalmaz
- Egy a sokhoz viszony (One-to-many relationships)
- Teljesítményi okok

Hogyan tároljuk az adatokat a MongoDB-ben?

A MongoDB-ben az adatokat BSON-dokumentumok tárolják (a bináris JSON rövidítése). Ezeket a dokumentumokat a MongoDB tárolja JSON (JavaScript Object Notation) formátumban. A JSON-dokumentumok támogatják a beágyazott mezőket, így a kapcsolódó adatok és adatlisták külső dokumentum helyett a dokumentummal együtt tárolhatók. A dokumentumok egy vagy több mezőt tartalmaznak, és mindegyik mező tartalmaz egy adott adattípus értékét, beleértve a tömböket, a bináris adatokat és az aldokumentumokat. A hasonló struktúrával rendelkező dokumentumok gyűjteményként vannak rendezve.

A JSON név / érték párként van formázva. A JSON-dokumentumokban a mezők nevét és értékét kettőspont választja el, a mezőneveket és az értékpárokat vesszők választják el, a mezőkészleteket pedig "göndör zárójelek" ({}) közé foglalják.

```
{
  "name": "notebook",
  "qty": 50,
  "rating": [ { "score": 8 }, { "score": 9 } ],
  "size": { "height": 11, "width": 8.5, "unit": "in" },
  "status": "A",
  "tags": [ "college-ruled", "perforated"]
}
```

Mi a különbség a MongoDB és az SQL-SERVER között?

A MongoDB az adatokat JSON formátumú dokumentumokban tárolja, de az SQL táblázat formátumban.

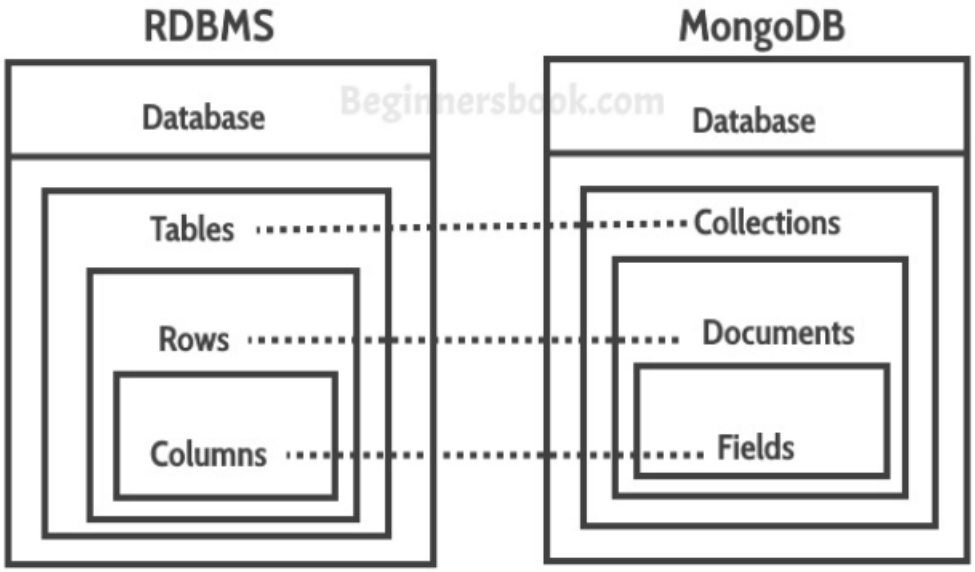
A MongoDB az SQL Server helyett nagy teljesítményt, magas rendelkezésre állást, könnyű skálázhatóságot stb. nyújt.

A MongoDB-ben egyszerűen megváltoztathatjuk a struktúrát azáltal, hogy hozzáadjuk, eltávolítjuk az oszlopot a meglévő dokumentumokból.

MongoDB and SQL Server Comparision Table

Base of Comparison	MS SQL Server	MongoDB
Storage Model	RDBMS	Document-Oriented
Joins	Yes	No

Base of Comparison	MS SQL Server	MongoDB
Transaction	ACID	Multi-document ACID Transactions with snapshot isolation
Agile practices	No	Yes
Data Schema	Fixed	Dynamic
Scalability	Vertical	Horizontal
Map Reduce	No	Yes
Language	SQL query language	JSON Query Language
Secondary index	Yes	Yes
Triggers	Yes	Yes
Foreign Keys	Yes	No
Concurrency	Yes	yes
XML Support	Yes	No



Hogyan érhet el tranzakciót és zárolást a MongoDB-ben?

A MongoDB-ben (4.2.) Egyetlen dokumentum művelete atomi. Azokban a helyzetekben, amelyek több dokumentumra (egy vagy több gyűjteményben) írási és olvasási atomiát igényelnek, a MongoDB támogatja a több dokumentumot tartalmazó tranzakciókat. Elosztott tranzakciókkal a tranzakciók több műveleten, gyűjteményeken, adatbázisokon, dokumentumokon is felhasználhatók.

A MongoDB lehetővé teszi több kliens számára ugyanazok az adatok olvasását és írását. Az egységesség biztosítása érdekében zárolást és más egyidejűség-ellenőrzési intézkedéseket alkalmaz annak megakadályozására, hogy több kliens egyszerre módosítsa ugyanazt az adatot.

A MongoDB több részletességű zárolást használ, amely lehetővé teszi a műveletek zárolását globális, adatbázis vagy gyűjtemény szinten, és lehetővé teszi az egyes tároló motorok számára, hogy a saját párhuzamosság-szabályozásukat a gyűjtemény szintje alatt valósítsák meg (például a WiredTiger dokumentumszintjén). A MongoDB olvasó-író zárakat használ, amelyek lehetővé teszik az olvasók egyidejű hozzáférését egy erőforráshoz, például adatbázishoz vagy gyűjteményhez.

Lock Mode	Description
R	Represents Shared (S) lock.
W	Represents Exclusive (X) lock.
r	Represents Intent Shared (IS) lock.
w	Represents Intent Exclusive (IX) lock.

```
const client = new MongoClient(uri);
await client.connect();

// Prereq: Create collections.

await client.db('mydb1').collection('foo').insertOne({ abc: 0 }, { w: 'majority' });

await client.db('mydb2').collection('bar').insertOne({ xyz: 0 }, { w: 'majority' });

// Step 1: Start a Client Session
const session = client.startSession();

// Step 2: Optional. Define options to use for the transaction
const transactionOptions = {
  readPreference: 'primary',
  readConcern: { level: 'local' },
  writeConcern: { w: 'majority' }
};

// Step 3: Use withTransaction to start a transaction, execute the callback, and commit (or abort on
error)
// Note: The callback for withTransaction MUST be async and/or return a Promise.
try {
  await session.withTransaction(async () => {
    const coll1 = client.db('mydb1').collection('foo');
    const coll2 = client.db('mydb2').collection('bar');

    // Important:: You must pass the session to the operations

    await coll1.insertOne({ abc: 1 }, { session });
    await coll2.insertOne({ xyz: 999 }, { session });
  }, transactionOptions);
```

```
} finally {
   await session.endSession();
   await client.close();
}
```

Mikor használja a MongoDB-t a MySQL helyett?

1. MongoDB

A MongoDB az egyik legnépszerűbb dokumentumorientált adatbázis a NoSQL adatbázis zászlaja alatt. A kulcs-érték párok formátumát használja, itt dokumentumtárnak hívják. A MongoDB-ben létrehozott dokumentumtárolók BSON fájlokban vannak tárolva, amelyek valójában a JSON fájlok kissé módosított változatai, és ezért az összes JS támogatott.

Nagyobb hatékonyságot és megbízhatóságot kínál, ami viszont megfelel a tárolási kapacitás és a sebesség igényeinek. A MongoDB sémamentes megvalósítása kiküszöböli a rögzített struktúra meghatározásának előfeltételeit. Ezek a modellek lehetővé teszik a hierarchikus kapcsolatok ábrázolását, és megkönnyítik a rekord szerkezetének megváltoztatását.

Előnyök

- A MongoDB lekérdezésenként alacsonyabb késleltetéssel rendelkezik, és kevesebb CPU-időt tölt lekérdezésenként, mert sokkal kevesebb munkát végez (pl. Nincs join, tranzakció). Ennek eredményeként nagyobb terhelést képes kezelni a másodpercenkénti lekérdezések szempontjából.
- A MongoDB-t könnyebb feldarabolni (cluster-ben használni), mert nem kell aggódnia a tranzakciók és a következetesség miatt.
- A MongoDB gyorsabb írási sebességgel rendelkezik, mert nem kell aggódnia tranzakciók vagy rollback miatt (és ezért nem kell aggódnia a zárolás miatt sem).
- Számos olyan funkciót támogat, mint az automatikus javítás, az egyszerűbb adatelosztás és az egyszerűbb adatmodellek, amelyek csökkentik az adminisztrációs és hangolási követelményeket a NoSQL-ben.
- A NoSQL adatbázisok olcsók és nyílt forráskódúak.
- A NoSQL adatbázis támogatja a gyorsítótárat a rendszermemóriában, így növeli az adatkimenet teljesítményét.

Hátrányok

- A MongoDB nem támogatja a tranzakciókat.
- Általánosságban elmondható, hogy a MongoDB több munkát (pl. Több CPU-költséget) teremt a kliens számára. Például az adatok összekapcsolásához több kérdést kell kiadnia, és meg kell tennie a csatlakozást a klienshez.
- Nincsenek tárolt eljárások mongoDB-ben (NoSQL adatbázis).

A NoSQL adatbázis használatának okai

- Nagy mennyiségű adat tárolása struktúra nélkül: A NoSQL adatbázis nem korlátozza a tárolható adattípusokat. Ezenkívül új típusokat is felvehet az üzleti igények változásával.

- A felhőalapú számítástechnika és a tárolás használata: A felhőalapú tárolás nagyszerű megoldás, de a méretezéshez az adatok könnyen elosztása szükséges több szerveren. A NoSQL adatbázisokat a megfizethető hardveres helyszíni teszteléshez, majd felhőben történő gyártáshoz használják.
- Gyors fejlesztés: Ha modern agilis módszertanok segítségével fejlesztesz, a relációs adatbázis lelassít. A NoSQL adatbázis nem igényli a relációs adatbázisokhoz általában szükséges előkészítési szintet.

2. MySQL

A MySQL egy népszerű nyílt forráskódú relációs adatbázis-kezelő rendszer (RDBMS), amelyet az Oracle Corporation fejleszt, terjeszt és támogat. A MySQL adatokat táblákban tárol, és strukturált lekérdezési nyelvet (SQL) használ az adatbázis-hozzáféréshez. A Strukturált Lekérdező Nyelv (Structured Query Language – SQL)-t használja az adatok és a parancsok, például a „SELECT", „UPDATE", „INSERT" és „DELETE" eléréséhez és továbbításához azok kezeléséhez.

A kapcsolódó információkat különböző táblákban tárolják, de a JOIN műveletek fogalma leegyszerűsíti annak összefüggését és a lekérdezések végrehajtását több táblázat között, és minimalizálja az adatok duplikálásának esélyét. Az ACID (Atomic, Consistent, Isolated és Durable) modellt követi. Ez azt jelenti, hogy amint a tranzakció befejeződik, az adatok konzisztensek és stabilak maradnak a lemezen, amely különálló memóriahelyeket tartalmazhat.

Előnyök

- Az SQL adatbázisok tábla alapú adatbázisok.
- Adattárolás sorokban és oszlopokban
- Minden sor egyedi példányt tartalmaz az oszlopok által definiált kategóriákhoz.
- elsődleges kulcsát a sorok egyedi azonosításához.

Hátrányok

- A felhasználóknak át kell méretezniük a relációs adatbázist olyan erős szervereken, amelyek drágák és nehezen kezelhetők. A relációs adatbázis méretezéséhez több szerverre kell elosztani. A táblázatok kezelése különböző szervereken nehéz.
- Az SQL szerver adatainak mindenképp táblákba kell illeszkedniük. Ha az adatai nem férnek be a táblákba, akkor meg kell terveznie az adatbázis-struktúrát, amely összetett és megint nehezen kezelhető.

Hogyan támogatja a MongoDB az ACID tranzakciókat és a zárolási funkciókat?

A MongoDB mindig támogatta az ACID tranzakciókat egyetlen dokumentumban, és a dokumentummodell megfelelő kihasználása során sok alkalmazásnak nincs szüksége ACID garanciára több dokumentumban.

A MongoDB dokumentum alapú NoSQL adatbázis rugalmas sémával. A tranzakciók nem olyan műveletek, amelyeket minden írási műveletnél végre kell hajtani, mivel nagyobb teljesítményköltséget jelentenek egyetlen dokumentumíráshoz képest. Dokumentum alapú felépítés és denormalizált adatmodell esetén minimális szükség van a tranzakciókra. Mivel a MongoDB lehetővé teszi a dokumentum beágyazását, nem feltétlenül kell tranzakciót használnia az írási művelet teljesítéséhez.

Példa: Többdokumentumos ACID-tranzakciók a MongoDB-ben

Ezek több állításból álló műveletek, amelyeket egymás után kell végrehajtani, anélkül, hogy befolyásolnák egymást. Az alábbiakban például két tranzakciót hozhatunk létre, az egyiket egy felhasználó hozzáadásához, a másikat pedig egy kor frissítéséhez.

```
$session.startTransaction()
   db.users.insert({_id:6, name “Ibrahim”})
   db.users.updateOne({_id:3 , {$set:{age:50}}})
session.commit_transaction()
```

A tranzakciókat lehet alkalmazni egy vagy több gyűjteményben / adatbázisban található több dokumentum elleni műveletekre. A dokumentumtranzakció miatti bármilyen változás nem befolyásolja a nem kapcsolódó munkaterhelések teljesítményét, vagy nem igényli őket. Amíg a tranzakciót nem hajtják végre, a nem kommitolt írásokat nem replikálják a másodlagos csomópontokra, és a tranzakciókon kívül sem olvashatók.

Melyek a legjobb gyakorlatok a MongoDB tranzakciókhoz?

A több dokumentumot tartalmazó tranzakciókat csak a WiredTiger tárolómotor támogatja. Egyetlen ACID tranzakció esetén, ha túl sok műveletet próbál végrehajtani, az nagy nyomást eredményezhet a WiredTiger gyorsítótárára. A gyorsítótár mindig a legrégebbi snapshot létrehozása óta diktálja az összes későbbi írás állapotának fenntartását. Ez azt jelenti, hogy az új írások a tranzakció teljes időtartama alatt felhalmozódnak a gyorsítótárban, és csak azután törlődnek, hogy a régi snapshot-okon futó tranzakciók végrehajtása vagy megszakítása megtörtént.

A tranzakció legjobb adatbázis-teljesítménye érdekében a fejlesztőknek mérlegelniük kell:

- A tranzakció során mindig módosítson kis számú dokumentumot. Ellenkező esetben a tranzakciót különböző részekre kell bontania, és a dokumentumokat különböző kötegekben kell feldolgoznia. Egyszerre legfeljebb 1000 dokumentumot dolgozhat fel.
- Ideiglenes kivételek (pl. hálózati), a tranzakció megszakadásához vezethetnek. A fejlesztőknek létre kell hozniuk egy logikát a tranzakció újrapróbálására, ha a definiált hibák megjelennek.
- Konfigurálja a tranzakció végrehajtásának optimális időtartamát a MongoDB által biztosított alapértelmezett 60 másodpercből. Ezenkívül alkalmazzon indexelést, hogy az gyors hozzáférést biztosítson a tranzakción belül.

- Bontsa szét a tranzakciót egy kis műveletsorozatba, hogy az megfeleljen a 16 MB-os méretkorlátoknak. Ellenkező esetben, ha a művelet az oplog leírásával együtt meghaladja ezt a korlátot, a tranzakció megszakad.
- Az entitásra vonatkozó összes adatot egyetlen, gazdag dokumentumstruktúrában kell tárolni. Ezzel csökkenthető a gyorsítótárba helyezett dokumentumok száma, amikor a különböző mezőket megváltoztatja.

Magyarázza meg a MongoDB tranzakciók korlátait?

A MongoDB tranzakciók csak viszonylag rövid ideig létezhetnek. Alapértelmezés szerint a tranzakciónak legfeljebb egy percnyi időtartamot kell lefednie. Ez a korlátozás a mögöttes MongoDB megvalósításból származik. A MongoDB az MVCC-t használja, de az olyan adatbázisokkal ellentétben, mint az Oracle, az adatok „régebbi" verzióit csak a memóriában tárolják.

- Nem hozhat létre vagy törölhet gyűjteményt egy tranzakcióban.
- A tranzakciók nem írhatnak egy korlátozott gyűjteménybe
- A tranzakciók végrehajtása sok időt vesz igénybe, és valahogy lassíthatják az adatbázis teljesítményét.
- A tranzakciók mérete 16 MB-ra korlátozódik, ehhez meg kell osztani azokat, amelyek ezt a méretet meghaladják, kisebb tranzakciókra.
- Ha nagy számú dokumentumot vetünk alá egy tranzakciónak, túlzott nyomás nehezedhet a WiredTiger motorra, és mivel az a snapshot képességre támaszkodik, nagy mûveletek megmaradnak a memóriában. Ez némi teljesítményköltséget jelent az adatbázisban.

Hogyan lehet egyesíteni a több gyűjteményből származó adatokat egy gyűjteményben?

$ lookup

left outer join-t végez egy ugyanabban az adatbázisban lévő gyűjteményhez, hogy a „joined" gyűjteményből származó dokumentumokat szűrje feldolgozás céljából. Minden bemeneti dokumentumhoz a $ lookup szakasz hozzáad egy új tömbmezőt, amelynek elemei a „joined" gyűjteményből származó megfelelő dokumentumok. A $ lookup szakasz ezeket az átalakított dokumentumokat a következő szakaszba továbbítja.

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

Keressen objektumokat két dátum között MongoDB?

A $ gte és a $ lt operátor két dátum közötti objektumok keresésére szolgál a MongoDB-ben.

Example: Creating a collection

```
>db.order.insert({"OrderId":1,"OrderAddrees":"US","OrderDateTime":ISODate("2020-02-19")};
WriteResult({ "nInserted" : 1 })

>db.order.insert({"OrderId":2,"OrderAddrees":"UK","OrderDateTime":ISODate("2020-02-26")};
WriteResult({ "nInserted" : 1 })
```

Display all documents from the collection using `find()` method.

```
> db.order.find().pretty();

// Output
{
   "_id" : ObjectId("5c6c072068174aae23f5ef57"),
   "OrderId" : 1,
   "OrderAddrees" : "US",
   "OrderDateTime" : ISODate("2020-02-19T00:00:00Z")
}
{
   "_id" : ObjectId("5c6c073568174aae23f5ef58"),
   "OrderId" : 2,
   "OrderAddrees" : "UK",
   "OrderDateTime" : ISODate("2020-02-26T00:00:00Z")
}
```

Here is the query to find objects between two dates:

```
> db.order.find({"OrderDateTime":{ $gte:ISODate("2020-02-10"), $lt:ISODate("2020-02-21") }
}).pretty();


// Output
{
   "_id" : ObjectId("5c6c072068174aae23f5ef57"),
   "OrderId" : 1,
   "OrderAddrees" : "US",
   "OrderDateTime" : ISODate("2020-02-19T00:00:00Z")
}
```

Hogyan lehet lekérdezni a MongoDB-t "like" -al?

```
db.users.insert({name: 'paulo'})
db.users.insert({name: 'patric'})
db.users.insert({name: 'pedro'})

db.users.find({name: /a/})  //like '%a%'
db.users.find({name: /^pa/}) //like 'pa%'
db.users.find({name: /ro$/}) //like '%ro'
```

Mi az upsert a MongoDB-ben?

A MongoDB-ben történő upsert műveletet használják a dokumentum gyűjteménybe mentésére. Ha a dokumentum megfelel a lekérdezési feltételeknek, akkor frissítési műveletet hajt végre, különben új dokumentumot illeszt be a gyűjteménybe.

Az Upsert művelet akkor hasznos, ha külső forrásból importál adatokat, amelyek frissítik a meglévő dokumentumokat, ha azok egyeznek, különben új dokumentumokat helyez be a gyűjteménybe.

Példa: A frissítéshez állítsa be az Upsert lehetőséget

Ez a művelet először megkeresi a dokumentumot, ha nincs, akkor beilleszti az új dokumentumot az adatbázisba.

```
> db.car.update(
...     { name: "Qualis" },
...     {
...        name: "Qualis",
...        speed: 50
...     },
...     { upsert: true }
... )
WriteResult({
           "nMatched" : 0,
           "nUpserted" : 1,
           "nModified" : 0,
           "_id" : ObjectId("548d3a955a5072e76925dc1c")
})
```

A Qualis nevű autó létezését ellenőrizzük, és ha nincs, akkor egy "Qualis" nevű és 50-es sebességű dokumentum kerül az adatbázisba. Az "1" értékkel feltüntetett nUpserted dokumentum új dokumentum beszúrását jelzi.

Van-e "upsert" opció a mongodb insert parancsban?

A db.collection.insert () nem kínál upsert lehetőséget. Ehelyett a mongo insert beszúr egy új dokumentumot a gyűjteménybe. A feltöltés csak a db.collection.update () és a db.collection.save () használatával lehetséges.

Mi az oplog?

Az OpLog (Operations Log - Műveleti napló) egy speciális, korlátozott gyűjtemény, amely nyilvántartást vezet minden olyan műveletről, amely módosítja az adatbázisokban tárolt adatokat.

A MongoDB az elsődleges adatbázis-műveleteket alkalmaz, majd rögzíti a műveleteket az elsődleges oplog-ján. Ezután a másodlagos tagok ezeket a műveleteket aszinkron folyamatban másolják és alkalmazzák. A replikakészlet minden tagja tartalmazza az oplog egy példányát a local.oplog.rs gyűjteményben, amely lehetővé teszi számukra az adatbázis aktuális állapotának fenntartását.

Az oplog minden művelete idempotens. Vagyis az oplog műveletek ugyanazokat az eredményeket produkálják, akár egyszer, akár többször alkalmazzák a cél adatkészletre.

```
MongoDB shell version: 2.0.4
connecting to: mongodb:27017/test
PRIMARY> use local
PRIMARY> db.oplog.rs.find()
```

A MongoDB immediately vagy lazily írja a lemezre az írásokat?

A MongoDB lazy módon az adatokat a lemezre. Frissíti a naplóba azonnal, de az adatok naplóról lemezre írása lazy történik.

Hogyan lehet végrehajtani a törlési műveletet a MongoDB-ben?

A MongoDB db.collection.deleteMany () és db.collection.deleteOne () metódusával törölhetők a dokumentumok a gyűjteményből. A törlési műveletek nem dobják el az indexeket, még akkor sem, ha az összes dokumentumot törlik a gyűjteményből. Az összes írási művelet a MongoDB-ben egyetlen dokumentum szintjén atomi.

```
// Create Inventory Collection
db.inventory.insertMany( [
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "P" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" },
] );


// Delete Commands
db.inventory.deleteMany({}) // Delete All Documents

db.inventory.deleteMany({ status : "A" }) // Delete All Documents that Match a Condition

db.inventory.deleteOne( { status: "D" } ) // Delete Only One Document that Matches a Condition
```

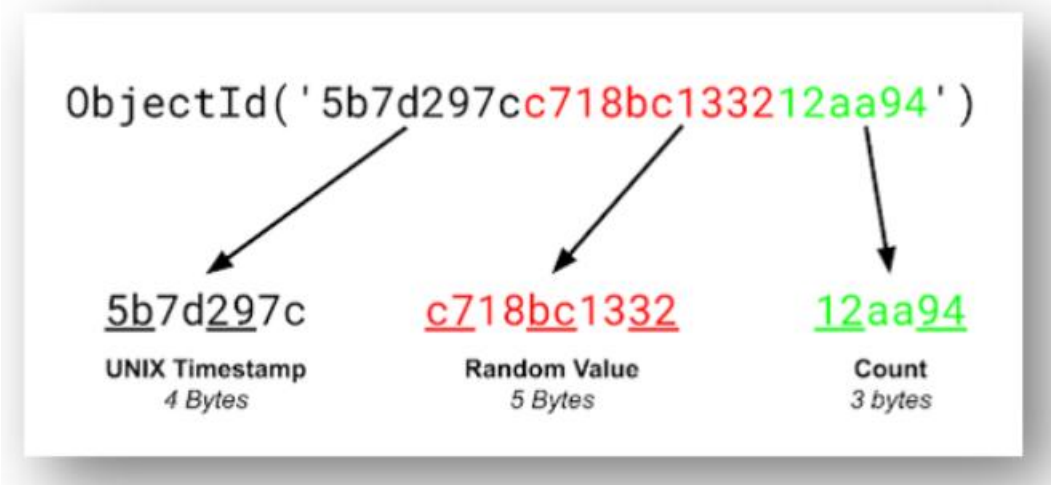
Ha eltávolít egy dokumentumot az adatbázisból, akkor a MongoDB eltávolítja a lemezről?

Igen. Ha eltávolít egy dokumentumot az adatbázisból, a MongoDB eltávolítja azt a lemezről is.

Magyarázza el az ObjectID felépítését a MongoDB-ben?

Az ObjectId (<hexadecimal>) osztály az alapértelmezett elsődleges kulcs egy MongoDB dokumentumhoz, és általában egy beillesztett dokumentum _id mezőjében található. Új ObjectId értéket ad vissza. A 12 bájtos ObjectId érték a következőkből áll:

- 4 bájtos időbélyeg-érték, amely az ObjectId létrehozását ábrázolja, másodpercek alatt mérve a Unix-korszak óta
- 5 bájtos véletlenszerű érték
- 3 bájtos növekményes számláló, véletlenszerű értékre inicializálva



Új objektumazonosító manuális létrehozásához a MongoDB-ben metódusként deklarálhatjuk az objectId () -t.

```
> newObjectId = ObjectId();

// Output
ObjectId("5349b4ddd2781d08c09890f3")
```

MongoDB provides three methods for ObjectId

Method	Description
ObjectId.getTimestamp()	Az objektum időbélyegző részét Dátumként adja vissza.
ObjectId.toString()	Visszaadja a JavaScript reprezentációt karakterlánc formájában "ObjectId (...)".
ObjectId.valueOf()	Az objektum reprezentációját adja vissza hexadecimális karakterláncként.

Mi a covered lekérdezés a MongoDB-ben?

A MongoDB által covered lekérdezés indexet használ, és nem kell egyetlen dokumentumot sem megvizsgálnia. Az index akkor fedi le a lekérdezést, ha megfelel a következő feltételeknek:

- A lekérdezés összes mezője egy index része.
- Az eredményekben visszaadott összes mező azonos indexű.
- a lekérdezés egyetlen mezője sem egyenlő null-al

Mivel a lekérdezésben szereplő összes mező egy index része, a MongoDB megegyezik a lekérdezés feltételeivel, és az eredményt ugyanazon index használatával adja vissza, anélkül, hogy a dokumentumokba nézne.

Példa:

A gyűjtemény-készlet a következő indexet tartalmazza a típus és az elem mezőkben:

```
db.inventory.createIndex( { type: 1, item: 1 } )
```

Ez az index a következő műveletet fedi le, amely a típus és az elem mezőkre kérdez, és csak az elem mezőt adja vissza:

```
db.inventory.find(
   { type: "food", item:/^c/ },
   { item: 1, _id: 0 }
)
```

Mi a Sharding a MongoDB-ben?

A Sharding egy módszer az adatok több gépen történő elosztására. A MongoDB a sharding-et használja a nagyon nagy adathalmazokkal műveletekkel rendelkező deployment támogatására.

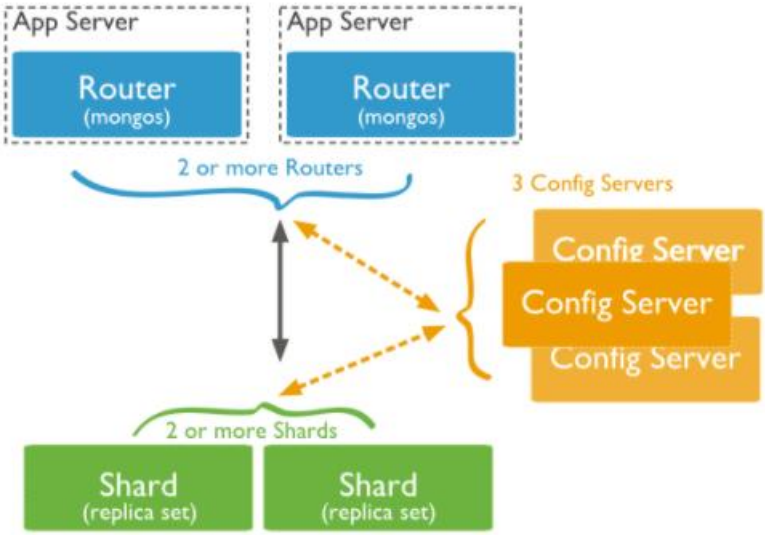
Nagy adathalmazokkal rendelkező alkalmazásokkal r adatbázis-rendszerek megkérdőjelezhetik egyetlen szerver kapacitását. Például a magas lekérdezési arány kimerítheti a szerver CPU-kapacitását. A rendszer RAM-jánál nagyobb méretû munkakészletek megterhelik a lemezmeghajtók I / O kapacitását. Két módszer létezik a rendszer növekedésének kezelésére: vertikális és horizontális skálázás.

1. Vertical Scaling

A Vertical Scalingmagában foglalja egyetlen szerver kapacitásának növelését, például egy erősebb CPU használatát, több RAM hozzáadását vagy a tárhely növelését.

2. Horizontal Scaling

A Horizontal Scaling magában foglalja a rendszer adatkészletének és terhelésének felosztását több szerveren, további kiszolgálók hozzáadásával a kapacitás növeléséhez szükség szerint. Míg egyetlen gép teljes sebessége vagy kapacitása nem biztos, hogy mindegyik magas, mindegyik gép kezeli a teljes munkaterhelés egy részét, ami potenciálisan jobb hatékonyságot biztosít, mint egyetlen nagy sebességű, nagy kapacitású szerver.



A MongoDB a horizontális skálázást sharding-el támogatja. A MongoDB sharded cluster a következő összetevőkből áll:

- Shards: Minden Shard a széttört adatok egy részhalmazát tartalmazzák. Minden Shard replikakészletként telepíthető.
- Mongos: A Mongo-k lekérdezési útválasztóként működnek, interfészt biztosítva a kliens alkalmazások és a sharded cluster között.
- Config Servers: A Config szerverek a cluster-ek metaadatait és konfigurációs beállításait tárolják.

Mi az Aggregation (összesítés) a MongoDB-ben?

Az Aggregation a MongoDB-ben olyan művelet, amelyet az adatok feldolgozására használnak, amelyek visszaadják a kiszámított eredményeket. Az összesítés alapvetően több dokumentum adatait csoportosítja, és sokféleképpen működik ezeken a csoportosított adatokon annak érdekében, hogy egy kombinált eredményt adjanak vissza.

Az összesített függvény csoportosítja a gyűjtemény rekordjait, és felhasználható a kiválasztott csoport összes számának (összegének), átlagának, minimumának, maximumának stb. Az összesített függvény MongoDB-ben történő végrehajtásához az aggregate () az a függvény, amelyet használni kell. A következő az összesítés szintaxisa.

```
db.collection_name.aggregate(aggregate_operation)
```

A MongoDB aggregációs keretrendszerét az adatfeldolgozó csővezetékek koncepciója mintázza. A dokumentumok egy többlépcsős folyamatba lépnek, amely a dokumentumokat összesített eredménnyé alakítja.

```
db.orders.aggregate([
```

```
   { $match: { status: "A" } },
   { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
]);
```

A $ match szakasz kiszűri a dokumentumokat az állapotmező alapján, és a következő szakaszba továbbítja azokat a dokumentumokat, amelyek állapota egyenlő "A" -val. A $ group szakasz a dokumentumokat a cust_id mező szerint csoportosítja, hogy kiszámolja az egyes egyedi cust_id összegeket.

Expression	Description
$sum	Összeadja a megadott értékeket a gyűjtemény összes dokumentumából
$avg	Kiszámítja az átlagértékeket a gyűjtemény összes dokumentuma alapján
$min	Visszaadja a gyűjtemény összes dokumentumának minimális értékét
$max	Visszaadja a gyűjtemény összes dokumentumának maximális értékét
$addToSet	Beilleszt egy értéket egy tömbbe, de nincs duplikátum a kapott dokumentumban
$push	Értékeket illeszt be egy tömbbe a kapott dokumentumban
$first	Visszaadja az első dokumentumot a forrásdokumentumból
$last	Visszaadja az utolsó dokumentumot a forrásdokumentumból

Hogyan lehet elkülöníteni a kurzorokat az írási műveletek beavatkozásától?

Mivel a kurzor élettartama alatt nincs elkülönítve, így a dokumentumok írásbeli műveleteinek beavatkozása olyan kurzort eredményezhet, amely többször is visszaküldi a dokumentumot. A snapshot () metódus használható egy kurzoron a művelet elkülönítésére egy nagyon konkrét esetre. A snapshot () bejárja az indexet a _id mezőben, és garantálja, hogy a lekérdezés minden dokumentumot legfeljebb egyszer ad vissza.

Korlátozás:

- Nem használhatunk snapshot ()-ot sharded collection-el.
- A snapshot () nem használható sort () vagy hint ()-el.

Milyen időközönként ír a MongoDB frissítéseket a lemezre?

Alapértelmezés szerint a MongoDB 60 másodpercenként írja a frissítéseket a lemezre. Ez azonban konfigurálható a bindIntervalMs és syncPeriodSecs beállításokkal.

Parancs az indexek eltávolítására és az összes index felsorolására egy adott gyűjteményben?

List all Indexes on a Collection

```
// To view all indexes on the people collection

db.people.getIndexes()
```

List all Indexes for a Database

```
// To list all the collection indexes in a database

db.getCollectionNames().forEach(function(collection) {
   indexes = db[collection].getIndexes();
   print("Indexes for " + collection + ":");
   printjson(indexes);
});
```

Remove Indexes

MongoDB provides two methods for removing indexes from a collection:

- `db.collection.dropIndex()`
- `db.collection.dropIndexes()`

1. Remove Specific Index

```
db.accounts.dropIndex( { "tax-id": 1 } )


// Output
{ "nIndexesWas" : 3, "ok" : 1 }
```

2. Remove All Indexes

```
// The following command removes all indexes from the accounts collection

db.accounts.dropIndexes()
```

Mi történik, ha egy index nem fér bele a RAM-ba?

Ha az indexek nem férnek bele a RAM-ba, a MongoDB a lemezről olvassa az adatokat, ami viszonylag sokkal lassabb, mint a RAM-ból olvasás.

Az indexeknek nem minden esetben kell teljesen beleilleszkedniük a RAM-ba. Ha az indexelt mező értéke minden beszúrással növekszik, és a legtöbb lekérdezés nemrégiben hozzáadott dokumentumokat választ; akkor a MongoDB-nek csak az index azon részeit kell a RAM-ban tartania,

amelyek a legfrissebb vagy a "jobb oldali" értékeket tárolják. Ez lehetővé teszi az index hatékony használatát az írási és olvasási műveleteknél, és minimalizálja az index támogatásához szükséges RAM mennyiségét.

Example: To check the size of indexes

```
> db.collection.totalIndexSize()

// Output (in bytes)
4294976499
```

Biztosít-e a MongoDB lehetőséget szöveges keresésre?

A MongoDB támogatja azokat a lekérdezési műveleteket, amelyek szöveges keresést hajtanak végre a karakterlánc tartalmában. Szöveges keresés végrehajtásához a MongoDB szövegindexet és a $ text operátort használ.

Példa

Egy gyűjtemény a következő dokumentumokat tárolja:

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

1. Text Index

A MongoDB szöveges indexeket biztosít a szöveges keresési lekérdezések támogatásához a karakterlánc tartalmán. a szöveges indexek bármely mezőt tartalmazhatnak, amelynek értéke karakterlánc vagy string elemek tömbje.

```
db.stores.createIndex( { name: "text", description: "text" } )
```

2. $ text Operator

Használja a $ text query operátort szöveges keresések végrehajtására szövegindexet tartalmazó gyűjteményen.

Example:

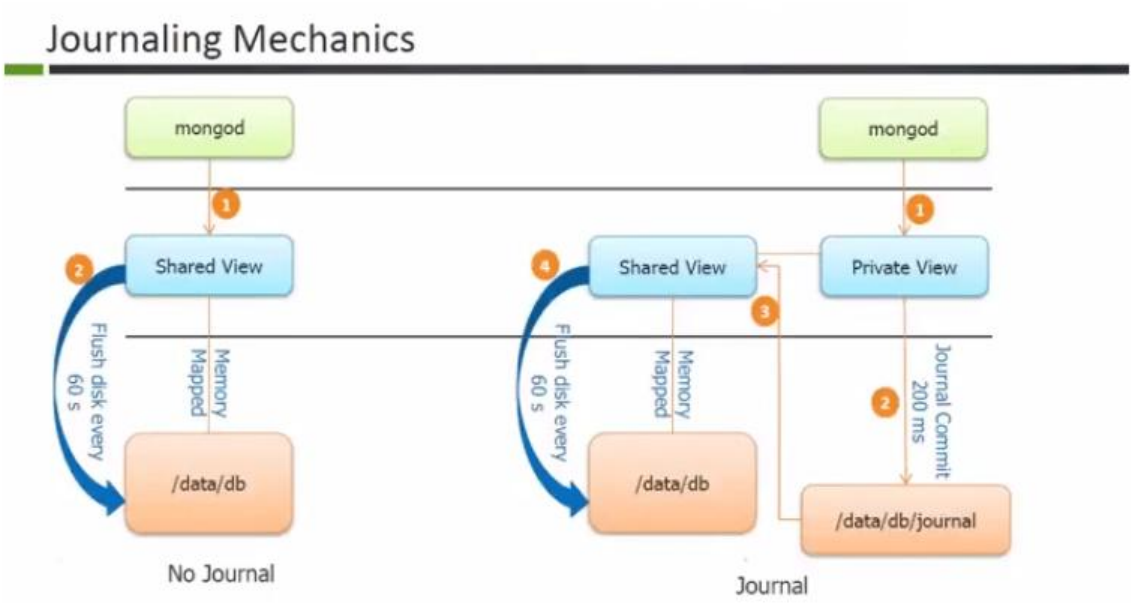
```
// Returns all stores containing any terms from the list “coffee”, “shop”, and “java”
db.stores.find( { $text: { $search: "java coffee shop" } } )


// Returns all documents containing “coffee shop”
db.stores.find( { $text: { $search: "\"coffee shop\"" } } )


// Returns all stores containing “java” or “shop” but not “coffee”
db.stores.find( { $text: { $search: "java shop -coffee" } } )
```

Hogyan működik a Journaling a MongoDB-ben?

Mongod elsősorban a memóriában található írási műveleteket osztja meg megosztott nézetben. Azért hívják megosztottnak, mert memória leképezéssel rendelkezik a tényleges lemezen. Ebben a folyamatban egy írási művelet történik a mongodban, amely aztán változásokat hoz létre a privát nézetben. Az első blokk a memória, a második a "saját lemezem". Egy meghatározott intervallum után, amelyet "naplókötési intervallumnak" neveznek, a privát nézet ezeket a műveleteket a naplókönyvtárba írja (a lemezen található).

Amint megtörténik a napló kommitolás, mongod az adatokat megosztott nézetbe (shared view) tolja. A folyamat részeként a tényleges adatkönyvtárba kerül a megosztott nézetből (shared view) (mivel ez a folyamat a háttérben történik). Alapvető előnye, hogy 60 másodpercről 200 milliszekundumra csökkent a ciklusunk.

Olyan esetekben, amikor a megszakadás bármely időpontban bekövetkezik, vagy a flash lemez az utolsó 59 másodpercig nem érhető el, akkor amikor a mongod legközelebb elindul, akkor alapvetően visszajátssza az összes írási műveletnaplót és ír a tényleges adatkönyvtárba.



A MongoDB séma nélküli?

NoSQL adatbázisként a MongoDB sématlannak tekinthető, mert nem igényel olyan merev, előre definiált sémát, mint egy relációs adatbázis. Az adatbázis-kezelő rendszer (DBMS) részleges sémát hajt végre az adatok írásakor, kifejezetten felsorolva a gyűjteményeket és indexeket.

A MongoDB egy dokumentum alapú adatbázis, amely nem használja a táblák és oszlopok fogalmát, ahelyett, hogy a dokumentumok és a gyűjtemények fogalmát használja. A különböző modulokra vonatkozó összes referenciaadatot egyetlen gyűjteményként tároljuk. A MongoDB által használt BSON adatstruktúrán túl könnyen különböző adatsorok és mezők lehetnek különböző típusúak.

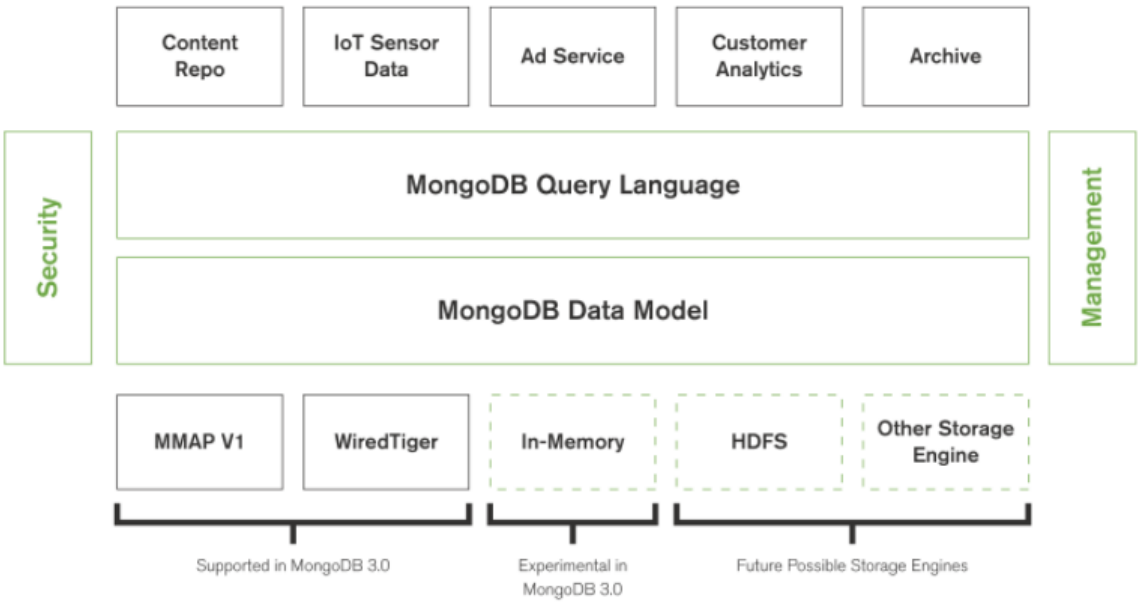
Ha sématlanságot mondunk, akkor valójában a dinamikusan beírt sémát értjük, szemben az RDBMS (SQL) adatbázisokban elérhető statikusan beírt sémákkal. A JSON egy teljesen séma nélküli adatstruktúra, szemben az XML-mel, amely lehetővé teszi az XSD megadását, ha szükséges.

Mi az a tároló motor (Storage Engine) a MongoDB-ben?

A tárolómotor az adatbázis összetevője, amely felelős az adatok tárolásának kezeléséért, mind a memóriában, mind a lemezen. A MongoDB több tárolómotort támogat, mivel a különböző motorok jobban teljesítenek bizonyos terhelések esetén.

Example: command to find storage engine

```
> db.serverStatus().storageEngine

// Output
{
    "name" : "wiredTiger",
    "supportsCommittedReads" : true,
    "oldestRequiredTimestampForCrashRecovery" : Timestamp(0, 0),
    "supportsPendingDrops" : true,
    "dropPendingIdents" : NumberLong(0),
    "supportsTwoPhaseIndexBuild" : true,
    "supportsSnapshotReadConcern" : true,
    "readOnly" : false,
    "persistent" : true,
    "backupCursorOpen" : false
}
```



A MongoDB főként 3 olyan tárolómotort támogat, amelyek teljesítménye bizonyos terheléseknek megfelelően eltér. A tároló motorok (Storage Engine):

- WiredTiger tároló motor
- Memóriatároló motor
- MMAPv1 tároló motor

1. WiredTiger tároló motor

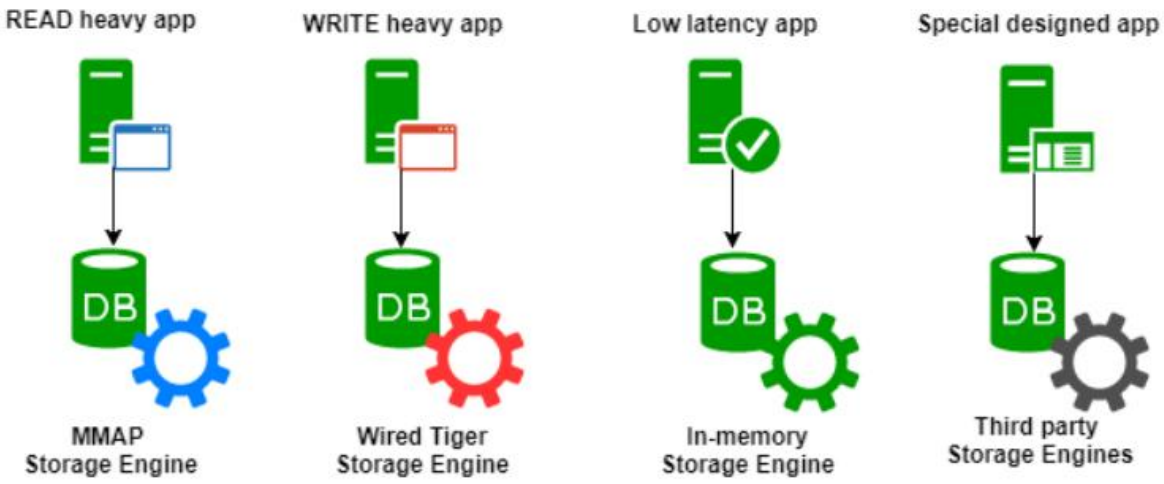
A WiredTiger az alapértelmezett tárolómotor, amely a MongoDB 3.2-ben indul. Jól alkalmazható a legtöbb terheléshez. A WiredTiger dokumentumszintű párhuzamossági modellt, ellenőrzőpontot (checkpoint) és tömörítést biztosít, többek között. A WiredTiger tároló motor B-fa alapú motorral és napló strukturált egyesítési fa alapú motorral rendelkezik.

2. Memóriatároló motor

Az In-Memory Storage Engine elérhető a MongoDB Enterprise rendszerben. Ahelyett, hogy dokumentumokat tárolna lemezen, a memóriában tartja őket a kiszámíthatóbb adatkésleltetések érdekében.

3. MMAPv1 tároló motor

Az MMAPv1 egy B-fa alapú rendszer, amely számos funkciót, például a tárolási interakciót és a memóriakezelést hajt az operációs rendszer segítségével. A neve onnan származik, hogy memóriába leképezett fájlokat használ az adatok eléréséhez. Ezt úgy teszi, hogy közvetlenül betölti és módosítja a virtuális memóriában lévő fájl tartalmát, az mmap () syscall módszertan segítségével.



Hogyan lehet nagy mennyiségű adatot sűríteni a Mongodb-ban?

compact-al

Átírja és töredezettségmentesíti a gyűjtemény összes adatait és indexeit. A WiredTiger adatbázisokban ez a parancs felesleges lemezterületet szabadít fel az operációs rendszer számára. Ez a parancs egy soros tömörítést hajt végre.

A MongoDB az alábbiak szerint tömöríti a fájlokat:

- a fájlok új helyre másolása
- a dokumentumok új sorrendjének készítése

- az eredeti fájlok cseréje az új fájlokkal

```
{ compact: <collection name> }
```

Lehetséges frissíteni a MongoDB mezőt egy másik mező értékével?

Az aggregációs függvénnyel frissíthető a MongoDB mező egy másik mező értékével.

```
db.collection.<update method>(
    {},
    [
        {"$set": {"name": { "$concat": ["$firstName", " ", "$lastName"]}}}
    ]
)
```

Hogyan ellenőrizhető, hogy egy mező tartalmaz-e részsztring-et?

A $ regex operátorral ellenőrizhető, hogy egy mező tartalmaz-e karakterláncot a MongoDB-ben.

```
db.users.findOne({"username" : {$regex : ".*some_string.*"}});
```

Hogyan lehet megtalálni a tömböt tartalmazó dokumentumot, amely egy adott értéket tartalmaz?

Feltöltés:

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

Ha azt szeretné lekérdezni, hogy a tömbmező legalább egy elemet tartalmaz-e a megadott értékkel, használja a {<field>: <value> } szűrőt, ahol az <value> az elem értéke.

```
db.inventory.find( { tags: "red" } )
```

Hogyan lehet megtalálni azokat a MongoDB rekordokat, ahol a tömbmező nem üres?

```
db.inventory.find({ pictures: { $exists: true, $ne: [] } })
```

Hogyan lehet megtalálni az utolsó N rekordot a find-al?

```
// Syntax
db.<CollectionName>.find().sort({$natural:-1}).limit(value)


// Example
db.employee.find().sort({$natural:-1}).limit(100)
```

Magyarázza a kapcsolatokat a mongodb-ben?

A MongoDB kapcsolatait annak meghatározására használják, hogy egy vagy több dokumentum hogyan kapcsolódik egymáshoz. A MongoDB-ben a kapcsolatok modellezhetők akár beágyazott (ebedded) módon, akár a referencia megközelítéssel. Ezek a kapcsolatok a következő formák lehetnek:

- One to One
- One to Many
- Many to Many

Példa: Vizsgáljuk meg a címek tárolásának esetét a felhasználók számára. Tehát egy felhasználónak több címe is lehet, így ez 1: N kapcsolattá válik.

User Collection

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Alex K",
   "contact": "987654321",
   "dob": "01-01-1990"
}
```

Address Collection

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

1. Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
> db.users.insert({
          {
                    "_id":ObjectId("52ffc33cd85242f436000001"),
                    "contact": "987654321",
                    "dob": "01-01-1991",
                    "name": "Alex K",
                    "address": [
                              {
                                        "building": "22 A, Indiana Apt",
                                        "pincode": 123456,
                                        "city": "Los Angeles",
                                        "state": "California"
                              },
                              {
                                        "building": "170 A, Acropolis Apt",
                                        "pincode": 456789,
                                        "city": "Chicago",
                                        "state": "Illinois"
                              }
                    ]
          }
})
```

Ez a megközelítés az összes kapcsolódó adatot egyetlen dokumentumban tárolja, ami megkönnyíti a visszakeresést és karbantartást. A teljes dokumentum lekérdezhető egyetlen lekérdezésben, például –

```
>db.users.findOne({"name":"Alex K"},{"address":1})
```

Hátránya, hogy ha a beágyazott dokumentum mérete túlságosan növekszik, az hatással lehet az olvasási / írási teljesítményre.

2. Modeling Referenced Relationships

Ez a megközelítés a normalizált kapcsolat tervezésének. Ebben a megközelítésben mind a felhasználói, mind a címdokumentumokat külön kezelik, de a felhasználói dokumentum tartalmazni fog egy mezőt, amely hivatkozni fog a címdokumentum id mezőjére.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Alex K",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

Ezzel a megközelítéssel két lekérdezésre lesz szükségünk: először a address_ids mezők beolvasásához a felhasználói dokumentumból, másodszor pedig a címek gyűjteményéből

```
>var result = db.users.findOne({"name":"Alex K"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

Mi a capped collection használata a MongoDB-ben?

A capped collection (korlátozott kollekciók) olyan rögzített méretű gyűjtemények, amelyek támogatják a nagy áteresztőképességű műveleteket, amelyek a beillesztési sorrend alapján beillesztik és visszakeresik a dokumentumokat. A korlátozott gyűjtemények a kör alakú pufferekhez hasonló módon működnek: amint egy gyűjtemény kitölti a kiosztott helyet, a gyűjtemény legrégebbi dokumentumainak felülírásával helyet biztosít az új dokumentumok számára.

A capped collection korlátozzák a dokumentumok frissítéseit, ha a frissítés nagyobb dokumentumméretet eredményez. Mivel a capped collection a dokumentumokat a lemeztárolási sorrendben tárolják, biztosítja, hogy a dokumentum mérete ne növelje a lemezen kiosztott méretet. A capped collection a naplóinformációk, a gyorsítótáradatok vagy más nagy mennyiségű adatok tárolására a legjobbak

Example:-

```
>db.createCollection( "log", { capped: true, size: 100000 } )
```

```
// specify a maximum number of documents for the collection
>db.createCollection("log", { capped: true, size: 5242880, max: 5000 } )


// check whether a collection is capped or not
>db.cappedLogCollection.isCapped()


// convert existing collection to capped
>db.runCommand({"convertToCapped": "posts", size: 10000})


// Querying Capped Collection
>db.cappedLogCollection.find().sort({$natural: -1})
```