

Be MEAN - MongoDB

Be MEAN

Published
with GitBook



Tabela de conteúdos

Introdução	0
Teoria	1
NoSQL	1.1
MongoDB	1.2
Instalação	2
Cliente	3
CRUD	4
insert	4.1
save	4.2
find	4.3
update	4.4
remove	4.5
drop	4.6
Paginação	4.7
Aggregation	5
Group	6
Replica	7
Sharding	8
GridFs	9

MongoDB

O MongoDB é um banco e dados NoSQL open-source e orientado a documentos JSON. Ele foi criado para ser escalado horizontalmente, conceito que veremos mais a frente. Nesse ebook será abordada a versão acima da 3.0.

Índice

- Teoria
 - NoSQL
 - MongoDB
- Instalação
- Cliente
- CRUD
 - insert
 - save
 - find
 - findOne
 - update
 - remove
 - drop
 - Paginação
- Aggregation
- Group
- Replica
- Sharding
- GridFs

Teoria

Primeiramente preciso explicar o que é um banco NoSQL, eu já enveredo por esse meio desde 2010 quando comecei a estudar MongoDB e CouchDb, acabando por optar pelo MongoDB, uma prova que faz tempo que escrevo e ensino sobre o assunto, e este artigo [NoSQL – você realmente sabe do que estamos falando?](#) de 28 de maio de 2010, publicado no iMasters, que por sinal foi a inauguração da área de NoSQL que eu gerenciava, sendo um dos primeiros artigos sobre MongoDB escritos no Brasil.

Desde lá para cá eu ja testei inúmeros bancos, porém o mais simples e adptável ao JavaScript, para mim, foi o MongoDB, por isso continuei trabalhando nele. Mas nunca deixando de estudar sempre as novidades, pior exemplo bancos de dados híbridos que são orientados tando a documento como grafo.

E eu sempre evangelizo que não devemos usar apenas **1** banco de dados NoSQL e sim alguns, pois cada um resolve um problema diferente, explico melhor esse conceito nesse artigo [NoSQL - Arquitetura híbrida para uma rede social](#).

NoSQL

[Conteúdo aqui.](#)

MongoDb

[Conteúdo aqui.](#)

Esse tipo de banco de dados já existe há um bom tempo, apenas não tinha essa nomenclatura, ela foi criada em 2009 em um evento sobre banco de dados não relacionais de código aberto, organizado por Johan Oskarsson da [Last.fm](#) e foi um funcionário do [Rackspace](#), Eric Evans, que cunhou o termo. Porém não significa que, não podemos usar *SQL*, até porque alguns bancos NoSQL usam um *tipo* de *SQL*, então como ele deveria ser chamado?

Bancos NoREL: Bancos Não Relacionais

Mas foi um golpe de marketing assim como o JavaScript também tem esse nome apenas para ter pego carona no Java, quando foi lançado.

Como sabemos os bancos relacionais são de propósito geral e qualquer coisa que é muito generalista, não consegue resolver um problema específico da melhor forma, para isso nós resolvemos com os banco NoSQL, bastante utilizados em projetos de [Business Intelligence](#), pois neles você tem essas 4 características:

- Velocidade de dados alta - lotes de dados que vêm muito rapidamente, possivelmente a partir de diferentes locais.
- Variedade de dados - armazenamento de dados que está estruturado, semi-estruturado e não estruturado.
- Volume de dados - dados que envolve muitos terabytes ou petabytes em tamanho.
- A complexidade dos dados - os dados que são armazenados e gerenciados em diferentes locais ou centros de dados.

Analogia

Eu faço uma analogia interessante sobre bancos de dados relacionais serem as cervejas de milho que encontramos aqui pelo Brasil e as cervejas artesanais serem os NoSQL. As cervejas de milho você acha em qualquer boteco e qualquer um bebe, agora as cervejas artesanais apenas poucos com bom gosto o fazem, assim é com os bancos de dados ehhehheh.

Vantagens

Normalmente as empresas utilizam os bancos NoSQL quando possuem um banco de dados em franco crescimento e precisam escalar horizontalmente com performance.

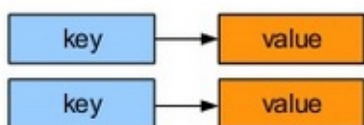
Tipos

Nesse universo de Banco de Dados NoSQL temos alguns grupos grandes com diversos bancos de dados e para as mais diversas finalidades, então vamos conhecer um pouco sobre eles para entender um pouco onde iremos nos enfiar. :p

Irei explicar um pouco dos seguintes grupos:

- Chave/Valor;
- Documento;
- Grafo;
- Coluna.

Chave/Valor



Esse tipo de banco de dados são utilizados em sua grande maioria para resolver o problema de cache, pois a estrutura que eles usam é bem simples, é a estrutura que temos em qualquer banco como **índice**.

Sabe quando você vai criar um índice na sua tabela de banco de dados para que ela tenha maior velocidade nas buscas?

Então é a mesma coisa aqui, a estrutura de uma *entidade* nesse tipo de banco segue a seguinte regra:

```
chave: valor
```

Então com uma chave específica você acessará diretamente essa entidade que pode ser apenas: um número, uma palavra, um array, um objeto, qualquer coisa. Porém você só consegue acessar essa entidade e seus **valores** a partir da **chave**, logo você não possui uma busca pelos valores internos. Vou dar um exemplo simples em JavaScript:

```
> var banco_chave_valor = [];  
undefined  
> var valor = {name: "Suissa", teacher: true};  
undefined  
> banco_chave_valor["minha-chave-unica-malandrinha"] = valor  
{ name: 'Suissa', teacher: true }  
> banco_chave_valor  
[ 'minha-chave-unica-malandrinha': { name: 'Suissa', teacher: true } ]
```

O que fiz foi criar um *array* vazio em `banco_chave_valor` e depois crio uma entidade chamada `valor` contendo o seguinte objeto: `{ name: 'Suissa', teacher: true }` e atribuo esse valor à minha chave `minha-chave-unica-malandrinha`.

Agora caso estivermos em um banco de Chave/Valor nós só podemos acessar os valores dessa entidade se buscarmos pela sua chave `minha-chave-unica-malandrinha` para depois acessarmos seus valores internamente:

```
> var busca_entidade_malandrinha = banco_chave_valor['minha-chave-unica-malandrinha']
undefined
> busca_entidade_malandrinha
{ name: 'Suissa', teacher: true }
> busca_entidade_malandrinha.name
'Suissa'
```

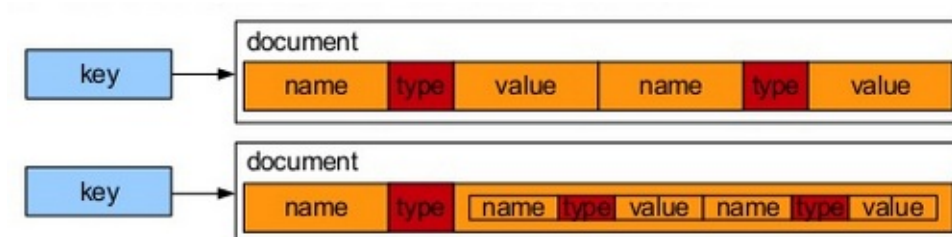
Bem simples esse conceito não? E como a maioria desses bancos funcionam operando apenas na RAM, para depois persistir, no caso de alguns, são largamente utilizados para **cache**, nesse meio contamos com nomes como:

- [Redis](#)
- [Riak](#)
- [LevelDb](#)

Para que usar?

Cache.

Documento



Um banco baseado em documento se assemelha bastante ao chave/valor pois também possui aquela estrutura:

```
chave: valor
```

Porém dessa vez também temos a busca pelos valores internos da nossa entidade persistida e para isso o MongoDB usa uma API bem simples e fácil de aprender que veremos a frente.

O tipo de documento em que o MongoDB é baseado é o JSON.

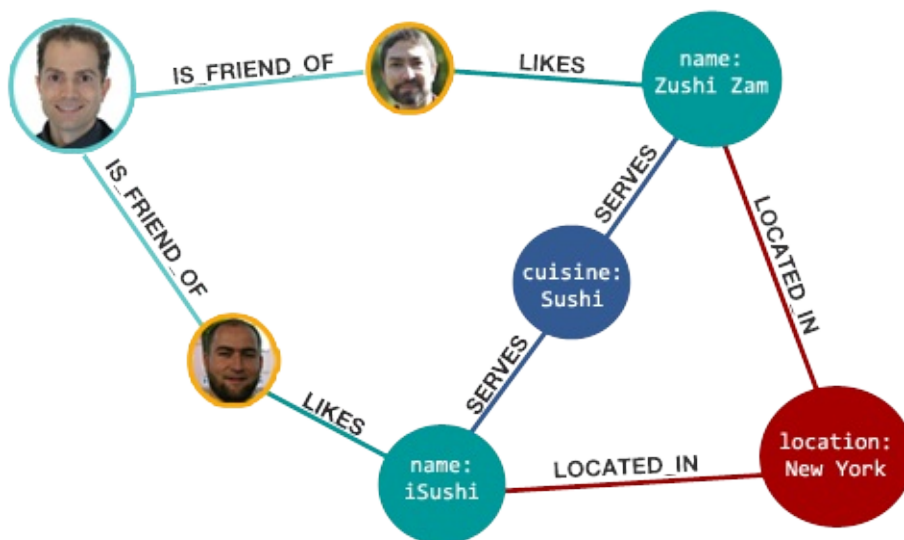


- [MongoDB](#)
- [CouchDB](#)

Para que usar?

Modelagem complexa e buscas dinâmicas.

Grafo



Um banco de dados orientado a [grafos](#) possui um base na teoria matemática dos grafos, mas que não é nada difícil, precisamos apenas pensar nas entidades como pontos(vértices) e que elas podem se relacionar com com outras entidades a partir de relações(arestas), como mostrado na imagem acima.

Esse tipo de banco é perfeito para redes sociais, caso você vá criar uma e não usar esse tipo de banco por favor **NUNCA DIGA QUE FOI MEU ALUNO**, LOL. Brincadeiras a parte, esse banco foi feito para isso, logo espero que o usem.

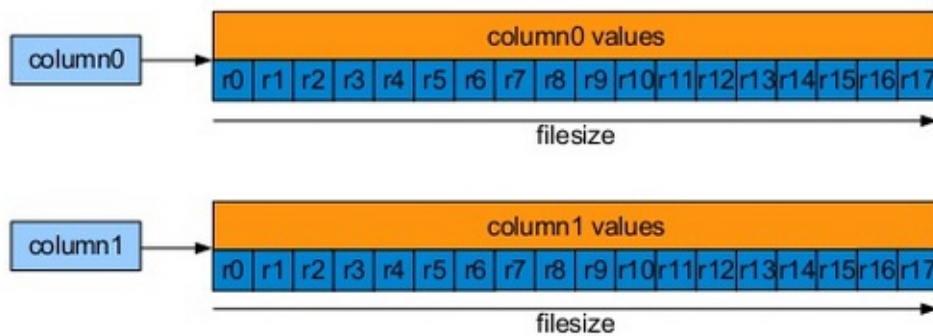
Caso você queira conhecer um pouco mais de um banco de grafos feito em Node.js, eu escrevi esse artigo há algum tempo atrás [Levelgraph - Um banco de dados de Grafos para Node.js - Parte 1](#).

- [Neo4J](#)
- [GraphDb](#)
- [Levelgraph](#)

Para que usar?

Dados inter-relacionados.

Coluna



Esse é o tipo que tive menos contato, confesso, logo não posso falar muito sobre além o do que li.

Esse modelo se tornou popular através do paper BigTable do Google, com o objetivo de montar um sistema de armazenamento de dados distribuído, projetado para ter um alto grau de escalabilidade e de volume de dados

A forma em que os dados são modelados lembra muito o relacional, porém mais complexo, é formado basicamente de 3 componentes:

- Keyspace: tem como função agrupar um conjunto de Famílias de Colunas. Semelhante a um banco de dados relacional.
- Família de Colunas: organiza as colunas. faz o uso de uma chave única, que traz flexibilidade ao modelo sem poluir as linhas com colunas nulas. Semelhante a uma tabela no modelo relacional.
- Coluna: que é uma tupla composta por nome, timestamp e valor, onde os dados são realmente armazenados.
- [Cassandra](#)
- [Hbase](#)

Para que usar?

Bl.

Híbridos

Depois dos 4 principais grupos ainda temos mais um que vem ganhando força, o dos bancos de dados híbridos, os 2 mais conhecidos são orientados por **documento e grafo**, o que os faz muito poderosos, pois a parte de relacionamento é o que peca no MongoDB e eu sempre aconselho a utilização de um banco de grafos para auxiliar nessa missão, agora você pode fazer as 2 coisas em 1 banco só, fiquem de olho pois são bem interessantes.

- [ArangoDb](#)

- [OrientedDB](#)

Para que usar?

Modelagem complexa e interconectada.

MongoDB (do inglês humongous, "gigantesco") é uma aplicação de código aberto, de alta performance, sem esquemas, orientado a documentos. Foi escrito na linguagem de programação C++.[1] Além de orientado a documentos, é formado por um conjunto de documentos JSON. Muitas aplicações podem, dessa forma, modelar informações de modo muito mais natural, pois os dados podem ser aninhados em hierarquias complexas e continuar a ser indexáveis e fáceis de buscar. O desenvolvimento de MongoDB começou em outubro de 2007 pela 10gen. A primeira versão pública foi lançada em fevereiro de 2009.[2]

fonte: <https://pt.wikipedia.org/wiki/MongoDB>

Acho que essa explicação da Wikipedia é bem clara e simples, porém também podemos adicionar que ele foi criado sendo pensado em escalar horizontalmente.

Escalabilidade Horizontal

Existem 2 tipos de escalabilidade, nesse caso, a horizontal e a vertical. Qual suas diferenças?

É bem simples, normalmente quando você utiliza bancos de dados relacionais sua performance do banco aumenta quando você aumenta o **poder do servidor** como adicionar mais memória RAM, HDs SSD, etc. Nesse caso você faz ele crescer para cima.

Na escalabilidade horizontal há um ganho na distribuição de dados, pois quanto mais dados forem armazenados, o número de servidores aumentarão (podendo ser de baixa ou alta performance) e há uma otimização no armazenamento dos dados, já que eles serão divididos entre todos os servidores, facilitando o gerenciamento e o processamento dos dados, assim, reduzindo o volume dos mesmos.

Schemaless

Capped Collection

Capped Collection são coleções de tamanho fixo que suportam as operações de alto rendimento que inserem e recuperam documentos com base em ordem de inserção.

Capped Collection trabalham de uma forma semelhante ao buffers circulares: uma vez que uma coleção preenche o seu espaço alocado, ele abre espaço para novos documentos, substituindo os documentos mais antigos na coleção.

- **Capped Collection tem os seguintes comportamentos:**
 - Capped Collection garanti a preservação da ordem de inserção. Como resultado, as consultas não precisam de um índice para devolver os documentos em ordem

de inserção. Sem essa sobrecarga de indexação, eles podem apoiar um maior rendimento de inserção.

- Capped Collection garante que a ordem de inserção é idêntica à ordem no disco (ordem natural) e faz isso através da proibição de atualizações que aumentam o tamanho do documento. Capped Collection só permite atualizações que se encaixam no tamanho do documento original, o que garante que o documento não altere a sua localização no disco.
- Capped Collection remove automaticamente os documentos mais antigos da coleção sem a necessidade de scripts ou operações de remoção explícitas.

fonte: <http://docs.mongodb.org/manual/core/capped-collections>

Memory-mapped files

O que são Memory-mapped files ?

Um memory-mapped file é um arquivo com dados, que o sistema operacional coloca em memória através da chamada do `mmap()`. `mmap()` então mapeia o arquivo para uma região da memória virtual. Memory-mapped files são a peça fundamental do mecanismo de armazenamento MMAPv1 no MongoDB. Ao usar memory-mapped files, o MongoDB consegue tratar os conteúdos dos arquivos como se eles estivessem em memória. Isso proporciona um método extremamente rápido e simples para acessar e manipular dados.

Como funcionam os memory-mapped files?

O MongoDB usa memory-mapped files para gerenciar e interagir com todos os dados.

O mapeamento de memória atribui arquivos para um bloco de memória com uma correlação direta byte a byte. O MongoDB mapeia para os arquivos para a memória assim acessados os documentos. Dados não acessados não são mapeados para a memória.

Uma vez mapeados, a relação entre arquivos e memória permite MongoDB para interagir com os dados no arquivo, como se fosse memória.

fonte: <http://docs.mongodb.org/manual/faq/storage/#mmapv1-storage-engine>

Auto-sharding

Replica

Cluster

GridFS

Geolocation

Modelagem

MongoDb University

Nossa querida [10gen](#) criou uma "Universidade" online para aprender MongoDB, a [MongoDB University](#).

Lá eles possuem cursos para:

- Node.js
- .Net
- Java
- DBAs

Então tem para todos os gostos, se você quiser se aprofundar **mais** no MongoDB eu aconselho a você ver essas aulas, se souber inglês.

Instalação

Instalar o MongoDB é mais fácil que mijar deitado. LOL

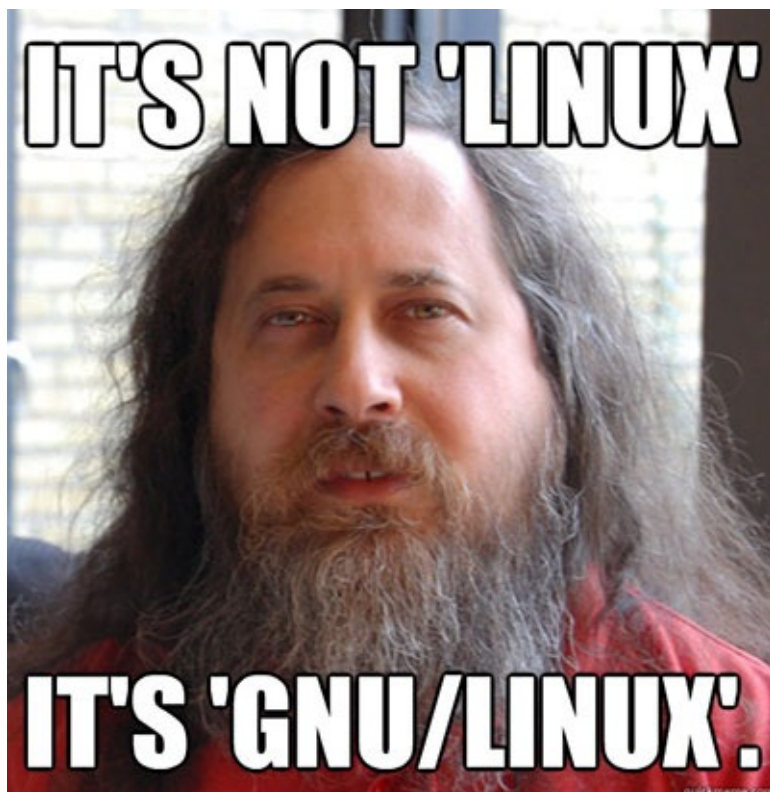


Primeiramente entre na página de download

<https://www.mongodb.org/downloads#production> e escolha lá embaixo o seu Sistema Operacional correto.

Depois basta descompactar e rodar.

Linux



Quem usa Linux do tipo Ubuntu da vida, como o Debian por exemplo, pode instalar via `apt-get` seguindo esses passos, caso o seu sistema seja 64 bits. Primeiro, para garantir a autenticidade e consistência dos pacotes do MongoDB:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

Depois precisa criar uma lista de arquivos para o MongoDB, no Ubuntu 12:

```
echo "deb http://repo.mongodb.org/apt/ubuntu precise/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

Ubuntu 14:

```
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

Ubuntu 15:

```
echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-org/3.0 main" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

Por fim, rodar o comando:

```
sudo apt-get update
sudo apt-get install -y mongodb-org
```

Se o teu OS for um Debian ou Ubuntu 32 bits, siga estes passos: [Install MongoDB on Ubuntu - Docs MongoDB](#)

openSUSE 64 bits

Adicionando o repositório:

```
sudo zypper addrepo --no-gpgcheck https://repo.mongodb.org/zypper/suse/11/mongodb-org/3.2
```

Instalando o mongodb:

```
sudo zypper -n install mongodb-org
```

Crie o diretório de dados, este diretório será usado apenas se não estiver rodando com usuário mongod:


```
sudo mkdir /data
sudo mkdir /data/db
sudo chmod 777 /data/db
```

Pode executar o mongod que verá o mongo rodando no seu terminal, control+C para sair.

```
mongod
```

O arquivo `/etc/mongod.conf` contém a configuração padrão do mongod. Também podemos rodar o mongod como serviço, neste caso o usuário padrão e o mongod e o diretório dos dados será em `/var/lib/mongo`, os logs ficarão em `/var/log/mongodb`

```
sudo service mongod start
```

Caso você use RedHat ou CentOS siga esses passos:

<http://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat/>

Mac



Quem usa Mac pode instalar via [brew](#) e para instalar o brew é bem fácil basta executar esse comando no seu terminal:

```
ruby -e "$(curl -fsSL https://macgithubusercontent.com/Homebrew/install/master/install)"
```

Depois basta executar o comando de `update` do `brew` :

```
brew update
```

Depois mandar ele instalar o `mongodb` :

```
brew install mongodb
```

Também tem um vídeo muito bom enviado por um aluno: [003 Installing MongoDB on a Mac](#).

Windows



Por incrível que pareça é bem simples no Windows, um aluno meu escreveu esse artigo que pode lhe ajudar <https://pablojuancruz.wordpress.com/2014/09/03/configurando-ambiente-mongodb-no-windows/>.

MongoDB Servidor

Caso você não tenha instalado ele com `apt-get` ou `brew` da vida você terá que executar ele diretamente da pasta onde ele foi descompactado, por isso de preferência descompacte em uma pasta chamada `mongodb` na sua raiz, sendo ela `/` ou `c:\`.

Caso você esteja usando Windows por favor use o `PowerShell`, vai no Executar e escreve `PowerShell`, ele roda comandos de **Linux** no Windows e é melhor que aquele lixo do `Console`.

Depois que entrar na pasta basta executar o binário `mongod` que é nosso **servidor**:

```
./mongod
```

Ou no `PowerShell`:

```
.\mongod
```

Caso ele de um erro falando sobre o `dbpath` é **muito simples** de resolver, basta criarmos uma pasta na sua raiz, `C:\`, com o nome `data` e dentro dela `db`, se usar Linux/Mac não esqueça de dar as permissões corretas, caso queira liberar geral basta um:

```
sudo chmod 777 -R /data
```

ps: por favor nunca faça isso em produção!

Acredito que depois ele não dará mais problema.

MongoDB Cliente

Depois de utilizarmos o MongoDB precisamos rodar seu cliente para que possamos integrar com ele via linha de comando, para isso basta executar o comando `mongo` no seu terminal:

```
mongo
```

Executando dessa forma ele irá se conectar em uma *database* chamada `test`, para que você execute o cliente diretamente em uma *database* específica, basta passar o nome dela logo após o comando:

```
mongo be-mean-instagram
```

Nesse caso já entramos com a *database* `be-mean-instagram` que será a base utilizada em nosso *workshop*.

Versão

Para você garantir que a versão 3 está instalada basta executar o seguinte comando:

```
mongod --version  
db version v3.0.6
```

E para o seu cliente:

```
mongo --version  
MongoDB shell version: 3.0.6
```

Agora estamos prontos para iniciar.

MongoDb Cliente

mongo-hacker

O [mongo-hacker](#) é uma extensão para seu terminal que adiciona algumas funcionalidades a mais, como por exemplo *syntax highlighter*, vou mostrar como é no meu terminal:



Instalação

Linux

Intalando o git. no ubuntu e Debian, funciona também nos derivados.

```
sudo apt-get install git -y
```

Intalando o git. no fedora 23

```
sudo dnf install git -y
```

Clonando o projeto mongo-hacker que pertence TylerBlock.

```
git clone https://github.com/TylerBrock/mongo-hacker
```

Algumas vezes no distro que está utilizando (Sistema Operacional) não vem instalado o gcc e g++ que são dependencias necessarias na instalação, vamos instalar.

No Ubuntu / Debian (e derivados).

```
sudo apt-get install gcc g++ -y
```

No Fedora 23.

```
sudo dnf install gcc g++ -y
```

Após a instalação do gcc e g++ vamos instalar o mongo-hacker.

```
cd mongo-hacker/  
sudo make install
```

Database

Para listarmos nossas *databases* precisamos apenas executar o seguinte comando no cliente do MongoDB:

```
show dbs  
local 0.0GB
```

E aparecerá a listagem das *databases* existentes.

Em versões anteriores a atual(3.2.1) a storage engine layer do mongoDB pré alocava `0.078GB`, o que dá algo em torno de 80MBs, ela fazia isso para melhorar a performance na hora da busca e sempre garantir um espaço sequencial para a persistência.

Após a compra da Wired Trigger(WT) o MongoDB começou a evoluir na sua Storage Engine Layer melhorando o uso de espaço em disco e a performance.

Sendo assim não se faz mais necessário a pré-alocação dos dados.

Agora vamos criar o nosso banco para iniciar o nosso Instagram, execute o seguinte comando:

```
use be-mean-instagram  
switched to db be-mean-instagram
```

Onde `be-mean-instagram` é o nome do nosso banco e ele está referenciado na variável `db`, então se quiser ver qual banco de dados estamos usando basta você executar `db` no cliente:

```
db  
be-mean-instagram
```

perceba que o banco `database-test` que possui já possui `0.078GB` de tamanho, porém não contém **nenhum** dado.

dropDatabase

Para apagarmos um banco de dados é bem simples, basta executarmos a função

```
dropDatabase()
```

 após termos definido nosso banco com `use nome_do_banco` :

```
use banco_a_remover
switched to db banco_a_remover

db.dropDatabase()

{
  "dropped": "banco_a_remover",
  "ok": 1
}
```

CUIDADO

Esse comando precisa de um *lock* de escrita **global** e irá bloquear outras operações até estar completa.

Collection

Import

Export

Replica

Sharding

GridFs

CRUD

Insert

Save

Find

FindOne

insert

Você deve ter notado que o database `worksop-be-mean` não foi criado né? Porque o MongoDB só irá realmente criar seu database quando você inserir um objeto em uma coleção. Então vamos fazer isso:

```
db.teste.insert({a: true})
```

Listamos novamente com `show dbs` e voiala!

Perceba que a sintaxe de um comando no MongoDB é:

```
database.coleção.função()
```

```
db.teste.insert()

// Inserindo diretamente via parametro
db.teste.insert({a: true})

// Inserindo via variável
var json = {b: 'TESTE'}
db.teste.insert(json)
```

Quando usamos o comando `use`, ele muda nosso database e o aponta para a variável `db` usada no inicio dos comandos, então ela sempre apontará para e database atual, como podemos ver executando apenas seu nome:

```
db
be-mean-instagram
```

Dica: instale o `mongo-hacker`, ver no github, manualmente.

```
db.teste.find()
{
  "_id": ObjectId("546142385b9f2b586cb31d06"),
  "a": true
}
{
  "_id": ObjectId("546142665b9f2b586cb31d07"),
  "b": "TESTE"
}
```

ObjectId

Você deve ter percebido esse campo após listarmos os objetos da nossa coleção

- [ObjectId](#)

Inserindo

Para inserir um objeto no MongoDB podemos criá-lo em uma variável e depois passar como parâmetro para a função `insert` ou `save` :

```
var pokemon = {'name':'Pikachu','description':'Rato elétrico bem fofinho','type': 'electr'}
db.pokemons.insert(pokemon)
Inserted 1 record(s) in 3ms
WriteResult({
  "nInserted": 1
})
```

Para inserir diversos registros de uma só vez podemos criar um array com nossos objetos como abaixo:

```
suissacorp(mongod-3.0.6) be-mean-instagram> var pokemons = [
{'name':'Bulbassau', 'description':'Chicote de trepadeira', 'type': 'grama', 'attack': 49

suissacorp(mongod-3.0.6) be-mean-instagram> pokemons
[
  {
    "name": "Bulbassau",
    "description": "Chicote de trepadeira",
    "type": "grama",
    "attack": 49,
    "height": 0.4
  },
  {
    "name": "Charmander",
    "description": "Esse é o cão chupando manga de fofinho",
    "type": "fogo",
    "attack": 52,
    "height": 0.6
  },
  {
    "name": "Squirtle",
    "description": "Ejeta água que passarinho não bebe",
    "type": "água",
    "attack": 48,
    "height": 0.5
  }
]

suissacorp(mongod-3.0.6) be-mean-instagram> db.pokemons.insert(pokemons)
Inserted 1 record(s) in 1ms
BulkWriteResult({
  "writeErrors": [ ],
  "writeConcernErrors": [ ],
  "nInserted": 3,
  "nUpserted": 0,
  "nMatched": 0,
  "nModified": 0,
  "nRemoved": 0,
  "upserted": [ ]
})
```

```
db.pokemons.find()
{
  "_id": ObjectId("564220f0613f89ac53a7b5d0"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 100,
  "height": 0.4
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d1"),
  "name": "Bulbassauro",
  "description": "Chicote de trepadeira",
  "type": "grama",
  "attack": 49,
  "height": 0.4
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d2"),
  "name": "Charmander",
  "description": "Esse é o cão chupando manga de fofinho",
  "type": "fogo",
  "attack": 52,
  "height": 0.6
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d3"),
  "name": "Squirtle",
  "description": "Ejeta água que passarinho não bebe",
  "type": "água",
  "attack": 48,
  "height": 0.5
}
Fetches 4 record(s) in 2ms
```

Dica: quando utilizar o comando `find` ou `findOne` e não tiver o mongo-hacker, utilize no final a função `pretty()` .

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

save

Nós também podemos inserir objetos utilizando o `save`, ele tanto insere como altera valores.

```
var pokemon = {'name':'Caterpie','description':'Larva lutadora','type': 'inseto', attack:  
  
suissacorp(mongod-3.0.6) be-mean-instagram> db.pokemons.save(pokemon)  
Inserted 1 record(s) in 1ms  
WriteResult({  
  "nInserted": 1  
})
```

Depois listamos para conferir:

```
db.pokemons.find()
{
  "_id": ObjectId("564220f0613f89ac53a7b5d0"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 100,
  "height": 0.4
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d1"),
  "name": "Bulbassauro",
  "description": "Chicote de trepadeira",
  "type": "grama",
  "attack": 49,
  "height": 0.4
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d2"),
  "name": "Charmander",
  "description": "Esse é o cão chupando manga de fofinho",
  "type": "fogo",
  "attack": 52,
  "height": 0.6
}
{
  "_id": ObjectId("56422345613f89ac53a7b5d3"),
  "name": "Squirtle",
  "description": "Ejeta água que passarinho não bebe",
  "type": "água",
  "attack": 48,
  "height": 0.5
}
{
  "_id": ObjectId("56422705613f89ac53a7b5d4"),
  "name": "Caterpie",
  "description": "Larva lutadora",
  "type": "inseto",
  "attack": 30,
  "height": 0.3
}
Fetched 5 record(s) in 40ms
```

Para alterarmos um valor com `save`, precisamos inicialmente buscar o objeto desejado com `findOne`, pois ele me retorna apenas o primeiro objeto achado. Caso eu usasse o `find`, mesmo retornando **um** objeto, ainda seria dentro de um *Array*.

Por isso usamos o `find` para listagem de registros e o `findOne` para consulta de registros.

Veja a diferença de retorno das duas funções:

```
var query = {name: 'Caterpie'}
suissacorp(mongod-2.4.8) be-mean> var p = db.pokemons.find(query)
suissacorp(mongod-3.0.6) be-mean-instagram> p
{
  "_id": ObjectId("56422705613f89ac53a7b5d4"),
  "name": "Caterpie",
  "description": "Larva lutadora",
  "type": "inseto",
  "attack": 30,
  "height": 0.3
}
suissacorp(mongod-3.0.6) be-mean-instagram> p.name
suissacorp(mongod-3.0.6) be-mean-instagram>
```

Não conseguimos acessar diretamente nosso objeto pois ele é retornado na forma de [cursor](#), que possui métodos especiais para acessar seus valores, [como visto aqui](#).

Então precisamos utilizar o `findOne` pois ele retorna um objeto comum.

```
var p = db.pokemons.findOne(query)
suissacorp(mongod-3.0.6) be-mean-instagram> p
{
  "_id": ObjectId("56422705613f89ac53a7b5d4"),
  "name": "Caterpie",
  "description": "Larva lutadora",
  "type": "inseto",
  "attack": 30,
  "height": 0.3
}
suissacorp(mongod-3.0.6) be-mean-instagram> p.name
Caterpie
suissacorp(mongod-3.0.6) be-mean-instagram> p.defense = 35
35
suissacorp(mongod-3.0.6) be-mean-instagram> p
{
  "_id": ObjectId("56422705613f89ac53a7b5d4"),
  "name": "Caterpie",
  "description": "Larva lutadora",
  "type": "inseto",
  "attack": 30,
  "height": 0.3,
  "defense": 35
}
suissacorp(mongod-3.0.6) be-mean-instagram> db.pokemons.save(p)
Updated 1 existing record(s) in 2ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```


find

Lembrando da aula anterior quando falei que a busca com `find` retorna um cursor onde você deve iterar nele para buscar seus dados, hoje veremos a diferença dele para o `findOne`.

Como havia dito, o Mongoose irá converter esse cursor para `Array` então sempre quando quisermos **LISTAR** algo iremos utilizar o `find`.

CUIDADO

Mesmo você buscando diretamente com o `_id`

_id

Esse `_id` que vocês devem ter visto nos registros inseridos nada mais é que um **UUID**.

Ele também é conhecido como `objectId` e ele é um tipo do BSON de 12-bytes, construído usando:

4-bytes: valor que representa os segundos desde a época Unix; 3-bytes: identificador de máquina; 2-bytes: ID do processo; 3-bytes: contador, começando com um valor aleatório. Sim essa porra é "**universalmente única**"!

O `_id` é nossa chave primária, olha aí relational-guys, é com ele que fazemos consultas específicas, por favor não esqueça disso!!!

query

Para facilitar nossa vida iremos criar um JSON para nossa *query*, para isso iremos criar um JSON da seguinte forma:

```
var query = {name: 'Pikachu'}
```

Isso significa que iremos pesquisar apenas os Pokemons com o `name` igual a `Pikachu`.

Esse nosso objeto de `query` tem a mesma funcionalidade do tão conhecido `SELECT` dos bancos relacionais.

Eu já escrevi sobre [isso em 2010 no iMasters - Como utilizar selects com MongoDB](#)

Claro que é bem defasado e escrito ainda com PHP ehhehehhehe.



fields

Se o nosso JSON de `query` é o `WHERE` do relacional, logo o JSON `fields` será o nosso `SELECT` onde o mesmo irá selecionar quais campos desejados na busca da `query`.

Para isso especificamos os campos desejados com `1` que significa `TRUE` ou os campos indesejados com `0` que significa `FALSE`.

```
suissacorp(mongod-3.0.6) be-mean-instagram> var query = {name: 'Pikachu'}
suissacorp(mongod-3.0.6) be-mean-instagram> var fields = {name: 1, description: 1}
suissacorp(mongod-3.0.6) be-mean-instagram> db.pokemons.find(query, fields)
{
  "_id": ObjectId("564220f0613f89ac53a7b5d0"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho"
}
Fetched 1 record(s) in 1ms
```

Operadores de Aritmética

< é \$lt - less than

`db.colecao.find({ "campo" : { $lt: value } });` Retorna documentos com valores menores que `value`.

<= ou \$lte - less than or equal

db.colecao.find({ "campo" : { \$lte: value } }); Retorna documentos com valores menores ou igual que value.

> ou \$gt - greater than

db.colecao.find({ "campo" : { \$gt: value } }); Retorna documentos com valores maiores que value.

>= ou \$gte - greater than or equal

db.colecao.find({ "campo" : { \$gte: value } }); Retorna documentos com valores maiores ou igual que value.

Operadores Lógicos

\$or

Retorna documentos caso a cláusula OU for verdadeira, ou seja, se **alguma das cláusulas forem verdadeiras**

Sintaxe

```
{ $or : [ { campo1 : valor } , { campo2 : valor } ] }
```

Uso

Vamos buscar os Pokemons que possuam **OU** o `{name: 'Pikachu'}` **OU** do tipo grama `{type: 'grama'}` .

\$nor

Retorna documentos caso a cláusula negação do OU for verdadeira, ou seja, retorna **apenas documentos que não satisfaçam as cláusulas.**

Sintaxe

```
{ $nor : [ { a : 1 } , { b : 2 } ] }
```

Uso

\$and

Retorna documentos caso a cláusula E for verdadeira, ou seja, somente se **todos as cláusulas forem verdadeiras**.

Sintaxe

```
{ $and: [ { a: 1 }, { a: { $gt: 5 } } ] }
```

Uso

Operadores "Existenciais"

\$exists

Sintaxe

```
db.colecao.find( { campo : { $exists : true } } );
```

Retorna o objeto caso o campo exista.

Uso

Operadores de Array

Antes de iniciar essa parte e já conhecendo sobre o `update`, pois foi dado anteriormente que esta parte, vamos deixar **todos** os pokemons com 1 ataque igual.

```
var query = {}
var mod = {$set: {moves: ['investida']}}
var options = {multi: true}
db.pokemons.update(query, mod, options)
```

Pronto agora todos pokemons possuem um campo do tipo *Array*, para finalizar vamos adicionar 1 ataque para: Charmander, Squirtle e Bulbassauo.

```

var query = {name: /pikachu/i}
var mod = {$push: {moves: 'choque do trovão'}}
db.pokemons.update(query, mod)

var query = {name: /squirtle/i}
var mod = {$push: {moves: 'hidro bomba'}}
db.pokemons.update(query, mod)

var query = {name: /charmander/i}
var mod = {$push: {moves: 'lança-chamas'}}
db.pokemons.update(query, mod)

var query = {name: /bulbassauro/i}
var mod = {$push: {moves: 'folha navalha'}}
db.pokemons.update(query, mod)

```

\$in

O operador `$in` retorna o(s) documento(s) que possui(em) algum dos valores passados no `[Array_de_valores]` .

Sintaxe

```
{ campo : { $in : [Array_de_valores] } }
```

Uso

Imaginemos que precisamos buscar todos Pokemons que possuam o ataque `choque do trovão` , pois o `investida` todos já possuem.

DICA: também pode usar **REGEX** aqui!

```

var query = {moves: {$in: [/choque do trovão/i]}}
db.pokemons.find(query)

```

Pronto com isso achamos apenas o Pikachu.

\$nin

Retorna documentos se nenhum dos valores for encontrado.

Sintaxe

```
{ campo : { $nin : [ [Array_de_valores] ] } }
```

Uso

Podemos trazer todos Pokemons que não possuem o ataque `investida` .

```
var query = {moves: {$nin: [/choque do trovão/i]}}
db.pokemons.find(query)
```

Nesse caso todos, excluindo o Pikachu.

\$all

Retorna documentos se **todos** os valores foram encontrados.

Sintaxe

```
{ campo : { $all : [ Array_de_valores ] } } )
```

Uso

Agora podemos buscar quais pokemons possuem os ataques `investida` e `hidro bomba` .

```
var query = {moves: {$all: ['hidro bomba', 'investida']}}
db.pokemons.find(query)
```

Dessa vez retornará apenas o Squirtle.

Operadores de Negação

\$ne - not Equal

Retorna documentos se o valor não for igual.

Sintaxe

```
{ campo : { $ne : valor } }
```

Uso

Podemos agora buscar **todos** os pokemons que não são do tipo `grama` .

```
var query = {type: {$ne: 'grama'}}
db.pokemons.find(query)
```

DICA: Não aceita **REGEX**!!!!

Error

Caso tente passar um valor como **REGEX** o MongoDB retornará esse erro:

```
Error: error: {
  "$err": "Can't canonicalize query: BadValue Can't have regex as arg to $ne.",
  "code": 17287
}
```

\$not

Retorna o objeto que não satisfaz a condição do campo, isso inclui documentos que não possuem o campo.

Sintaxe

```
{ campo : { $not : { $gt: 666 } } }
```

Uso

Com esse operador iremos buscar os pokemons que não tem um `attack` acima de 50.

```
var query = {attack: { $not : { $gt: 50 } } }
db.pokemons.find(query)
```

Percebeu que os documentos que não possuem o campo `attack` também retornaram e que `null` é menor que `50`.

E podemos usar **REGEX** para trazer todos os pokemons que não possuam o nome `Pikachu`.

```
var query = { name : { $not : /pikachu/i } }
db.pokemons.find(query)
```

Filtro com Data

Usando o conhecimento sobre `$and`, `$gte` e `$lt` podemos realizar busca de datas.

Exemplo: Gostaria de puxar o cadastro de um pokemon que foi criado em 17/01/2016 cuja o nome seja 'Bulbassauo'.

Inserindo dados na coleção.

```
var pokemons = {'name':'Bulbassau', 'description':'Chicote de trepadeira', 'type': 'grama'}

db.pokemons.insert(pokemons);

writeResult({ "nInserted" : 1 })
```

Depois da inserção buscaremos o campo `created_at`, que consta a data da criação do documento.

```
> var query = {$and: [{created_at: {$gte: new Date(2016, 0, 17), $lt: new Date(2016, 0, 18)}}]}

> db.pokemons.findOne(query).pretty();
{
  "_id" : ObjectId("569bb4441b2a879aee8fb0f1"),
  "name" : "Bulbassau",
  "description" : "Chicote de trepadeira",
  "type" : "grama",
  "attack" : 49,
  "height" : 0.4,
  "created_at" : ISODate("2016-01-17T15:33:14.893Z")
}
```


Update

Para alteramos um documento no MongoDB possuímos duas formas:

- save
- update.

Recordando que para utilizar o `save` eu preciso antes buscar o documento necessário antes de poder modificá-lo, com o `update` isso não será necessário.

A função `update` recebe 3 parâmetros:

- query
- modificação
- options

```
db.colecao.update(query, mod, options);
```

Para iniciarmos vamos criar um Pokemon novo:

```
var poke = {name: "Testemon", attack: 8000, defense: 8000, height: 2.1, description: "Pok  
db.pokemons.save(poke)
```

```
Inserted 1 record(s) in 48ms  
WriteResult({  
  "nInserted": 1  
})
```

Após inserido, vamos buscar esse documento para termos a certeza e já pegarmos seu `_id`, já já você entenderá o porquê.

```
var query = {name: /testemon/i}  
db.pokemons.find(query)  
{  
  "_id": ObjectId("5648970669bd5df270cc7e01"),  
  "name": "Testemon",  
  "attack": 8000,  
  "defense": 8000,  
  "height": 2.1,  
  "description": "Pokemon de teste"  
}  
Fetched 1 record(s) in 1ms
```

Depois de inserido vamos tentar fazer o nosso primeiro `update` , para isso iremos criar uma `query` para buscar nosso Pokemon e posteriormente, modificá-lo:

```
var query = {"_id": ObjectId("5648970669bd5df270cc7e01")}
var mod = {description: "Mudei aqui mermaoooo"}
db.pokemons.update(query, mod)
Updated 1 existing record(s) in 2ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Opa mas olha que simples, já alterou. Então vamos buscar novamente nosso documento pelo seu `_id` :

```
db.pokemons.find(query)
{
  "_id": ObjectId("5648970669bd5df270cc7e01"),
  "description": "Mudei aqui mermaoooo"
}
Fetched 1 record(s) in 1ms
```

PORRA SUISSA C FODEU O BAGUIO MANOOOOO C EH LOCO CACHORRERA??

Então, fiz de propósito hihihihhi.



Para evitarmos que o nosso documento seja sobrescrito pelo objeto de modificação nós deveremos utilizar os **operadores** de modificação.

\$set

O operador `$set` modifica um valor ou cria caso não exista.

```
{ $set : { campo : valor } }
db.pokemons.update( { name: 'Pikachu'}, { $set: { attack: 120
} } );
```

Então vamos reaproveitar nossa `query` que já possui nosso `_id` e vamos adicionar agora os campos faltantes e arrumar a `description` :

```
var mod = {$set: {name: 'Testemon', attack: 8000, defense: 8000, height: 2.1, description
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 1ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Então vamos buscar novamente nosso documento reaproveitando a `query` :

```
db.pokemons.find(query)
{
  "_id": ObjectId("5648970669bd5df270cc7e01"),
  "description": "Pokemon de teste",
  "name": "Testemon",
  "attack": 8000,
  "defense": 8000,
  "height": 2.1
}
Fetched 1 record(s) in 1ms
```

Perceba que além dele modificar o valor já existente de `description` ele também criou os campos faltantes.

\$unset

Bom se temos um operador para modificar e criar campos novos, obviamente temos um operador para remover os campos, que é o caso do `$unset` .

A sintaxe desse operador é a seguinte:

```
{ $unset : { campo : 1} }
```

Então vamos eliminar um campo do nosso Testemon :

```
var mod = {$unset: {height: 1}}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 3ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Bem simples a alteração de documentos no MongoDB não?

\$inc

O operador `$inc` incrementa um valor no campo com a quantidade desejada. Caso o campo não exista, ele irá criar o campo e setar o valor. Para decrementar, basta passar um valor negativo.

```
{ $inc : { campo : valor } }
```

Então vamos utilizar o nosso pokemon de teste modificado anteriormente para incrementar seu *attack*.

```
var mod = {$inc: { attack: 1 }}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 2ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Bem simples né? E podemos passar o valor que quisermos, não apenas incrementar de 1 em 1.

Por exemplo, quando algum Pokemon for evoluir ele ganhará 100 de attack, então para criar esse cenário nós fazemos:

```
var mod = {$inc: { attack: 100 }}

db.pokemons.update(query, mod)
```

E para decrementar o valor basta que passemos um valor negativo para o operador `$inc`.

Operadores de Arrays

Para iniciarmos a alteração em arrays vamos modificar o **Pikachu** para adicionar ao seu documento um *Array* de movimentos/ataques.

```
var query = {name: /pikachu/i}
var mod = {$set: { moves: ['investida'] }}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 7ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Para conferirmos nossa modificação vamos fazer a busca pelo Pikachu.

```
db.pokemons.find({name: /pikachu/i})
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "investida"
  ]
}
Fetched 1 record(s) in 1ms
```

Pronto agora temos um *Array* para nossos ataques.

\$push

O operador `$push` adiciona um valor ao campo, caso o **campo seja um *Array* existente**. Caso **não exista irá criar o campo novo, do tipo *Array* com o valor passado** no `$push`. Caso o **campo exista e não for um *Array***, irá retornar um erro.

Sintaxe

```
{ $push : { campo : valor } }
```

Uso

Então vamos adicionar o **Choque do Trovão** ao Pikachu:

```
var mod = {$push: {moves: 'choque do trovão'}}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 2ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Após a modificação vamos buscar o Pikachu e ver se alteramos corretamente:

```
db.pokemons.find(query)
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "investida",
    "choque do trovão"
  ]
}
Fetched 1 record(s) in 0ms
```

Erro

```
The field 'type' must be an array but is of type String in document {_id: ObjectId('56422')}
WriteResult({
  "nMatched": 0,
  "nUpserted": 0,
  "nModified": 0,
  "writeError": {
    "code": 16837,
    "errmsg": "The field 'type' must be an array but is of type String in document {_id: ObjectId('56422')}"
  }
})
```

\$pushAll

DEPRECIADO (Usar [\\$each](#))

O operador `$pushAll` adiciona cada valor do `[Array_de_valores]`, caso o **campo seja um Array existente**. Caso **não exista** irá criar o campo novo, do tipo **Array** com o valor passado no `$pushAll`. Caso o **campo exista e não for um Array**, irá retornar um erro.

Sintaxe

```
{ $pushAll : { campo : valor } }
```

Uso

Agora vamos adicionar 3 ataques novos ao Pikachu, para isso criamos um *Array* para seus valores e logo após passamos ele para o `$pushAll`:

```
var attacks = ['choque elétrico', 'ataque rápido', 'bola elétrica']
var mod = {$pushAll: {moves: attacks}}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 24ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Vamos conferir a modificação.

```
db.pokemons.find(query)
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "investida",
    "choque do trovão",
    "choque elétrico",
    "ataque rápido",
    "bola elétrica"
  ]
}
```

\$pull

O operador `$pull` retira um valor do campo, caso o **campo seja um Array existente**.

Caso **não exista** ele não fará nada. Caso o **campo exista e não for um Array**, irá retornar um erro.

Sintaxe

```
{ $pull : { campo : valor } }
```

Uso

Dessa vez iremos retirar um ataque do Pikachu.

```
var mod = {$pull: {moves: 'bola elétrica'}}
db.pokemons.update(query, { $pull: { moves: 'bola elétrica' } } )

Updated 1 existing record(s) in 17ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Consultando o Pikachu conferimos que o ataque `bola elétrica` foi removido.


```
db.pokemons.find(query)
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "investida",
    "choque do trovão",
    "choque elétrico",
    "ataque rápido"
  ]
}
```

\$pullAll

O operador `$pullAll` retira cada valor do `[Array_de_valores]`, caso o **campo seja um Array existente**. Caso **não exista** ele não fará nada. Caso o **campo exista e não for um Array**, irá retornar um erro.

Sintaxe

```
{ $pullAll : { campo : valor } }
```

Uso

Vamos remover 2 ataques de uma só vez: Choque Elétrico e Choque do Trovão.

```
var attacks = ['choque elétrico', 'bola elétrica']
var mod = {$pullAll: {moves: attacks}}
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 24ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

\$addToSet

O operador `$addToSet` adiciona um valor ao campo, caso o **campo seja um Array existente**. Caso o valor já esteja presente não vai adicionar o valor, o operador garante que o valor não vai ser adicionado caso ele já exista no campo. Caso o campo exista e não for um Array, irá retornar um erro.

Uso

Vamos deixar o Pikachu com `choque do trovão`. Agora vamos adicionar o ataque `choque elétrico` usando o `$addToSet`.

```
var mod = { $addToSet : { moves : "choque elétrico" } }
db.pokemons.update(query, mod)
Updated 1 existing record(s) in 1ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Vamos ver como ficou:

```
db.pokemons.find(query)
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "electric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "choque do trovão",
    "choque elétrico"
  ]
}
```

Igual ao `$push` né? Agora vamos tentar adicionar esse ataque novamente.

```
db.pokemons.update(query, mod)
Updated 1 existing record(s) in 1ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 0
})
```

Perceba como `nModified` está 0. Ele não alterou o array `moves`. Mas e se fosse o `$push` ?

```
var mod = { $push : { moves : "choque elétrico" } }
db.pokemons.update(query, mod)
Updated 1 existing record(s) in 1ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

O Pikachu vai ficar com duplicidade em `choque elétrico` e fica assim:

```
db.pokemons.find(query)
{
  "_id": ObjectId("56832c197ecdbeff48ee7b5d"),
  "name": "Pikachu",
  "description": "Rato elétrico.",
  "type": "Eletric",
  "attack": 30,
  "defense": 20,
  "height": 0.4,
  "moves": [
    "choque do trovão",
    "choque elétrico",
    "choque elétrico"
  ]
}
```

\$each

O modificador `$each` pode ser usado com `$addToSet` e com o operador `$push`. Como o operador `$pushAll` foi depreciado, agora utiliza-se `$each` para adicionar múltiplos valores ao campo. E com o `$addToSet` adicionar o múltiplos valores que não existem no campo.

Sintaxe

```
{ $push : { campo : {$each: [Array_de_valores] } } }
```

```
{ $addToSet: {campo : {$each: [Array_de_valores] } } }
```

Uso

Vamos adicionar 3 novos ataques ao Pikachu, para isso criamos um *Array* para seus valores e logo após passamos ele para o `$push`:

```
var attacks = ['choque elétrico', 'ataque rápido', 'bola elétrica']
var mod = { $push : { moves : { $each: attacks } } }
db.pokemons.update(query, mod)

Updated 1 existing record(s) in 29ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Vamos conferir a modificação.

```
db.pokemons.find(query)
{
  "_id": ObjectId("56422c36613f89ac53a7b5d5"),
  "name": "Pikachu",
  "description": "Rato elétrico bem fofinho",
  "type": "eletric",
  "attack": 55,
  "height": 0.4,
  "moves": [
    "investida",
    "choque do trovão",
    "choque elétrico",
    "ataque rápido",
    "bola elétrica"
  ]
}
```

Vamos remover os ataques de `moves` deixando apenas `investida` . E usar o operador `$addToSet` com `$each` para adicionar 4 ataques.

```
var attacks = ['investida', 'choque elétrico', 'ataque rápido', 'bola elétrica']
var mod = { $addToSet : { moves: { $each: attacks } } }
```

Olha como ficou o Pikachu. Não adicionou o `investida` .

```
db.pokemons.find(query)
{
  "_id": ObjectId("56832c197ecdbeff48ee7b5d"),
  "name": "Pikachu",
  "description": "Rato elétrico.",
  "type": "Eletric",
  "attack": 30,
  "defense": 20,
  "height": 0.4,
  "moves": [
    "investida",
    "choque elétrico",
    "ataque rápido",
    "bola elétrica"
  ]
}
```

options

Para que eles serve?

O objeto `options` servirá para configurarmos alguns valores diferentes do padrão para o `update`.

Sintaxe

```
{
  upsert: boolean,
  multi: boolean,
  writeConcern: document
}
```

upsert

O parâmetro `upsert` serve para caso o documento não seja encontrado pela `query` ele insira o objeto que está sendo passado como modificação.

Ele por padrão é `FALSE`.

Imagine que precisamos ativar, para ler suas informações, os Pokemons na nossa pokeagenda.

```
var query = {name: /squirtle/i}
var mod = {$set: {active: true}}
var options = {upsert: true}
db.pokemons.update(query, mod, options)
```

```
Updated 1 existing record(s) in 2ms
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
```

Então perceba que se o Pokemon existir ele só fará a alteração, agora vamos ver com um Pokemon que não exista na pokeagenda.

```
var query = {name: /PokemonInexistente/i}
var mod = {$set: {active: true}}
var options = {upsert: true}
db.pokemons.update(query, mod, options)

Updated 1 new record(s) in 3ms
WriteResult({
  "nMatched": 0,
  "nUpserted": 1,
  "nModified": 0,
  "_id": ObjectId("564a94aa3888e5da82899ccc")
})
```

Agora como percebemos no `WriteResult` ele não achou nenhum `"nMatched": 0` e inseriu 1 `"nUpserted": 1`. Retornando o `_id` do documento inserido.

```
db.pokemons.find(query)
{
  "_id": ObjectId("56422345613f89ac53a7b5d3"),
  "name": "Squirtle",
  "description": "Ejeta água que passarinho não bebe",
  "type": "água",
  "attack": 48,
  "height": 0.5,
  "active": true
}
```

\$setOnInsert

Com esse operador você pode definir valores que serão adicionados apenas se ocorrer um **upsert**, ou seja, se o objeto for inserido pois não foi achado pela **query**.

Vamos pegar um cenário onde buscaremos um pokemon em nossa pokeagenda, porém o mesmo não se encontra nos registros, então inserimos ele com valores padrões.

```
var query = {name: /NaoExisteMon/i}
var mod = {
  $set: {active: true},
  $setOnInsert: {name: "NaoExisteMon", attack: null, defense: null, height: null, descrip
}
var options = {upsert: true}
db.pokemons.update(query, mod, options)

Updated 1 new record(s) in 90ms
WriteResult({
  "nMatched": 0,
  "nUpserted": 1,
  "nModified": 0,
  "_id": ObjectId("564a89f33888e5da82899ccb")
})

db.pokemons.find(query)
{
  "_id": ObjectId("564a89f33888e5da82899ccb"),
  "active": true,
  "name": "NaoExisteMon",
  "attack": null,
  "defense": null,
  "height": null,
  "description": "Sem maiores informações"
}
```

multi

Quem nunca de um UPDATE SEM WHERE na vida que atire a primeira pedra ehhehehehe.



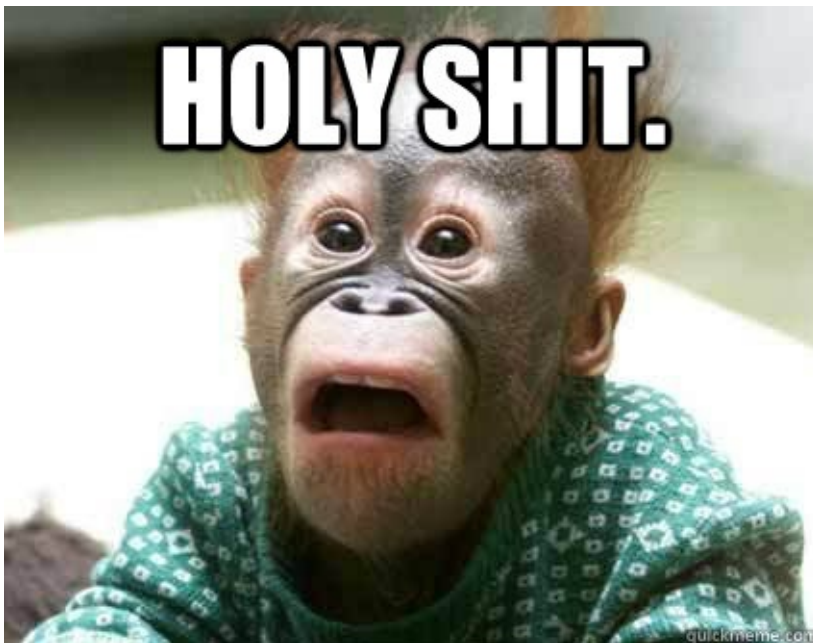
Não, não é o [canal Update Sem Where](#), é dar um **UPDATE** na sua tabela sem ter passado um **WHERE** na sua SQL.

Ué mas por que isso é ruim?

Se você se perguntou isso nunca deve ter trabalho com bancos de dados relacionais.

Pois quando você não passa a cláusula do **WHERE** o banco entende que você quer atualizar **TODOS** os registros.

Então imagine que você só ia atualizar o email de um usuário, não passando o **WHERE** você vai atualizar **TODOS OS EMAILS DE TODOS OS USUÁRIOS** para aquele email específico.



O MongoDB não deixa você fazer essa cagada, pois ele por padrão só deixa você alterar um documento por vez, caso você realmente deseje alterar **vários** de uma só vez, terá que passar esse parâmetro como `true`.

Vamos adicionar o campo `active: false` para todos os Pokemons.

```
var query = {}
var mod = {$set: {active: false}}
var options = {multi: true}
db.pokemons.update(query, mod, options)
```

```
Updated 8 existing record(s) in 3ms
WriteResult({
  "nMatched": 8,
  "nUpserted": 0,
  "nModified": 8
})
```

writeConcern

O `writeConcern` descreve a garantia de que MongoDB fornece ao relatar o sucesso de uma operação de escrita.

A força dos *write concerns* determinam o nível de garantia. Quando inserções, atualizações e exclusões têm um *write concern* fraco, operações de escrita retornam rapidamente.

Em alguns casos de falha, as operações de gravação emitidas com *write concerns* fracos podem não persistir.

Com os *write concerns* mais fortes, os clientes esperam após o envio de uma operação de escrita para o MongoDB confirmar as operações de escrita.

Caso queira saber mais como criar o documento a ser passado nessa opção [leia mais aqui](#).

remove

Para apagarmos os dados dessa coleção usaremos o `remove` .

O `remove` apenas apaga os dados, porém a coleção continua existindo, como podemos ver abaixo:

```
var query = {name: \squirtle\i}
db.pokemons.remove(query)
```

drop

A função `drop` irá apagar completamente a coleção eliminando ela do nosso database, como visto abaixo:

```
suissacorp(mongod-2.4.8) be-mean> show collections
system.indexes
teste

suissacorp(mongod-2.4.8) be-mean> db.teste.drop()
true

suissacorp(mongod-2.4.8) be-mean> show collections
system.indexes
```

pagination

A paginação no MongoDB é conseguida através do uso de duas outras funções: `.limit()` e `.skip()`.

limit

O método `limit`, como o próprio nome indica, **limita** o retorno de documentos. Recebe um inteiro como argumento, e devolve essa quantidade de documentos de uma coleção.

skip

O método `skip`, **pula** um número `n` de documentos, ou seja, retorna apenas os próximos, pulando os `n` primeiros. Também recebe um inteiro como argumento.

Utilizando os dois métodos acima juntos, conseguimos paginar, ou seja, retornar uma quantidade limitada de documentos a cada interação. Para paginar de 10 em 10, utilizamos assim:

```
.limit(10).skip(0 * 10); //primeira página
.limit(10).skip(1 * 10); //segunda página
.limit(10).skip(2 * 10); //terceira página
```

E para paginar de 5 em 5:

```
.limit(5).skip(0 * 5); //primeira página
.limit(5).skip(1 * 5); //segunda página
.limit(5).skip(2 * 5); //terceira página
```

Pois se fizermos as contas, ao pular 0 5 documentos, *não estamos pulando ninguém, então retornarmos os primeiros documentos*, ao pular 1 5, pulamos, os 5 primeiros, ou seja, pulamos a quantidade que já foi vista na primeira página, e ao pularmos 2 * 5, pulamos 10 documentos, que são aqueles da primeira e segunda página, trazendo assim apenas os da terceira página, e assim por diante.

Aggregation

No MongoDB podemos utilizar o [Aggregation Framework](#) para agruparmos valores para alguma finalidade, por exemplo fazer uma query que retorne a média salarial dos seus funcionários.

Um agrupamento pode ser feito com um [operador aritmético](#):

- Valor absoluto `$abs`
- Adição `$add`
- Arredondamento "para baixo" `$ceil`
- Arredondamento "para cima" `$floor`
- Truncar para inteiro `$trunc` sem arredondar
- dentre outros

Com um [acumulador](#):

- Soma `$sum`
- Média `$avg`
- Mínimo `$min`
- Máximo `$max`
- Desvio padrão `$stdDevPop`
- dentre outros

Com [strings](#)

- Concatenação `$concat`
- Retornar parte de uma string `$substr`
- Converter para minúsculas `$toLowerCase`
- Converter para maiúsculas `$toUpperCase`
- Comparar `$strcasecmp`

Com [arrays](#):

- Junta arrays e retorna um novo com todos os elementos `$concatArrays`
- Retornar o número de elementos de um array `$size`
- Reparte um array, retirando um pedaço dele `$slice`
- dentre outros

Com [datas](#):

- Dia do ano de uma data (de 1 a 366) `$dayOfYear`
- Número de milissegundos (de 0 a 999) `$millisecond`

- dentre outros

Vamos dar uma olhada na sintaxe geral do método **aggregate**:

pipeline

```
db.collection.aggregate(pipeline, options)
```

Sendo estas abaixo todas as [instruções do pipeline](#):

```
[
  { $project: <RETORNA UM DOCUMENTO ADICIONANDO OU REMOVENDO CAMPOS PARA A STREAM DO PIPEL
  { $match: <CONDIÇÃO DE FILTRO> },
  { $redact: <RETORNA O DOCUMENTO PARA A STREAM DO PIPELINE> },
  { $limit: <LIMITA A QUANTIDADE DE DOCUMENTOS A SEREM CONSIDERADOS> },
  { $skip: <PULA OS PRIMEIROS n DOCUMENTOS> },
  { $unwind <DESCONTOI UM ARRAY EM UM DOCUMENTO> },
  { $group: { _id: <KEY AGRUPADORA>, <OUTROS CAMPOS A SEREM AGRUPADOS> },
  { $sample: <AMOSTRA ALEATÓRIA> },
  { $sort: <ORDENAÇÃO A SER CONSIDERADA> },
  { $geoNear: <BUSCA UTILIZANDO LATITUDE E LONGITUDE> },
  { $lookup: <REALIZA UM 'left outer join' COM OUTRA COLEÇÃO DO MESMO BANCO DE DADOS> },
  { $out: <ESCREVE O RESULTADO EM UMA COLLECTION> },
  { $indexStats: <INDICA SE ALGUM ÍNDICE FOI UTILIZADO> }
]
```

Como o próprio nome `pipeline` indica, a **order** das declarações importa e modifica o resultado do `aggregate`. Então, se você quer por exemplo, ordenar o retorno, deixe o parâmetro `sort` por último, e não antes do `group`, entendeu?

Não vamos utilizar todas as opções disponíveis de uma só vez, e podemos fazer qualquer combinação delas, enquanto estamos construindo a nossa agregação, para atingirmos o resultado esperado.

options

E os `options` (argumento opcional que não precisa ser enviado):

```
{
  explain: boolean,
  allowDiskUse: boolean,
  cursor: boolean,
  bypassDocumentValidation: boolean,
  readConcern: boolean
}
```

Basicamente, agregações mais simples podem ser feitas utilizando o método `.group()`, com uma sintaxe bem diferente, e um pouco mais manualmente, já que no group, recorreremos a estratégia de map/reduce, em vez de termos operadores prontos que fazem o trabalho de média por exemplo para nós.

A agregação *pipeline* consiste em etapas, ou seja, cada estágio transforma os documentos a medida que passa através do pipeline.

A melhor definição de **Pipeline** é uma unidade de processamento de dados.

exemplo: 

Temos a seguinte leitura na imagem acima:

Obtemos os documentos da nossa coleção e em seguida temos as fases `stages` de cada um dos quais realiza a operação de cada entrada. Para cada fase temos uma modificação na nossa saída dos documentos. É **importante planejar** cada fase para que possamos realmente obter os documentos desejados.

Explicarei melhor com exemplos, logo em seguida e para entendermos sobre o que podemos ter em cada **stages** e facilitar nosso dia a dia.

Como estou fazendo o curso [Be-mean](#), usarei o banco **be-mean** e a coleção **pokemons** para prosseguir com os exemplos. Então para uma melhor aprendizagem sugiro seguir esses passos abaixo:

- Importar o arquivo [json aqui](#) para o banco be-mean. Puts, mas eu não sei como vou fazer para importar esses arquivos para o meu mongodb. Segue esse [vídeo](#) em 2:03 minutos de video o professor [William Bruno](#) pode te ajudar, é lógico, se você estiver com o mongodb instalado. Caso ainda não instalou siga essa [documentação](#) para instalar em seu S.O.

Banco Relacional - ordem lógica de execução

Como estou vindo do banco relacional, primeiramente vale lembrar de como o banco de dados relacional trabalha em relação a ordem lógica de execução.

Mas o que tem a ver a ordem lógica de execução do banco relacional com `aggregation` do `mongodb`?

Muita calma nessa hora, o que eu quero mostrar é a ordem da estrutura no `mongodb`, onde dependendo do posicionamento dos operadores `$limit`, `$skip` e `$sort` o resultado pode não ser o esperado.

Ordem lógica de execução

5: SELECT	< <u>select list</u> >
1: FROM	< <u>table source</u> >
2: WHERE	< <u>search condition</u> >
3: GROUP BY	< <u>group by list</u> >
4: HAVING	< <u>search condition</u> >
6: ORDER BY	< <u>order by list</u> >

Na imagem acima, temos a ordem lógica de execução em um banco de dados SQL SERVER. Vale lembrar que esse é o modelo padrão da estrutura sql que utilizo normalmente no relacional. Temos a mesma estrutura em outros bancos como MySql, Postgres, DB2 e etc.

Aggregation Operators

OU operadores de agregação

Iremos conhecer os operadores que podem fazer parte do método `aggregate`.

\$match

Filtra os documentos que correspondem à condição especificada, para seguir no `stages` do pipeline seguinte. O `$match` nada mais é que o método `find()`.

exemplo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}}
])
```

Estamos utilizando o método **aggregate** e realizando um `find` com o operador `$match`, onde irá retornar da nossa coleção pokemons, todos pokemons onde o `speed` é maior ou igual 40.

\$project

Se quisermos informar os campos que queremos na nossa saída dos dados `output`. Vale ressaltar que estamos adicionando mais uma etapa na nossa `stages`.

exemplo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$project: {
    _id: 0,
    name: 1,
    speed: 1
  }}
])
```

Nesse caso o `_id` não será apresentado em nossa saída, apenas os nomes dos pokemons e `speed` utilizando o método de agregação.

\$limit

Limita o número de documentos passados para a próxima etapa no pipeline.

exemplo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$limit: 5},
  {$project: {
    _id: 0,
    name: 1
  }}
])
```

Obtemos a quantidade de 5 documentos no processo de cada `stages` no pipeline. Apresentando na saída apenas os nomes dos pokemons.

Podemos perceber que podemos trocar a ordem de cada `stages` , como o exemplo abaixo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$project: {
    _id: 0,
    name: 1
  }},
  {$limit: 5}
])
```

Temos o mesmos documentos do exemplo anterior. Mas cuidado dependendo dos operadores utilizados os resultados não serão os mesmos.

\$sort

Ordena os documentos de acordo com a classificação de entrada. onde:

- 1 para especificar ordem crescente.
- -1 para especificar ordem decrescente.

exemplo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$sort: {name: 1, speed: -1}},
  {$limit: 5},
  {$project: {
    _id: 0,
    name: 1
  }}
])
```

Nesse caso estamos ordenando os documentos, onde `name` em ordem crescente e `speed` em ordem decrescente.

No exemplo abaixo iremos trocar o **\$sort** no lugar do **\$limit**. Podemos perceber que os documentos não são os mesmos.

ATENÇÃO!!!, nem sempre os valores que queremos é conforme a ordem que aplicamos no método **aggregate**. Fiquem ligados na ordem de cada `stages` .

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$limit: 5},
  {$sort: {name: 1, speed: -1}},
  {$project: {
    _id: 0,
    name: 1
  }}
])
```

\$skip

Ignora o número especificado de documentos que passam para o `stages` e passa os documentos restantes para a próxima fase do `pipeline`.

exemplo:

```
db.pokemons.aggregate([
  {$match: {speed: {$gte: 40}}},
  {$sort: {name: 1, speed: -1}},
  {$skip: 5},
  {$limit: 5},
  {$project: {
    _id: 0,
    name: 1
  }}
])
```

Adicionamos mais um operador em nossa `stages` para ignorar o número de documentos especificados no método `aggregate`. Nesse caso o `$skip` ignorou os 5 primeiros documentos no processo do pipeline. Perceba que temos cinco `stages` para o filtro na nossa coleção pokemons, `$match === find()`, `$sort`, `$skip`, `$limit`, e `$project`.

\$group

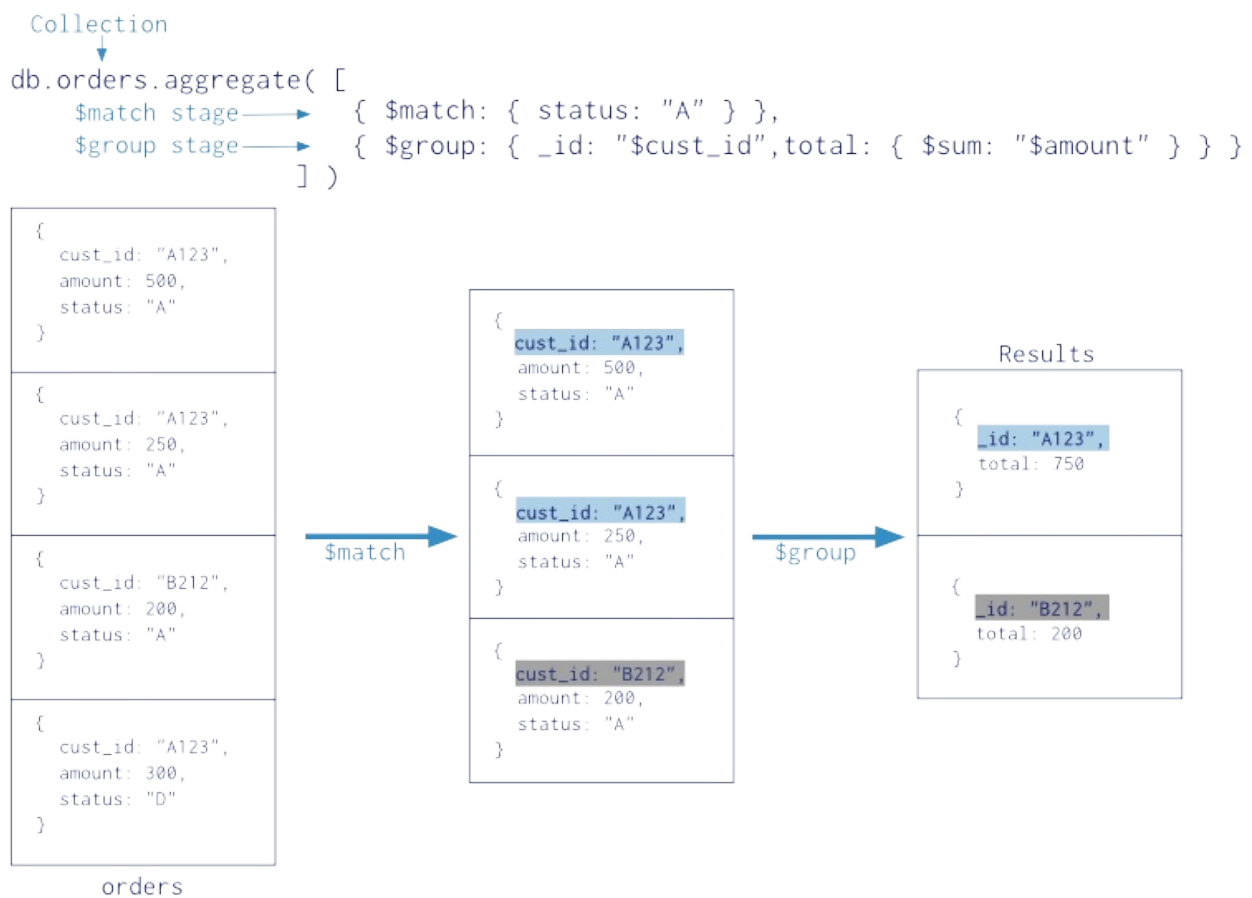
É o agrupamento de documentos por alguma expressão especificada e saída para a próxima fase de um documento para cada grupo distinto.

exemplo:

```
db.pokemons.aggregate(
  [
    {
      $group : {
        _id : null,
        totalPokemons: {$sum: 1}
      }
    }
  ]
)
```

Apenas para exemplificar o uso do `$group` no método **aggregate**.

A imagem abaixo temos um melhor entendimento no que se refere ao método `aggregate`.



Relacional vs. mongodb

Percebamos que temos no relacional uma estrutura onde não podemos trocar de lugares conforme as `stages` da estrutura do mongodb. Fora que no mongoDB podemos trabalhar com [Embedded Documents](#) de One-to-Many ou One-to-One. No banco relacional precisaríamos usar JOIN entre várias tabelas.

Performace

Pessoal, em outro post falarei sobre a criação de [INDEX](#) no mongodb, para obter uma melhor performance ao utilizar uma busca.

É isso aí e até a próxima!!!!

Referências

1 - [Documentação MongoDB](#)

group

O [método group](#) é outra ferramenta que temos no MongoDB para extrairmos informações agrupadas da nossa coleção, tais como média, soma, etc.

A sintaxe é:

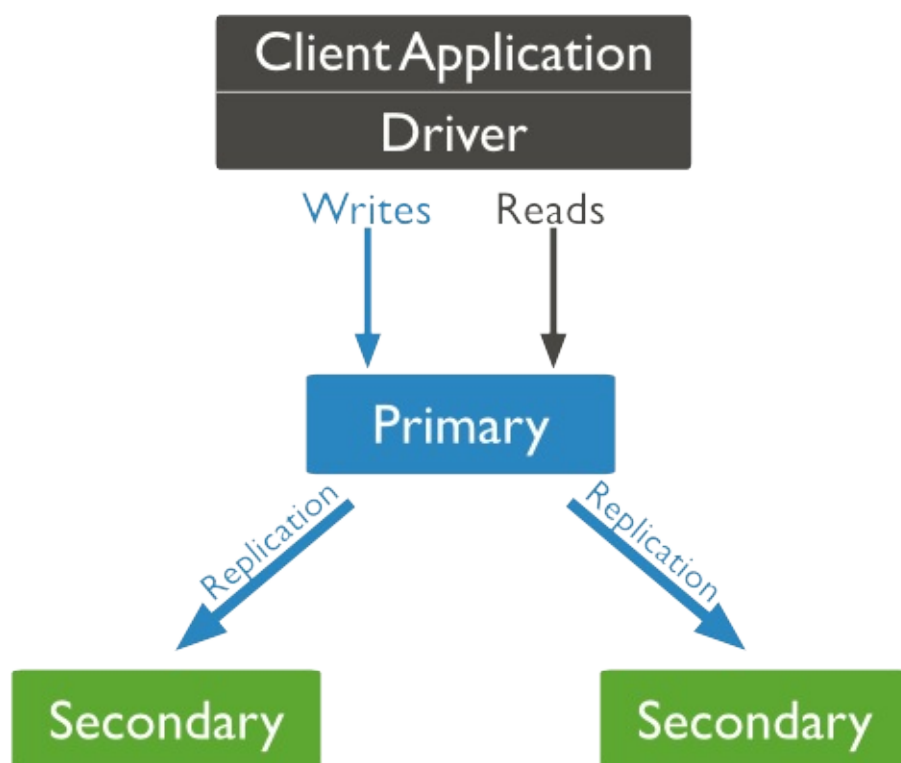
```
db.collection.group({
  key: <CAMPO OU CAMPOS QUE AGRUPARÃO A COLLECTION>,
  reduce: <FUNÇÃO ONDE O AGRUPAMENTO SERÁ FEITO, RECEBE DOIS ARGUMENTOS, SENDO O DOCUMENT>,
  initial: <INICIALIZA OS VALORES A SEREM UTILIZADOS>,
  keyf: <FUNÇÃO PARA CRIAR UMA NOVA CHAVE AGRUPADORA DINAMICAMENTE>,
  cond: <FILTRO QUE LIMITA QUAIS DOCUMENTOS SERÃO AGRUPADOS>,
  finalize: <FUNÇÃO QUE SERÁ EXECUTADA AO FINAL DO GROUP>,
})
```

Note que se o `key` for omitido, a coleção será agrupada como um todo, retornando um único documento como resposta. As opções `keyf`, `cond` e `finalize` são opcionais, e podem ser omitidas, se você não precisar delas.

O `group` não trabalha com coleções em shardings, então nesse caso, use o aggregation.

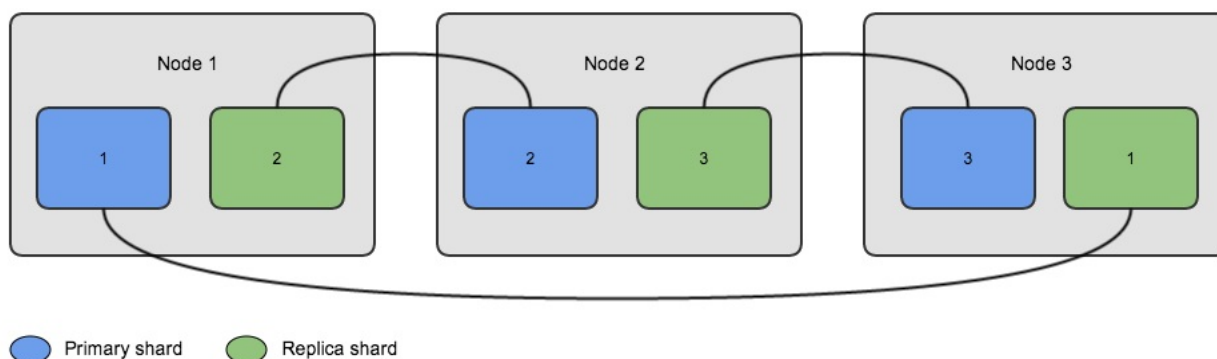
Replica

Possuímos *Replicas* na maioria dos bancos de dados relacionais também, ela faz o espelhamento dos seus dados de um servidor para outro, no MongoDB uma *ReplicaSet* pode conter 50 membros, ou seja, 50 *Replicas* contando com os árbitros.



Todas as operações de escrita são feitas no primário e replicada para os secundários, no MongoDB devemos também replicar os *Shards*.

3 Shards - 1 Replica - 3 Nodes



A replicação ocorre em 2 etapas:

- Initial Sync;
- Replication.

Initial Sync

O Initial Sync ocorre no início, quando uma *Replica* copia todos os dados de outra. Uma *Replica* utiliza-se do Initial Sync quando ela é nova ou não tem nenhum dado ou possui dados mas está faltando o histórico de replicação.

Quando a *Replica* executa um *Initial Sync* o MongoDB irá:

- Clonar todos os bancos de dados. Para clonar, o mongod consulta cada coleção em cada banco de dados de origem e insere todos os dados em suas próprias cópias dessas coleções. Neste momento, os índices `_id` também são construídos. O processo de clonagem apenas copia os dados válidos, omitindo documentos inválidos.
- Aplicar todas as alterações para o conjunto de dados. Usando o *oplog* a partir da fonte, o mongod atualiza seus dados para refletir o estado atual do conjunto de *Replicas*.
- Construir todos os índices em todas as coleções (exceto índices `_id`, que já foram concluídos).
- Quando o mongod acabar de construir todos os índices, o membro pode fazer a transição para um estado normal, ou seja secundário.

Replication

Membros do conjunto de *Replicas* replicam os dados continuamente após a sincronização inicial. Este processo mantém os membros atualizados com todas as alterações para os dados do conjunto de *Replicas*. Na maioria dos casos, secundários sincronizam a partir do primário. Secundários podem mudar automaticamente os seus alvos de sincronização, se necessário com base em mudanças no tempo de ping e estado de replicação de outros membros.

oplog

O **oplog** (log de operações) é uma *capped collection* especial que mantém os registros de todas as operações de modificação de dados.

MongoDB aplica as operações no primário e, em seguida, registra as operações no oplog do primário. Os membros secundários, em seguida, copiam e aplicam essas operações em um processo assíncrono.

Todos os membros do conjunto de *Replicas* contém uma cópia do oplog, na coleção [local.oplog.rs](#), o que lhes permite manter o estado atual da base de dados.

Para facilitar a replicação, todos os membros do conjunto de *Replicas* enviam batimentos cardíacos (pings) para todos os outros membros. Qualquer membro pode importar entradas oplog de qualquer outro membro.

Por que usar?

Porque sempre devemos ter uma garantia dos nossos dados e uma *Replica* serve exatamente para isso, garantir que seus dados existam em outro lugar também, caso o seu servidor principal caia você poderá levantar outro com os dados da sua *Replica*.

Quando usar?

SEMPRE! Pois você sempre precisa de uma segurança adicional para seus dados, nenhum servidor é 100% a prova de falhas por isso **sempre** se garanta.

Como usar?

Vou demonstrar localmente como fazer um conjunto de 3 *Replicas* bem simples, inicie criando 3 pastas novas dentro de `data`, as quais armazenarão os dados das *Replicas*.

```
mkdir /data/rs1
mkdir /data/rs2
mkdir /data/rs3
```

Agora vamos iniciar nossos processos do `mongod`, pare todos que você estiver rodando antes, só precisamos levantar o `mongod` com `--replSet` como visto abaixo:

```
mongod --replSet replica_set --port 27017 --dbpath /data/rs1
mongod --replSet replica_set --port 27018 --dbpath /data/rs2
mongod --replSet replica_set --port 27019 --dbpath /data/rs3
```

Caso você queira rodar eles em *background* basta passar o atributo `--fork` como visto no script [create-replicaset.sh](#):

Executando cada linha acima em um terminal diferente você podderá ver algo assim:

```
2015-11-20T13:12:48.187-0200 I CONTROL [initandlisten] options: {
  net: { port: 27019 },
  replication: { replSet: "replica_set" },
  storage: { dbPath: "/data/rs3" }
}
2015-11-20T13:12:48.209-0200 I NETWORK [initandlisten] waiting for connections on port 2
```

Configurando e iniciando

Depois você deve conectar em cada uma para iniciar o serviço de *Replica* com [rs.initiate\(\)](#), com uma configuração:

```
rsconf = {
  _id: "replica_set",
  members: [
    {
      _id: 0,
      host: "127.0.0.1:27017"
    }
  ]
}
rs.initiate(rsconf)
```

O objeto de configuração segue o seguinte modelo:

```
{
  _id: <string>,
  version: <int>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
    },
    ...
  ],
  settings: {
    getLastErrorDefaults : <document>,
    chainingAllowed : <boolean>,
    getLastErrorModes : <document>,
    heartbeatTimeoutSecs: <int>
  }
}
```

Após a execução desse comando vá até o terminal que está rodando o `rs1` e você verá algo assim:

```
2015-11-27T12:04:22.801-0200 I REPL      [conn1] replSet replSetInitiate config object wit
2015-11-27T12:04:22.817-0200 I REPL      [ReplicationExecutor] New replica set config in u
2015-11-27T12:04:22.817-0200 I REPL      [ReplicationExecutor] This node is 127.0.0.1:2701
2015-11-27T12:04:22.817-0200 I REPL      [ReplicationExecutor] transition to STARTUP2
2015-11-27T12:04:22.817-0200 I REPL      [conn1] *****
2015-11-27T12:04:22.817-0200 I REPL      [conn1] creating replication oplog of size: 192MB
2015-11-27T12:04:22.817-0200 I STORAGE  [FileAllocator] allocating new datafile /data/rs1
2015-11-27T12:04:53.404-0200 I STORAGE  [FileAllocator] done allocating datafile /data/rs
2015-11-27T12:04:53.429-0200 I STORAGE  [conn1] MmapV1ExtentManager took 30 seconds to op
2015-11-27T12:04:53.440-0200 I REPL      [conn1] *****
2015-11-27T12:04:53.440-0200 I REPL      [conn1] Starting replication applier threads
2015-11-27T12:04:53.440-0200 I COMMAND  [conn1] command admin.$cmd command: replSetInitia
2015-11-27T12:04:53.440-0200 I REPL      [ReplicationExecutor] transition to RECOVERING
2015-11-27T12:04:53.441-0200 I REPL      [ReplicationExecutor] transition to SECONDARY
2015-11-27T12:04:53.441-0200 I REPL      [ReplicationExecutor] transition to PRIMARY
2015-11-27T12:04:54.443-0200 I REPL      [rsSync] transition to primary complete; database
2015-11-27T12:06:55.711-0200 I INDEX     [conn1] allocating new ns file /data/rs1/test.ns,
2015-11-27T12:06:57.202-0200 I STORAGE  [FileAllocator] allocating new datafile /data/rs1
2015-11-27T12:07:05.268-0200 I STORAGE  [FileAllocator] done allocating datafile /data/rs
2015-11-27T12:07:05.521-0200 I STORAGE  [conn1] MmapV1ExtentManager took 8 seconds to ope
2015-11-27T12:07:05.557-0200 I WRITE     [conn1] insert test.teste query: { _id: ObjectId(
```

Adicionando *Replicas*

Depois de termos iniciado nossa *Replica* primária vamos adicionar as outras *Replicas* nessa *ReplicaSet*:

```
rs.add("127.0.0.1:27018")
rs.add("127.0.0.1:27019")
```

Após nossas *Replicas* estarem rodando, vamos conectar em cada uma:

```
mongo --port 27017
MongoDB shell version: 3.0.6
connecting to: 127.0.0.1:27017/test
Mongo-Hacker 0.0.3
Server has startup warnings:
2015-11-20T10:34:58.383-0200 I CONTROL [initandlisten]
2015-11-20T10:34:58.383-0200 I CONTROL [initandlisten] ** WARNING: soft rlimits too low.
suissacorp(mongod-3.0.6)[PRIMARY] test>
```

Vamos para a segunda que deve ser `SECONDARY` já que o servidor da porta `27017` é o `PRIMARY`.

```
mongo --port 27018
MongoDB shell version: 3.0.6
connecting to: 127.0.0.1:27018/test
Mongo-Hacker 0.0.3
Server has startup warnings:
2015-11-20T10:34:58.472-0200 I CONTROL [initandlisten]
2015-11-20T10:34:58.472-0200 I CONTROL [initandlisten] ** WARNING: soft rlimits too low.
suissacorp(mongod-3.0.6)[SECONDARY] test>
```

E para confirmar que o terceiro também é `SECONDARY`.

```
mongo --port 27019
MongoDB shell version: 3.0.6
connecting to: 127.0.0.1:27019/test
Mongo-Hacker 0.0.3
Server has startup warnings:
2015-11-20T10:34:58.556-0200 I CONTROL [initandlisten]
2015-11-20T10:34:58.557-0200 I CONTROL [initandlisten] ** WARNING: soft rlimits too low.
suissacorp(mongod-3.0.6)[SECONDARY] test>
```

Gerenciando

Status da ReplicaSet

Para vermos o *status* de cada instância executamos o seguinte comando no `mongo` :

```
rs.status()
{
  "set": "replica_set",
  "date": ISODate("2015-11-20T12:37:19.505Z"),
  "myState": 1,
  "members": [
    {
      "_id": 0,
      "name": "localhost:27017",
      "health": 1,
      "state": 1,
      "stateStr": "PRIMARY",
      "uptime": 141,
      "optime": Timestamp(1448023002, 1),
      "optimeDate": ISODate("2015-11-20T12:36:42Z"),
      "electionTime": Timestamp(1448023005, 1),
      "electionDate": ISODate("2015-11-20T12:36:45Z"),
      "configVersion": 1,
      "self": true
    },
    {
      "_id": 1,
      "name": "localhost:27018",
      "health": 1,
      "state": 2,
      "stateStr": "SECONDARY",
      "uptime": 53,
      "optime": Timestamp(1448023002, 1),
      "optimeDate": ISODate("2015-11-20T12:36:42Z"),
      "lastHeartbeat": ISODate("2015-11-20T12:37:17.799Z"),
      "lastHeartbeatRecv": ISODate("2015-11-20T12:37:18.015Z"),
      "pingMs": 0,
      "configVersion": 1
    },
    {
      "_id": 2,
      "name": "localhost:27019",
      "health": 1,
      "state": 2,
      "stateStr": "SECONDARY",
      "uptime": 53,
      "optime": Timestamp(1448023002, 1),
      "optimeDate": ISODate("2015-11-20T12:36:42Z"),
      "lastHeartbeat": ISODate("2015-11-20T12:37:17.830Z"),
      "lastHeartbeatRecv": ISODate("2015-11-20T12:37:18.015Z"),
      "pingMs": 0,
      "configVersion": 1
    }
  ],
  "ok": 1
}
```

Para conhecer mais sobre as configuração da *ReplicaSet* [entre aqui no - replSetGetConfig](#).

Status do oplog

Também possuímos a função `rs.printReplicationInfo()` que mostra um relatório do *oplog* da sua *ReplicaSet*:

```
rs.printReplicationInfo()
configured oplog size: 192MB
log length start to end: 1796secs (0.5hrs)
oplog first event time: Fri Nov 27 2015 00:17:37 GMT-0200 (BRST)
oplog last event time: Fri Nov 27 2015 00:47:33 GMT-0200 (BRST)
now: Fri Nov 27 2015 11:00:30 GMT-0200 (BRST)
```

Rebaixando a Replica Primária

Caso você deseje rebaixar a *Replica Primária* basta executar o comando `rs.stepDown()` como visto abaixo, forçando o MongoDB a eleger uma Secundária como Primária:

```
suissacorp(mongod-3.0.6)[PRIMARY] test> rs.stepDown()
2015-11-27T11:03:56.373-0200 I NETWORK DBClientCursor::init call() failed
2015-11-27T11:03:56.376-0200 E QUERY Error: error doing query: failed
    at DBQuery._exec (src/mongo/shell/query.js:83:36)
    at DBQuery.hasNext (src/mongo/shell/query.js:240:10)
    at DBCollection.findOne (src/mongo/shell/collection.js:187:19)
    at DB.runCommand (src/mongo/shell/db.js:58:41)
    at DB.adminCommand (src/mongo/shell/db.js:66:41)
    at Function.rs.stepDown (src/mongo/shell/utils.js:1006:15)
    at (shell):1:4 at src/mongo/shell/query.js:83
2015-11-27T11:03:56.378-0200 I NETWORK trying reconnect to 127.0.0.1:27119 (127.0.0.1) f
2015-11-27T11:03:56.378-0200 I NETWORK reconnect 127.0.0.1:27119 (127.0.0.1) ok
suissacorp(mongod-3.0.6)[SECONDARY] test>
```

Logo após a execução do comando o `mongo` desconecta e conecta novamente, porém dessa vez como `SECONDARY`, como visto na última linha acima.

Sincronizando de outro servidor

Caso queira mudar de qual *Replica* a sincronização é feita debes usar o comando `rs.syncFrom()`, por exemplo:


```
suissacorp(mongod-3.0.6)[SECONDARY] test> rs.syncFrom("127.0.0.1:27119")
{
  "syncFromRequested": "127.0.0.1:27119",
  "ok": 1
}
```

Isso é interessante para você testar diferentes padrões e situações onde uma *Replica* não esteja replicando do *host* desejado.

Recapitulando

1. Criar um diretório em `/data` para cada *Replica*.
2. Levantar cada *Replica* com `--replSet nome_ReplicaSet` em uma porta diferente.
3. Criar um JSON de configuração.
4. Conectar no **primário** e executar `rs.initiate(JSON_de_config)`.
5. Adicionar as outras *Replicas* caso não tenha as colocado no JSON de configuração.
6. Pronto.

Árbitro

É um serviço que não possui a réplica dos dados e nem pode virar primário, mas tem poder do voto de Minerva, onde ele terá um poder decisivo na votação de qual *Replica* secundária deve virar primária.

Comunicação

A única comunicação entre os árbitros e os outros membros da *ReplicaSet* são:

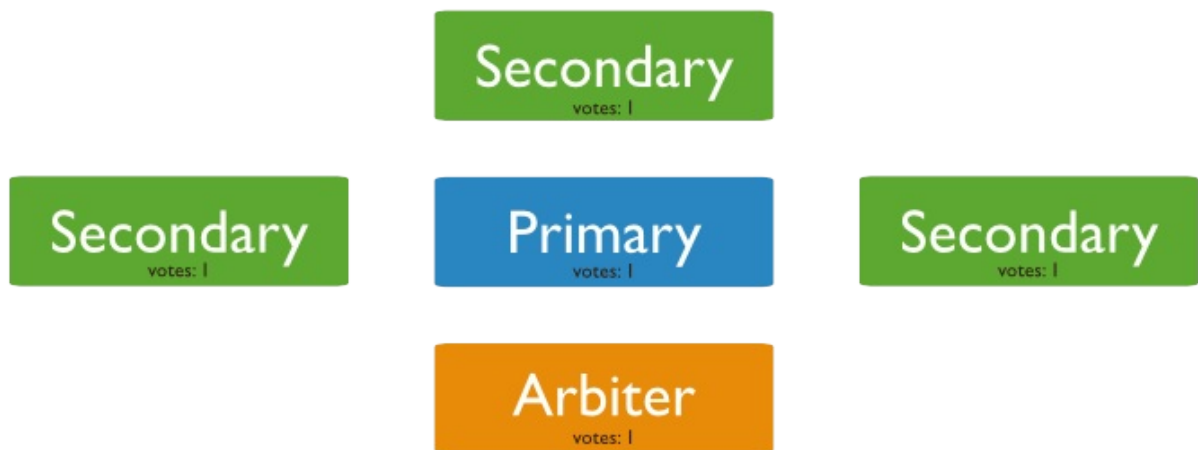
- votar durante eleições;
- heartbeats;
- dados de configuração.

Por que usar?

Porque quando uma **Replica primária** cair o MongoDB deverá eleger uma **Replica secundária** para virar primária.

Quando usar?

Só adicione um árbitro em uma *ReplicaSet* com um número **PAR** de membros, para que o árbitro seja o desempate.



Como usar?

Primeiramente crie um diretório que conterà os dados de configuração.

```
mkdir /data/arb
```

Depois precisa levantar o `mongod` utilizando `--replSet nomeDaReplicaSet` com seu diretório anteriormente criado.

```
mongod --port 30000 --dbpath /data/arb --replSet replica_set
```

Após levantar seu árbitro, conecte na *Replica* primária e adicione o árbitro criado com [rs.addArb\(\)](#):

```
rs.addArb("127.0.0.1:30000")
```

Sharding

Sharding é o processo de armazenamento de registros de dados em várias máquinas, é a abordagem que o MongoDB faz para atender o crescimento dos dados.

À medida que o tamanho dos dados aumenta, uma única máquina pode não ser suficiente para armazenar os dados, nem proporcionar uma leitura aceitável e rendimento na escrita, o Sharding resolve o problema com a escalabilidade horizontal, com sharding, você deve adicionar mais máquinas para suportar o crescimento de dados e as demandas de leitura e escrita.

Qual diferença entre escalabilidade horizontal e vertical?

Por que usar?

Porque o seu servidor não aguentará quando alguma coleção sua for maior que sua memória RAM, fazendo com que o MongoDB tenha que paginar os dados quando for ler, impactando na performance.

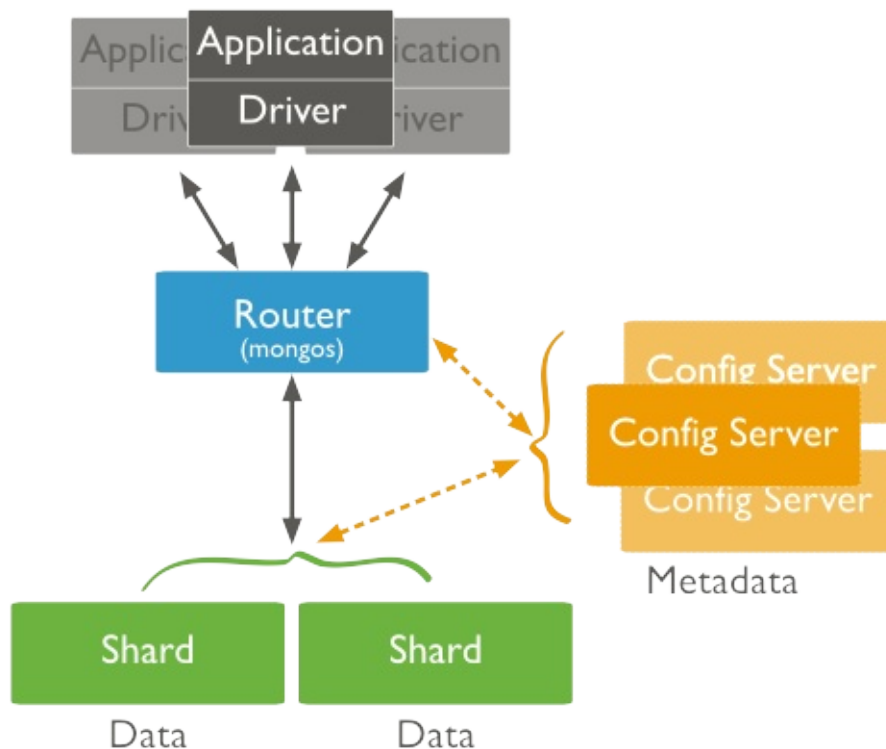
Quando usar?

Quando você analisar seu banco de dados e verificar que uma coleção está chegando perto do tamanho que o servidor tem de memória disponível para o MongoDB.

Como usar?

Para usar precisamos entender como é a arquitetura de um cluster com MongoDB, nele possuímos 3 serviços diferentes que são:

- shards
- config servers
- router



Shards

Cada shard é uma instância do MongoDB que guarda um pedaço dos dados da coleção.

Config Servers

Cada config server é uma instância do MongoDB que guarda os metadados sobre o cluster. Os metadados mapeiam os *chunks* de dados para os shards.

Router

Cada router é uma instância `mongos` que faz o roteamento das escritas e leituras para os shards. A aplicação não acessa diretamente os shards.

Para verificar todas as conexões do seu `mongos` basta conectar nele e rodar o seguinte comando:

```
db._adminCommand("connPoolStats");
```

Criando um cluster local

Criando o Config Server

Primeiramente criamos um *Config Server* utilizando o próprio `mongod`, porém usando o atributo `--configsvr` e setando a porta `27010`.

```
mkdir \data\configdb
$ mongod --configsvr --port 27010
```

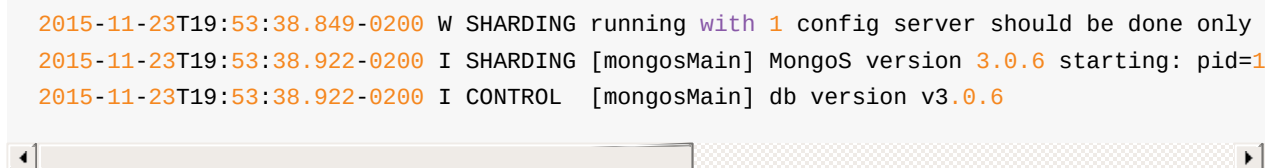
Como estamos fazendo para testar iremos criar apenas 1, **porém a indicação oficial é de criar pelo menos 3 Config Server para não ter 1 ponto único de falha.**

Criando o Router

Depois disso precisamos criar o *Router* utilizando o `mongos`, setando o *Config Server* que ele acessará para ter as informações dos *Shards*.

```
mongos -c-onfigdb localhost:27010 --port 27011
```

Quando rodar você verá o começo das mensagens assim:



```
2015-11-23T19:53:38.849-0200 W SHARDING running with 1 config server should be done only
2015-11-23T19:53:38.922-0200 I SHARDING [mongosMain] MongoDB version 3.0.6 starting: pid=1
2015-11-23T19:53:38.922-0200 I CONTROL [mongosMain] db version v3.0.6
```

Para você configurar mais de 1 *Config Server* basta passar seu `ip:porta` separado por vírgula após o `--configdb`, por exemplo:

```
mongos --configdb localhost:27010,190.1.1.10:666,190.1.1.11:666, --port 27011
```

Criando os Shards

Agora vamos criar 3 *Shards* que conterão nossos dados, por favor abra 3 terminais separados, podemos colocar os processos em background com `&` mas eu quero que vocês vejam o que acontece em cada.

Antes de tudo vamos criar as pastas onde os *Shards* irão persistir nossos dados:

```
mkdir /data/shard1 && mkdir /data/shard2 && mkdir /data/shard3
```

Depois de criado nossos diretórios rode cada comando em um terminal diferente.

Shard 1

```
mongod --port 27012 --dbpath /data/shard1
```

Shard 2

```
mongod --port 27013 --dbpath /data/shard2
```

Shard 3

```
mongod --port 27014 --dbpath /data/shard3
```

Resgistrando os Shards no Router

Vamos conectar no *Router* para poder registrar os *Shards*.

```
mongo --port 27011 --host localhost
MongoDB shell version: 3.0.6
connecting to: localhost:27011/test
Mongo-Hacker 0.0.3
mongos> sh.addShard("localhost:27012")
{ "shardAdded" : "shard0000", "ok" : 1 }
mongos> sh.addShard("localhost:27013")
{ "shardAdded" : "shard0001", "ok" : 1 }
suissacorp(mongos-3.0.6)[mongos] test> sh.addShard("localhost:27014")
{
  "shardAdded": "shard0002",
  "ok": 1
}

/***
mongos> sh.enableSharding("students")
{ "ok" : 1 }
mongos> sh.shardCollection("students.grades", {"student_id" : 1})
{ "collectionsharded" : "students.grades", "ok" : 1 }
***/
```

Depois disso vamos especificar qual *database* iremos *shardear*:

```
sh.enableSharding("be-mean")
{
  "ok": 1
}
```

E depois especificamos qual **coleção** dessa *database* será *shardeada* com

```
sh.shardCollection :
```

```
sh.shardCollection("be-mean.notas", {"_id" : 1})
{
  "collectionsharded": "be-mean.notas",
  "ok": 1
}
```

Enviando os dados para o Router

Vamos conectar no *Router* e adicionar dados na nossa *database* e coleção:

```
for ( i = 1; i < 100000; i++ ) {
  db.notas.insert({tipo: "prova", nota : Math.random() * 100, estudante_id: i, active: tr
}
```

Lembrando que devemos enviar os dados sempre para o *Router* para ele decidir o que fazer.

DICA

O tamanho padrão do *chunk* de cada *shard* é 64MB, logo a coleção precisar **ser maior que 64MB** para que ocorra a divisão dos seus dados pela `shard key` .

Dependendo do número de *shards* do seu *cluster* o MongoDB pode esperar que tenha pelo menos 10 *chunks* para disparar a migração.

Você pode rodar `db.printShardingStatus()` para ver todos os *chunks* presentes no servidor.

GridFS

GridFS é o sistema de arquivos do MongoDB, ele irá armazenar os binários diretamente no banco.

Por que usar?

Você pode querer guardar algum binário no banco porém o limite de cada documento **BSON** é de 16 MB, logo se você quiser armazenar algo maior o GridFS é a ferramenta correta pro serviço.

E também **se você não quiser que todo o arquivo vá para a memória RAM**, isso é algo muito importante quando você está trabalhando com uma coleção grande de arquivos.

Quando usar?

Tudo bem entendi que é para usar para armazenar arquivos maiores que 16 MB e que não vão para a memória, mas quando vou usar?



Em algumas situações, o armazenamento de arquivos grandes podem ser mais eficiente no MongoDB do que em um sistema de arquivos.

- Se seu sistema de arquivos limita o número de arquivos em um diretório, você pode usar GridFS para armazenar quantos arquivos quiser.
- Quando você quiser manter seus arquivos e metadados automaticamente sincronizados. Ao usar réplicas distribuídas geograficamente o MongoDB pode distribuir arquivos e seus metadados automaticamente.
- Quando você quiser acessar informações de partes de arquivos grandes sem ter que carregar todos os arquivos em memória, você pode usar GridFS buscar seções dos arquivos sem ler o arquivo inteiro na memória.

Não use GridFS se você precisar atualizar o conteúdo de todo o arquivo atômicamente. Como alternativa, você pode armazenar várias versões de cada arquivo e especificar a versão atual do arquivo nos metadados. Você pode atualizar o campo de metadados que indica o status de "último" em uma atualização atômica após o upload de uma nova versão do arquivo, e depois remover versões anteriores, se necessário.

Além disso, se seus arquivos são todos menores de 16MB, considere armazenar o arquivo manualmente dentro de um único documento. Você pode usar o tipo de dados `BinData` para armazenar os dados binários. Consulte a documentação de drivers para detalhes sobre como usar `BinData`. Pois se vc armazenar um arquivo pequeno, só para ele retornar esses 16MB o MongoDB irá retornar 65 documentos pelo menos de 255Kb, logo nada aconselhável né?

Como usar?

Para utilizar o GridFS, no terminal, usaremos o `mongofiles` passando o atributo `-d` `nome_database` para o nome da database onde iremos inserir o arquivo e `put` `nome_do_arquivo` para enviarmos o arquivo selecionado. Além disso pode ser necessário passar `-h 127.0.0.1` para definir nosso host como local.

Vamos fazer isso então na pasta `apostila/module-mongodb/data` onde se encontra o vídeo `0s_Raios_do_Pikachu.mp4` que iremos inserir no GridFS.

```
mongofiles -d be-mean-files put 0s_Raios_do_Pikachu.mp4 -h 127.0.0.1
2015-11-19T15:44:38.964-0200    connected to: 127.0.0.1
added file: 0s_Raios_do_Pikachu.mp4
```

O GridFS irá automaticamente irá gerar 2 coleções dentro do database informado:

- `fs.chunks`
- `fs.files`

Na coleção `fs.chunks` fica nosso arquivo binário dividido em pequenas partes, chamadas de `chunks`, cada *chunk* é um documento contendo 255KB de dados seguindo essa estrutura:

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary>
}
```

Na coleção `fs.files` temos os metadados do arquivo armazenado, como:

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>,
  "uploadDate" : <timestamp>,
  "md5" : <hash>,
  "filename" : <string>,
}
```

Caso você queira inserir seus arquivos com mais metadados terá que usar algum driver do MongoDB na sua programação que suporte o GridFS.

Você deve ter notado que temos o campo `md5`, para que o `md5` do arquivo pode ser interessante nesse caso?

Bom, você pode fazer uma busca pelo `md5` e caso encontre mais de 1 registro, é porque existem arquivos duplicados, aí você decide o que fazer com ele, como por exemplo removê-los.

[DICA] Se for usar o GridFS, utilize-o em um servidor próprio para configurá-lo da melhor forma possível.