

MongoDB

Construa novas aplicações com
novas tecnologias



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-5519-043-8

EPUB: 978-85-5519-044-5

MOBI: 978-85-5519-045-2

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Agradeço a você por pensar fora da caixa e escolher uma excelente alternativa à tecnologia de 1970: os bancos relacionais!

Agradeço ao Leandro Pincini por me mostrar o MongoDB e pela ajuda nos exemplos de Mac OS X.

Agradeço também a todas as pessoas que se dedicam ao software livre, pois, sem elas, não teríamos excelentes sistemas operacionais, banco de dados, servidores de aplicação, browsers, ferramentas e tudo mais de ótima qualidade.

Agradeço à minha esposa por sempre estar ao meu lado, aos meus pais e a Deus por tudo.

E segue o jogo!

QUEM É FERNANDO BOAGLIO

Uma imagem fala mais do que mil palavras. Veja quem eu sou na figura:



Figura 1: Quem é Fernando Boaglio?

PREFÁCIO

Por que construir aplicações novas com tecnologia antiga? É impressionante como aprendemos o que bancos de dados relacionais são e não são. E não há nada que possa ser feito sobre isso.

Sua aplicação pode usar a mais nova tecnologia existente, mas quando for persistir os dados, necessitará do banco de dados relacional usando a mesma tecnologia dos anos setenta. Existe espaço para todos e, com certeza, em vários casos os bancos de dados NoSQL, como o MongoDB, se sobressaem em relação aos tradicionais bancos relacionais.

Público-alvo

Este livro foi feito para desenvolvedores de sistemas que usam bancos de dados relacionais e procuram alternativas melhores. Também foi escrito para os interessados em aprender sobre o MongoDB, que é o mais famoso e mais usado banco de dados NoSQL, para explicar por que as grandes empresas estão investindo terabytes nessa tecnologia.

No site do MongoDB, temos uma excelente documentação que, no entanto, apenas explica como o comando funciona e não faz nenhuma comparação com o SQL que todo desenvolvedor conhece. Aqui, caro leitor, você sempre encontrará um comparativo com o SQL relacional que vai facilitar muito o funcionamento e as vantagens do MongoDB.

Quickstart — a primeira parte do livro

Para rapidamente configurar o seu ambiente e disponibilizar o seu banco de dados MongoDB modelado corretamente para a sua aplicação, não será preciso ler todos os capítulos, apenas os cinco primeiros.

Melhorando seu banco de dados — a segunda parte do livro

Os capítulos restantes complementam com a parte de migração de outro banco de dados para o MongoDB, performance, administração, comandos avançados de busca e utilização de particionamento e cluster.

Apêndices — instalação,upgrade e FAQ

Foram criados dois apêndices focados em instalação: o apêndice A, para instalação do banco de dados do MongoDB; e o apêndice B, para a ferramenta cliente RoboMongo.

Existe também um terceiro apêndice, com as perguntas e respostas mais frequentes sobre o MongoDB, como por exemplo, se ele suporta transações, ou quais as grandes empresas que o usam. O quarto apêndice orienta a migrar o seu banco de dados de uma versão mais antiga do MongoDB.

Código-fonte

O código-fonte deste livro está disponível no endereço <https://github.com/boaglio/mongodb-casadocodigo>.

Sumário

1 Por que criar aplicações novas com conceitos antigos?	1
1.1 O sistema na maneira tradicional	2
1.2 Próximos passos	5
2 JSON veio para ficar	7
2.1 Próximos passos	10
3 MongoDB básico	11
3.1 Conceitos	11
3.2 Acessando ao MongoDB	13
3.3 Exemplo da Mega-Sena	14
3.4 Buscar registros	17
3.5 Adicionar registros	22
3.6 Atualizar registros	27
3.7 Remover registros	33
3.8 Criar e remover collections	34
3.9 Alterando uma coluna de uma collection	35
3.10 Validação dos dados	36
3.11 Melhorando as buscas	38

Sumário	
Casa do Código	
3.12 Capped Collection	43
3.13 Próximos passos	44
4 Schema design	46
4.1 Relacionando uma collection para muitas	48
4.2 Relacionando muitas collection para muitas	50
4.3 Tudo em uma collection	50
4.4 Schema design na prática	51
4.5 Sistema Meus filmes relacional	54
4.6 Sistema Meus filmes no MongoDB	56
4.7 Próximos passos	57
5 Conversando com MongoDB	59
5.1 O sistema de seriados	59
5.2 Seriados em PHP	60
5.3 Java	67
5.4 Play Framework	76
5.5 Ruby on Rails	79
5.6 Node.js	81
5.7 Qt	83
5.8 Próximos passos	86
6 Migrando o seu banco de dados	88
6.1 IMDB simplificado	88
6.2 Migrando de um banco de dados relacional	92
6.3 Migrando para nuvem	94
6.4 Próximos passos	105
7 Buscas avançadas	107

Casa do Código	Sumário
7.1 Operadores de comparação	108
7.2 Operador distinct	109
7.3 Expressões regulares	110
7.4 Operadores lógicos	110
7.5 Operadores unários	111
7.6 Operador estilo LIKE	112
7.7 Incrementando valores	116
7.8 Próximos passos	116
8 Busca geoespacial	118
8.1 O banco de dados	118
8.2 Usando o sistema web	121
8.3 Entendo o sistema web	122
8.4 Próximos passos	124
9 Aggregation Framework	125
9.1 Por que não usar Map Reduce	125
9.2 Explorando o Aggregation Framework	127
9.3 Próximos passos	134
10 Aumentando a performance	135
10.1 Criar um índice	137
10.2 Listar os índices criados	141
10.3 Remover um índice criado	142
10.4 Índice textual	142
10.5 Criar índice em background	145
10.6 Próximos passos	146
11 MongoDB para administradores	147

11.1 Tipos de storage	147
11.2 Ajuste de performance	148
11.3 Gerenciando espaço em disco	148
11.4 Autenticação	150
11.5 Programas externos	155
11.6 Backup	156
11.7 Restore	158
11.8 Exibir operações rodando	160
11.9 Próximos passos	161
12 MongoDB em cluster	162
12.1 Alta disponibilidade	162
12.2 Testando dois replica sets	163
12.3 Particionamento	165
12.4 Próximos passos	174
13 Continue seus estudos	175
14 Apêndice A — Instalando MongoDB	176
15 Apêndice B — Robomongo	196
16 Apêndice C — Perguntas e respostas	210
17 Apêndice D — Upgrade da versão 2.6 para MongoDB 3.x	215

Versão: 20.8.24

CAPÍTULO 1

POR QUE CRIAR APLICAÇÕES NOVAS COM CONCEITOS ANTIGOS?

Nas últimas décadas, a criação de um sistema evoluiu apenas de um lado (o da interface com o usuário), começando com sistemas na arquitetura de cliente/servidor (como os feitos em Visual Basic ou Delphi) até sistemas em três camadas (como a maioria dos sites na Internet, feitos em PHP, Java ou ASP), e terminando nos sistema com celular.

De um lado, a tela desktop evoluiu para a tela web e, finalmente, para a tela do celular. Mas por trás de todas elas, quase sempre estava algum banco de dados relacional.

Se sempre foi assim, é natural que, ao projetarmos um sistema, sempre assumimos que o lado dos dados, da informação, da persistência, não mudará suas regras tão cedo. Portanto, desenhamos o problema em cima das regras (ou melhor, limitações) relacionais e, a partir dele, criamos uma camada de manipulação desses dados pela nossa aplicação, seja ela desktop, web ou mobile.

1.1 O SISTEMA NA MANEIRA TRADICIONAL

Desde os anos 90, a Universidade de Harvard oferece um prêmio dado aos autores de pesquisas, experimentos e outras atividades inusitadas, nas diversas áreas da ciência. Essa homenagem é chamada Prêmio IgNobel, que é o oposto ao Prêmio Nobel.

O nosso sistema é a lista de ganhadores do prêmio IgNobel (http://pt.wikipedia.org/wiki/Anexo:Lista_de_ganhadores_do_Prêmio_IgNobel), como o "sucesso no treino de pombos para distinguirem entre pinturas de Picasso e Monet", ou "depenagem de galinhas como meio de medir a velocidade do vento de um tornado".

Pela lista da Wikipedia, conseguimos separar quatro informações: ano, tipo, autor e descrição do prêmio. Partindo para modelagem relacional, temos o modelo da figura adiante.

Basicamente, pegamos as quatro informações e criamos uma tabela para cada um, e como um prêmio IgNobel pode ter vários autores, criamos uma tabela auxiliar `premio_autor`.

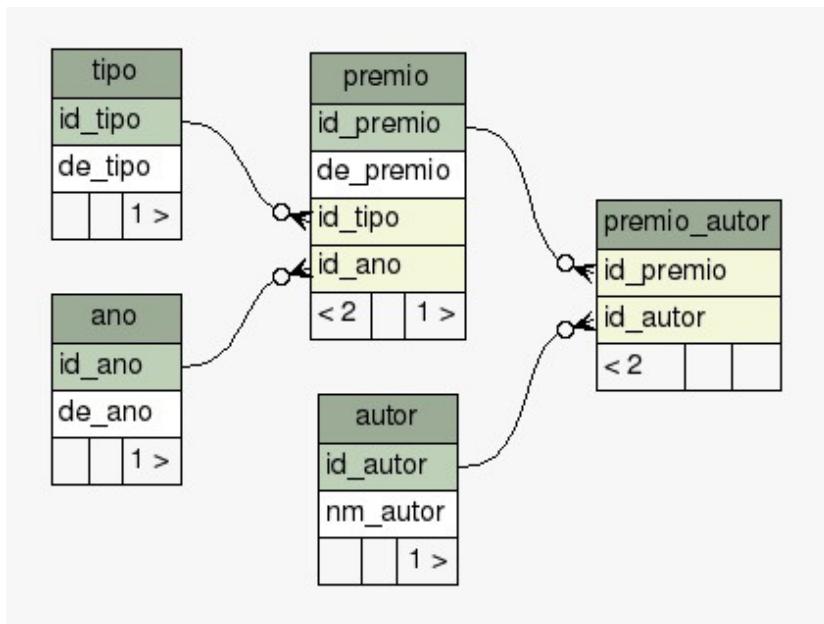


Figura 1.1: Modelo relacional dos ganhadores

Vamos listar algumas reflexões sobre esse modelo:

1. ao montarmos o modelo, pensamos em desnormalizar toda informação, isolando em quatro tabelas distintas;
2. como um mesmo prêmio pode ter vários autores, precisamos criar uma tabela auxiliar `premio_autor` ;
3. montamos toda estrutura baseada nas limitações de um banco de dados (no mundo real não existe representação da tabela auxiliar `premio_autor`);
4. não pensamos no que a aplicação vai fazer, pensamos apenas em arrumar os dados;
5. em caso de lentidão, revemos os SQLs e criamos índices.

Para exibir uma página como da Wikipedia, é preciso fazer

uma consulta envolvendo todas as cinco tabelas criadas:

```
select
p.de_premio, t.de_tipo, a.de_ano ,au.nm_autor
from premio p, tipo t, ano a, premio_autor pa, autor au
where p.id_premio = pa.id_premio
and p.id_tipo = t.id_tipo
and p.id_ano = a.id_ano
and pa.id_autor = au.id_autor
```

Como a página da Wikipedia tem muitos acessos, se eles tivessem feito da maneira convencional, o site com certeza não seria tão rápido. Portanto, pensando na aplicação e não nos dados, o ideal seria que tudo estivesse organizado de acordo com a necessidade do negócio, e não com *formas normais* do mundo relacional.

Se a página da Wikipedia exibe tudo de uma vez, o correto seria a informação estar concentrada em apenas um lugar (uma tabela). Essa prática já é conhecida no mundo do Data Warehouse, chamada de desnormalização.

No MongoDB, organizamos os dados em função da aplicação. Não temos tabelas, temos conjuntos de dados chamados *collections*, que, ao contrário de tabelas, não têm *constraints* (chave primária, chave estrangeira) nem transações. Além disso, também não têm as limitações de uma tabela relacional. Dentro de uma coluna, você pode ter um array ou uma lista de valores; algo impossível em uma tabela convencional.

Resumindo: da maneira convencional, a sua aplicação obedece às regras do seu banco de dados. No MongoDB, é o contrário: é a sua aplicação que manda, e os dados são organizados conforme a necessidade do sistema.

Nesse exemplo da Wikipedia, precisamos ter uma collection com todas as informações. As informações são organizadas dessa maneira:

```
{  
    "ano" : 1992,  
    "tipo" : "Medicina",  
    "autores" : [  
        "F. Kanda",  
        "E. Yagi",  
        "M. Fukuda",  
        "K. Nakajima",  
        "T. Ohta",  
        "O. Nakata"],  
    "premio" : "Elucidação dos Componentes Químicos Responsáveis  
    pelo Chulé do Pé (Elucidation of Chemical  
    Compounds Responsible for Foot Malodour),  
    especialmente pela conclusão de que as pessoas  
    que pensam que têm chulé, têm, e as que pensam  
    que não têm, não têm."  
}
```

Assim, em um único registro temos *todas* as informações de que precisamos.

Resumindo:

- De quantas tabelas precisamos para exibir um prêmio? Cinco.
- De quantas collections precisamos para exibir um prêmio? Uma.

1.2 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- enumerar as principais práticas da modelagem relacional tradicional;

- comparar as diferenças da modelagem relacional tradicional e do MongoDB.

Talvez existam algumas dúvidas sobre alguns conceitos ou aplicações do MongoDB; não é preciso ler o livro inteiro para esclarecê-las. Consulte o *Apêndice C — Perguntas e respostas*.

No próximo capítulo, vamos aprender a linguagem usada pelo MongoDB: o JSON e todo potencial que ele oferece.

CAPÍTULO 2

JSON VEIO PARA FICAR

Com o crescimento de serviços no começo do século, o tráfego de informações também aumentou. Foi preciso criar uma forma simples de enviar informação de um servidor para um web browser, sem a necessidade de nenhum plugin para funcionar (como Flash).

Por esse motivo, Douglas Crockford identificou uma prática usada desde 1996 pela Netscape como a solução desse problema. Ele criou uma especificação para ela e batizou-a como Notação de Objetos JavaScript (*JavaScript Object Notation*), ou simplesmente JSON.

A ideia é manter a simplicidade para transferir as informações, suportando tipos de dados bem simples:

1. `null` — valor vazio;
2. `Boolean` — `true` ou `false` ;
3. `Number` — número com sinal que pode ter um notação com `E` exponencial;
4. `String` — uma sequência de um ou mais caracteres Unicode;
5. `Object` — um array não ordenado com itens do tipo chave-valor, em que todas as chaves devem ser strings distintas no

mesmo objeto;

6. Array — lista ordenada de qualquer tipo, inteira entre colchetes e com cada elemento separado por vírgulas.

Um exemplo das informações do Brasil em formato JSON, um elemento sempre começa com chaves:

```
{  
    "país": "Brasil",  
    "população": 206081432,  
    "PIB total em trilhões de dólares": 3.101,  
    "faz fronteira com": [  
        "Argentina",  
        "Bolívia",  
        "Colômbia",  
        "Guiana Francesa",  
        "Guiana",  
        "Paraguai",  
        "Peru",  
        "Suriname",  
        "Uruguai",  
        "Venezuela"  
    ],  
    "cidades": {  
        "capital" : "Brasília",  
        "mais populosa": "São Paulo"  
    }  
}
```

Em uma lista de elementos JSON (lista de países), sempre começa com colchetes:

```
[  
{  
    "país": "Brasil",  
    "população": 206081432  
},  
{  
    "país": "Argentina",  
    "população": 41281631  
},  
{
```

```
        "país": "Bolívia",
        "população": 10426160
    }
]
```

Entretanto, a especificação JSON não padroniza o formato de data, ou como trabalhar com dados binários. Por esse motivo, o MongoDB trabalha com BSON (*Binary JSON*), que é uma extensão do JSON.

Além de todos os formatos do JSON, o BSON suporta:

1. MinKey, MaxKey, Timestamp — tipos utilizados internamente no MongoDB;
2. BinData — array de bytes para dados binários;
3. ObjectId — identificador único de um registro do MongoDB;
4. Date — representação de data;
5. Expressões regulares.

Veja um exemplo:

```
{
  "_id" : ObjectId("57e08da696535fff4a345c67"),
  "timestamp" : Timestamp(1474334118, 1),
  "data" : ISODate("2016-09-20T01:16:41.720Z"
}
```

É importante entender a sintaxe e os tipos, pois toda a comunicação feita entre você e o MongoDB, ou entre sua aplicação e o MongoDB, será nesse formato.

Se a expressão em JSON for muito extensa, você pode usar ferramentas online para formatação, como os sites:

- <http://jsonprettyprint.com>

- <https://jsonformatter.curiousconcept.com>

2.1 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- o que é JSON;
- os tipos existentes no JSON;
- a sintaxe de um elemento JSON;
- a sintaxe de uma lista de elementos JSON.

No próximo capítulo, faremos as operações básicas do MongoDB para manipulação de dados.

CAPÍTULO 3

MONGODB BÁSICO

Para iniciar este capítulo, é preciso antes instalar o software do MongoDB. Consulte *Apêndice A — Instalando MongoDB*.

3.1 CONCEITOS

O MongoDB é um *document database* (banco de dados de documentos), mas não são os documentos da família Microsoft, e sim documentos com informações no formato JSON. A ideia é o documento representar toda a informação necessária, sem a restrição dos bancos relacionais.

Em um documento, pode existir um valor simples, como um número, uma palavra ou uma data, e também uma lista de valores. Os documentos são agrupados em *collections*. Um conjunto de collections forma um *database* (banco de dados).

Se for necessário, esse database pode ser duplicado em outros servidores, e cada cópia é chamada de *replica set* (conjunto de réplica). A figura a seguir mostra uma *replica set* rs1, que contém dois databases, e cada um deles possui duas collections.

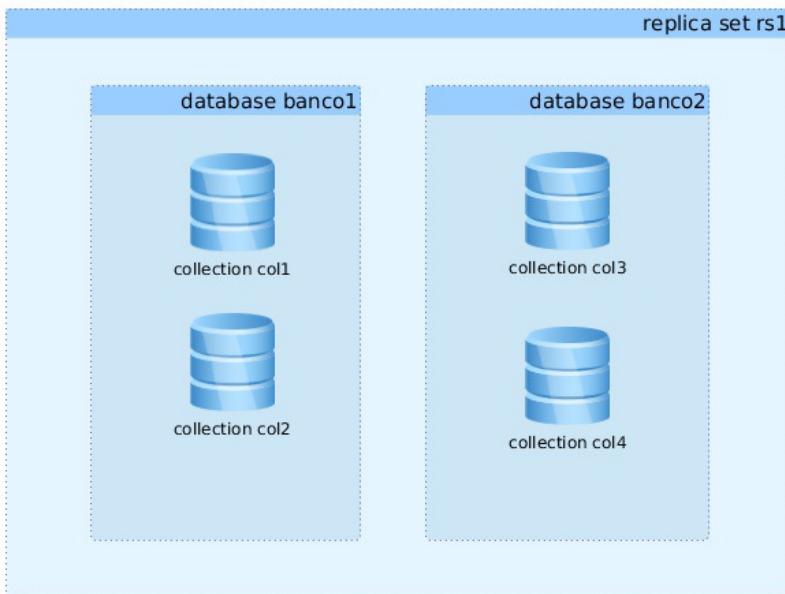


Figura 3.1: Conceitos básicos

Outro conceito mais avançado é o de *sharding* (particionamento — será exemplificado no *capítulo 12 - MongoDB em cluster*), que é usado quando sua collection passou dos bilhões de registros e há vantagem em dividir os dados por servidor.

A figura adiante mostra um exemplo com uma collection única de três terabytes que pode ser particionada em três partições de um terabyte cada, espalhada em três máquinas distintas. Neste exemplo, a collection de visitas de um site foi separada pela data, dividindo dados por trimestre. Na máquina 1, ficaram os dados de janeiro até abril; na máquina 2, de maio até agosto; e na máquina 3, de setembro até dezembro.

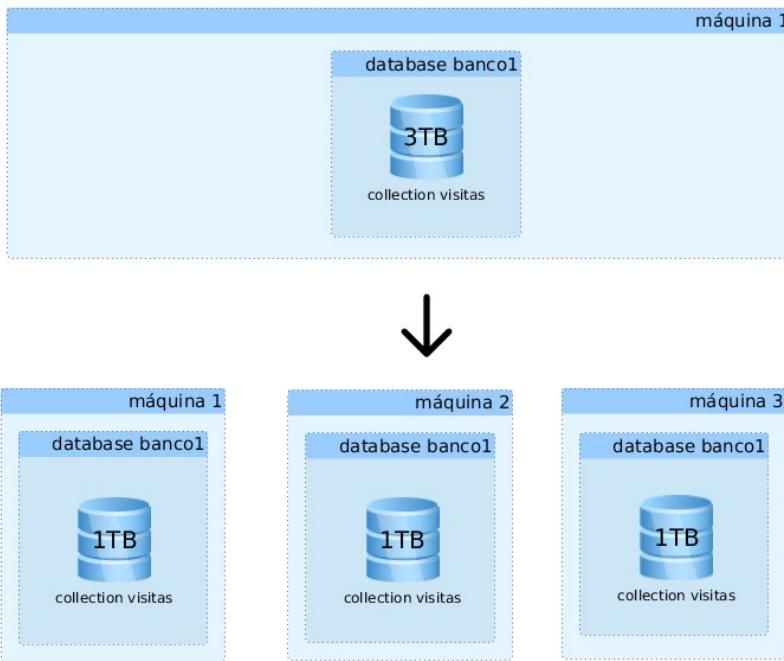


Figura 3.2: Sharding

3.2 ACESSANDO AO MONGODB

O acesso ao MongoDB pode ser feito via console (executável `mongo`) ou via Robomongo (para instalar, consulte *Apêndice B — Robomongo*).

Qualquer comando funcionará nos dois, mas o Robomongo é mais amigável. Além disso, possui autocomplete, que é bem útil quando não lembramos do comando.

Podemos resumir a grande maioria dos comandos na seguinte sintaxe:

```
db.<nome-da-collection>.<operacao-desejada>;
```

Veja um exemplo:

```
db.collection.count();
```

3.3 EXEMPLO DA MEGA-SENA

Vamos utilizar um exemplo mais concreto para entender melhor os conceitos e conhecer melhor a sintaxe do MongoDB, além de comparar com o SQL dos bancos relacionais.

No site da Caixa, estão disponíveis para download todos os resultados da Mega-Sena em formato HTML (<http://loterias.caixa.gov.br/wps/portal/loterias/landing/megasena>). Copiando o HTML, colando em uma planilha e gravando no formato CSV, podemos facilmente importar os valores para o MongoDB.

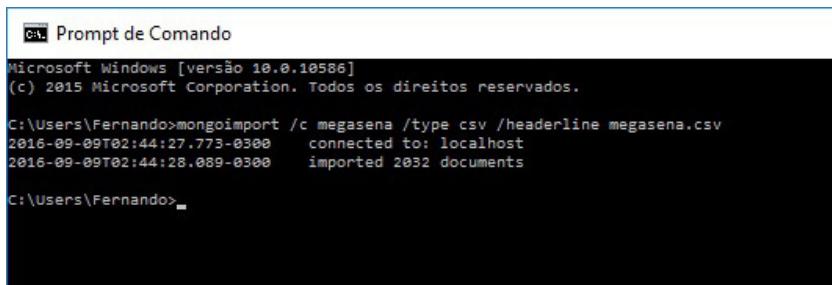
Baixe o arquivo `megasena.csv` (<https://github.com/boaglio/mongodb-casadocodigo/blob/master/capitulo-03/megasena.csv>), copie para um diretório de testes e execute o comando:

```
mongoimport  
  -c <nome-da-collection>  
  --type csv  
  --headerline <nome-do- arquivo-CSV>
```

Neste exemplo dos dados da Mega-Sena, os parâmetros são:

```
mongoimport -c megasena --type csv --headerline megasena.csv
```

O resultado é semelhante em Windows é o da figura a seguir.



```
Microsoft Windows [versão 10.0.10586]
(c) 2015 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Fernando>mongoimport /c megasena /type csv /headerline megasena.csv
2016-09-09T02:44:27.773-0300      connected to: localhost
2016-09-09T02:44:28.089-0300      imported 2032 documents

C:\Users\Fernando>
```

Figura 3.3: Importar dados da Mega-Sena

Em Linux, o resultado é parecido com esse. Note que aparece o total de documentos (objetos) importados: 2032 registros dentro da collection `megasena` .

```
fb@cascao > mongoimport
          -c megasena
          --type csv
          --headerline megasena.csv
connected to: 127.0.0.1
imported 2032 documents
```

Se a collection já existir, você pode usar o parâmetro `drop` para remover a existente:

```
fb@cascao > mongoimport
          -c megasena
          --type csv
          --drop
          --headerline megasena.csv
connected to: 127.0.0.1
dropping: test.megasena
imported 2032 documents
```

Em um banco relacional tradicional, para qualquer carga de dados, é necessário criar a estrutura (a tabela) que receberá os dados. No MongoDB, não precisamos de nada disso: a collection `megasena` , seja a estrutura que for, será criada automaticamente.

Para conferirmos a quantidade de registros importados, vamos executar o comando `db.megasena.count()` :

```
fb@cascao ~ > mongo
MongoDB shell version: 3.2.9
connecting to: test
> db.megasena.count()
2032
> exit
bye
fb@cascao ~ >
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.count()
```

- SQL Relacional:

```
select count(*) from megasena
```

Para mostrar mais detalhes da collection `megasena` , vamos executar o comando `db.megasena.stats()` no Robomongo (figura seguinte).

The screenshot shows the Robomongo interface. On the left, the database structure is displayed with 'localhost (2)' expanded, showing 'System', 'test' (expanded), 'Collections (1)' (expanded), 'megasena' (selected), 'Functions', and 'Users'. In the main panel, a query is being run in the command line:

```
db.getCollection('megasena').stats()
```

The results are shown in a table:

Key	Value	Type
ns	test.megasena	String
count	2032	Int32
size	920239	Int32
avgObjSize	452	Int32
storageSize	290816	Int32
capped	false	Boolean
wiredTiger	{ 13 fields }	Object
nindexes	1	Int32
totalIndexSize	28672	Int32
indexSizes	{ 1 field }	Object
ok	1.0	Double

Figura 3.4: Exibindo collection megasena no Robomongo

3.4 BUSCAR REGISTROS

Para exibir todos os registros de collection, usamos o comando `find`:

```
db.<nome-da-collection>.find()
```

O console exibirá os primeiros 20 registros:

```
> db.megasena.find();
{ "_id" : ObjectId("57d08d348688b893a04c3760"), "Concurso" : 1,
  "Data Sorteio" : "11/03/1996", "1ª Dezena" : 4, "2ª Dezena" : 5,
  "3ª Dezena" : 30, "4ª Dezena" : 33, "5ª Dezena" : 41,
  "6ª Dezena" : 52, "Arrecadacao_Total" : 0,
  "Ganhadores_Sena" : 0, "Rateio_Sena" : 0,
  "Ganhadores_Quina" : 17, "Rateio_Quina" : "39158,92",
```

```
"Ganhadores_Quadra" : 2016,  
"Rateio_Quadra" : "330,21", "Acumulado" : "SIM",  
"Valor_Acumulado" : "1714650,23",  
"Estimativa_Prêmio" : 0, "Acumulado_Mega_da_Virada" : 0 }  
...  
19 restantes  
...  
Type "it" for more  
>
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.find()
```

- SQL Relacional:

```
select * from megasena
```

Usando o comando `findOne`, é exibido apenas o primeiro registro.

```
> db.megasena.findOne()  
{  
    "_id" : ObjectId("57d08d348688b893a04c3760"),  
    "Concurso" : 1,  
    "Data_Sorteio" : "11/03/1996",  
    "1ª Dezena" : 4,  
    "2ª Dezena" : 5,  
    "3ª Dezena" : 30,  
    "4ª Dezena" : 33,  
    "5ª Dezena" : 41,  
    "6ª Dezena" : 52,  
    "Arrecadacao_Total" : 0,  
    "Ganhadores_Sena" : 0,  
    "Rateio_Sena" : 0,  
    "Ganhadores_Quita" : 17,  
    "Rateio_Quita" : "39158,92",  
    "Ganhadores_Quadra" : 2016,  
    "Rateio_Quadra" : "330,21",  
    "Acumulado" : "SIM",
```

```
        "Valor_Acumulado" : "1714650,23",
        "Estimativa_Prêmio" : 0,
        "Acumulado_Mega_da_Virada" : 0
    }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.findOne()
```

- MySQL:

```
select * from megasena limit 1
```

Note que, além dos campos dos sorteios, é exibido também um campo chamado `_id`. Este é conhecido como `ObjectId`, um campo inserido automaticamente pelo MongoDB, que garante a unicidade do registro na collection.

O valor do `ObjectId` pode ser fornecido explicitamente pelo usuário, ou implicitamente pelo MongoDB. O valor gerado não é sequencial, mas é gerado considerando o `timestamp`, o ID da máquina, ID do processo e um contador local.

O comando `find` pode receber parâmetros para filtrar o resultado:

```
db.<nome-da-collection>.find(<campo1>:<valor1>,
<campo2>:<valor2>, ...);
```

Neste exemplo, listamos o sorteio do concurso 73:

```
> db.megasena.find({"Concurso":73})
{ "_id" : ObjectId("57d08d348688b893a04c37b5"), "Concurso" : 73,
  "Data Sorteio" : "27/07/1997", "1ª Dezena" : 25,
  "2ª Dezena" : 26, "3ª Dezena" : 28, "4ª Dezena" : 45,
  "5ª Dezena" : 51, "6ª Dezena" : 57, "Arrecadacao_Total" : 0,
  "Ganhadores_Sena" : 1, "Rateio_Sena" : "21026575,4",
```

```
"Ganhadores_Quina" : 95, "Rateio_Quina" : "21205,27",
"Ganhadores_Quadra" : 8222, "Rateio_Quadra" : "244,52",
"Acumulado" : "NÃO", "Valor_Acumulado" : 0,
"Estimativa_Prêmio" : 0, "Acumulado_Mega_da_Virada" : 0 }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.find({"Concurso":73})
```

- MySQL:

```
select * from megasena where "Concurso"=73
```

A exibição padrão dos resultados de busca do MongoDB nem sempre é legível, pois o console exibe o JSON sem nenhum espaço ou quebra de linha. Para resolver esse problema, basta adicionar no comando o sufixo `.pretty()`:

```
> db.megasena.find({"Concurso":73}).pretty()
{
    "_id" : ObjectId("57d08d348688b893a04c37b5"),
    "Concurso" : 73,
    "Data_Sorteio" : "27/07/1997",
    "1ª Dezena" : 25,
    "2ª Dezena" : 26,
    "3ª Dezena" : 28,
    "4ª Dezena" : 45,
    "5ª Dezena" : 51,
    "6ª Dezena" : 57,
    "Arrecadacao_Total" : 0,
    "Ganhadores_Sena" : 1,
    "Rateio_Sena" : "21026575,4",
    "Ganhadores_Quina" : 95,
    "Rateio_Quina" : "21205,27",
    "Ganhadores_Quadra" : 8222,
    "Rateio_Quadra" : "244,52",
    "Acumulado" : "NÃO",
    "Valor_Acumulado" : 0,
    "Estimativa_Prêmio" : 0,
    "Acumulado_Mega_da_Virada" : 0
```

}

Para listarmos os dois sorteios da Mega-Sena com cinco ganhadores, faremos a consulta:

```
> db.megasena.find({ "Ganhadores_Sena": 5 })  
{ "_id" : ObjectId("57d08d348688b893a04c385c"),  
  "Concurso" : 233, "Data Sorteio" : "19/08/2000",  
  "1ª Dezena" : 3, "2ª Dezena" : 7, "3ª Dezena" : 24,  
  "4ª Dezena" : 32, "5ª Dezena" : 36, "6ª Dezena" : 45,  
  "Arrecadacao_Total" : 0, "Ganhadores_Sena" : 5,  
  "Rateio_Sena" : "3196547,03", "Ganhadores_Quita" : 512,  
  "Rateio_Quita" : "3790,28", "Ganhadores_Quadra" : 21452,  
  "Rateio_Quadra" : "90,2", "Acumulado" : "NÃO",  
  "Valor_Acumulado" : 0, "Estimativa_Prêmio" : 0,  
  "Acumulado_Mega_da_Virada" : 0 }  
{ "_id" : ObjectId("57d08d348688b893a04c3d1a"),  
  "Concurso" : 1350, "Data Sorteio" : "31/12/2011",  
  "1ª Dezena" : 3, "2ª Dezena" : 4, "3ª Dezena" : 29,  
  "4ª Dezena" : 36, "5ª Dezena" : 45, "6ª Dezena" : 55,  
  "Arrecadacao_Total" : NumberLong(549326718),  
  "Ganhadores_Sena" : 5, "Rateio_Sena" : "35523497,52",  
  "Ganhadores_Quita" : 954, "Rateio_Quita" : "33711,3",  
  "Ganhadores_Quadra" : 85582, "Rateio_Quadra" : "536,83",  
  "Acumulado" : "NÃO", "Valor_Acumulado" : 0,  
  "Estimativa_Prêmio" : 2500000, "Acumulado_Mega_da_Virada" : 0 }
```

O comando `find` exibe todas as colunas por padrão. Para restringir os campos, devemos informar conforme a sintaxe:

```
db.<nome-da-collection>.find(  
  {<campo1>:<valor1>,  
   <campo2>:<valor2>,  
   ...},  
  {<campoParaExibir>:<exibeOuNaoExibe>,  
   <campoParaExibir>:<exibeOuNaoExibe>,  
   ...})
```

O valor de `exibeOuNaoExibe` pode ser nos formatos numérico e booleano: 1 ou true exibem o campo; 0 ou false , não.

Vamos exibir apenas a coluna `Concurso` na lista dos cinco ganhadores:

```
> db.megasena.find({"Ganhadores_Sena":5}, {"Concurso":1})
{ "_id" : ObjectId("57d08d348688b893a04c385c"),
  "Concurso" : 233 }
{ "_id" : ObjectId("57d08d348688b893a04c3d1a"),
  "Concurso" : 1350 }
```

Por padrão, a coluna do `ObjectId` é sempre exibida. É preciso explicitamente inativar a sua exibição, conforme os dois exemplos:

```
> db.megasena.find({"Ganhadores_Sena":5}, {"Concurso":true,
                           "_id":false})
{ "Concurso" : 233 }
{ "Concurso" : 1350 }

> db.megasena.find({"Ganhadores_Sena":5}, {"Concurso":1,
                           "_id":0});
{ "Concurso" : 233 }
{ "Concurso" : 1350 }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.find({"Ganhadores_Sena":5}, {"Concurso":1,
                           "_id":0})
```

- MySQL:

```
select "Concurso" from megasena where "Ganhadores_Sen
a"=5
```

3.5 ADICIONAR REGISTROS

Para adicionar novos registros, usamos o comando `insert`:

```
db.<nome-da-collection>.insert(
```

```
{ <campo1>:<valor1>,
  <campo2>:<valor2>,
  ...
});
```

Vamos inserir um novo sorteio, fazendo uma analogia com o MySQL:

- MongoDB:

```
db.megasena.insert(
{
  "Concurso" : 99999,
  "Data_Sorteio" : "07/09/2016",
  "1a Dezena" : 1,
  "2a Dezena" : 2,
  "3a Dezena" : 3,
  "4a Dezena" : 4,
  "5a Dezena" : 5,
  "6a Dezena" : 6,
  "Arrecadacao_Total" : 0,
  "Ganhadores_Sena" : 0,
  "Rateio_Sena" : 0,
  "Ganhadores_Quina" : 1,
  "Rateio_Quina" : "88000",
  "Ganhadores_Quadra" : 55,
  "Rateio_Quadra" : "76200",
  "Acumulado" : "NAO",
  "Valor_Acumulado" : 0,
  "Estimativa_Prêmio" : 0,
  "Acumulado_Mega_da_Virada" : 0
})
```

- MySQL:

```
INSERT INTO MEGASENA
  ("Concurso",
   "Data_Sorteio",
   "1a Dezena",
   "2a Dezena",
   "3a Dezena",
   "4a Dezena",
   "5a Dezena",
```

```
"6ª Dezena",
"Arrecadacao_Total",
"Ganhadores_Sena",
"Rateio_Sena",
"Ganhadores_Quina",
"Rateio_Quina",
"Ganhadores_Quadra",
"Rateio_Quadra",
"Acumulado",
"Valor_Acumulado",
"Estimativa_Prêmio",
"Acumulado_Mega_da_Virada" )
VALUES
(99999,
"07/09/2016",
1,
2,
3,
4,
5,
6,
0,
0,
0,
1,
"88000",
55,
"76200",
"NAO",
0,
0,
0);
```

Até o momento, sem nenhuma novidade, apenas uma diferença na sintaxe entre o MongoDB e o SQL relacional, certo? Pois bem, vamos mostrar algumas vantagens agora.

Além dos sorteios, vamos anotar o CPF de ganhador, no formato número do concurso e CPF.

- MongoDB:

```
db.ganhadores.insert({"Concurso":99999,  
                      "CPF":12345678900})
```

- MySQL:

```
create table ganhadores (Concurso double, CPF do  
uble);  
insert into ganhadores values (99999,12345678900  
);
```

Não é preciso criar a estrutura da tabela, o MongoDB faz isso automaticamente! E não é só isso. Imagine que agora precisamos adicionar o nome do ganhador também:

- MongoDB:

```
db.ganhadores.insert({"Concurso":99999,  
                      "CPF":12345678900,  
                      "Nome":"Coffin Joe"})
```

- MySQL:

```
alter table ganhadores add nome varchar(100);  
insert into ganhadores  
values (99999,12345678900,'Coffin Joe');
```

Perceba que, no MongoDB, além de não precisarmos criar a collection para armazenar os dados, não precisamos alterá-la também caso desejemos adicionar ou remover colunas.

Veja o exemplo a seguir:

```
connecting to: test  
> db.ganhadores.count()  
0  
> db.ganhadores.insert({"Concurso":99999,  
                           "CPF":12345678900})  
WriteResult({ "nInserted" : 1 })  
> db.ganhadores.insert({"Concurso":99999,  
                           "CPF":12345678900,  
                           "Nome":"Coffin Joe"})
```

```
WriteResult({ "nInserted" : 1 })
> db.ganhadores.find().pretty()
{
  "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 12345678900
}
{
  "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 12345678900,
  "Nome" : "Coffin Joe"
}
>
```

Inicialmente, verificamos com `count` que a collection está vazia (ou não existe). Em seguida, adicionamos dois registros: o primeiro com duas colunas e o segundo com três.

Finalmente, quando exibimos o resultado com `find`, percebemos que, mesmo que cada registro tenha uma estrutura diferente, eles são armazenados e exibidos na mesma collection sem nenhum problema. Essa é uma das grandes flexibilidades do MongoDB: sua estrutura se adapta à sua necessidade, e não o contrário, como o que normalmente acontece com os bancos de dados relacionais.

Para exemplificar o identificador único `_id`, inserimos três registros diferentes para mostrar que qualquer valor pode ser inserido, desde que seja único.

```
> db.valores.insert({"_id" : 111, "valor" : 1000})
WriteResult({ "nInserted" : 1 })
> db.valores.insert({"_id" : "importante", "valor" : 2000})
WriteResult({ "nInserted" : 1 })
> db.valores.insert({"valor" : 3000})
WriteResult({ "nInserted" : 1 })
> db.valores.find();
{"_id":111, "valor" : 1000 }
```

```
{"_id":"importante", "valor" : 2000 }
{"_id":ObjectId("57d092d3e741513365fdbb2d"), "valor" : 3000 }
```

Note que, no primeiro `insert`, inserimos um número; no segundo, um texto; e no terceiro, nada. Logo, o MongoDB cria implicitamente um valor único de forma automática.

3.6 ATUALIZAR REGISTROS

Para atualizar os registros de uma collection, a sintaxe é:

```
db.<nome-da-collection>.update(
  {<criterioDeBusca1>:<valor1>, ...},
  {<campoParaAtualizar1>:<novoValor1>, ...})
);
```

Ao contrário dos demais comandos, que são possíveis de se associarem com exemplos do SQL, o comando `update` apresenta um comportamento diferente de um banco relacional. Além de atualizar registros, com ele é possível também alterar a estrutura de uma collection e até remover colunas.

No exemplo a seguir, atualizamos o campo nome de um registro (buscando pelo campo `_id`):

```
>db.ganhadores.find(
  { "_id" : ObjectId("57d09065e741513365fdbb2c")})
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 12345678900,
  "Nome" : "Coffin Joe" }
>
>db.ganhadores.update(
  {"_id" : ObjectId("57d09065e741513365fdbb2c")},
  {"Nome":"Zé do caixão"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

O resultado mostra que um registro foi encontrado (`nMatched`) e um foi modificado (`nModified`). Entretanto, realizando a busca, temos como resultado a exibição apenas da coluna `_id` e `Nome` :

```
> db.ganhadores.find(  
{ "_id" : ObjectId("57d09065e741513365fdbb2c")})  
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),  
  "Nome:" : "Zé do caixão" }  
>
```

Antes do comando `update`, a linha possuía as colunas `Concurso` e `CPF`. O que aconteceu?

Este é o comportamento padrão do comando `update`: substituir as colunas informadas no comando pelas linhas encontradas. Se a coluna não foi especificada, ela será eliminada.

Comparando, temos:

- MongoDB:

```
db.ganhadores.update(  
  {"_id" : ObjectId("57d09065e741513365fdbb2c")},  
  {"Nome:" :"Zé do caixão"})
```

- MySQL:

```
alter table ganhadores drop column concurso;  
alter table ganhadores drop column cpf;  
update ganhadores  
set nome = 'Zé do caixão'  
where _id = "57d09065e741513365fdbb2c";
```

Existe, porém, uma opção que se assemelha ao comando SQL, que é uso do `set` no `update`:

```
db.<nome-da-collection>.update(  
  { <criterioDeBusca1>:<valor1>, ... },  
  { $set: { <campoParaAtualizar1>:<novoValor1>, ... } })
```

```
)
```

Executando os mesmos comandos, temos um resultado sem remover nenhuma coluna:

```
> db.ganhadores.find(  
{ "_id" : ObjectId("57d09065e741513365fdbb2c")})  
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),  
  "Concurso" : 99999,  
  "CPF" : 12345678900,  
  "Nome" : "Coffin Joe" }  
>  
> db.ganhadores.update(  
{ "_id" : ObjectId("57d09065e741513365fdbb2c")},  
  { $set: {"Nome":"Zé do caixão"} })  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
>  
> db.ganhadores.find(  
{ "_id" : ObjectId("57d09065e741513365fdbb2c")});  
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),  
  "Concurso" : 99999,  
  "CPF" : 12345678900,  
  "Nome" : "Zé do caixão" }  
>
```

Portanto, em comparação, temos:

- MongoDB:

```
db.ganhadores.update(  
  { "_id" : ObjectId("57d09065e741513365fdbb2c")},  
  { $set: {"Nome":"Zé do caixão"} })
```

- MySQL:

```
update ganhadores  
set nome = 'Zé do caixão'  
where _id = "57d09065e741513365fdbb2c";
```

Outro ponto do comando update que é importante saber é que, por padrão, ele atualiza apenas o primeiro registro que obedecer ao critério de busca especificado. Para que ele altere

todos os registros, é preciso adicionar mais um parâmetro booleano, o `multi`, que por padrão é `false`.

Por que existe essa opção? Quem nunca fez um `UPDATE` em uma base relacional e, esquecendo-se da cláusula `WHERE`, detonou todos os dados da tabela?

Pensando nisso, sem nenhuma cláusula, no MongoDB o padrão é atualizar um único registro / documento. Já nas bases relacionais, o padrão é atualizar todos os registros.

Neste primeiro exemplo, apenas uma linha é alterada:

```
> db.ganhadores.find()
{ "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 12345678900 }
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 12345678900,
  "Nome" : "Zé do caixão" }

>
>
> db.ganhadores.update({}, { $set: {"CPF": 555555555555 }});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.ganhadores.find();
{ "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 555555555555 }
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 12345678900,
  "Nome" : "Zé do caixão" }

>
```

Agora, alterando `multi` para `true`, todas as linhas são alteradas.

```
> db.ganhadores.update({},
```

```

{ $set: {"CPF": 555555555555 }}, {multi:true});
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 1 })
>
> db.ganhadores.find();
{ "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 555555555555 }
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 555555555555,
  "Nome" : "Zé do caixão" }
>
```

Comparando, temos:

- MongoDB:

```
db.ganhadores.update({},
{ $set: {"CPF": 555555555555 }}, {multi:true})
```

- MySQL:

```
update ganhadores
set cpf = 555555555555;
```

Outro parâmetro bem interessante é o `upsert`. Quantas vezes em nossos sistemas temos rotinas do tipo: *"Busca um registro. Se ele existir, atualiza; caso contrário, cadastrá"*? Sempre existe um *script*, uma *trigger*, ou uma rotina fora do banco de dados para fazer essa operação.

No MongoDB não é necessária nenhuma rotina para isso, basta apenas ativar o parâmetro `upsert`, como no exemplo a seguir. Inicialmente, temos um `update` comum, que não encontra o registro para alterar, e não altera nada (note que tanto o `nMatched` como o `nModified` têm valor zero):

```
> db.ganhadores.update({"Nome" : "Mula sem cabeça"}, 
  { $set: {"CPF": 3333333333 }}, 
  {multi:0, upsert:0});
```

```

WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
>
> db.ganhadores.find();
{ "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 55555555555 }
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 5555555555555,
  "Nome" : "Zé do caixão" }

```

O mesmo comando update com o parâmetro upsert ativo adicionou um novo registro (note o nUpserted com valor 1):

```

> db.ganhadores.update({"Nome" : "Mula sem cabeça"},
                        { $set: {"CPF": 33333333333 } },
                        {multi:0,upsert:1});
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : ObjectId("57d09a248688b893a04c3f50" )
})
>
> db.ganhadores.find();
{ "_id" : ObjectId("57d09058e741513365fdbb2b"),
  "Concurso" : 99999,
  "CPF" : 55555555555 }
{ "_id" : ObjectId("57d09065e741513365fdbb2c"),
  "Concurso" : 99999,
  "CPF" : 5555555555555,
  "Nome" : "Zé do caixão" }
{ "_id" : ObjectId("57d09a248688b893a04c3f50"),
  "Nome" : "Mula sem cabeça",
  "CPF" : 33333333333 }

```

Portanto, comparando, temos:

- MongoDB:

```
db.ganhadores.update({"Nome" : "Mula sem cabeça"
},
```

```
{ $set: {"CPF": 333333333333 },  
{multi:0,upsert:1});
```

- MySQL:

```
CREATE TRIGGER seNaoExisteCadastra  
BEFORE UPDATE ON ganhadores  
FOR EACH ROW  
BEGIN  
    IF NEW.nome not in (  
        select nome  
        From ganhadores A  
        where NEW.nome = A.nome  
    ) THEN  
        INSERT INTO ganhadores(nome)  
        VALUES (NEW.nome);  
    END IF;  
END;  
update ganhadores  
set cpf = 3333333333  
where nome='Mula sem cabeça';
```

3.7 REMOVER REGISTROS

Para remover registros, usamos o comando `remove` com a seguinte sintaxe:

```
db.<nome-da-collection>.remove(  
    { <criterioDeBusca1>:<valor1>, ... }  
)
```

Para remover todos os registros com CPF 333333333333 , fazemos:

```
> db.ganhadores.count()  
3  
> db.ganhadores.find({"CPF" : 333333333333}).count()  
1  
> db.ganhadores.remove({"CPF" : 333333333333})  
WriteResult({ "nRemoved" : 1 })  
> db.ganhadores.count()
```

Apenas um registro foi removido, o total count diminuiu de 3 para 2. Comparando com SQL relacional, temos:

- MongoDB:

```
db.ganhadores.remove({"CPF" : 3333333333})
```

- MySQL:

```
delete from ganhadores
where CPF=3333333333;
```

Para removermos todos os registros de uma collection, basta colocar uma condição vazia:

```
> db.ganhadores.count()
2
> db.ganhadores.remove({})
WriteResult({ "nRemoved" : 2 })
>
> db.ganhadores.count()
0
```

Note que o count inicial de 2 (igual ao nRemoved), e depois do comando tornou-se zero.

3.8 CRIAR E REMOVER COLLECTIONS

Não existe comando para criar uma collection, pois, ao adicionar um registro, automaticamente a collection é criada com a mesma estrutura.

Para remover uma collection, usamos o comando drop :

```
db.<nome-da-collection>.drop();
```

No exemplo a seguir, criamos uma collection e a removemos em seguida:

```
fb@cascao ~ > mongo
MongoDB shell version: 3.2.9
connecting to: test
> db.collectionNovaNaoPrecisaCriarAntes.insert(
  {"uma coluna":"só"})
WriteResult({ "nInserted" : 1 })
> db.collectionNovaNaoPrecisaCriarAntes.count()
1
> db.collectionNovaNaoPrecisaCriarAntes.find()
{ "_id" : ObjectId("57d09bf00e8c303b4eda456a"),
  "uma coluna" : "só" }
> db.collectionNovaNaoPrecisaCriarAntes.drop()
true
> exit
bye
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.drop();
```

- MySQL:

```
drop table megasena;
```

3.9 ALTERANDO UMA COLUNA DE UMA COLLECTION

Se for necessário remover uma coluna, a sintaxe do comando é:

```
db.<collection>.update( {},
  { $unset : { <campo>: 1 }},
  false,true);
```

O parâmetro `false` avisa que não é um `upsert`. E o parâmetro `true` é a confirmação para remover em todos os

documentos, e não apenas no primeiro. Veja um exemplo:

```
> db.messages.update( {},  
  { $unset : { titulo: 1 }},  
  false,true);  
WriteResult({ "nMatched" : 120477,  
  "nUpserted" : 0,  
  "nModified" : 120476 })
```

Se for necessário apenas alterar o nome, a sintaxe é semelhante:

```
db.<collection>.update( {},  
  { $rename :  
    { "<nome-da-coluna-atual>" : "<nome-da-coluna-novo>" }},  
  false,true);
```

Veja um exemplo:

```
db.messages.update( {},  
  { $rename :  
    { "mailboxx" : "mailbox" }},  
  false,true);  
WriteResult({ "nMatched" : 120477,  
  "nUpserted" : 0,  
  "nModified" : 120477 })
```

3.10 VALIDAÇÃO DOS DADOS

A partir da versão 3.2, foi adicionada a opção de adicionar um sistema de validação (`validator`) dentro de uma collection. No nosso exemplo, vamos adicionar uma validação que verifica se uma coluna `CPF` existe:

```
> db.ganhadores.find()  
> db.runCommand({  
  collMod:"ganhadores",  
  validator: {  
    "CPF":{$exists:true}  
  }})  
{ "ok" : 1 }
```

No cadastro de um registro com CPF , tudo ocorre normalmente:

```
> db.ganhadores.insert({"nome":"Sortudo","CPF":123456})  
WriteResult({ "nInserted" : 1 })
```

Ao tentarmos cadastrar um registro sem o CPF , o validador retorna um erro e não permite o cadastro:

```
> db.ganhadores.insert({"nome":"Sortudo sem CPF"})  
WriteResult({  
    "nInserted" : 0,  
    "writeError" : {  
        "code" : 121,  
        "errmsg" : "Document failed validation"  
    }  
})
```

Podemos adicionar uma validação com expressão regular, por exemplo, para validar somente e-mails do domínio boaglio.com :

```
> db.runCommand({ collMod:"ganhadores", validator:  
    { "CPF":{$exists:true},  
      "email": { $regex: /@boaglio\.com$/ }  
    }  
  }  
  { "ok" : 1 })
```

Tentando cadastrar um e-mail de outro domínio, o validador retorna erro:

```
> db.ganhadores.insert({"nome":"Sortudo","CPF":123,  
    "email":"email@gmail.com"})  
WriteResult({  
    "nInserted" : 0,  
    "writeError" : {  
        "code" : 121,  
        "errmsg" : "Document failed validation"  
    }  
})
```

No e-mail com domínio correto, sem problemas:

```
> db.ganhadores.insert({"nome":"Sortudo","CPF":123,
  "email":"email@boaglio.com"})
WriteResult({ "nInserted" : 1 })
```

Podemos também validar o tipo de dado que está entrando. Vamos adicionar uma validação para o campo `CPF` ser do tipo texto (string):

```
> db.runCommand({ collMod:"ganhadores", validator:
  { "CPF":{$exists:true,$type:"string"}, 
    "email": { $regex: /@boaglio\.com$/ }
  }
}, { "ok" : 1 })
```

Um registro válido é inserido sem problemas:

```
> db.ganhadores.insert({"nome":"Sortudo","CPF":"W123W","email":"e
mail@boaglio.com"})
WriteResult({ "nInserted" : 1 })
```

Já um registro com o campo `CPF` numérico retorna erro de validação:

```
> db.ganhadores.insert({"nome":"Sortudo","CPF":123,
  "email":"email@boaglio.com"})
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 121,
    "errmsg" : "Document failed validation"
  }
})
```

Existem outros tipos de dados para usar com `$type`. Para mais, consulte a documentação em:
<https://docs.mongodb.com/manual/reference/operator/query/type/>

3.11 MELHORANDO AS BUSCAS

Se desejarmos contar todos os sorteios de 2009, precisamos filtrar o campo `Data Sorteio` de maneira que só considere as datas que terminem com "2009". Isso em SQL seria o correspondente ao comando `LIKE`.

No MongoDB, para fazermos esse tipo de filtro, usamos expressões regulares. Esse tipo de consulta pode ser feito de duas maneiras:

```
db.<nome-da-collection>.find({ <campo>:/<texto-para-buscar>/})
```

Ou:

```
db.<nome-da-collection>.find(  
{<campo>:{$regex:<texto-para-buscar>}})
```

Veja um exemplo:

```
fb@cascao ~ > mongo  
MongoDB shell version: 3.2.9  
connecting to: test  
> db.megasena.find({"Data Sorteio":/2009/}).count()  
105  
> db.megasena.find({"Data Sorteio":{$regex:'2009'}}).count()  
105  
> exit  
bye
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.find({"Data Sorteio":/2009/}).count()  
)
```

- MySQL:

```
SELECT COUNT(*)  
FROM megasena  
WHERE "Data Sorteio" like '%2009';
```

Entretanto, se fizermos uma busca para contar quantas megasena s foram acumuladas, temos o resultado:

```
> db.megasena.find({"Acumulado":/SI/}).count()  
1409  
> db.megasena.find({"Acumulado":/si/}).count()  
0
```

Para fazer uma busca ignorando letras maiúsculas e minúsculas, a sintaxe é um pouco diferente:

```
db.<nome-da-collection>.find({ <campo>:/<texto-para-buscar>/i})
```

Ou:

```
db.<nome-da-collection>.find(  
{<campo>:{$regex:<texto-para-buscar>, $options:'i'}})
```

Veja um exemplo:

```
fb@cascao ~ > mongo  
MongoDB shell version: 3.2.9  
connecting to: test  
> db.megasena.find({"Acumulado":/si/i}).count()  
1409  
> db.megasena.find({  
"Acumulado":{$regex:'si', $options:'i'}}).count()  
1409  
> exit  
bye
```

A lista contém outros operadores de busca. Consulte na figura a seguir e no site oficial do MongoDB.

- **\$gt** — maior do que (*greater-than*)
- **\$gte** — igual ou maior do que (*greater-than or equal to*)
- **\$lt** — menor do que (*less-than*)
- **\$lte** — igual ou menor do que (*less-than or equal to*)

- `$ne` — não igual (*not equal*)
- `$in` — existe em uma lista
- `$nin` — não existe em uma lista
- `$all` — existe em todos elementos
- `$not` — traz o oposto da condição
- `$mod` — calcula o módulo
- `$exists` — verifica se o campo existe
- `$elemMatch` — compara elementos de array
- `$size` — compara tamanho de array

OPÇÕES DE BUSCA

✓: valor encontrado na busca

✗: valor não encontrado na busca

Operador	busca	documentos
\$gt, \$gte, \$lt, \$lte, \$ne	{Concurso : {\$lt:3}}	✓ {Concurso: 1} ✗ {Concurso: "hello"} ✗ {Concurso: 99}
\$in, \$nin	{Concurso : {\$in : [1, 2] }}	✓ {Concurso: 1} ✓ {Concurso: 2} ✗ {Concurso: 9}
\$all	{"1ª Dezena" : {\$all : [1,3]}}	✓ {"1ª Dezena": [1, 2,3,4,5]} ✗ {"1ª Dezena": [1,2]}
\$not	{Acumulado : {\$not : /sim/i}}	✓ {Acumulado:"NÃO"} ✗ {Acumulado:"SIM"}
\$mod	{Concurso : {\$mod : [100,3]}}	✓ {Concurso: 203} ✗ {Concurso: 222}
\$exists	{Valor_Acumulado: {\$exists: true}}	✓ {Valor_Acumulado: 0"} ✗ {"valor_acumulado" 871973]}
\$elemMatch	{sorteio:{\$elemMatch: {\$gte: 30, \$lt:40 }}}	✓ {sorteio:[31,32]} ✗ {sorteio:[20,31,32]}
\$size	{"sorteio":{\$size:3}}	✓ {"sorteio": [23,31, 32]} ✗ {"sorteio": [23,31, 32,44,45,51]}

Figura 3.5: Operadores de busca

MAIS EXEMPLOS

Na documentação oficial do MongoDB, há mais exemplos de comandos MongoDB comparando com SQL relacional:
http://info.mongodb.com/rs/mongodb/images/sql_to_mongo.pdf.

3.12 CAPPED COLLECTION

O MongoDB possui um recurso bem interessante, as *capped collections* (collections "tampadas"). Elas são collections com tamanhos predefinidos e com o seu conteúdo rotativo. Normalmente, em um banco de dados relacional, esse tipo de comportamento é feito manualmente ou através de uma aplicação.

A sintaxe para criar uma collection desse tipo é:

```
db.createCollection("<collection>",
  {capped: true, size: <tamanho-em-bytes>,
   max: <número-de-documentos>})
```

O tamanho deve ser no mínimo 4096 e, opcionalmente, podemos limitar o número de documentos com `max`. Vamos criar uma collection com o limite de dois documentos:

```
> db.createCollection("cacheDeDoisdocumentos",
  { capped: true, size: 4096,
    max: 2 })
{ "ok" : 1 }
```

Em seguida, inserimos quatro documentos:

```
> db.cacheDeDoisdocumentos.insert({ "nome": "teste 1" })
```

```
WriteResult({ "nInserted" : 1 })
> db.cacheDeDoisdocumentos.insert({"nome":"teste 2"})
WriteResult({ "nInserted" : 1 })
> db.cacheDeDoisdocumentos.insert({"nome":"teste 3"})
WriteResult({ "nInserted" : 1 })
> db.cacheDeDoisdocumentos.insert({"nome":"teste 4"})
WriteResult({ "nInserted" : 1 })
```

Ao consultar a collection, percebemos que apenas os dois últimos estão armazenados:

```
> db.cacheDeDoisdocumentos.count()
2
> db.cacheDeDoisdocumentos.find()
{ "_id" : ObjectId("57d09f45e05366ca97bcc91e"),
  "nome" : "teste 3" }
{ "_id" : ObjectId("57d09f4fe05366ca97bcc91f"),
  "nome" : "teste 4" }
```

Note que em nenhum momento ocorreu erro ao inserir um documento. Ao alcançar o limite da collection em tamanho ou número de documentos, o MongoDB automaticamente elimina os documentos para dar espaço aos novos.

3.13 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- operações de busca com o comando `find` ;
- operações de adição com o comando `insert` ;
- operações de alterar com o comando `update` ;
- operações de remover com o comando `remove` e `drop` ;
- validação de dados nas collections;
- criar `capped collections` .

No próximo capítulo, analisaremos como modelar o *schema* do

MongoDB da maneira correta.

CAPÍTULO 4

SCHEMA DESIGN

Quando se pensa em modelar um sistema em um banco de dados, sempre são consideradas as regras de normalização de dados, independente do sistema usado. A normalização evita a redundância de dados armazenados.

Entretanto, ela pode ser um problema quando for necessário consultar essas informações normalizadas e separadas em várias tabelas, e mostrá-las em um site em uma única página. Se as informações estão separadas em várias tabelas, isso possivelmente estará mais espalhado no disco rígido e, provavelmente, exigirá mais processamento da CPU para juntar tudo ao retornar para uma consulta.

Quando se trabalha com MongoDB, a primeira coisa para se considerar é como a aplicação precisa dos dados agrupados, para somente depois as collections serem organizadas. Em um modelo relacional, é comum separar tudo em tabelas, já no MongoDB separa-se por "entidades", em que todos dados necessários (ou quase) estão juntos.

Lembre-se de que, no MongoDB, não temos tabelas compostas por colunas que armazenam apenas um tipo de informação. Trabalhamos com documentos, que não têm esse tipo de limitação

em sua estrutura. É importante também saber que o MongoDB não suporta nenhum tipo de constraint de chave estrangeira (*foreign key*), pois ele espera que essa validação exista do lado da aplicação.

Além disso, o MongoDB também não suporta transação, já que a ideia é que, em vez de termos uma transação que envolva várias tabelas, a sua aplicação tenha um documento que seja armazenado em uma collection.

- MongoDB:

```
db.seriados.insert({  
    "_id":4,  
    "nome":"Chaves",  
    "personagens":[  
        "Seu Barriga",  
        "Quico",  
        "Chaves",  
        "Chiquinha",  
        "Nhonho",  
        "Dona Florinda"]})
```

- SQL Relacional:

```
INSERT INTO SERIADO(ID,NOME)  
VALUES (4, 'Chaves');  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (55, 'Seu Barriga', 4);  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (56, 'Quico', 4);  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (57, 'Chaves', 4);  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (58, 'Chiquinha', 4);  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (59, 'Nhonho', 4);  
INSERT INTO PERSONAGEM(ID,NOME,SERIADO_ID)  
VALUES (60, 'Dona Florinda', 4);  
COMMIT;
```

Analizando o exemplo das tabelas SERIADO e PERSONAGEM desse comparativo, percebemos que separamos essas informações apenas pela limitação do banco de dados relacional. Entretanto, existem algumas maneiras de relacionar as collections, tendo um funcionamento comparado à chave estrangeira do banco de dados relacional.

Apesar de o MongoDB possuir opções de índices para melhorar a sua busca, nada supera um bom *schema design*, com collections que refletem exatamente o que a aplicação espera, agrupando as informações da maneira mais adequada.

4.1 RELACIONANDO UMA COLLECTION PARA MUITAS

Existem duas maneiras de representar esse tipo de relacionamento. Vamos imaginar um exemplo de um sistema de venda de livros, com vários comentários para cada livro.

Certamente, pensando mais do jeito relacional, é possível dividir as informações em duas collections, inicialmente cadastrando um livro:

```
db.livros.insert({  
    _id:"A menina do Vale",  
    autor: "Bel Pesce",  
    tags: ["empreendedorismo","inspiração","virar a mesa" ]});
```

Depois, cadastrando dois comentários para este livro:

```
db.comentarios.insert({  
    livro_id: "A menina do Vale",  
    autor: "Amit Garg",  
    texto: "A Menina do Vale tem o poder de energizar qualquer  
    pessoa. É um livro sobre ação e mostra que qualquer
```

```
pessoa nesse mundo pode realizar os seus sonhos."});  
  
db.comentarios.insert({  
    livro_id: "A menina do Vale",  
    autor: "Eduardo Lyra",  
    texto: "Pare tudo e leia A Menina do Vale agora mesmo. Te  
        garanto que você vai aprender demais com essa  
        leitura e vai se surpreender com o quanto é capaz  
        de fazer."});
```

Essa abordagem, entretanto, não tem nenhuma vantagem para o MongoDB, pois separa as informações em locais distintos, exigindo mais CPU e operações em disco quando a aplicação necessitar exibir tudo de uma vez. Nesse caso, o ideal era embutir as informações de comentários dentro de cada livro, dessa maneira:

```
db.livros.insert({  
    _id:"A menina do Vale",  
    autor: "Bel Pesce",  
    tags: ["empreendedorismo","inspiração","virar a mesa" ],  
    comentarios: [  
        {  
            autor: "Amit Garg",  
            texto: "A Menina do Vale tem o poder de energizar qualquer  
                pessoa. É um livro sobre ação e mostra que qualquer  
                pessoa nesse mundo pode realizar os seus sonhos."  
        },  
        {  
            autor: "Eduardo Lyra",  
            texto: "Pare tudo e leia A Menina do Vale agora mesmo. Te  
                garanto que você vai aprender demais com essa  
                leitura e vai se surpreender com o quanto é capaz  
                de fazer."  
        }  
    ]  
})
```

Perceba que sempre a forma como a aplicação necessita das informações é essencial para organizar as suas collections. Ela é a chave de uma boa performance de seu sistema.

4.2 RELACIONANDO MUITAS COLLECTION PARA MUITAS

No MongoDB, também é possível representar o relacionamento de muitos para muitos. Vamos exemplificar um cenário em que temos uma loja de livros com várias categorias, e uma categoria tenha vários livros.

Inicialmente, cadastramos um livro referenciando três categorias:

```
db.livros.insert({  
  _id:"A menina do Vale",  
  autor: "Bel Pesce",  
  categorias: ["empreendedorismo","inspiração","virar a mesa"]});
```

Em seguida, cadastramos uma categoria referenciando dois livros:

```
db.categorias.insert({nome: "empreendedorismo",  
  lista_de_livros:  
    ["A menina do Vale",  
     "28 Mentes Que Mudaram o Mundo"]  
});
```

Note que, para representar a mesma informação em um banco de dados relacional, seria necessária uma tabela intermediária. Provavelmente teríamos ao final as tabelas: LIVRO , CATEGORIA e LIVRO_CATEGORIA .

4.3 TUDO EM UMA COLLECTION

Se tudo ficar em uma collection, há a vantagem de termos as informações de maneira mais intuitiva e melhor performance. Entretanto, podemos ter algumas desvantagens, como complicar

demais ao fazer uma busca, principalmente para obter resultados parciais.

Além disso, cada registro/ documento possui um limite de 16 Mb. Saiba mais sobre os limites do MongoDB no *Apêndice C — Perguntas e respostas*.

Em uma situação com vários documentos dentro de documentos (*embedded document*), a criação de índices fica um pouco mais complexa (consulte o *capítulo 10 — Aumentando a performance*).

4.4 SCHEMA DESIGN NA PRÁTICA

Vamos analisar um caso real: o banco de dados utilizado é uma simplificação do site Internet Movie Database (IMDB). Através do software JMDB (<http://www.jmdb.de/>), é possível baixar a base de dados localmente no seu computador em MySQL, com milhares de filmes cadastrados.

Protótipo

Nosso protótipo do sistema "Meus filmes" tem como objetivo exibir filmes e atores. Na tela inicial, como vemos na figura a seguir, é possível fazer uma busca por somente filmes, somente atores ou ambos.



Meus Filmes

O que procura ?

- Só filmes
- Só atores
- Tudo

Buscar

Resultado da busca

Matrix Reloaded

The Matrix

Figura 4.1: Protótipo do Meus filmes — página inicial

Saindo da tela inicial, podemos detalhar um ator, como é exibido na figura seguinte. Além do nome do ator, é exibida a lista de filmes de que ele participa.



Reeves, Keanu

Filmes
1990 Bill & Ted's Excellent Adventures
1999 The Matrix

Figura 4.2: Protótipo do Meus filmes — detalhe de ator

Podemos detalhar um filme, como é exibido na figura adiante. Além do nome do filme, são exibidas a quantidade de votos, a nota média e a lista de categorias de filme, diretores e atores.



Filme

The Matrix (1999)

Votos: 849782

Nota: 8.7

Action

Sci-Fi

Diretores

Wachowski, Andy

Wachowski, Lana

Atores

Reeves, Keanu

Fishburne, Laurence

Figura 4.3: Protótipo do Meus filmes — detalhe de filme

4.5 SISTEMA MEUS FILMES RELACIONAL

O sistema relacional que exibe essas informações está dividido em 7 tabelas:

- `actors` — atores
- `movies` — filmes
- `directors` — diretores
- `genres` — gêneros dos filmes
- `movies2actors` — tabela associativa de filmes e atores
- `ratings` — notas dos filmes

- movies2directors — tabela associativa de filmes e diretores

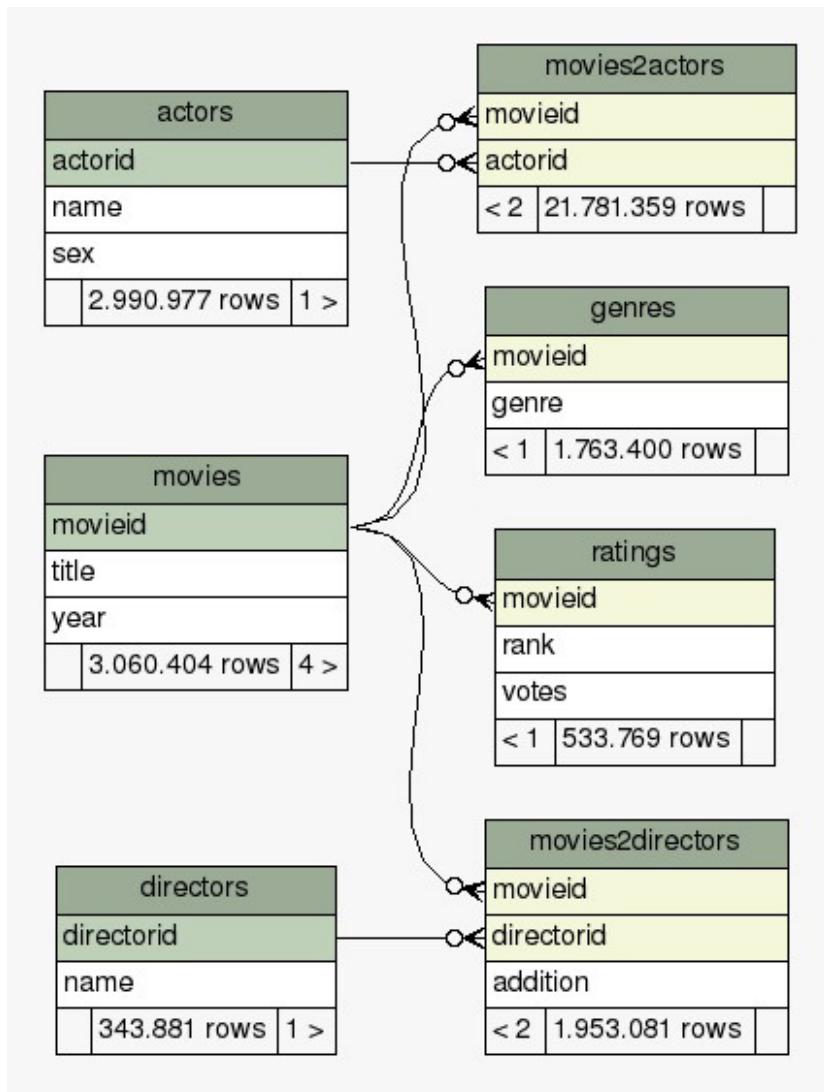


Figura 4.4: Modelo relacional de meus filmes

Como todo sistema relacional, esse modelo foi criado levando em conta apenas as formas normais, sem levar em consideração como a aplicação foi montada.

4.6 SISTEMA MEUS FILMES NO MONGODB

No MongoDB, fazemos uma análise inicial da aplicação e, a partir dela, montamos o nosso *schema* de collections. Percebemos que a aplicação como um todo exige:

- busca por atores;
- busca por filmes;
- exibir todas informações de um ator;
- exibir todas informações de um filme.

Portanto, percebemos que tudo dentro de uma mesma collection é a melhor solução, já que a aplicação não faz nenhuma consulta complexa que necessite separar as informações. O exemplo de registro do nosso protótipo ficaria dessa maneira:

```
db.filmes.insert({  
  _id:2719007,  
  title: "The Matrix",  
  year: 1999,  
  rank: 8.7,  
  votes: 849782,  
  genres: ["Action", "Sci-Fi" ],  
  directors: ["Wachowski, Andy", "Wachowski, Lana"],  
  actors: [  
    {"name": "Arahanga, Julian", sex:"M"},  
    {"name": "Aston, David (I)", sex:"M"},  
    {"name": "Ball, Jeremy (I)", sex:"M"},  
    {"name": "Butcher, Michael (I)", sex:"M"},  
    {"name": "Chong, Marcus", sex:"M"},  
    {"name": "Dodd, Steve", sex:"M"},  
    {"name": "Doran, Matt (I)", sex:"M"},  
    {"name": "Fishburne, Laurence", sex:"M"},
```

```
{"name": "Goddard, Paul (I)", sex: "M"},  
 {"name": "Gray, Marc Aden", sex: "M"},  
 {"name": "Harbach, Nigel", sex: "M"},  
 {"name": "Lawrence, Harry (I)", sex: "M"},  
 {"name": "Ledger, Bernard", sex: "M"},  
 {"name": "O'Connor, David (I)", sex: "M"},  
 {"name": "Pantoliano, Joe", sex: "M"},  
 {"name": "Parker, Anthony Ray", sex: "M"},  
 {"name": "Pattinson, Chris", sex: "M"},  
 {"name": "Quinton, Luke", sex: "M"},  
 {"name": "Reeves, Keanu", sex: "M"},  
 {"name": "Simper, Robert", sex: "M"},  
 {"name": "Taylor, Robert (VII)", sex: "M"},  
 {"name": "Weaving, Hugo", sex: "M"},  
 {"name": "White, Adrynn", sex: "M"},  
 {"name": "Witt, Rowan", sex: "M"},  
 {"name": "Woodward, Lawrence", sex: "M"},  
 {"name": "Young, Bill (I)", sex: "M"},  
 {"name": "Brown, Tamara (I)", sex: "F"},  
 {"name": "Foster, Gloria (I)", sex: "F"},  
 {"name": "Gordon, Deni", sex: "F"},  
 {"name": "Johnson, Fiona (I)", sex: "F"},  
 {"name": "McClory, Belinda", sex: "F"},  
 {"name": "Morrison, Rana", sex: "F"},  
 {"name": "Moss, Carrie-Anne", sex: "F"},  
 {"name": "Nicodemou, Ada", sex: "F"},  
 {"name": "Pender, Janaya", sex: "F"},  
 {"name": "Tjen, Natalie", sex: "F"},  
 {"name": "Witt, Eleanor", sex: "F"}  
 ]})
```

4.7 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- avaliar a aplicação antes de montar o *schema*;
- representar collections em relacionamento um para muitos;
- representar collections em relacionamento muitos para muitos.

No próximo capítulo, veremos como os sistemas interagem com o MongoDB, com exemplos práticos em várias linguagens.

CAPÍTULO 5

CONVERSANDO COM MONGODB

O que é um repositório de dados sem uma aplicação para conversar com ele? Neste capítulo, veremos como diversas linguagens de programação se comunicam com o MongoDB.

Está fora do escopo deste livro se aprofundar nas linguagens ou nas configurações de ambiente. Aqui apenas citaremos as fontes para instalação e focaremos mais no exemplo da linguagem utilizada.

Os exemplos também se destinam ao aprendizado da integração da linguagem com o MongoDB, e não na melhor prática da linguagem em si em um sistema.

5.1 O SISTEMA DE SERIADOS

O sistema usado como exemplo é composto de um cadastro de seriados, usando apenas uma collection. Por meio da aplicação, é possível fazer a operação de buscar, adicionar, remover e alterar documentos (registros).

O exemplo a seguir ilustra o documento de um seriado,

composto de alguns campos simples e um array:

```
{  
    "_id" : ObjectId("57d5d1a8a73374d64eebf4e9"),  
    "nome" : "Breaking Bad",  
    "personagens" : [  
        "Walter White",  
        "Skyler White",  
        "Jesse Pinkman",  
        "Hank Schrader",  
        "Marie Schrader ",  
        "Saul Goodman"  
    ]  
}
```

5.2 SERIADOS EM PHP

Ambiente

Os fontes deste projeto estão disponíveis em <https://github.com/boaglio/mongodb-php-casadocodigo>.

Prerrequisitos:

- Servidor HTTP Apache – <http://httpd.apache.org>
- PHP – <http://php.net>
- PHP MongoDB database driver – <http://pecl.php.net/package/mongodb>

A instalação correta exibirá nas informações do PHP o driver do MongoDB instalado, conforme a figura:

The screenshot shows a browser window titled "phpinfo" with the URL "127.0.0.1/seriados/info.php". The main content is a table titled "mongodb" under the heading "mongodb support". It contains four rows with the following data:

	enabled
mongodb version	1.1.8
mongodb stability	stable
libmongoc version	1.3.5

Below this is another table with three columns: "Directive", "Local Value", and "Master Value". It has one row:

Directive	Local Value	Master Value
mongodb.debug	no value	no value

Figura 5.1: phpInfo com MongoDB driver

SOBRE O PHP

Em 1994, foi criada a linguagem de programação PHP (*Personal Home Page*) focada na criação de conteúdo dinâmico na Internet. Hoje ela é usada em muitas aplicações, entre elas a Wikipedia, Facebook e Wordpress.

A lista de seriados

A lista de seriados está no arquivo `index.php` e inicia-se criando uma variável `conexao`, que recebe o driver de acesso ao MongoDB e já define o nome do servidor e sua porta:

```
<?php  
$conexao = new MongoDB\Driver\Manager("mongodb://localhost:27017");
```

Em seguida, criamos uma variável de busca (query):

```
$query = new MongoDB\Driver\Query([]);
```

Finalmente, usamos uma variável `cursor` que armazenará o resultado da busca de todos os documentos da collection com o

método `find`:

```
$cursor = $conexao->executeQuery("test.seriados",$query);
```

Depois, os documentos são exibidos na tela através de um loop da variável `cursor`, que devolve uma lista de documentos, tratados um por um:

```
<?php  
foreach ($cursor as $documento) {
```

Os valores do documento compõem um array na variável `documento`. São exibidos apenas os valores `_id` para chamar a página de alterar os valores e o `nome` para exibir na tela.

```
echo "<a class=\"alert alert-success\" href=\"detalhe.php?id=".  
=".  
$documento->_id."\">".  
$documento->nome."</a>";  
}  
?>
```

O resultado obtido é exibido na figura:



Meus seriados preferidos

Chaves

Carga Pesada

Breaking Bad

novo seriado

Figura 5.2: Lista de seriados em PHP

Um novo seriado

Os valores de um novo seriado são enviados em um formulário HTML via `POST` e capturados no PHP através da variável `$_POST`.

O cadastro de seriados está no arquivo `novo.php`. Inicialmente, abrimos a conexão no MongoDB e especificamos que vai ocorrer o processo de escrita na collection com a variável `bulk` desejada:

```
$conexao = new MongoDB\Driver\Manager("mongodb://localhost:27017");
$bulk = new MongoDB\Driver\BulkWrite;
```

Em seguida, montamos o documento para cadastrar. Primeiro montamos a lista de personagens dentro da variável `personagens`:

```
$personagens =array($_POST['personagem1'],$_POST['personagem2'],
                     $_POST['personagem3'],$_POST['personagem4'],
                     $_POST['personagem5'],$_POST['personagem6'])
;
```

Depois o documento final é montado, composto pelos campos `ID`, `nome` e a lista de personagens:

```
$documento = array("_id" => new MongoDB\BSON\ObjectId,
                  "nome" => $_POST['nome'],
                  "personagens" => $personagens);
```

Finalmente, o documento é inserido pelo método `insert` e, com o `executeBulkWrite`, definimos a collection de destino:

```
$bulk->insert($documento);
$conexao->executeBulkWrite('test.seriados', $bulk);
```

O resultado obtido é exibido na figura:



Meus seriados preferidos

Novo seriado

Nome	Breaking Bad
Personagem 1	Walter White
Personagem 2	Skyler White
Personagem 3	Jesse Pinkman
Personagem 4	Hank Schrader
Personagem 5	Marie Schrader
Personagem 6	Saul Goodman

cadastrar !

lista de seriados

Figura 5.3: Cadastro de novo seriado em PHP

Alterar um seriado

O identificador único da collection de seriados (`_id`) é enviado como parâmetro via `GET` ao arquivo `detalhe.php` , e capturado no PHP pela variável `$_GET` .

A página de detalhe do seriado inicia abrindo uma conexão no

MongoDB da mesma maneira que as páginas anteriores:

```
$conexao = new MongoDB\Driver\Manager("mongodb://localhost:27017");
$bulk = new MongoDB\Driver\BulkWrite;
```

Em seguida, faremos a busca do documento usando o mesmo método `find`, porém com o parâmetro `_id`. Nesse caso, como a busca é feita pelo identificador único, é preciso criar um objeto do tipo `MongoId`:

```
$id = $_GET['id'];
$query = new MongoDB\Driver\Query(['_id' => new \MongoDB\BSON\ObjectID($id)]);
$cursor = $conexao->executeQuery("test.seriados",$query);
```

Finalmente, o registro é atribuído à variável `documento`:

```
foreach ($cursor as $documento) {}
```

A opção de alteração de documento é semelhante à de cadastro. Inicialmente, o documento é montado na variável `documento`:

```
$personagens =array($_POST['personagem1'],
$_POST['personagem2'],$_POST['personagem3'],
$_POST['personagem4'],$_POST['personagem5'],
$_POST['personagem6']);
```

Usamos o método `update` para alterar o documento de maneira semelhante ao `insert`:

```
$bulk = new MongoDB\Driver\BulkWrite;
$bulk->update(['_id' => new MongoDB\BSON\ObjectID($id)],
[ "nome" => $_POST['nome'], "personagens" => $personagens]);
$conexao->executeBulkWrite('test.seriados', $bulk);
```

A opção de remover o documento é semelhante também, usando o método `remove` e enviando como parâmetro o

identificador único:

```
$bulk = new MongoDB\Driver\BulkWrite;  
$bulk->delete(['_id' => new MongoDB\BSON\ObjectID($id)]);  
$conexao->executeBulkWrite('test.seriados', $bulk);
```

O resultado obtido da tela de alteração é exibido na figura:



Meus seriados preferidos

Alterar seriado

Nome	Chaves
Personagem 1	Seu Barriga
Personagem 2	Quico
Personagem 3	Chaves
Personagem 4	Chiquinha
Personagem 5	Nhonho
Personagem 6	Dona Florinda

alterar !

remover !

lista de seriados

Figura 5.4: Alterar um seriado em PHP

5.3 JAVA

SOBRE O JAVA

Em 1995, foi criada a linguagem de programação focada na criação de aplicações para diversas plataformas: desktop, servidor e mobile. Hoje ela é usada em diversas soluções, de ERPs, sites de bancos, celulares e até eletrodomésticos.

Existem várias opções para trabalhar com Java e MongoDB, mas vamos nos focar nas duas principais.

Driver oficial

Os códigos-fontes deste projeto estão disponíveis em <https://github.com/boaglio/mongodb-javapuro-casadocodigo>.

Os pré-requisitos são:

- Servidor Apache Tomcat — <http://tomcat.apache.org>
- Java — <http://www.oracle.com/technetwork/java>
- Java MongoDB database driver — <http://www.mongodb.org>

Em nosso sistema de seriados, a classe `SeriadosDAO` concentra as rotinas que acessam o banco de dados. A conexão com o MongoDB é controlada pela classe `MongoClient`, como no exemplo:

```
public Mongo mongo() throws Exception {  
  
    MongoClient mongoClient = new MongoClient(  
        new ServerAddress("localhost", 27017)  
    );
```

```
    return mongoClient;
}
```

A busca de dados é feita através de um loop em um cursor `MongoCursor` (antigo `DBCursor`), que retorna uma linha/documento `Document` (antigo `DBObject`).

```
public List<Seriado> findAll() {

    List<Seriado> seriados = new ArrayList<Seriado>();
    MongoCursor<Document> cursor = seriadosCollection.find().iterator();
    while (cursor.hasNext()) {
        Document resultElement = cursor.next();
```

Esse objeto do tipo `Document` é um `Map` de objetos, e depois convertido para um objeto do tipo `Seriado`:

```
    Seriado seriado = new Seriado();
    seriado.setId((ObjectId) resultElement.get("_id"));
    seriado.setNome((String) resultElement.get("nome"));
    seriado.setPersonagens((List<String>) resultElement.get("personagens"));

    seriados.add(seriado);

    System.out.println("Seriado lido = " + seriado);
}
return seriados;
}
```

No final, temos uma lista de objetos do tipo `Seriado` retornados.

Pelo sistema em Java, a página `index.jsp` cria uma instância da classe `SeriadosDAO` pela chamada:

```
<jsp:useBean
    id="dao"
    class="com.boaglio.casadocodigo.mongodb.SeriadosDAO" />
```

Depois, é montada a lista com os nomes de seriados:

```
<c:forEach var="seriado" items="<%= dao.findAll() %>">
    <a class="alert alert-success"
        href="detalhe.jsp?id=<c:out value="${seriado.id}" />">
        <c:out value="${seriado.nome}" />
    </a>
</c:forEach>
```

Um novo seriado

Os valores de um novo seriado são enviados em um formulário HTML via POST e capturados no JSP por meio da variável request . O cadastro de seriados está no arquivo novo.jsp . Criamos uma instância da classe SeriadoDAO , e depois verificamos se o conteúdo de nome existe, para iniciar uma inclusão:

```
String nome = request.getParameter("nome");
SeriadosDAO dao = new SeriadosDAO();

if ( nome!=null && nome.length()>0 ) {
```

Em seguida, recuperamos os valores restantes enviados pelo formulário e atribuímos a um objeto do tipo `Seriado`:

```
Seriado seriadoNovo = new Seriado();
seriadoNovo.setNome(nome);

List<String> personagens = new ArrayList<String>();
personagens.add(request.getParameter("personagem1"));
personagens.add(request.getParameter("personagem2"));
personagens.add(request.getParameter("personagem3"));
personagens.add(request.getParameter("personagem4"));
personagens.add(request.getParameter("personagem5"));
personagens.add(request.getParameter("personagem6"));
seriadoNovo.setPersonagens(personagens);
```

Finalmente, inserimos o novo seriado no MongoDB chamando

o método `insert` criado em `SeriadoDAO` :

```
dao.insert(seriadoNovo);
```

Alterar um seriado

Os valores de um novo seriado são enviados em um formulário HTML via `POST` e capturados no JSP através da variável `request`.

Na alteração, recebemos o parâmetro `id` para buscar os dados do seriado para exibir na página, usando o método `findById`:

```
String id = request.getParameter("id");  
  
SeriadosDAO dao = new SeriadosDAO();  
Seriado seriado = dao.findById(id);
```

Ajustamos o identificador único do seriado (`objectId`) para o parâmetro `id` recebido na variável `seriadoParaAlterar`:

```
Seriado seriadoParaAlterar = new Seriado();  
seriadoParaAlterar.setId(new ObjectId(id));
```

Depois, atribuímos os valores recebidos do formulário para a variável `seriadoParaAlterar`:

```
seriadoParaAlterar.setNome( request.getParameter("nome") );  
  
List<String> personagens = new ArrayList<String>();  
personagens.add(request.getParameter("personagem1"));  
personagens.add(request.getParameter("personagem2"));  
personagens.add(request.getParameter("personagem3"));  
personagens.add(request.getParameter("personagem4"));  
personagens.add(request.getParameter("personagem5"));  
personagens.add(request.getParameter("personagem6"));  
seriadoParaAlterar.setPersonagens(personagens);
```

Finalmente, chamamos o método `update` criado em

SeriadoDAO :

```
dao.update(seriadoParaAlterar);
```

Se for escolhida a opção de remover o seriado, usamos o método remove :

```
if(opt.equals("remover"))
{
    dao.remove(id);
}
```

Spring Data

SOBRE O SPRING FRAMEWORK

Em 2002, foi criado um framework totalmente baseado em injeção de dependências como alternativa aos sistemas em EJB. Hoje o framework evoluiu bastante e possui diversos módulos, desde segurança, web services até de mobile. Para aprofundar no uso Spring, procure pelo livro *Vire o jogo com Spring Framework* (<http://www.casadocodigo.com.br/products/livro-spring-framework>).

Saber como usar o driver oficial é a melhor forma de aprender todo o poder de acesso que uma aplicação Java pode ter em utilizar o MongoDB. Entretanto, quando precisamos de produtividade, partimos para um framework. Uma excelente opção de abstração do driver do MongoDB é o Spring Data, que facilita alguns passos no uso do MongoDB.

No exemplo disponível em <https://github.com/boaglio/mongodb-java-springdata-casadocodigo>, temos uma versão da aplicação de seriados com o MongoDB e Spring Data. Ela usa também outros módulos do Spring, como o Spring MVC, mas eles não estão configurados da forma mais adequada para uso em produção. O foco dessa aplicação é mostrar a integração do sistema ao MongoDB através do Spring Data.

A parte de tela (JSP) é igual ao exemplo anterior, a diferença é a facilidade em manipular os dados. Inicialmente, temos a mesma classe *POJO* *Seriado* usada anteriormente, mas com uma anotação que indica o nome da collection do MongoDB a que essa classe pertence:

```
@Document(collection = "seriados")
public class Seriado implements Serializable {
```

Em seguida, temos uma classe que contém as configurações de conexão ao banco de dados, que deve ser uma classe filha de *AbstractMongoConfiguration* :

```
@Configuration
public class SpringMongoConfig extends
AbstractMongoConfiguration {
```

Assim como no exemplo anterior, a conexão ao banco de dados retorna um objeto do tipo *MongoClient* , e devemos sobrescrever o método *mongo* e colocar os valores adequados:

```
private static final String SERVIDOR_MONGODB = "127.0.0.1";

@Override
@Bean
public Mongo mongo() throws Exception {
    return new MongoClient(SERVIDOR_MONGODB);
}
```

Para manipular os dados, semelhante ao comportamento da classe `SeriadoDAO` do exemplo anterior, temos agora a `SeriadoRepository`, que possui os mesmos métodos, porém com algumas diferenças. A classe recebe uma anotação de ser um repositório de dados do Spring (`Repository`) e tem um atributo do tipo `MongoTemplate`:

```
@Repository  
public class SeriadosRepository {  
  
    @Autowired  
    private MongoTemplate mongoTemplate;
```

Com esse atributo, fazemos as operações de acesso ao MongoDB com bem menos código que no exemplo anterior.

Para retornar a lista de todos os seriados, usamos apenas o método `findAll` e informamos o tipo de dado que será retornando (`Seriado.class`):

```
List<Seriado> seriados = new ArrayList<Seriado>();  
seriados = mongoTemplate.findAll(Seriado.class);  
return seriados;
```

Dessa maneira, o Spring Data se conecta ao banco de dados, acessa a collection `seriados` especificada na classe `Seriado` (em `@Document`), e depois retorna a lista de registros, fazendo automaticamente a conversão do documento JSON do MongoDB para uma lista de instâncias da classe `Seriado`. Sim, com certeza essa é uma maneira bem produtiva de se trabalhar!

As outras operações também são bem mais simples, como a busca pela chave, em que criamos um critério de busca pelo `id` e usamos o método `findOne`, que retorna apenas um documento, nesse caso do tipo `Seriado`. Da mesma maneira como no

`findAll` , a conversão automática do documento JSON para a instância da classe `Seriado` também acontece.

```
Seriado seriado = new Seriado();
Query queryDeBuscaPorID =
    new Query(Criteria.where("id").is(id));
seriado =
    mongoTemplate.findOne(queryDeBuscaPorID,Seriado.class);
return seriado;
```

As operações de cadastrar/ atualizar também são bem simples. Passamos o objeto para gravar e ele é automaticamente convertido:

```
// cadastrar
mongoTemplate.insert(seriado);

// alterar
mongoTemplate.save(seriado);
```

Para remover um seriado pelo `id` , criamos uma instância do tipo `Seriado` e colocamos no atributo `id` o valor de `new Object(id)` :

```
Seriado seriadoParaRemover = new Seriado();
seriadoParaRemover.setId(new ObjectId(id));
mongoTemplate.remove(seriadoParaRemover);
```

Outras opções

Uma opção interessante é o *Jongo* (<http://jongo.org/>), que tem como objetivo oferecer comandos no Java semelhantes aos do Mongo shell. Uma busca no Mongo shell é feita dessa maneira:

```
db.seriados.find({nome: "Chaves"})
```

Usando o Jongo, seria:

```
seriados.find("{nome:'Chaves'}").as(Seriado.class)
```

A alternativa semelhante ao Spring Data é o Morphia (da MongoDB — <https://github.com/mongodb/morphia>):

```
Morphia morphia = new Morphia();
Datastore ds = morphia.createDatastore("test");
List<Seriado> seriados = ds.find(Seriado.class).asList();
```

Por último, e não menos importante, é a *Hibernate OGM* (Hibernate Object/Grid Mapper — <http://ogm.hibernate.org>), que no momento possui uma versão um pouco limitada do MongoDB, mas totalmente funcional.

5.4 PLAY FRAMEWORK

SOBRE O PLAY FRAMEWORK

Em 2007, foi criado um framework para aplicações web no padrão MVC, visando à produtividade com convenção sobre configuração. A sua versão 2 foi reescrita inteiramente em Scala e oferece recursos interessantes dessa linguagem para os sistemas: é totalmente RESTful, integrada com JUnit e Selenium, possui I/O assíncrono e arquitetura modular. Para aprofundar no uso do Play Framework, procure pelo livro *Play Framework* (<http://www.casadocodigo.com.br/products/livro-play-framework-java>).

Os códigos-fontes do sistema estão disponíveis em <https://github.com/boaglio/play2-casadocodigo>. O Play Framework possui alguns templates de exemplos com MongoDB:

```
$ activator list-templates | grep mongo
play-2.4-crud-with-reactive-mongo
activator-play-autosource-reactivemongo
play-authenticate-mongo
play-mongo-knockout
play-reactive-mongo
play-reactive-mongo-db
play-reactivemongo-polymer
play-tepkin-mongo-example
Play2.3-Spring-PlayAuthenticate-deadbolt2-and-mongo-with-morphi
a
playing-reactive-mongo
scalatra-mongodb-seed
```

Vamos criar um exemplo mais simples.

A aplicação em Play tem quase tudo o que é preciso para fazer um sistema completo. Precisamos apenas adicionar uma dependência do módulo para conectar ao MongoDB no arquivo `build.sbt`:

```
libraryDependencies +=
  "net.vz.mongodb.jackson" %
  "play-mongo-jackson-mapper_2.10" %
  "1.1.0"
```

No arquivo de configurações `application.conf`, adicionamos duas linhas:

```
mongodb.servers="127.0.0.1:27017"
mongodb.database="test"
```

O mapeamento da `collection` é feito na classe `Seriado`:

```
private static
JacksonDBCollection<Seriado, String> collection =
MongoDB.getCollection("seriados", Seriado.class, String.class);
```

Para buscar todos os seriados, chamamos o método `find` da variável `collection` declarada:

```
public static List<Seriado> all() {  
    return Seriado.collection.find().toArray();  
}
```

A busca pelo id também é feita pela mesma variável collection :

```
public static Seriado findById(String id) {  
    Seriado seriado = Seriado.collection.findOneById(id);  
    return seriado;  
}
```

As operações de cadastro, alteração e remoção também são semelhantes:

```
// cadastro  
Seriado.collection.insert(seriado);  
  
// alteração  
Seriado.collection.save(seriado);  
  
// remoção  
Seriado.collection.removeById(id);
```

As telas são baseadas no template do Play. A página inicial que exibe os seriados recebe a lista como parâmetro:

```
@(seriados: List[Seriado])  
  
@import helper._  
  
@main("Seriados") {  
  
@for(seriado <- seriados) {  
  
<a class="alert alert-success" href="@{seriado.id}">  
    @seriado.nome  
</a>  
}  
}
```

5.5 RUBY ON RAILS

SOBRE O RUBY ON RAILS

Em 2004, foi criado um framework em Ruby para acelerar o desenvolvimento de aplicações Web MVC. Para aprofundar no uso do Ruby on Rails, procure pelo livro *Ruby on Rails: coloque sua aplicação web nos trilhos* (<http://www.casadocodigo.com.br/products/livro-ruby-on-rails>).

Os códigos-fontes do sistema estão disponíveis em <https://github.com/boaglio/mongodb-rails-casadocodigo>. O nosso exemplo usará Mongoid, o driver escrito em Ruby (<http://mongoid.org/>).

Inicialmente, criamos o sistema assim:

```
rails new mongodb-rails-casadocodigo --skip-active-record
```

Em seguida, alteramos o arquivo `Gemfile` adicionando duas linhas:

```
gem 'mongoid', '~> 4', github: 'mongoid/mongoid'  
gem 'bson_ext'
```

Criamos o arquivo de configuração de acesso ao MongoDB:

```
rails g mongoid:config
```

Com isso, será criado o arquivo `mongoid.yml` com as configurações de acesso:

```
development:
  sessions:
    default:
      database: mongodb_rails_casadocodigo_development
      hosts:
        - localhost:27017
test:
  sessions:
    default:
      database: mongodb_rails_casadocodigo_test
      hosts:
        - localhost:27017
    options:
      read: primary
      max_retries: 1
      retry_interval: 0
```

Finalmente, geramos a tela de cadastro de seriados:

```
rails generate scaffold seriados nome personagens
```

Depois para subir o servidor:

```
rails server
```

O resultado esperado da tela inicial é a figura:



Meus seriados preferidos

Listing seriados

Nome	Personagens	
Chaves	Seu Barriga, Chaves	Show Edit Destroy
Breaking Bad	Walter White	Show Edit Destroy
Carga Pesada	Pedro da boleia , Bino	Show Edit Destroy

[New Seriado](#)

Figura 5.5: Tela inicial do MongoDB com Ruby on Rails

5.6 NODE.JS

SOBRE O NODE.JS

Em 2009, foi criado um interpretador JavaScript baseado na engine do Google Chrome (chamada V8). A ideia é usar esse interpretador no servidor, sendo capaz de disponibilizar aplicações web com milhares de conexões simultâneas. Para aprofundar no uso do Node.js, procure pelo livro *Aplicações web real-time com Node.js* (<http://www.casadocodigo.com.br/products/livro-nodejs>).

Os códigos-fontes do sistema estão disponíveis em <https://github.com/boaglio/mongodb-nodejs-casadocodigo>. Para mostrarmos um exemplo mais simples com o Node.js, usamos o framework express para gerenciar as operações mais comuns.

Dentro da pasta seriados , existe o arquivo package.json com as dependências necessárias:

```
{  
  "name": "seriados",  
  "version": "1.0.0",  
  "private": true,  
  "scripts": {  
    "start": "node ./bin/www"  
  },  
  "dependencies": {  
    "express": "~4.9.0",  
    "body-parser": "~1.8.1",  
    "cookie-parser": "~1.3.3",  
    "morgan": "~1.3.0",  
    "serve-favicon": "~2.1.3",  
    "debug": "~2.0.0",  
    "jade": "~1.6.0",  
    "mongodb": "*",  
    "monk": "*"  
  }  
}
```

O arquivo principal do node.js (o app.js) possui algumas variáveis para conectar ao banco de dados:

```
var mongo = require('mongodb');  
var monk = require('monk');  
var db = monk('localhost:27017/test');
```

A página inicial chama o script index.js , que faz a busca da collection seriados e joga na variável seriados :

```
var collection = db.get('seriados');  
collection.find({},{},function(e,docs){  
  res.render('seriados', {
```

```
"seriados" : docs
});
});
```

Finalmente, no arquivo de template `seriados.jade`, temos a exibição dos seriados:

```
extends layout

block content
each seriado, i in seriados
  .btn.btn-primary.btn-lg
    <b>#{seriado.nome}</b> com: #{seriado.personagens}
```

O resultado é a figura a seguir.

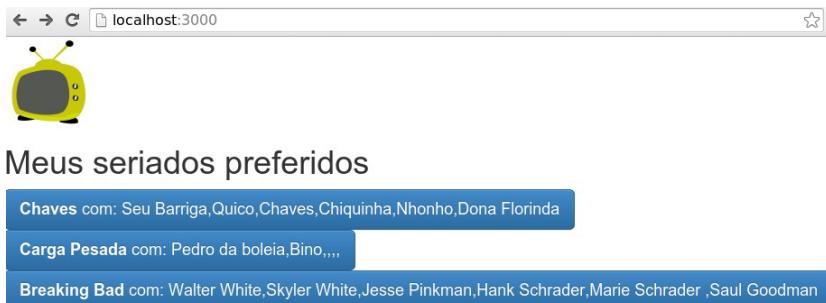


Figura 5.6: MongoDB com Node.js

5.7 QT

SOBRE O QT

Em 1995, foi criada a linguagem de programação baseada em C++, focada na criação de interfaces gráficas. Hoje ela é usada em muitas aplicações, entre elas o KDE, Skype, Opera, Google Earth, VLC e Virtual Box.

A aplicação em Qt possui implementação através de C++ e outra através de scripts na linguagem QML. Usando essa linguagem, vamos conectar ao MongoDB usando um componente não oficial (<http://qt5.jp/mongodb-plugin-for-qml.html>):

```
import me.qtquick.MongoDB 0.1
import QtQuick 2.0
import QtQuick.Controls 1.0
```

Em seguida, configuramos o acesso ao banco de dados e à collection de seriados:

```
Database {
    id: db
    host: '127.0.0.1'
    port: 27017
    name: 'test'

    property Collection
        seriados: Collection { name: 'seriados' }
}
```

O Qt, assim como vários frameworks web, utiliza o conceito de MVC, portanto a lista dos seriados (model) é delegada ao componente visual `Text`, que por sua vez usa o método `stringify` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify)

) para converter o documento em texto.

```
ListView {  
  
    id: lista  
    anchors.fill: parent  
  
    clip: true  
  
    model: db.seriados.find({"nome" : "Breaking Bad"})  
  
    delegate: Text {  
        text: JSON.stringify(model.modelData, null, 4)  
    }  
}
```

O resultado é a figura:



The screenshot shows a window titled "Qt com MongoDB!". Inside the window, a JSON document is displayed:

```
{  
    "_id": "53fd3aa68fc5e9cd4b8b4567",  
    "nome": "Breaking Bad",  
    "personagens": [  
        "Walter White",  
        "Skyler White",  
        "Jesse Pinkman",  
        "Hank Schrader",  
        "Marie Schrader",  
        "Saul Goodman"  
    ]  
}
```

Figura 5.7: MongoDB com Qt

5.8 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- integração do MongoDB com PHP;
- integração do MongoDB com Java com driver oficial;
- integração do MongoDB com Java com Spring Data;
- integração do MongoDB com Node.js;
- integração do MongoDB com Qt.

No próximo capítulo, analisaremos uma migração de MySQL para MongoDB, quando apenas scripts não são suficientes e é mais interessante desenvolver uma ferramenta para realizar a migração.

CAPÍTULO 6

MIGRANDO O SEU BANCO DE DADOS

A migração de bases relacionais para o MongoDB pode ser simples, na qual alguns SQLs gerem comandos para inserir e o seu sistema está migrado. Entretanto, normalmente em um *schema design* adequado, é normal termos várias tabelas convergindo para uma collection.

Esse tipo de informação não é possível representar em uma planilha para importar depois, e também será bem complexo (ou talvez impossível) rodar um `SELECT` para gerar comandos `db.collection.insert`. Nesse caso, é interessante criar um programa auxiliar para executar essa migração, no qual é feita uma leitura na base relacional e um cadastro no MongoDB.

6.1 IMDB SIMPLIFICADO

Os códigos-fontes do sistema estão disponíveis em <https://github.com/boaglio/mongodb-migra-imdb-casadocodigo>. O IMDB (Internet Movie DataBase — <http://imdb.com>) é a maior referência da internet do cinema mundial, que possui um gigantesco banco de dados com todos os filmes, seus atores e diretores.

O interessante é que eles disponibilizam os dados gratuitamente para baixar (<ftp://ftp.berlin.de/pub/misc/movies/database/>). O programa JMDB (<http://www.jmdb.de>) é um software que baixa esses dados e instala em uma base de dados MySQL local.

A base de dados é enorme, pois, além de filmes, inclui seriados (cada um dos episódios existentes) e videogames relacionados a filmes. Para o nosso exemplo, a base de dados foi simplificada tanto nos dados (temos apenas filmes) como em sua estrutura.

Nesse modelo simplificado, existem sete tabelas:

```
mysql> show tables;
+-----+
| Tables_in_jmdb |
+-----+
| actors         |
| directors      |
| genres         |
| movies         |
| movies2actors |
| movies2directors |
| ratings        |
+-----+
7 rows in set (0,00 sec)
```

A tabela que possui o cadastro de atores:

```
mysql> desc actors;
+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default |
+-----+-----+-----+-----+
| actorid | mediumint(8) | NO   | PRI | NULL    |
| name    | varchar(250)  | NO   |     | NULL    |
| sex     | enum('M','F') | YES  |     | NULL    |
+-----+-----+-----+-----+
+-----+
| Extra           |
```

```
+-----+  
| auto_increment |  
|  
|  
+-----+  
3 rows in set (0,00 sec)
```

A tabela que possui o cadastro de diretores:

```
mysql> desc directors;  
+-----+-----+-----+-----+  
| Field      | Type       | Null | Key | Default |  
+-----+-----+-----+-----+  
| directorid | mediumint(8) | NO   | PRI | NULL    |  
| name        | varchar(250) | NO   |     | NULL    |  
+-----+-----+-----+-----+  
  
+-----+  
| Extra      |  
+-----+  
| auto_increment |  
|  
+-----+  
2 rows in set (0,00 sec)
```

A tabela de tipos ou gêneros de filmes:

```
mysql> desc genres;  
+-----+-----+-----+-----+-----+-----+  
| Field      | Type       | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+-----+  
| movieid    | mediumint(8) | NO   | MUL | NULL    |       |  
| genre       | varchar(50)  | NO   |     | NULL    |       |  
+-----+-----+-----+-----+-----+-----+  
2 rows in set (0,00 sec)
```

A tabela de tipos ou gêneros de filmes:

```
mysql> desc movies;  
+-----+-----+-----+-----+-----+  
| Field      | Type           | Null | Key | Default |  
+-----+-----+-----+-----+-----+  
| movieid    | mediumint(8)  | NO   | PRI | NULL    |  
| title       | varchar(400)  | NO   |     | NULL    |  
+-----+-----+-----+-----+-----+
```

```

| year      | varchar(100)          | YES   |      | NULL    |
+-----+-----+-----+-----+
| Extra     |                         |
+-----+-----+
| auto_increment |                         |
|               |                         |
|               |                         |
+-----+-----+
3 rows in set (0,01 sec)

```

A tabela auxiliar que relaciona filmes e atores:

```

mysql> desc movies2actors;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| movieid | mediumint(8) | NO   | MUL | NULL    |      |
| actorid | mediumint(8) | NO   | MUL | NULL    |      |
+-----+-----+-----+-----+-----+
2 rows in set (0,00 sec)

```

A tabela auxiliar que relaciona filmes e diretores:

```

mysql> desc movies2directors;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| movieid | mediumint(8) | NO   | MUL | NULL    |      |
| directorid | mediumint(8) | NO   | MUL | NULL    |      |
| addition | varchar(1000) | YES  |      | NULL    |      |
+-----+-----+-----+-----+-----+
3 rows in set (0,00 sec)

```

A tabela de notas (de 0 a 10) e de quantidade de votos dos filmes:

```

mysql> desc ratings;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| movieid | mediumint(8) | NO   | MUL | NULL    |      |
| rank    | char(4)    | NO   |      | NULL    |      |
+-----+-----+-----+-----+-----+

```

```
| votes | mediumint(8) | YES |      | NULL   |      |
+-----+-----+-----+-----+-----+
3 rows in set (0,01 sec)
```

```
mysql>
```

Perceba que todas essas informações se relacionam a um filme. Isso significa que, no MongoDB, elas devem ficar em uma única uma collection.

Sua estrutura deverá ficar como nesse exemplo:

```
{
  "_id" : NumberLong(2646365),
  "titulo" : "TRON (1982)",
  "ano" : "1982",
  "nota" : 6.8,
  "votos" : NumberLong(78310),
  "categorias" : ["Action", "Adventure", "Sci-Fi"],
  "diretores": ["Lisberger, Steven"],
  "atores" : [{"nome" : "Berns, Gerald", "sexo": "M"},
    {"nome" : "Bostwick, Jackson", "sexo" : "M"}, 
    {"nome" : "Boxleitner, Bruce", "sexo" : "M"}, 
    {"nome" : "Bridges, Jeff (I)", "sexo" : "M"}, 
    {"nome" : "Brubaker, Tony", "sexo" : "M"}, 
    {"nome" : "Cass Sr., David S.", "sexo" : "M"}]
}
```

Perceba que categorias e diretores são um array simples com alguns valores, enquanto cada ator é um documento composto por nome e sexo , sendo que um array de atores está atribuído para atores .

6.2 MIGRANDO DE UM BANCO DE DADOS RELACIONAL

A rotina de migração é bem simples. Ela basicamente consulta

o MySQL para montar os dados, e depois insere o documento na collection do MongoDB. A parte inicial estabelece a conexão com o MongoDB e, em seguida, remove a collection de filmes (se ela já existir):

```
mongoClient = new MongoClient(  
    new MongoClientURI("mongodb://localhost"));  
database = mongoClient.getDatabase("test");  
filmesCollection = database.getCollection("filmes");  
filmesCollection.drop();
```

Depois, usamos a classe auxiliar `MySQLDAO` para consultas ao MySQL, inicialmente com a lista de filmes:

```
List<Filme> filmes = mysqlDAO.getFilmes();
```

Depois fazemos um loop de filme por filme, chamando a rotina auxiliar `adicionarFilme` :

```
for (Filme filme : filmes) {  
    adicionarFilme(filme.getId(),  
                   filme.getTitulo(),  
                   filme.getAno(),  
                   filme.getNota(),  
                   filme.getVotos());  
    contador++;  
}
```

A rotina `adicionarFilme` monta um documento adicionando valores a um `BasicDBObject`. As listas de categorias e diretores são preenchidas chamando as rotinas auxiliares `getcategorias` e `getDiretores` da classe `MySQLDAO` :

```
Document documento = new Document();  
documento.put("_id", movieid);  
documento.put("titulo", title);  
documento.put("ano", year);  
documento.put("nota", rank);  
documento.put("votos", votes);  
documento.put("categorias",
```

```
mysqlDAO.getcategorias(Long.valueOf(movieid));
documento.put("diretores",
mysqlDAO.getDiretores(Long.valueOf(movieid)));
```

Por último, temos a lista de atores, na qual inicialmente lemos com a rotina auxiliar `getAtores` da classe `MySQLDAO` :

```
List<Document> atoresMongoDB =
new ArrayList<Document>();
List<Ator> atoresMySQL =
mysqlDAO.getAtores(Long.valueOf(movieid));
```

Temos um loop de ator por ator retornado do MySQL, que é adicionado à lista `atoresMongoDB` :

```
for (Ator ator : atoresMySQL) {
    Document atorMongoDB = new Document();
    atorMongoDB.put("nome", ator.getNome());
    atorMongoDB.put("sexo", ator.getSexo());
    atoresMongoDB.add(atorMongoDB);
}
```

A lista de documentos de atores é inserida ao documento:

```
documento.put("atores", atoresMongoDB);
```

Finalmente, todo o documento é inserido na collection de filmes:

```
filmesCollection.insertOne(document);
```

O processo pode demorar algumas horas, mas a com certeza a base será migrada com sucesso.

6.3 MIGRANDO PARA NUVEM

Existem várias opções de disponibilizar o MongoDB na nuvem, a oficial (e paga) é o MongoDB Atlas (<http://www.mongodb.com/cloud>). Entretanto, nos focaremos

aqui no OpenShift da Red Hat. Vamos criar uma conta gratuita pelo site <https://www.openshift.com/>.

Ao entrarmos no sistema, vemos a lista de aplicações instaladas, e opção de adicionar uma nova, como ilustrado na figura:

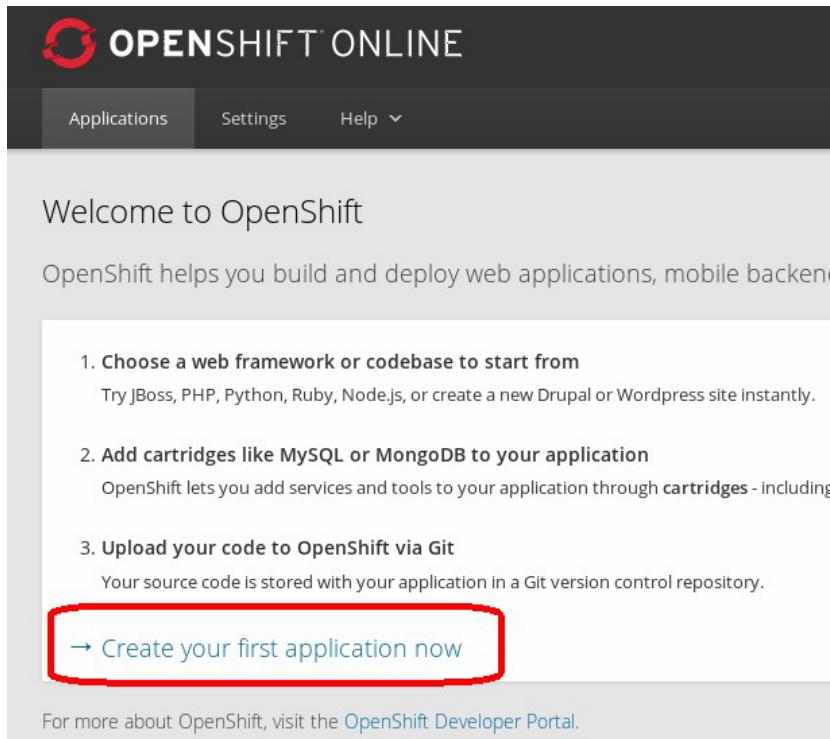


Figura 6.1: OpenShift — lista de aplicações

Diversas opções serão oferecidas, mas escolheremos um contêiner leve como o Tomcat , conforme a figura a seguir.

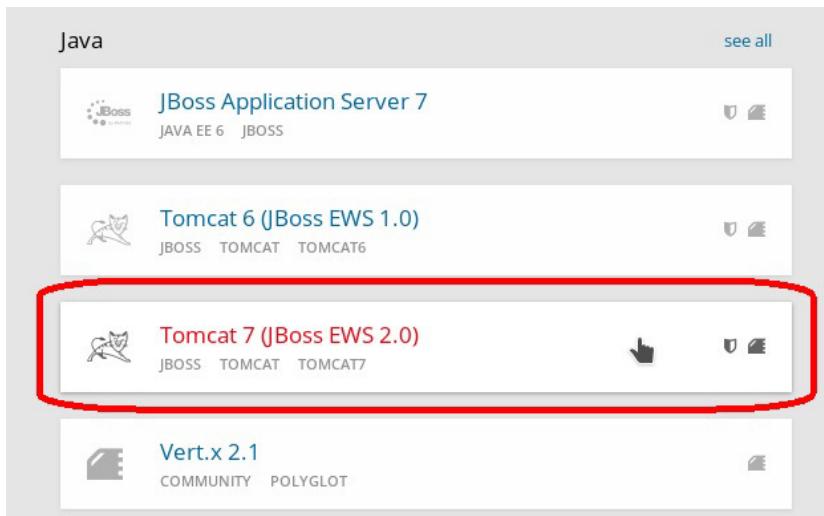


Figura 6.2: OpenShift — lista de componentes

Depois de escolher o servidor, é oferecida a opção de escolher a URL pública, na qual informamos o valor de seriados . Veja a figura:

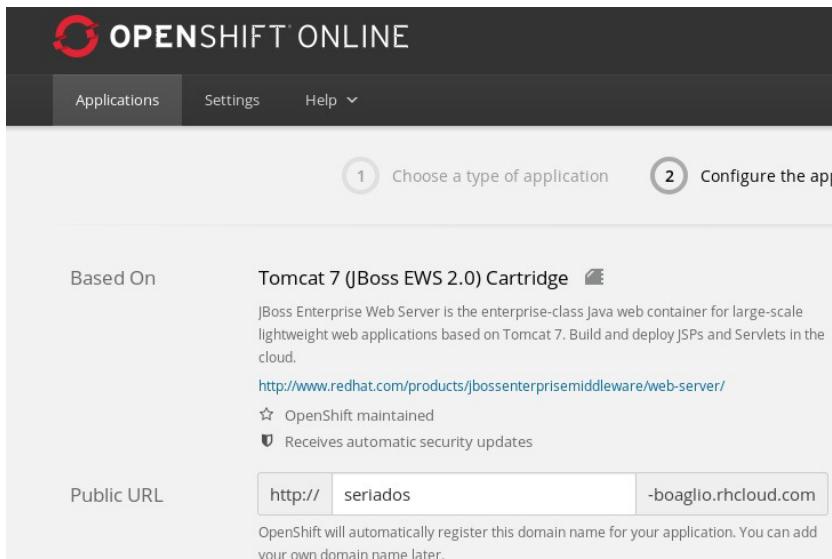


Figura 6.3: OpenShift — registrando aplicação

Uma vez criada a aplicação de seriados, adicionaremos o componente do MongoDB (figura adiante).

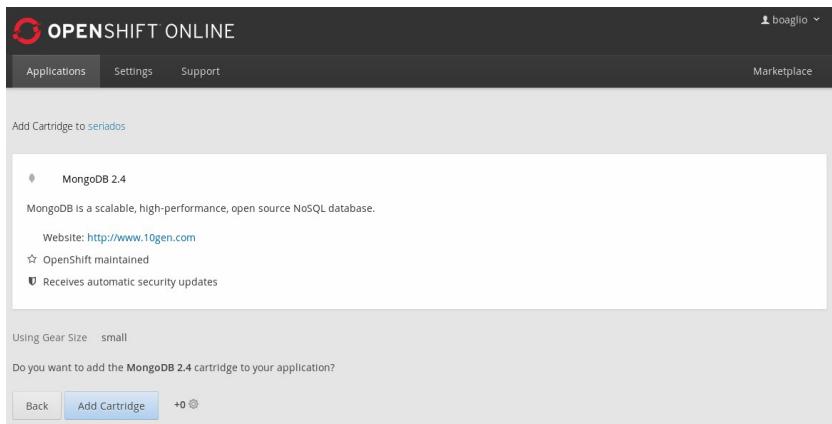


Figura 6.4: OpenShift — adicionando MongoDB

Agora é informada a senha de acesso (figura a seguir).

The screenshot shows the OpenShift Online interface. At the top, there's a navigation bar with 'OPENSHIFT ONLINE' and links for 'Applications', 'Settings', and 'Help'. Below the header, the URL 'seriados-boaglio.rhcloud.com' is displayed, along with a 'change' link and a note that it was created 8 minutes ago in the domain 'boaglio' and the 'aws-us-east-1' region.

A green notification box contains a checkmark icon and the message: 'MongoDB 2.4 database added. Please make note of these credentials:' followed by the root user ('admin'), root password ('admin'), and database name ('seriados'). It also provides a connection URL: 'mongodb://\$OPENSHIFT_MONGODB_DB_HOST:\$OPENSHIFT_MONGODB_DB_PORT/'.

The 'Cartridges' section lists two components: 'Tomcat 7 (JBoss EWS 2.0)' and 'MongoDB 2.4'. The MongoDB entry includes details: 'Database: seriados', 'User: admin', and 'Password: show'.

At the bottom, there are sections for 'Continuous Integration' (with a 'Enable Jenkins' link) and 'Tools and Support' (with a 'Add RockMongo 1.1' link).

Figura 6.5: OpenShift — MongoDB adicionado

Para administrar o MongoDB remotamente, adicionamos o componente RockMongo :

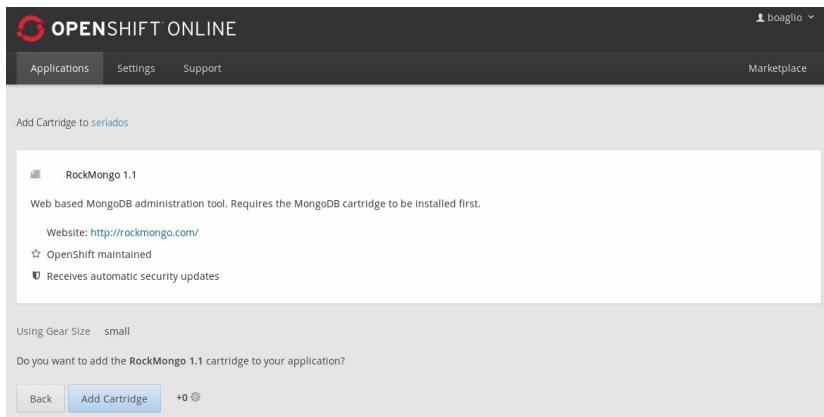


Figura 6.6: OpenShift — adicionando RockMongo

Depois de adicionado, é informada a senha de acesso:

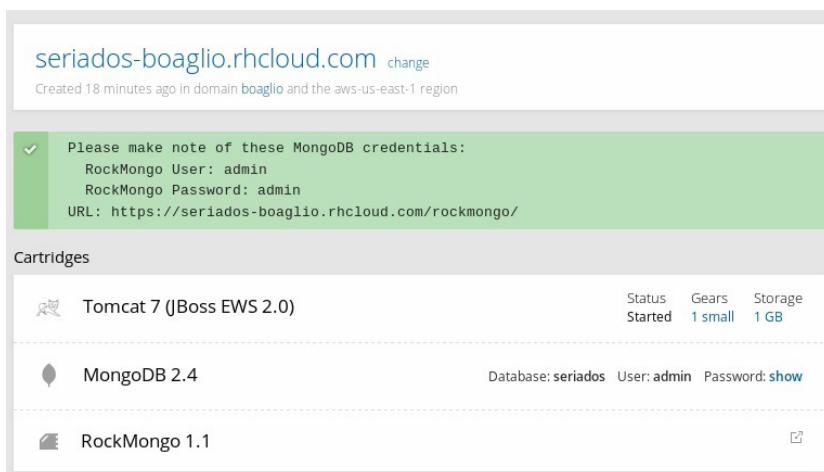


Figura 6.7: OpenShift — RockMongo adicionado

Conseguimos acessar o RockMongo com a senha fornecida:

Log-in:

Host [▼]

UserName:

Password:

Non-Admin users:

DB Name(s) :

[More »](#)

[Login and Rock](#)

Powered by [RockMongo v1.1.7](#), check
out new version here. [\[Donate 10
dollars\]](#)

Figura 6.8: OpenShift — entrar no RockMongo

Ao entrar no RockMongo, visualizamos todas as collections:

The screenshot shows the OpenShift interface with the 'Bancos' section selected. Under 'seriados', the 'Estatísticas' tab is active, displaying database statistics. The table includes rows for Size (0), Is Empty? (No), Collections (3 collections: openshift, system.indexes, system.users), Objects (11), Avg Object Size (66.181818181818), Data Size (728B), Storage Size (24k), Extents (4), Indexes (3), Index Size (23.95k), Total File Size (16m), Namespace Size (16m), and Data File Version ({"major": NumberInt(4), "minor": NumberInt(5)}).

Figura 6.9: OpenShift — visão geral

Através dele, podemos importar uma base local com a opção `import :`

The screenshot shows the 'Importar' (Import) feature for the 'seriados' database. It has two sections: one for a '.js' file (using RockMongo) and one for a '.json' file (using mongoexport). The '.js' section shows a 'JS File:' input field ('Selecionar arquivo...') with the message 'Nenhum arquivo selecionado.' and a 'Importar' button. The '.json' section shows an 'Import to collection name:' input field ('seriados'), a 'JSON File:' input field ('Selecionar arquivo...') with the value 'seriados.json', and an 'Importar' button.

Figura 6.10: OpenShift — importar banco de dados

Podemos também criar collections novas com um tamanho específico e, se necessário, com a opção `capped`, que deixa a

collection com um tamanho fixo e conteúdo rotativo, semelhante a um log de servidor (figura a seguir).

The screenshot shows the OpenShift web interface. On the left, there's a sidebar with 'Servidor' and 'Overview'. Under 'seriados (3)', it lists 'keyword', 'openshift (1)', 'system.indexes (3)', and 'system.users (1)'. There's also a 'Create' button. On the right, the main panel shows 'Bancos > Bancos > Criar nova Coleção'. It has a 'Name' field set to 'seriados'. Below it is a 'Capped Collection Options' section with three fields: 'Is Capped' (unchecked), 'Size' (0 bytes), and 'Max' (0 documents). A 'Create' button is at the bottom of this section. A link 'Here for more details »' is visible below the options.

Figura 6.11: OpenShift — criar collection

Depois de configurado o ambiente no OpenShift, podemos subir nossa aplicação para o Tomcat.

O primeiro passo para permitir acesso aos servidores do OpenShift é a instalação do client deles. Consulte o link para instalar no sistema operacional utilizado:
<https://developers.openshift.com/managing-your-applications/client-tools.html>.

Com o ambiente configurado, conseguimos clonar o repositório remoto com git:

```
git clone  
ssh://541bab@seriados-boaglio.rhcloud.com/~/git/seriados.git/  
cd seriados/
```

Veja um exemplo:

```
$ git clone ssh://57db@seriados-boaglio.rhcloud.com  
/~/git/seriados.git/
```

```
Cloning into 'seriados'...
Enter passphrase for key '/home/fb/.ssh/id_rsa':
remote: Counting objects: 41, done.
remote: Compressing objects: 100% (32/32), done.
remote: Total 41 (delta 1), reused 41 (delta 1)
Checking connectivity... done.
$ cd seriados/
```

E depois adicionarmos a nossa aplicação nesse diretório:

```
git add .
git commit -m 'Minha app'
git push
```

A nossa aplicação de seriados sofreu uma pequena alteração para funcionar no OpenShift. Os códigos-fontes podem ser baixados de <https://github.com/boaglio/mongodb-java-springdata-openshift-casadocodigo>, e depois subidos para a sua conta do OpenShift .

A única alteração ocorre na classe do `SpringMongoConfig` , que trata da autenticação do MongoDB. Felizmente, todas as informações de acesso ao banco de dados estão em variáveis de ambiente `OPENSIFT_MONGODB_DB_HOST` , `OPENSIFT_MONGODB_DB_PORT` , `OPENSIFT_APP_NAME` , `OPENSIFT_MONGODB_DB_USERNAME` , `OPENSIFT_MONGODB_DB_PASSWORD` , facilmente acessíveis pelo sistema:

```
host = System.getenv("OPENSIFT_MONGODB_DB_HOST");
if (host == null) {
    host = "127.0.0.1";
}
String sport = System.getenv("OPENSIFT_MONGODB_DB_PORT");
db = System.getenv("OPENSIFT_APP_NAME");
if (db == null) {
    db = "test";
}
user = System.getenv("OPENSIFT_MONGODB_DB_USERNAME");
```

```
password = System.getenv("OPENSIFT_MONGODB_DB_PASSWORD");
if (sport != null) {
    port = Integer.decode(sport);
} else {
    port = 27017;
}
```

Note que os valores vão funcionar tanto para uma conexão local quanto no OpenShift. Na autenticação, verificamos se foi informado o usuário e, em caso positivo, usamos a autenticação com usuário e senha:

```
if (user != null) {
    credential =
        MongoCredential.createMongoCRCredential
            (user, db, password.toCharArray());
    mongoClient = new MongoClient(
        new ServerAddress(host, port),
        Arrays.asList(credential)
    );
} else {
    mongoClient = new MongoClient(
        new ServerAddress(host, port)
    );
}
return mongoClient;
```

Em seguida, um exemplo de como seria subir no OpenShift a aplicação de seriados:

```
> git commit -m "seriados openshift"
[master 96a6c16] seriados openshift
 20 files changed, 701 insertions(+), 637 deletions(-)
...
  rewrite pom.xml (93%)
> git push
Counting objects: 31, done.
Compressing objects: 100% (23/23), done.
Writing objects: 100% (31/31), 11.72 KiB | 0 bytes/s, done.
Total 31 (delta 2), reused 0 (delta 0)
remote: Stopping RockMongo cartridge
remote: Waiting for stop to finish
```

```
remote: Waiting for stop to finish
remote: Stopping MongoDB cartridge
...
remote: [INFO] BUILD SUCCESS
remote: Preparing build for deployment
remote: Deployment id is 9fd7164b
remote: Activating deployment
remote: Starting RockMongo cartridge
remote: Starting MongoDB cartridge
remote: Deploying JBoss
remote: Starting jbossas cartridge
remote: Found 127.11.181.1:8080 listening port
remote: Found 127.11.181.1:9999 listening port
a35cfbc..96a6c16 master -> master
```

Acessamos a aplicação no endereço <http://seriados-boaglio.rhcloud.com/>, obtendo um resultado semelhante à figura:



Figura 6.12: OpenShift aplicação de seriados

6.4 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- adaptar uma rotina em Java para ler em uma base relacional e cadastrar no MongoDB;
- criar uma aplicação no OpenShift com MongoDB.

No próximo capítulo, analisaremos as buscas mais avançadas do MongoDB, sendo algumas impossíveis de se realizar em um banco de dados relacional.

CAPÍTULO 7

BUSCAS AVANÇADAS

Como nem todo sistema faz buscas apenas pela chave, é essencial entendermos os diversos tipos de busca existentes. No *capítulo 3 - MongoDB básico*, aprendemos a seguinte sintaxe de busca:

```
db.<nome-da-collection>.find(  
    {<campo1>:<valor1>,  
     <campo2>:<valor2>,  
     ...},  
    {<campoParaExibir>:<exibeOuNaoExibe>,  
     <campoParaExibir>:<exibeOuNaoExibe>,  
     ...});
```

Entretanto, para fazer uma busca mais abrangente, com mais opções, o critério de busca é diferente da maneira relacional. Vamos para um exemplo de como buscar quantos sorteios tiveram mais do que cinco ganhadores. Nos exemplos a seguir, vamos usar o operador `count` apenas para simplificar os resultados.

```
db.<nome-da-collection>.find(  
    {<campo>:  
        { <operador> : <valor1>}  
    }.count();
```

Nesse capítulo, além da sintaxe, usamos o exemplo da collection de sorteios da Mega-Sena. Vamos continuar com esse exemplo para ilustrar melhor o funcionamento dos operadores de

busca.

7.1 OPERADORES DE COMPARAÇÃO

No nosso exemplo, temos o operador `greater than` (ou maior do que) abreviado como `gt`. Para o MongoDB entender que ele é um operador e não um campo, ele é prefixado com dólar (`$`):

- MongoDB:

```
db.megasena.find({ "Ganhadores_Sena" :  
    { $gt:4} }).count();
```

- MySQL:

```
select count(*)  
from megasena  
where "Ganhadores_Sena">>4
```

Temos outros operadores semelhantes:

- `$gt` : maior do que
- `$gte` : igual ou maior do que
- `$lt` : menor do que
- `$lte` : igual ou menor do que

Além disso, temos outros operadores para usar também, como o operador `in`, que na busca a seguir conta o total de sorteios com 5, 6 ou 7 ganhadores:

```
> db.megasena.find({"Ganhadores_Sena":{$in:[5,6,7]} }).count()  
4
```

Outro operador interessante é o `ne` (*not equal/ não igual*), que traz o resultado oposto ao critério especificado. No exemplo a

seguir, trazemos a quantidade de sorteios que não têm 7 ganhadores.

Note que, utilizando o operador `in`, obtemos o valor complementar do total de documentos:

```
> db.megasena.find({"Ganhadores_Sena":{$ne:7}}).count()
2031
> db.megasena.find({"Ganhadores_Sena":{$in:[7]}}).count()
1
> db.megasena.find().count()
2032
```

Se no operador `ne` precisarmos usar uma lista de valores, usamos o operador `nin` (`not in`):

```
> db.megasena.find().count()
2032
> db.megasena.find({"Ganhadores_Sena":{$nin:[5,6,7]}}).count();
2028
> db.megasena.find({"Ganhadores_Sena":{$in:[5,6,7]}}).count();
4
```

7.2 OPERADOR DISTINCT

Um operador comum nos bancos relacionais é o `DISTINCT`, que elimina as repetições do resultado de uma consulta.

```
db.<collection>.distinct(<campo>)
```

Veja um exemplo da quantidade de ganhadores da Mega-Sena:

```
> db.megasena.distinct("Ganhadores_Sena")
[ 0, 1, 2, 3, 4, 5, 15, 7, 6 ]
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.distinct("Ganhadores_Sena")
```

- MySQL:

```
select distinct("Ganhadores_Sena")
from megasena
```

7.3 EXPRESSÕES REGULARES

O MongoDB oferece o poderoso recurso de busca com expressões regulares, que permite fazer buscas interessantes, como usar várias máscaras.

Uma limpeza em dados é algo bem comum. No caso da nossa collection de `megasena`, talvez accidentalmente um dado de números foi inserido em uma coluna:

```
db.megasena.insert({ "Acumulado": "123" })
```

Como encontrar valores numéricos em um campo texto? Isso é possível usando uma máscara do `regex`, por meio da sintaxe:

```
db.collection>.find({ <campo>: { $regex: <máscara> } })
```

Veja um exemplo:

```
db.megasena.find({ "Acumulado": { $regex: /^\\d.*$/ } })
{
    "_id" : ObjectId("57d08d348688b893a04c3e95"),
    "Acumulado" : "123"
}
```

7.4 OPERADORES LÓGICOS

Os operadores lógicos `and`, `nor`, `not` e `or` trabalham de maneira semelhante:

```
db.<nome-da-collection>.find({  
    <operador lógico>:  
        [ <condição 1>, <condição 2>, ... ]  
})  
.count();
```

Neste exemplo, os operadores retornam exatamente o mesmo valor:

```
> db.megasena.find({ $or:  
    [ {"Ganhadores_Sena":{ $eq:5 } },  
      {"Ganhadores_Sena":{ $eq:7 } }]  
}).count()  
3  
> db.megasena.find({ $or:  
    [ {"Ganhadores_Sena":5},  
      {"Ganhadores_Sena":7}]  
}).count()  
3  
> db.megasena.find({"Ganhadores_Sena":{$in:[5,7]}} ).count()  
3
```

7.5 OPERADORES UNÁRIOS

Outro operador interessante é o `exists`, que verifica a existência de um campo. Como no MongoDB o schema é flexível, podendo ser criado um novo campo a qualquer momento, é interessante saber como procurar os documentos com esses campos novos.

No exemplo a seguir, inicialmente inserimos um documento novo com o campo novo `obs`:

```
> db.megasena.insert({"obs": "sem sorteio"})  
WriteResult({ "nInserted" : 1 })
```

Agora, alterando os valores do `exists` para `true` e `false`, podemos diferenciar os registros com o novo campo:

```
> db.megasena.find().count()
2033
> db.megasena.find({"obs":{$exists:true}}).count()
1
> db.megasena.find({"obs":{$exists:false}}).count()
2032
```

7.6 OPERADOR ESTILO LIKE

O operador like é muito comum nas bases relacionais e permite fazer buscas por trechos de texto nas tabelas. No MongoDB, existe um correspondente, mas sem o mesmo nome. Inicialmente, listando os nomes que contêm a palavra ad :

```
> db.seriados.find({ "nome": /ad/}, {"nome":1, "_id":0})
{ "nome" : "Carga Pesada" }
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({ "nome": /ad/},
                  {"nome":1, "_id":0})
```

- MySQL:

```
select nome
from megasena
where "nome" like '%ad%';
```

Para a busca sem considerar maiúsculas ou minúsculas, é preciso colocar i (de *case Insensitive*):

```
> db.seriados.find({ "nome": /bad/}, {"nome":1, "_id":0})
> db.seriados.find({ "nome": /bad/i}, {"nome":1, "_id":0})
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({ "nome": /bad/i},
                  {"nome":1,"_id":0});
```

- MySQL:

```
select nome
      from megasena
     where upper("nome") like upper('%ad%')
```

Para buscar por palavras que terminam com um trecho de caracteres, é preciso colocar \$:

```
> db.seriados.find({ "nome": /ad/}, {"nome":1,"_id":0})
{ "nome" : "Carga Pesada" }
{ "nome" : "Breaking Bad" }
> db.seriados.find({ "nome": /ad$/}, {"nome":1,"_id":0})
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({ "nome": /ad$/},
                  {"nome":1,"_id":0})
```

- MySQL:

```
select nome
      from megasena
     where "nome" like '%ad'
```

Para buscar por palavras que se iniciam com um trecho de caracteres, é preciso colocar ^ :

```
> db.seriados.find({ "nome": /^Ba/}, {"nome":1,"_id":0})
> db.seriados.find({ "nome": /^Br/}, {"nome":1,"_id":0})
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({ "nome": /^Br/ },
                  {"nome":1, "_id":0})
```

- MySQL:

```
select nome
      from megasena
     where upper("nome") like upper('Br%')
```

Para ordenar o resultado da consulta, usamos o sufixo `sort`, especificando as colunas que desejamos ordenar em ordem crescente (com o valor 1):

```
> db.seriados.find({}, {_id:0, nome:1}).sort({nome:1})
{ "nome" : "Breaking Bad" }
{ "nome" : "Carga Pesada" }
{ "nome" : "Chaves" }
```

Ou em ordem decrescente (com o valor -1):

```
> db.seriados.find({}, {_id:0, nome:1}).sort({nome:-1})
{ "nome" : "Chaves" }
{ "nome" : "Carga Pesada" }
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({}, {_id:0, nome:1})
              .sort({nome:-1})
```

- MySQL:

```
select nome
      from megasena
     order by nome desc
```

Para limitar os resultados, podemos usar `limit` informando a

quantidade de documentos para exibir:

```
> db.seriados.find({}, {_id:0, nome:1})  
{ "nome" : "Chaves" }  
{ "nome" : "Carga Pesada" }  
{ "nome" : "Breaking Bad" }  
> db.seriados.find({}, {_id:0, nome:1}).limit(1)  
{ "nome" : "Chaves" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({}, {_id:0, nome:1})  
.limit(1)
```

- MySQL:

```
select nome  
from megasena  
limit 1
```

Além de limitar com `limit`, podemos informar a quantidade de documentos para pular antes de exibir com `skip`:

```
> db.seriados.find({}, {_id:0, nome:1})  
{ "nome" : "Chaves" }  
{ "nome" : "Carga Pesada" }  
{ "nome" : "Breaking Bad" }  
> db.seriados.find({}, {_id:0, nome:1}).limit(1).skip(2)  
{ "nome" : "Breaking Bad" }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.seriados.find({}, {_id:0, nome:1})  
.limit(1).skip(2)
```

- MySQL:

```
select nome
```

```
from megasena  
limit 1  
skip 2
```

O comando `sort` pode ser usado também para exibir o último registro cadastrado:

```
db.<collection>.find().sort({"_id": -1}).limit(1)
```

Veja um exemplo:

```
> db.seriados.find({}, {_id:0, nome:1}).sort({"_id": -1}).limit(1)  
{ "nome" : "Breaking Bad" }
```

7.7 INCREMENTANDO VALORES

É comum usarmos valores armazenados no banco de dados para alguma operação de atualização, como um aumento de salário em uma collection de funcionários. Entretanto, esse tipo de operação é um pouco diferente no MongoDB. É preciso usar um operador específico. Por exemplo, em incrementar um campo:

```
db.<collection>.update({<filtro-de-busca>}  
, { $inc: { <campo> : <valor> }})
```

Agora veja um exemplo de dar um aumento de R\$ 500 a um funcionário:

```
db.funcionarios.update(  
  {"_id" : ObjectId("57d08d348688b893a04c3e8d")},  
  { $inc: { "salario" : 500 }  
)
```

7.8 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- operações de busca com operadores de comparação;
- operações de busca com operadores lógicos;
- operações de busca com operadores unários;
- operações de busca estilo LIKE;
- operações de ordenação.

No próximo capítulo, usaremos o suporte geoespacial do MongoDB para trabalhar facilmente com coordenadas e distâncias.

CAPÍTULO 8

BUSCA GEOESPACIAL

Quando falamos sobre geoespacial, significa manipular informações em duas ou três dimensões. Trabalhar com coordenadas em um banco de dados não é algo muito incomum. Entretanto, usar rotinas nativas que calculam automaticamente a distância entre coordenadas simplificam muito o desenvolvimento de sistemas.

Vamos mostrar um exemplo de calcular a distância de duas cidades americanas. O código-fonte da aplicação e o *dump* do banco de dados estão em <https://github.com/boaglio/mongodb-java-geospatial-springdata-casadocodigo>.

8.1 O BANCO DE DADOS

Para restaurar o banco de dados, faça os passos a seguir (entenda com mais detalhes no *capítulo 11 — MongoDB para administradores*):

```
cd <diretório-do-projeto>
mongorestore
```

Veja um exemplo:

```
> cd /home/fb/workspace/projeto/dump/
> mongorestore
```

```
using default 'dump' directory
building a list of dbs and collections to restore from dump dir
reading metadata for test.zipcodes
  from dump/test/zipcodes.metadata.json
restoring test.zipcodes from dump/test/zipcodes.bson
restoring indexes for collection test.zipcodes from metadata
finished restoring test.zipcodes (25704 documents)
done
```

O nosso banco de dados possui 25704 cidades cadastradas, cada uma delas com o documento `loc` com coordenadas de latitude (Y) e longitude (X). Veja um exemplo de busca pela cidade de Orlando no estado da Flórida:

```
db.zipcodes.find({ "city" : "ORLANDO", "state" : "FL"})
{
  "_id" : ObjectId("544edc2d5e0a44b1d3daa0d0"),
  "city" : "ORLANDO",
  "state" : "FL",
  "loc" : {
    "x" : 81.408162,
    "y" : 28.487102
  }
}
```

Agora, buscando a cidade de Miami, também na Flórida:

```
db.zipcodes.find({ "city" : "MIAMI", "state" : "FL"})
{
  "_id" : ObjectId("544edc2c5e0a44b1d3da974a"),
  "city" : "MIAMI",
  "state" : "FL",
  "loc" : {
    "x" : 80.441031,
    "y" : 25.661502
  }
}
```

Até então, é um banco de dados comum do MongoDB. O que ativa o uso das informações geoespaciais é a criação de um índice desse tipo.

Veremos mais detalhes de criação de índices no *capítulo 10 — Aumentando a performance*, mas nesse caso basta informar o campo para indexar as coordenadas com essa sintaxe:

```
db.<collection>.ensureIndex( { <campo-com-coordenadas> : "2d" } )
```

No nosso banco de dados, temos:

```
db.zipcodes.ensureIndex( { loc : "2d" } )
```

Com esse índice, conseguimos buscar as cidades próximas em radianos com o comando `near` usando a sintaxe:

```
db.<collection>.find( { <campo-com-coordenadas>: { $near : [ <coordenada>, <coordenada> ], $maxDistance: <distancia> } } )
```

Veja um exemplo das cidades a 0.1 radianos de Miami:

```
db.zipcodes.find( { 'loc': { $near : [ 80.441031, 25.661502 ], $maxDistance: .1 } }, { _id:0, "city":1, "state":1 } )
{
  "city" : "MIAMI",
  "state" : "FL"
}
{
  "city" : "OLYMPIA HEIGHTS",
  "state" : "FL"
}
```

Se aumentarmos a distância, consequentemente aparecem mais cidades vizinhas:

```
db.zipcodes.find( { 'loc': { $near : [ 80.441031, 25.661502 ], $maxDistance: .1 } } ).count()
2
db.zipcodes.find( { 'loc':
```

```
{$near : [ 80.441031,25.661502],  
 $maxDistance: .5 } }).count()  
31  
db.zipcodes.find( { 'loc':  
 {$near : [ 80.441031,25.661502],  
 $maxDistance: 1 } }).count()  
54
```

8.2 USANDO O SISTEMA WEB

Inicialmente, o sistema busca duas listas de cidades para escolher, como mostra a figura:

The screenshot shows a web browser window with the URL `localhost:8080/home`. On the left, there is a small map of the United States with a grid overlay, showing the route from Miami to Orlando. A blue callout box labeled "Distância entre cidades americanas" points to this map. Below the map, there is a search form with two dropdown menus and a button. The first dropdown menu is labeled "Origem:" and contains the option "MIAMI - FL". The second dropdown menu is labeled "Destino:" and contains the option "ORLANDO - FL". Below these dropdowns is a blue button labeled "calcular...".

Figura 8.1: Calculando a distância entre Miami e Orlando

Em seguida, exibe o mapa das duas cidades e a distância entre elas e suas cidades vizinhas, como mostra a figura.

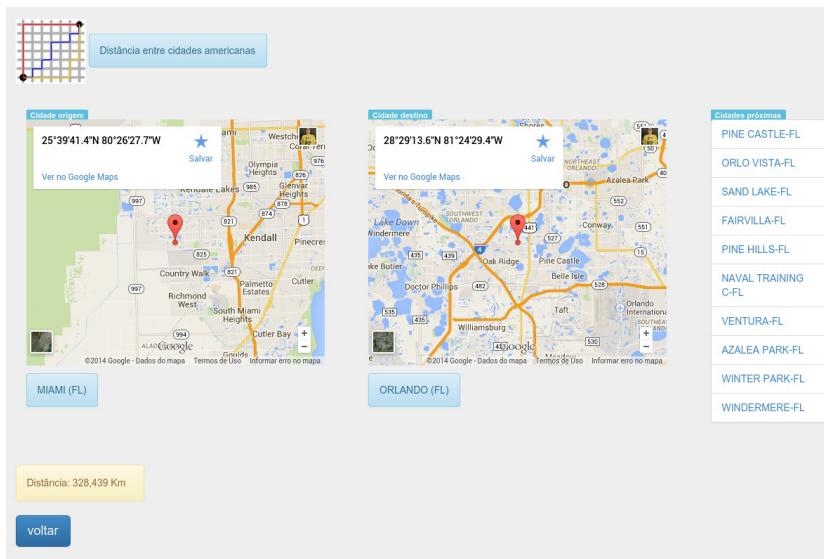


Figura 8.2: Exibindo o resultado da distância entre Miami e Orlando

8.3 ENTENDO O SISTEMA WEB

Seguindo o padrão do Spring Data, o sistema mapeia as cidades com a classe:

```
@Document(collection = "zipcodes")
public class Zip {

    @Id
    private ObjectId id;

    private String city;

    private String state;

    private Loc loc;
```

Na página inicial, exibe duas listas ordenadas de cidades, que ele busca da classe `ZipsRepository`:

```
List<Zip> zips = new ArrayList<Zip>();
Query query = new Query();
query.with(new Sort(Sort.Direction.ASC, "city"));
zips = mongoTemplate.find(query, Zip.class);
```

O usuário escolhe a cidade origem e a destino, e as informações são enviadas à classe `ZipcodesController`. Usamos a rotina auxiliar `CalculaDistancia` para retornar os dados em quilômetros:

```
Zip zip1 = repository.findById(idCidadeOrigem);
Zip zip2 = repository.findById(idCidadeDestino);
double distancia = CalculaDistancia.distance(
    zip1.getLoc().getX(), zip1.getLoc().getY(),
    zip2.getLoc().getX(), zip2.getLoc().getY());
```

Até aqui, não temos nada demais do que já foi usado e qualquer cadastro de coordenadas poderia fazer. Entretanto, o diferencial é essa consulta de cidades vizinhas, na qual informamos apenas as coordenadas da cidade e o tamanho do raio da distância. O banco de dados automaticamente calcula a lista das cidades próximas.

Criamos uma classe do tipo `Criteria` para definir a busca geoespacial com o comando `near`:

```
public List<Zip> findCidadesProximas(Double x, Double y) {
    List<Zip> zips = new ArrayList<Zip>();

    Criteria criteria = new Criteria("loc")
        .near(new Point(x,y))
        .maxDistance(
            CalculaDistancia.getInKilometer(RAIO_DE_DISTANCIA_EM_KM)
    );
}
```

Executamos a busca e limitamos o resultado das cidades vizinhas com `limit`:

```
Query buscaCidades = new Query(criteria);
zips = mongoTemplate.find(
    buscaCidades.limit(11), Zip.class);
```

Indo além

Com certeza, o sistema pode ser melhorado. Além disso, o MongoDB oferece vários recursos para trabalhar com coordenadas, como polígonos e estruturas mais complexas. Para tal, existe o GeoJSON (<http://geojson.org/>) e seus operadores.

8.4 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- criar um índice geoespacial em campo com coordenadas;
- utilizar o operador `near` .

No próximo capítulo, usaremos o *aggregation* framework para facilmente extrair importantes informações do banco de dados.

CAPÍTULO 9

AGGREGATION FRAMEWORK

De que adianta conter muitos dados se não for possível extrair informação deles? É por esse motivo que agrupamos os dados conforme a necessidade para conseguir o detalhamento necessário, o que nos bancos relacionais normalmente é feito com o comando `GROUP BY`.

Entretanto, no MongoDB, não existe apenas um comando semelhante. Existe, na verdade, algo bem mais robusto e completo chamado `aggregation framework` (framework de agrupamento), o que veremos adiante como usar.

9.1 POR QUE NÃO USAR MAP REDUCE

O aggregation framework surgiu na versão 2.2 do MongoDB e, desde então, se tornou uma versão mais simples e com mais performance do que o tradicional `map reduce`. O `map reduce` é um modelo de programação criado pela Google para trabalhar com muitos dados, e ser capaz de executar tarefas em paralelo com o objetivo de atingir o resultado de maneira mais rápida e eficiente.

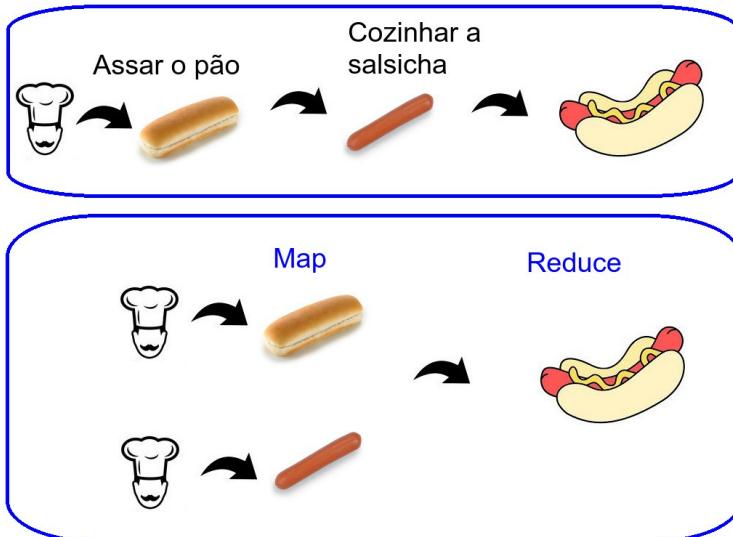


Figura 9.1: Fazendo hot dog com e sem map reduce

Nessa figura, vemos duas maneiras de fazer um hot dog. A primeira é a tradicional (e sequencial), em que uma tarefa é feita após a outra: primeiro assamos o pão, depois cozinhamos a salsicha. Com *map reduce*, conseguimos paralelizar as tarefas: um cozinheiro assa o pão enquanto o outro cozinha a salsicha.

Se a ideia for fazer cinco hot dogs, não tem muita diferença em usar qualquer uma das maneiras. Porém, se o objetivo for cinco mil unidades, com certeza a segunda opção é a mais eficiente.

Vamos usar o exemplo do *capítulo 3 — MongoDB básico* da collection de sorteios da Mega-Sena. Inicialmente definimos a função de mapeamento da quantidade de ganhadores agrupado pelos pela *flag* de acumulado (S ou N):

```
map = function() {
```

```
        emit(this.Acumulado , this.Ganhadores_Sena );
}
```

Em seguida, definimos a função `reduce` que será chamada para cada agrupamento, sendo que ela soma a quantidade de ganhadores de Mega-Sena.

```
reduce = function(Acumulado, Ganhadores_Sena) {
    return Array.sum(Ganhadores_Sena);
}
```

Finalmente, criaremos a collection `ganhadores`, em que aplicaremos o *map reduce*:

```
result = db.runCommand({
    "mapreduce" : "megasena",
    "map" : map,
    "reduce" : reduce,
    "out" : "ganhadores",
    "bypassDocumentValidation" : true
})
```

Fazendo uma consulta na collection criada, percebemos que ninguém ganhou nos sorteios em que acumulou (SIM). Já nos sorteios em que não acumulou (NÃO), 627 pessoas ganharam a Mega-Sena.

```
> db.ganhadores.find();
{ "_id" : "NÃO", "value" : 627 }
{ "_id" : "SIM", "value" : 0 }
```

9.2 EXPLORANDO O AGGREGATION FRAMEWORK

A sintaxe do aggregation framework é bem diferente do tradicional `GROUP BY`. Em nosso exemplo mais simples, vamos aplicar uma função de grupo em um campo, com a seguinte

sintaxe:

```
db.collection.aggregate( { $group :  
  { _id : null,  
    <nome-do-campo>:{  
      <função-de-grupo>:"$<nome-do-campo-para-agrupar>"  
    }  
  }  
});
```

Vamos, por exemplo, somar os ganhadores da Mega-Sena:

```
> db.megasena.aggregate( { $group :  
  { _id:null,  
    soma:{$sum:"$Ganhadores_Sena"}  
  } } )  
{ "_id" : null, "soma" : 627 }
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.aggregate( { $group :  
  { _id:null,  
    soma:{$sum:"$Ganhadores_Sena"}  
  } } );
```

- MySQL:

```
select sum(Ganhadores_Sena)  
from megasena
```

Vamos adicionar também a opção para exibir o total de sorteios e a média de ganhadores:

```
> db.megasena.aggregate( { $group: {  
  _id: null,  
  total: { $sum: 1 } ,  
  soma: { $sum: "$Ganhadores_Sena" } ,  
  avg: { $avg: "$Ganhadores_Sena" }  
} } )  
{ "_id" : null, "total" : 2032,
```

```
"soma" : 627, "avg" : 0.3383702104695089 }
```

Pelo resultado, percebemos que, além dos 627 ganhadores, temos ao todo 2032 sorteios e, em média, 0.33 de ganhadores por sorteio. Ou seja, aproximadamente um ganhador para cada 3 sorteios.

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.aggregate( { $group :  
    { _id:null,  
        total: { $sum: 1 } ,  
        soma: { $sum: "$Ganhadores_Sena" } ,  
        avg: { $avg: "$Ganhadores_Sena" }  
    } );
```

- MySQL:

```
select count(Ganhadores_Sena),  
       sum(Ganhadores_Sena),  
       avg(Ganhadores_Sena),  
  from megasena
```

Vamos agora agrupar por um campo. A sintaxe muda um pouco:

```
db.collection.aggregate( { $group :  
    { _id : <nome-do-campo>,  
      <campo>:{  
        <função-de-grupo>:"$<campo-para-agrupar>"  
      }  
    }  
})
```

O exemplo a seguir retorna exatamente o mesmo resultado que o exemplo de *map reduce* do começo do capítulo.

```
> db.megasena.aggregate({ $group: {  
    _id: "$Acumulado",
```

```
        soma: { $sum: "$Ganhadores_Sena" }
    } })
{ "_id" : "NÃO", "soma" : 627 }
{ "_id" : "SIM", "soma" : 0 }
```

Bem mais simples que usar o *map reduce*, não?

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.aggregate({ $group: {
    _id: "$Acumulado",
    soma: { $sum: "$Ganhadores_Sena" }
} })
```

- MySQL:

```
select Ganhadores_Sena,
       sum(Ganhadores_Sena)
  from megasena
 group by Acumulado;
```

Se depois de agrupados os resultados precisarmos fazer um filtro, devemos usar o `match`:

```
db.collection.aggregate( { $group :
  { _id : <nome-do-campo>,
    <nome-do-campo>:{ 
      <função-de-grupo>:"$<nome-do-campo-para-agrupar>" 
    }
  },
  { $match : {<filtro>} }
);
```

No exemplo, vamos listar apenas os resultados com soma maior do que zero:

```
db.megasena.aggregate({ $group: {
    _id: "$Acumulado",
    soma: { $sum: "$Ganhadores_Sena" }
} })
```

```

    } ,
    { $match : { soma : { $gt : 0 }}}
)
{ "_id" : "NÃO", "soma" : 627 }

```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```

db.megasena.aggregate({ $group: {
    _id: "$Acumulado",
    soma: { $sum: "$Ganhadores_Sena" }
} },
{ $match : { soma : { $gt : 0 }}}
)

```

- MySQL:

```

select Acumulado,
       sum(Ganhadores_Sena)
  from megasena
 group by Acumulado
 having sum(Ganhadores_Sena)>0;

```

Vamos agora agrupar por mais de um campo e ordenar o resultado:

```

db.collection.aggregate( { $group :
  { _id : { <apelido1> : $<nome-do-campo1>,
            <apelido2> : $<nome-do-campo2>,
            ...
        }
<nome-do-campo>:{ 
  <função-de-grupo>:"$<nome-do-campo-para-agrupar>" 
}
},
  { $sort : { _id: -1 } }
} )

```

Listaremos a quantidade de ganhadores da Mega-Sena agrupados também pela *flag* de se o prêmio está acumulado, tudo ordenado pela quantidade de ganhadores em ordem decrescente:

```
> db.megasena.aggregate({ $group: {
    _id: { ganhadores_sena:"$Ganhadores_Sena",
           acumulado: "$Acumulado" },
    soma: { $sum: "$Ganhadores_Sena" }
} },
{ $sort : { _id: -1 } } )
{"_id": {"ganhadores_sena":15,"acumulado":"NÃO"}, "soma":15}
{"_id": {"ganhadores_sena":7,"acumulado":"NÃO"}, "soma":7}
{"_id": {"ganhadores_sena":6,"acumulado":"NÃO"}, "soma":6}
{"_id": {"ganhadores_sena":5,"acumulado":"NÃO"}, "soma":10}
{"_id": {"ganhadores_sena":4,"acumulado":"NÃO"}, "soma":36}
{"_id": {"ganhadores_sena":3,"acumulado":"NÃO"}, "soma":60}
{"_id": {"ganhadores_sena":2,"acumulado":"NÃO"}, "soma":166}
{"_id": {"ganhadores_sena":1,"acumulado":"NÃO"}, "soma":327}
{"_id": {"ganhadores_sena":0,"acumulado":"SIM"}, "soma":0}
```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```
db.megasena.aggregate({ $group: {
    _id: { ganhadores_sena:"$Ganhadores_Sena",
           acumulado: "$Acumulado" },
    soma: { $sum: "$Ganhadores_Sena" }
} },
{ $sort : { _id: -1 } } )
```

- MySQL:

```
select Ganhadores_Sena,
       Acumulado,
       sum(Ganhadores_Sena)
  from megasena
 group by Ganhadores_Sena, Acumulado
 order by Ganhadores_Sena desc;
```

Para efetuarmos agrupamentos de agrupamentos, basta adicionar mais uma chave `group` ao final. No exemplo anterior, vamos agrupar novamente para somar todos os ganhadores da Mega-Sena:

```
> db.megasena.aggregate({ $group: {
```

```

_id: { ganhadores_sena:"$Ganhadores_Sena",
       acumulado: "$Acumulado" },
       soma: { $sum: "$Ganhadores_Sena" }
} },
{
  $group: {
    _id: null,
    soma_total: { $sum: "$soma" }
  }
}
{ "_id" : null, "soma_total" : 627 }

```

Fazendo uma analogia aos bancos de dados relacionais, temos:

- MongoDB:

```

db.megasena.aggregate({ $group: {
  _id: { ganhadores_sena:"$Ganhadores_Sena",
         acumulado: "$Acumulado" },
         soma: { $sum: "$Ganhadores_Sena" }
} },
{
  $group: {
    _id: null,
    soma_total: { $sum: "$soma" }
  }
});

```

- MySQL:

```

select sum(soma) soma_total
from ( select Ganhadores_Sena,
            Acumulado,
            sum(Ganhadores_Sena) soma
      from megasena
     group by Ganhadores_Sena, Acumulado)

```

Para mais exemplos, acesse a documentação oficial em
<http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/>.

9.3 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- o conceito geral do *map reduce*;
- o funcionamento e uso do aggregation framework.

No próximo capítulo, veremos como melhorar a performance do MongoDB, analisando o sistema como um todo e melhorando o tempo de consultas.

CAPÍTULO 10

AUMENTANDO A PERFORMANCE

Com o crescimento das collections, é normal as buscas que eram rápidas ficarem cada vez mais lentas. Entretanto, é muito simples de identificar a lentidão de uma consulta e fazer o ajuste necessário.

O MongoDB oferece o método stats existente em cada collection para informar várias coisas. Vamos destacar apenas os atributos que nos interessa, usando-os na collection de filmes do *capítulo 6 — Migrando o seu banco de dados*, que contém informações de milhares de filmes, com seus atores e diretores.

```
> db.filmes.stats()
{
  ...
  "count" : 1317734,
  "nindexes" : 1,
  "indexSizes" : {
    "_id_" : 15831040
  },
  ...
}
```

Nessa lista, temos:

- count : total de registros da collection;

- `nindexes` : quantidade de índices criados;
- `indexSizes` : nome e tamanho dos índices, por padrão temos um índice para o campo `_id` .

Usando o comando `explain` , podemos extrair informações importantes de uma consulta. Mas precisamos passar o parâmetro `executionStats` para exibir os tópicos interessantes. Novamente vamos destacar apenas o que interessa:

```
> db.filmes.find({ "ano" : "1999" }).explain('executionStats')
{
  ...
  "winningPlan" : {
    "stage" : "COLLSCAN",
    "filter" : {
      "ano" : {
        "$eq" : "1999"
      }
    },
    ...
  },
  ...
  "nReturned" : 13832,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 1317734,
  "executionTimeMillis" : 974,
  ...
}
```

A estratégia de como o banco de dados buscará os registros de uma consulta (query) é chamada de plano de execução (execution plan). Os bancos de dados modernos testam diferentes planos de execução e, em seguida, escolhem o melhor deles para executar. No caso do MongoDB, essa informação aparece no campo `winninPlan` .

Ali percebemos uma importante informação no campo `tage` , que possui o valor `COLLSCAN` . Isso é uma abreviação de Collection Scan, que significa que a consulta percorreu toda a collection (mais

de um milhão de registros) para retornar o resultado.

Podemos ter essa informação comparando os resultados de contar os registros, e comparar com o resultado do `explain`.

```
> db.filmes.find({"ano" :"1999"}).count()  
13832  
> db.filmes.count()  
1317734
```

Verificamos que o campo `nReturned` corresponde à quantidade de documentos retornados da busca, ou seja, a collection possui 13.832 filmes do ano de 1999. O campo `totalDocsExamined` é a quantidade de documentos que o MongoDB leu antes de retornar o resultado.

Comparando com a consulta anterior, percebemos que a quantidade de registros que ele leu é exatamente o tamanho da collection. Ou seja, ele varreu a collection inteira para retornar o resultado, o que nos bancos relacionais chamamos de *Full Table Scan*.

O tempo da execução é medido pelo campo `executionTimeMillis`. Neste caso, foi 974 milissegundos. Pelo campo `totalKeysExamined` com valor zero, percebemos também que nenhum índice foi usado para a consulta.

Em uma collection pequena, esse tempo não faz muita diferença. Entretanto, quando passamos dos milhões ou bilhões de registros, precisamos minimizar a quantidade de registros acessados pelo MongoDB.

10.1 CRIAR UM ÍNDICE

Para resolver esse problema, vamos criar um índice com a seguinte sintaxe:

```
db.collection.ensureIndex(  
{ <campo1> : <ordem>,  
  <campo2> : <ordem>,  
  ... } );
```

A ordem pode ser definida com 1 para crescente, ou -1 para decrescente, mas ela é importante apenas para índice composto. Em campo com índice simples, não importa a ordem.

Vamos criar um índice para o campo que usamos na busca:

```
db.filmes.ensureIndex( { "ano" : 1 } )
```

Em seguida, rodamos novamente o `explain` para verificar as diferenças:

```
> db.filmes.find({ "ano" : "1999" }).explain('executionStats')  
{  
  ...  
  "winningPlan" : {  
    "stage" : "FETCH",  
    "inputStage" : {  
      "stage" : "IXSCAN",  
      "keyPattern" : {  
        "ano" : 1.0  
      },  
      ...  
    "nReturned" : 13832,  
    "executionTimeMillis" : 54,  
    "totalKeysExamined" : 13832,  
    "totalDocsExamined" : 13832,  
    ...  
  }  
}
```

Aqui percebemos que finalmente o índice foi usado, já que no plano de execução aparece `IXSCAN`. Verificamos também que a quantidade de registros retornados (`nReturned`) continua 13832.

Entretanto, o campo `totalDocsExamined` retornou 13832 em vez de 1317734. Pelo campo `executionTimeMillis`, vemos que o tempo caiu de 974 para 54 milissegundos, e que o índice foi usado, já que existe valor maior do que zero no campo `totalKeysExamined`.

Para criarmos um índice em um campo de array, como as categorias de filme, o procedimento é exatamente o mesmo:

```
> db.filmes.ensureIndex( { "categorias" : 1 } )  
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 2,  
  "numIndexesAfter" : 3,  
  "ok" : 1  
}
```

Se executamos o `explain`, verificamos que o índice foi usado para listar os filmes da categoria `Crime`:

```
> db.filmes.find({ "categorias" : "Crime" }).explain('executionStats')  
...  
"winningPlan" : {  
  "stage" : "FETCH",  
  "inputStage" : {  
    "stage" : "IXSCAN",  
    ...  
  }  
  "nReturned" : 45151,  
  "executionTimeMillis" : 84,  
  "totalKeysExamined" : 45151,  
  "totalDocsExamined" : 45151,  
  ...  
}
```

Entretanto, se precisamos criar um índice para um campo dentro de um documento interno (*embedded document*), a complexidade aumenta.

No nosso exemplo, vamos tentar melhorar a consulta de todos

os atores do sexo masculino criando um índice para o campo atores:

```
> db.filmes.ensureIndex( { "atores" : 1 } )
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 3,
  "numIndexesAfter" : 4,
  "ok" : 1
}
```

Fazendo o `explain` da consulta, notamos que o índice não foi usado:

```
db.filmes.find({ "atores.sexo" :"M"}).explain('executionStats')
...
"winningPlan" : {
  "stage" : "COLLSCAN",
...
"nReturned" : 937854,
"executionTimeMillis" : 828,
"totalKeysExamined" : 0,
"totalDocsExamined" : 1317734,
...
```

Para esse caso, precisamos criar um índice junto com o campo desejado na pesquisa:

```
> db.filmes.ensureIndex( { "atores.sexo" : 1 } )
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 4,
  "numIndexesAfter" : 5,
  "ok" : 1
}
```

Dessa vez, notamos que o índice foi usado com sucesso:

```
db.filmes.find({ "atores.sexo" :"M"}).explain('executionStats')
...
"winningPlan" : {
  "stage" : "FETCH",
```

```
"inputStage" : {  
    "stage" : "IXSCAN",  
    ...  
    "nReturned" : 937854,  
    "executionTimeMillis" : 714,  
    "totalKeysExamined" : 937854,  
    "totalDocsExamined" : 937854,  
    ...
```

Se existir uma consulta constante por mais de um campo, como por exemplo o ano e as melhores nota do filme, é interessante criar um índice composto pelos campos ano com ordem crescente e nota com ordem decrescente:

```
db.filmes.ensureIndex( { "ano" : 1 , "nota" : -1 } )
```

10.2 LISTAR OS ÍNDICES CRIADOS

Para listar os índices criados, usamos a sintaxe:

```
db.collection.getIndexes()
```

Veja um exemplo:

```
> db.filmes.getIndexes()  
[  
{  
    "v" : 1,  
    "key" : { "_id" : 1 },  
    "name" : "_id_",  
    "ns" : "test.filmes"  
,  
...  
{  
    "v" : 1,  
    "key" : {  
        "atores" : 1  
,  
        "name" : "atores_1",  
        "ns" : "test.filmes"  
,
```

```
{  
  "v" : 1,  
  "key" : {  
    "atores.sexo" : 1  
  },  
  "name" : "atores.sexo_1",  
  "ns" : "test.filmes"  
}  
]
```

10.3 REMOVER UM ÍNDICE CRIADO

Para remover um índice existente, usamos a sintaxe:

```
db.collection.dropIndex(<nome-do-índice>);
```

Vamos remover o índice de atores que não ajuda nas nossas consultas:

```
> db.filmes.dropIndex("atores_1");  
{ "nIndexesWas" : 5, "ok" : 1 }
```

O índice do campo `_id` não pode ser removido, pois ele é usado internamente pelo MongoDB. Ao tentar removê-lo, o MongoDB exibirá uma mensagem de erro.

```
> db.filmes.dropIndex("_id_")  
{ "nIndexesWas" : 1, "ok" : 0, "errmsg" : "cannot drop _id index"}
```

10.4 ÍNDICE TEXTUAL

A busca textual (*text search*, ou *full text search*) existe no MongoDB desde a versão 2.4 e é ativa por padrão desde a versão 2.6. Com ela, é possível fazer uma busca não apenas por um trecho de texto, mas também por aproximações do mesmo texto.

A sintaxe para criar um índice textual, ou *text index*, é:

```
db.collection.ensureIndex({<campo>: "text"},  
{default_language: <idioma>} );
```

O parâmetro `default_language` é opcional. Se omitido, o índice é criado no idioma inglês. Vamos criar um índice textual em português para a collection de textos:

```
> db.textos.ensureIndex( {texto: "text"},  
{default_language: "portuguese"} )  
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 1,  
  "numIndexesAfter" : 2,  
  "ok" : 1  
}
```

Vamos cadastrar uma frase simples:

```
> db.textos.insert({texto: "Eu gosto de São Paulo"})  
WriteResult({ "nInserted" : 1 })
```

Fazendo uma busca por trechos parecidos, como "ele gosta" ou "gostar", o texto é encontrado:

```
> db.textos.find( { $text: { $search: "ele gosta" } } )  
{ "_id" : ObjectId("57de772c33ffffba25e714073"),  
  "texto" : "Eu gosto de São Paulo" }  
> db.textos.find( { $text: { $search: "gostar" } } )  
{ "_id" : ObjectId("57de772c33ffffba25e714073"),  
  "texto" : "Eu gosto de São Paulo" }
```

Vamos adicionar mais um documento:

```
> db.textos.insert({texto:"Eu gosto de São Paulo e Rio Claro"})  
WriteResult({ "nInserted" : 1 })
```

Buscando por "gostar", encontramos os dois documentos cadastrados:

```
> db.textos.find( { $text: { $search: "gostar" } } )  
{ "_id" : ObjectId("57de779633ffffba25e714074"),  
  "texto" : "Eu gosto de São Paulo e Rio Claro" }
```

```
{ "_id" : ObjectId("57de772c33ffffba25e714073"),
  "texto" : "Eu gosto de São Paulo" }
```

Pela busca textual, conseguimos remover resultados. Por exemplo, buscar por "gostar", mas sem os documentos que contenham "claro":

```
> db.textos.find( { $text: { $search: "gostar -claro" } } )
{ "_id" : ObjectId("57de772c33ffffba25e714073"),
  "texto" : "Eu gosto de São Paulo" }
```

Talvez exista a necessidade de criar um índice para vários campos da mesma collection. No caso da collection `textos`, seria:

```
db.textos.ensureIndex({ "$**": "text" },
  {default_language: "portuguese" } )
```

O MongoDB não permite que existam dois índices de *text search* na mesma coluna, logo, precisamos primeiro remover o existente e depois criar o novo:

```
> db.textos.dropIndex("texto_text");
{ "nIndexesWas" : 2, "ok" : 1 }
> db.textos.ensureIndex({ "$**": "text" },
  {default_language: "portuguese" } )
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Se for necessário fazer uma busca também por outro campo que não seja de *text search*, é interessante criar um índice composto. Se tivermos mais esse registro:

```
> db.textos.insert({texto:"Eu gosto de Trantor", ficção:true})
```

Precisamos de uma maneira eficiente de fazer uma busca nos textos de ficção. Para esse caso, criamos o índice dessa maneira:

```
> db.textos.ensureIndex( {ficação: 1, texto: "text"},  
{default_language: "portuguese"} )
```

Com esse índice, conseguimos fazer buscas eficientes como essa:

```
> db.textos.find({$text:{$search:"gosto"}, "ficação":true})  
{ "_id" : ObjectId("57de7f3633ffffba25e714075"),  
"texto" : "Eu gosto de Trantor", "ficação" : true }
```

10.5 CRIAR ÍNDICE EM BACKGROUND

No momento em que o índice é criado, as operações de leitura e escrita são bloqueadas até que o índice seja criado completamente.

Para evitar esse bloqueio, é possível criar o índice em background, conforme o exemplo de índice simples:

```
> db.textos.ensureIndex( { "texto": 1},  
{background: true} );  
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 2,  
  "numIndexesAfter" : 3,  
  "ok" : 1  
}
```

E índice textual:

```
> db.textos.ensureIndex( { "$**": "text"},  
{default_language: "portuguese"} ,  
{background: true} );  
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 1,  
  "numIndexesAfter" : 2,  
  "ok" : 1  
}
```

A criação de um índice em background demora um pouco mais do que o padrão. Pelo log, é possível acompanhar a criação do log e a porcentagem do que já foi criado. Existe também o comando `db.currentOp()` que mostra a mesma informação, como veremos no *capítulo 11 — MongoDB para administradores*.

10.6 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- fazer o *explain* de uma consulta;
- analisar o resultado de um *explain*;
- criar e remover índices simples e compostos;
- diferença de índices de *embedded document*;
- criar índices de busca textual;
- criar índices em background.

No próximo capítulo, veremos algumas tarefas administrativas, como ativar autenticação, gerenciar *backups* e *restores* do banco de dados.

CAPÍTULO 11

MONGODB PARA ADMINISTRADORES

Se você é administrador de banco de dados, certamente achou um absurdo o MongoDB ser tão aberto e não exigir nenhuma autenticação para trabalhar.

Ele foi feito dessa maneira por padrão para facilitar o desenvolvimento, mas evidentemente é possível ativar a autenticação, assim como definir perfis (*roles*) diferentes para executar tarefas específicas.

Veremos adiante o dia a dia de algumas tarefas de admins como uso de *storage*, gerenciamento de usuários e backup/restore.

11.1 TIPOS DE STORAGE

Até a versão 2, tínhamos apenas um tipo, o *MMAPv1*, que apesar de rápido exigia muito espaço em disco. A versão 3 introduziu a nova opção *WiredTiger*, que é a melhor opção de storage com um uso de espaço muito mais eficiente que a anterior.

Para qualquer tipo de ambiente (desenvolvimento, aceite, produção), esqueça o antigo *NNMAPv1* e utilize apenas o

11.2 AJUSTE DE PERFORMANCE

A maioria dos bancos de dados delega duas tarefas aos administradores: ajustar o sistema operacional da máquina além do banco de dados.

É comum ver servidores subutilizados com bancos de dados, por exemplo, o MySQL 5 tem em seu parâmetro `innodb_buffer_pool_size` o valor padrão de 128Mb. Se instalado em um servidor parrudo de 64Gb de memória RAM, ele utilizará menos de 1% da capacidade da máquina. Em alguns casos, apenas na instalação que os parâmetros são ajustados.

Nesse caso, cabe ao DBA ajustar os parâmetros do banco de dados para se adequar ao servidor. Com MongoDB, é diferente. Depois de configurar o sistema operacional, o MongoDB assume que "posso usar a máquina à vontade", e assim ele vai consumir bem a parte de memória de disco, o que é ideal para garantir uma boa performance de acesso aos dados das collections do banco de dados.

Se for possível alocar a collection inteira na memória para garantir uma boa performance, o MongoDB fará isso sem dó de seu servidor e o que estiver rodando junto com ele, desde que ele perceba que o recurso (memória) esteja livre para uso. Como o MongoDB tem esse comportamento, é uma boa prática ter um servidor (ou uma máquina virtual) dedicado a isso.

11.3 GERENCIANDO ESPAÇO EM DISCO

Para obter melhor performance e evitar problemas com fragmentação de arquivos, o MongoDB pré-aloca seus arquivos de dados. Conforme o banco de dados cresce, ele vai alocando mais e mais espaço, mesmo que não necessite naquele momento.

Como o MongoDB possui esse comportamento, é interessante de tempos em tempos executar o comando `repairDatabase` para reescrever todo o banco de dados e otimizar o espaço utilizado.

```
> db.repairDatabase();  
{ "ok" : 1 }
```

Se ocorreu um *crash* no servidor, ou aquela eventual falta de energia, mesmo que o banco de dados suba normalmente, é uma boa prática executar o comando `repairDatabase` para corrigir algum problema, se existir.

É importante saber que, para esse comando ser executado com sucesso, é necessário existir o espaço livre de, pelo menos, o tamanho do banco de dados atual e mais 2 Gb. Isso acontece porque o comando reescreve todo o banco de dados em novos arquivos, e depois efetua a troca dos antigos pelos novos.

Veja alguns exemplos:

- tamanho do banco de dados: 10Gb = espaço livre mínimo necessário: 22Gb
- tamanho do banco de dados: 30Gb = espaço livre mínimo necessário: 62Gb.

Existe também o comando `compact` com a mesma finalidade que o `repairDatabase`, mas é executado para cada collection e não para o banco de dados inteiro.

```
> db.runCommand ( { compact: '<nome-da-collection>' } )
```

Exemplo:

```
> db.runCommand ( { compact: 'filmes' } )  
{ "ok" : 1 }
```

11.4 AUTENTICAÇÃO

Por padrão, o banco de dados não oferece nenhuma autenticação, pois o foco é facilidade no uso. Contudo, em uma empresa é muito arriscado deixar o banco de dados dessa maneira.

O MongoDB oferece autenticação em vários níveis, por banco de dados ou por collections, além de suporte a perfis (*roles*).

Existem diferentes estratégias para montar uma autenticação como um todo. Vamos listar aqui as tarefas mais comuns que envolvem praticamente todas as necessidades de um DBA.

Adicionar autenticação

Para adicionar autenticação, usamos o banco de dados `admin` e adicionamos `roles` existentes. Em seguida, alteramos o arquivo de configuração do MongoDB e reiniciamos o serviço.

A sintaxe para adicionar a autenticação é:

```
use admin  
db.createUser({user: "<nome-do-usuario>",  
    pwd: "<senha-do-usuario>",  
    roles:[ "userAdminAnyDatabase",  
        "dbAdminAnyDatabase",  
        "readWriteAnyDatabase" ]})
```

Exemplo:

```
use admin
db.createUser({user: "admin",
    pwd: "minhasenha",
    roles:[ "userAdminAnyDatabase",
        "dbAdminAnyDatabase",
        "readWriteAnyDatabase"]})
db.createUser({user: "outroAdmin",
    pwd: "minhasenha",
    roles:[ "userAdminAnyDatabase",
        "dbAdminAnyDatabase",
        "readWriteAnyDatabase"]})
```

Depois, é necessário alterar o arquivo de configuração do MongoDB (normalmente em /etc/mongod.conf) e adicionar o parâmetro de autenticação, que pode ser security.authorization ou auth , ambos com o valor true .

A lista completa de opções de configuração está disponível em <http://docs.mongodb.org/manual/reference/configuration-options/>.

Autenticação por banco de dados

Como é comum existir uma aplicação que utilize diversas collections, é uma boa abordagem criar um banco de dados por aplicação e criar uma autenticação dentro dele, permitindo ler e escrever.

Para cadastrar um usuário administrador de todo o banco de dados:

```
use admin
db.createUser(
{
    user: "<nome-do-usuario-admin>",
    pwd: "<senha-do-usuario-admin>",
    roles: [ "root" ]
})
```

)

Exemplo:

```
use meubanco
db.createUser(
{
    user: "usuarioAdmin",
    pwd: "minhasenha",
    roles: [ "root" ]
}
)
```

Existem diversos perfis (roles) no MongoDB para definição de usuários com menos privilégios que o perfil `root`, confira a lista em <https://docs.mongodb.com/manual/reference/built-in-roles/>.

Para cadastrar um usuário administrador de um único banco de dados:

```
use <meu-banco-de-dados>
db.createUser(
{
    user: "<nome-do-usuario-admin>",
    pwd: "<senha-do-usuario-admin>",
    roles: [
        { role: "readWrite", db: "<meu-banco-de-dados>" }
    ]
}
)
```

Exemplo:

```
use meubanco
db.createUser(
{
    user: "usuarioAdmin",
    pwd: "minhasenha",
    roles: [
        { role: "readWrite", db: "meubanco" }
    ]
}
```

)

Outra necessidade comum é criar um usuário só de leitura. Nesse caso, a sintaxe é parecida, apenas a `role` muda:

```
use <meu-banco-de-dados>
db.createUser(
{
    user: "<nome-do-usuario-de-leitura>",
    pwd: "<senha-do-usuario-de-leitura>",
    roles: [
        { role: "read", db: "<meu-banco-de-dados>" }
    ]
}
)
```

Exemplo:

```
use meubanco
db.createUser(
{
    user: "usuarioSomenteLeitura",
    pwd: "minhasenha",
    roles: [
        { role: "read", db: "meubanco" }
    ]
}
)
```

11.5 PROGRAMAS EXTERNOS

O MongoDB oferece alguns programas que auxiliam na análise de performance, vamos destacar dois deles a seguir.

mongostat

O programa `mongostat` funciona de maneira semelhante ao `vmstat` existente em alguns sistemas operacionais UNIX, que tem o objetivo de informar de maneira geral as operações de

consulta, atualização, alocação de memória virtual, operações de rede e conexões existentes.

Para chamar o programa, digite:

```
mongostat
```

Consulte a documentação oficial para mais detalhes:
<http://docs.mongodb.org/manual/reference/program/mongostat/>.

Na figura a seguir, temos o exemplo da base test em uso na carga da collection filmes usada no *capítulo 6 - Migrando o seu banco de dados*.

	insert	query	update	delete	getmore	command	% dirty	% used
12	*0	*0	*0	0	3 0	0.0	15.4	
insert	query	update	delete	getmore	command	% dirty	% used	
1	*0	*0	*0	0	2 0	0.0	15.4	
16	*0	*0	*0	0	4 0	0.0	15.4	
6	*0	*0	*0	0	2 0	0.0	15.4	
4	*0	*0	*0	0	2 0	0.0	15.4	
2	*0	*0	*0	0	3 0	0.0	15.4	
2	*0	*0	*0	0	2 0	0.0	15.4	
8	*0	*0	*0	0	4 0	0.0	15.4	
3	*0	*0	*0	0	3 0	0.0	15.4	
4	*0	*0	*0	0	3 0	0.0	15.4	
15	*0	*0	*0	0	5 0	0.0	15.4	
insert	query	update	delete	getmore	command	% dirty	% used	
6	*0	*0	*0	0	2 0	0.0	15.4	
10	*0	*0	*0	0	4 0	0.0	15.4	
5	*0	*0	*0	0	2 0	0.0	15.4	
5	*0	*0	*0	0	2 0	0.0	15.4	
*0	*0	*0	*0	0	2 0	0.0	15.4	
*	*	*	*	*	*	*	*	*

Figura 11.1: mongostat exibindo operações no banco test

mongotop

O programa mongotop funciona de maneira semelhante ao top existente em alguns sistemas operacionais UNIX, que tem o objetivo de informar os processos mais pesados, que estão consumindo mais recurso do banco de dados.

Para chamar o programa, digite:

```
mongotop
```

Consulte a documentação oficial para mais detalhes:
<http://docs.mongodb.org/manual/reference/program/mongotop/>.

Na figura seguinte, temos o exemplo da collection filmes da base test em uso.

ns	total	read	write
test.filmes	1ms	0ms	1ms
admin.system.roles	0ms	0ms	0ms
admin.system.version	0ms	0ms	0ms
local.startup_log	0ms	0ms	0ms
local.system.replset	0ms	0ms	0ms
messages.events	0ms	0ms	0ms
messages.messages	0ms	0ms	0ms
st.cacheDeDoisdocumentos	0ms	0ms	0ms
test.collection	0ms	0ms	0ms
test.comentarios	0ms	0ms	0ms

Figura 11.2: mongotop exibindo operações na collection filmes do banco test

Programas externos

Se a autenticação for ativada, é preciso conceder ao usuário o privilégio da role root para executar esses programas externos, através da seguinte sintaxe:

```
db.grantRolesToUser("<usuario>", [{"role": "root", "db": "admin"}]);
```

Exemplo:

```
db.grantRolesToUser("admin", [{"role": "root", "db": "admin"}]);
```

Em seguida, para iniciar o programa externo, é necessário informar os parâmetros para autenticação:

```
mongotop --authenticationDatabase <banco-de-dados-admin>
-u <usuário-administrador>
-p <senha-do-administrador>
```

Exemplo:

```
mongotop --authenticationDatabase admin -u admin -p admin
```

Os outros programas, como o mongo console ou o mongostat , recebem os mesmos parâmetros para efetuar autenticação.

11.6 BACKUP

Operações de backup dos dados são essenciais para garantir o bom funcionamento e uma restauração rápida se necessário.

Backup frio

O backup de tudo com banco fora do ar (conhecido como *backup frio*) é efetuado com o comando mongodump , com essa sintaxe:

```
<derruba-serviço-do-MongoDB>
mkdir <diretorio-de-backup>
cd <diretorio-de-backup>
mongodump --dbpath <diretorio-de-dados-do-mongodb>
<sobe-serviço-do-MongoDB>
```

Exemplo:

```
service mongod stop
mkdir /bkp/dados/
cd /bkp/dados/
mongodump --dbpath /var/lib/mongodb/
service mongod start
```

Os arquivos do banco de dados serão gerados dentro do

diretório dump em que foi executado o mongodump . Dentro dele, serão criados subdiretórios de cada banco de dados, e dentro de cada um deles teremos dois arquivos para cada collection: um arquivo pequeno com os metadados da collection, nome e informação dos índices (<nome-da-collection>.metadata.json) e outro maior, que contém os dados da collection em formato Binary JSON (BSON).

Uma alternativa mais lenta, mas também interessante, é o comando mongoexport , que permite exportar os dados em formato CSV ou JSON .

A sua sintaxe simplificada é:

```
mongoexport -d <banco-de-dados>
            -c <collection>
            --out <arquivo-de-saida>
```

Exemplo de exportar para o arquivo seriados.json :

```
fb@cascao > mongoexport -d test -c seriados --out seriados.json
connected to: 127.0.0.1
exported 3 records
```

Backup quente

O backup de tudo com banco no ar (conhecido como *backup quente*) também é efetuado com o comando mongodump , com essa sintaxe:

```
service mongod start
mkdir <diretorio-de-backup>
cd <diretorio-de-backup>
mongodump
```

Exemplo:

```
service mongod start  
mkdir /bkp/dados/  
cd /bkp/dados/  
mongodump
```

Backup de apenas um banco de dados

Informando o parâmetro `db` , podemos fazer o backup de apenas um banco de dados:

```
mkdir <diretorio-de-backup>  
cd <diretorio-de-backup>  
mongodump --db <nome-do-banco>
```

Exemplo:

```
mkdir /bkp/dados/  
cd /bkp/dados/  
mongodump --db test
```

Backup de apenas uma collection

Informando o parâmetro `db` e `collection` , podemos fazer o backup de apenas uma collection:

```
mkdir <diretorio-de-backup>  
cd <diretorio-de-backup>  
mongodump --db <nome-do-banco>  
          --collection <nome-da-collection>
```

Exemplo:

```
mkdir /bkp/dados/  
cd /bkp/dados/  
mongodump --db test  
          --collection filmes
```

11.7 RESTORE

Para restaurar os backups feitos com `mongodump`, utilizamos o `mongorestore`, que deve ser sempre executado com o banco de dados fora do ar.

restore full

O *restore full* ou completo é a restauração de todos os bancos de dados do MongoDB e é feita com a seguinte sintaxe:

```
<derruba-serviço-no-MongoDB>
cd <diretório-de-backup>
mongorestore --dbpath <diretório-de-dados-do-mongodb> dump
```

Exemplo:

```
service mongod stop
cd /bkp/dados/
mongorestore --dbpath /var/lib/mongo dump
```

restore parcial

Para restaurar apenas um banco de dados específico, a sintaxe é semelhante:

```
<derruba-serviço-no-MongoDB>
cd <diretório-de-backup>
mongorestore --dbpath <diretório-de-dados-do-mongodb>
    --db <nome-do-banco>
    dump/<nome-do-banco>
```

Exemplo de *restore* do banco de dados `test`:

```
service mongod stop
cd /bkp/dados/
mongorestore --dbpath /var/lib/mongo
    --db test
    dump/test
```

Se o banco já existir, o MongoDB fará um *merge* do atual com

o *dump* existente.

Para restaurar removendo o banco existente, usamos o parâmetro `drop`:

```
<derruba-serviço-no-MongoDB>
cd <diretório-de-backup>
mongorestore --drop
    --dbpath <diretório-de-dados-do-mongodb>
    --db <nome-do-banco>
    dump/<nome-do-banco>
```

Exemplo:

```
service mongod stop
cd /bkp/dados/
mongorestore --drop
    --dbpath /var/lib/mongo
    --db test
    dump/test
```

11.8 EXIBIR OPERAÇÕES RODANDO

O MongoDB oferece dois comandos bem interessantes que ajudam bastante na administração do banco de dados.

O comando `db.currentOp` exibe as operações em execução no momento:

```
mongo> db.currentOp()
{"inprog" :
[{
    "opid" : 123,
    "op" : "query"
    ...
}]
}
```

Para exibir mais detalhes na busca, use `db.currentOp(true)`.

Se necessário, é possível derrubar um processo desses com o comando `db.killOp`:

```
mongo> db.killOp(123)
{ "info" : "attempting to kill op" }
```

Em ambientes com muitas conexões simultâneas, o resultado pode ser centenas de linhas, pois se uma conexão durar alguns milisegundos, ela será listada também.

Nessa situação, uma boa opção é exibir as operações que estão demorando mais de cinco segundos:

```
mongo> db.currentOp().inprog.forEach(
  function(op) {
    if(op.secs_running > 5) printjson(op);
  }
)
```

11.9 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- a ativar autenticação;
- a criar usuários e seus acessos aos banco de dados;
- a fazer backup do banco de dados completo e parcial;
- a restaurar o backup do banco de dados completo e parcial;
- a exibir operações rodando.

No próximo capítulo analisaremos a questão de `replica set` e `sharding`, quando usar e como usar.

MONGODB EM CLUSTER

Se o seu banco ficou grande demais para uma única máquina ou necessita de alta disponibilidade, chegou a hora de entender um pouco mais sobre os conceitos de `replica set` e `sharding`.

12.1 ALTA DISPONIBILIDADE

Imagine que você tem seus dados replicados em diferentes lugares (chamados **nós**). E se algum nó cair, outro assumirá seu lugar. Chamamos isso de alta disponibilidade, pois a sua aplicação não deixa de funcionar.

Essa arquitetura é chamada de *replica set* (ou conjunto de servidores replicados), em que podemos ter entre 2 e 12 servidores (mas o mínimo sugerido é 3).

Na figura a seguir, temos o exemplo de uma arquitetura de 3 nós: o primeiro é o nó primário (em que os dados são lidos e escritos), e os outros dois são os nós secundários (os dados são apenas copiados do nó primário e são usados apenas para consulta).

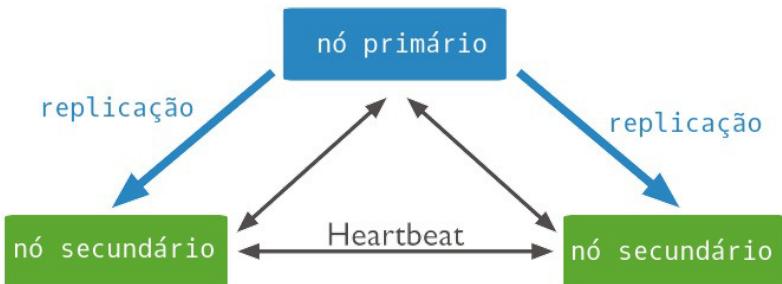


Figura 12.1: MongoDB replica set com três nós

A cada dois segundos, os nós se conversam e verificam se estão ativos. Isso é chamado de *heartbeat*.

Se o nó primário cair, um dos nós secundários é eleito para ser o novo nó primário. Novos nós secundários podem ser adicionados a qualquer instante, sem interromper o *cluster* inteiro.

Um nó pode estar na mesma máquina que outro nó, desde que em portas distintas. Entretanto, em termos de alta disponibilidade, isso não é interessante, já que uma falha de hardware poderia comprometer mais de um nó ao mesmo tempo.

Para informar ao MongoDB que se está usando `replicaset`, é necessário adicionar ao arquivo de configuração (`mongodb.conf`) o parâmetro `replSet`, ou `replication.replSetName:`, informando o nome do cluster criado.

12.2 TESTANDO DOIS REPLICA SETS

Para teste, vamos iniciar dois serviços do MongoDB na mesma máquina em diferentes portas para simular um *replica set* de duas

máquinas. Em um terminal, iniciamos o primeiro nó na porta 27017 do cluster `rs0` :

```
mkdir /tmp/mongodb/rs0-0 /tmp/mongodb/rs0-1 -p
mongod --port 27017
--dbpath /tmp/mongodb/rs0-0
--replSet rs0
```

Em outro inicial, iniciamos o segundo nó na porta 27018 do mesmo cluster `rs0` :

```
mongod --port 27018
--dbpath /tmp/mongodb/rs0-1
--replSet rs0
```

Agora precisamos definir quem é o nó primário e quem é o secundário. Para isso, vamos nos conectar ao nó primário e executar o comando `rs.initiate` para ativar o cluster:

```
> mongo --port 27017
MongoDB shell version: 3.2.9
connecting to: 127.0.0.1:27017/test

> rs.initiate()
{
  "info2" : "no configuration specified.
  Using a default configuration for the set",
  "me" : "cascao:27017",
  "ok" : 1
}
```

Podemos consultar quantos nós temos em nosso cluster no array `members` :

```
> rs.conf()
{
  "_id" : "rs0",
  "version" : 1,
  ...
  "members" : [
    {
```

```
"_id" : 0,  
"host" : "cascao:27017",  
...  
}
```

Vamos adicionar o nó secundário da porta 27018 com o comando `rs.add`:

```
rs0:PRIMARY> rs.add("cascao:27018")  
{ "ok" : 1 }
```

Com isso, verificando novamente a configuração, percebemos que o array de `members` contém um novo elemento:

```
rs0:PRIMARY> rs.conf()  
{  
  "_id" : "rs0",  
  "version" : 2,  
  ...  
  "members" : [  
    {  
      "_id" : 0,  
      "host" : "cascao:27017",  
      ...  
    },  
    {  
      "_id" : 1,  
      "host" : "cascao:27018",  
      ...  
    }  
  ]
```

Para adicionar novos nós, utilize o mesmo comando `rs.add`.

12.3 PARTICIONAMENTO

Sua collection chegou à casa dos bilhões de registros e fisicamente não cabe mais em um único servidor. Nesse caso, chegou a hora de quebrar a sua collection por uma chave, o que é chamado de *sharding*, ou particionamento. O termo *shard* do

inglês vem de caco de vidro.

Um exemplo é se existe 30Tb de dados que não cabem em um servidor, então é possível particionar a collection e dividir em três máquinas de 10Tb cada. Para tal, é preciso escolher a melhor maneira de distribuir uniformemente a informação entre os servidores. Esse critério é feito na criação do particionamento definindo uma chave (um filtro) para dividir as informações.

A figura seguinte mostra um exemplo com uma collection única de três terabytes que pode ser particionada em três partições de um terabyte cada, espalhada em três máquinas distintas.

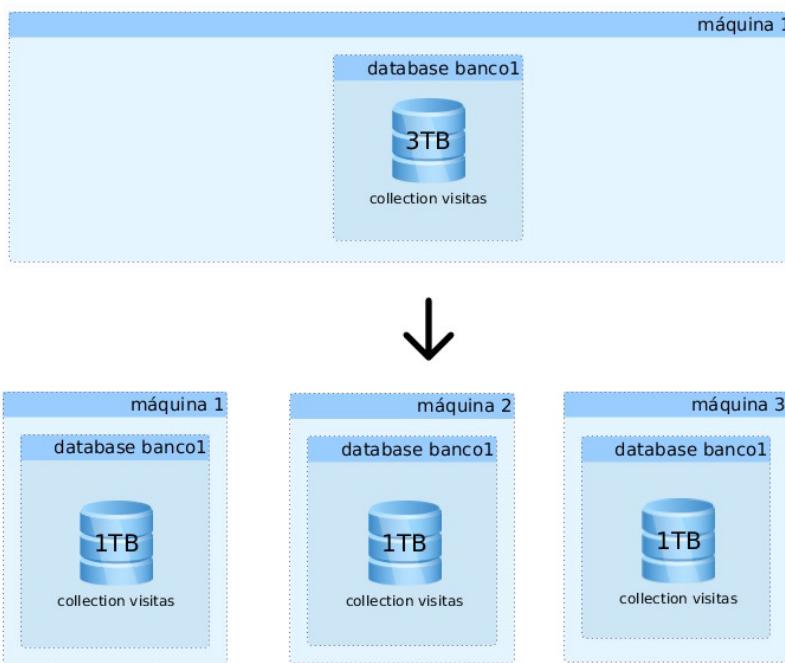


Figura 12.2: Sharding

Arquitetura de sharding

O MongoDB com particionamento exige três diferentes serviços:

- Shards : são as instâncias do MongoDB que contêm os seus dados particionados. Cada shard pode ser um *replica set*.
- Config Servers : são os servidores que têm mapeados os metadados de toda a arquitetura.
- Query Routing Instances : essa instância com que sua aplicação vai se comunicar, é ela que direciona as leituras e escritas para os shards (nenhuma aplicação acessa os shards diretamente).

Essa arquitetura é ilustrada na figura a seguir. Repare que a aplicação acessa apenas as instâncias de `query router`, e ela faz a distribuição dos acessos aos dados nos shards. Note que, em vez do executável `mongod`, é utilizado para sharding o `mongos`.

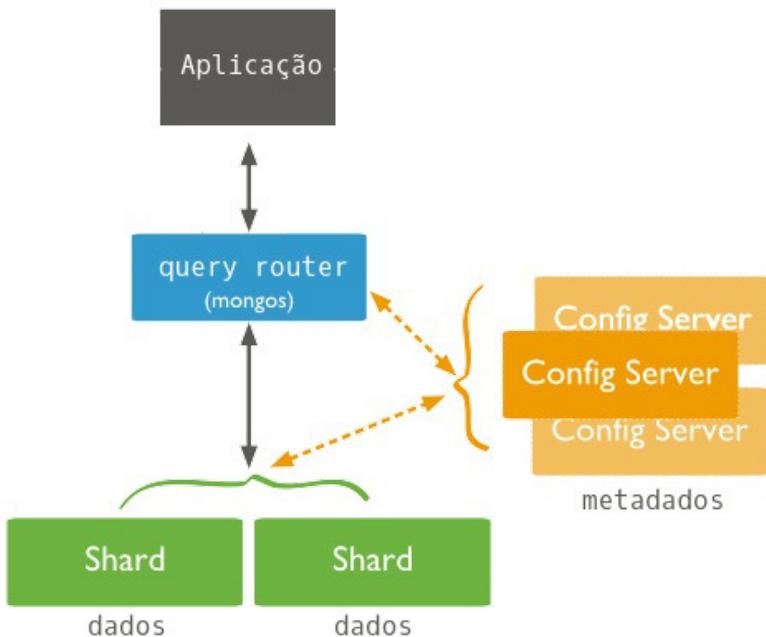


Figura 12.3: MongoDB com particionamento (sharding)

Em ambiente de produção, é recomendado:

- `Shards instances` : no mínimo dois serviços, e cada shard replicado com *replica set*;
- `Config instances` : no mínimo três serviços;
- `Query routing instances` : no mínimo dois serviços.

Sharding de exemplo

Vamos fazer uma configuração mínima de sharding para exemplificar o funcionamento:

- `Config instances` : um serviço
- `Query routing instances` : um serviço
- `Shards instances` : dois serviços

Tudo rodará na mesma máquina, com os serviços em portas distintas.

Exemplo de config instance

Vamos criar o exemplo mais simples possível de `sharding` , na ordem a seguir:

1. criamos três instâncias de *config server*;
2. criamos três instância de um *replica set*;
3. conectamos na primeira e configuramos as replica sets;
4. criamos uma instância do *router*;
5. adicionamos as replica sets ao shard.

Vamos configurar um cluster na mesma máquina, para tal cada serviço terá uma porta TCP/IP diferente. O primeiro passo é criar as instâncias do `config server` na porta 27020, com essas configurações:

```
> cd /var/lib/mongodb  
> mkdir configSrv1 configSrv2 configSrv3  
> mongod --configsvr --port 26050  
  --dbpath /var/lib/mongodb/configSrv1  
> mongod --configsvr --port 26051  
  --dbpath /var/lib/mongodb/configSrv2  
mongod --configsvr --port 26052  
  --dbpath /var/lib/mongodb/configSrv3
```

Em seguida, criamos os dois replica sets `replicaSetTest` e `replicaSetTest2` , o primeiro com três instâncias e o segundo com duas:

```
> cd /var/lib/mongodb  
> mkdir replSet1 replSet2 replSet3 replSet4 replSet5  
> mongod --port 31000 --replSet replicaSetTest --dbpath /var/lib/mongodb/replSet1  
> mongod --port 31001 --replSet replicaSetTest --dbpath /var/lib/mongodb/replSet2  
> mongod --port 31002 --replSet replicaSetTest --dbpath /var/lib/mongodb/replSet3  
> mongod --port 31003 --replSet replicaSetTest2 --dbpath /var/lib/mongodb/replSet4  
> mongod --port 31004 --replSet replicaSetTest2 --dbpath /var/lib/mongodb/replSet5
```

Conectamos à primeira instância para criar o replica set `replicaSetTest` :

```
> mongo --port 31000  
rs.initiate()  
rs.add("cascao:31001")  
rs.add("cascao:31002")  
rs.status()
```

Conectamos à segunda instância para criar o replica set `replicaSetTest2` :

```
> mongo --port 31003  
rs.initiate()  
rs.add("cascao:31004")  
rs.status()
```

Vamos criar o nosso `router` com o executável `mongos` (`s` de sharding) se contendo aos config servers :

```
mongos --configdb cascao:26050,cascao:26051,cascao:26052
```

O passo final é conectar-se ao `router` e adicionar as replica sets como sharding:

```
mongos> sh.addShard("replicaSetTest/cascao:31000")  
{ "shardAdded" : "replicaSetTest", "ok" : 1 }  
mongos> sh.addShard("replicaSetTest2/cascao:31003")  
{ "shardAdded" : "replicaSetTest2", "ok" : 1 }
```

```
mongos> sh.status()
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  ...
shards:
{ "_id" : "replicaSetTest",
  "host" : "replicaSetTest/cascao:31000,cascao:31001,cascao:31002"}
{ "_id" : "replicaSetTest2",
  "host" : "replicaSetTest2/cascao:31003,cascao:31004"}
...

```

Finalmente, temos o nosso MongoDB completo configurado com suporte ao *sharding*. No final, teremos uma lista de processos do MongoDB no ar:

```
> ps -ef | grep mongo
mongod --port 31000 --replSet replicaSetTest --dbpath /var/lib/mo
ngodb/replSet1
mongod --port 31001 --replSet replicaSetTest --dbpath /var/lib/mo
ngodb/replSet2
mongod --port 31002 --replSet replicaSetTest --dbpath /var/lib/mo
ngodb/replSet3
mongod --port 31003 --replSet replicaSetTest2 --dbpath /var/lib/mo
ngodb/replSet4
mongod --port 31004 --replSet replicaSetTest2 --dbpath /var/lib/mo
ngodb/replSet5
mongod --configsvr --port 26050 --dbpath /var/lib/mongodb/configS
rv1
mongod --configsvr --port 26051 --dbpath /var/lib/mongodb/configS
rv2
mongod --configsvr --port 26052 --dbpath /var/lib/mongodb/configS
rv3
mongos --configdb cascao:26050,cascao:26051,cascao:26052
```

Usando sharding

Com a nossa arquitetura do MongoDB ligada, precisamos ativar as collections que desejamos particionar. Isso é feito com o comando `h.enableSharding`:

```
mongos> use test
switched to db test
mongos> sh.enableSharding("test");
{ "ok" : 1 }
```

Na aplicação, não existe mudança alguma. As consultas continuam sendo feitas da mesma maneira:

```
mongos> db.filmes.findOne();
{
  "_id" : NumberLong(140744),
  "titulo" : "\"Back from the Edge\" (2010)",
  "ano" : "2010",
  "nota" : 0,
  "votos" : NumberLong(0),
  "categorias" : [ ],
  "diretores" : [ ],
  "atores" : [ ]
}
```

Vamos criar um índice para dividir os filmes pelo ano:

```
mongos> db.filmes.ensureIndex({ "ano":"hashed"});
{
  "raw" : {
    "replicaSetTest/casco:31000,casco:31001,casco:31002" : {
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1,
      "$gleStats" : {
        "lastOpTime" : Timestamp(1474253813, 1),
        "electionId" : ObjectId("7fffffff0000000000000007")
      }
    }
  },
  "ok" : 1
}
```

Em seguida, vamos distribuir (particionar) esses dados:

```
mongos> sh.shardCollection("test.filmes", {"ano":"hashed"})
{ "collectionssharded" : "test.filmes", "ok" : 1 }
mongos> db.filmes.count()
```

1111000

Finalmente, vamos identificar como os dados estão distribuídos:

```
mongos> db.filmes.getShardDistribution()

Shard replicaSetTest at
replicaSetTest/cascao:31000,cascao:31001,cascao:31002
data : 420.79MiB docs : 1026720 chunks : 8
estimated data per chunk : 52.59MiB
estimated docs per chunk : 128340

Shard replicaSetTest2 at
replicaSetTest2/cascao:31003,cascao:31004
data : 223.28MiB docs : 548297 chunks : 7
estimated data per chunk : 31.89MiB
estimated docs per chunk : 78328

Totals
data : 644.07MiB docs : 1575017 chunks : 15
Shard replicaSetTest contains 65.33% data,
65.18% docs in cluster, avg obj size on shard : 429B
Shard replicaSetTest2 contains 34.66% data,
34.81% docs in cluster, avg obj size on shard : 427B
```

Observe que a distribuição ficou aproximadamente 65% e 35% em cada shard . O primeiro replica set, por ter um nó a mais, ficou com a maior parte. No final, o nosso ambiente de exemplo será semelhante ao da figura:

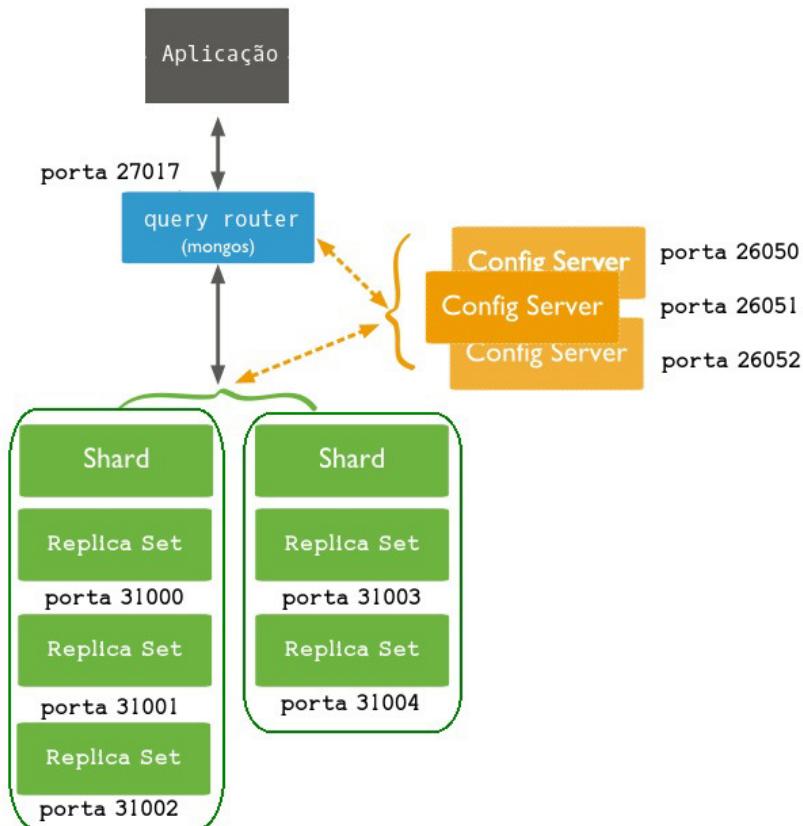


Figura 12.4: Exemplo do MongoDB com particionamento (sharding)

12.4 PRÓXIMOS PASSOS

Certifique-se de que aprendeu a:

- criar *clusters* com *replica set*;
- particionar dados com *sharding*;
- adicionar uma collection a um *sharding*.

CAPÍTULO 13

CONTINUE SEUS ESTUDOS

Agora em diante, para aprimorar os conhecimentos no MongoDB:

- Faça os excelentes treinamentos gratuitos na *MongoDB University* — <https://university.mongodb.com>.
- Participe do grupo internacional de usuários do MongoDB — <https://groups.google.com/forum/#!forum/mongodb-user>.
- Participe do grupo brasileiro de usuários do MongoDB — <https://groups.google.com/forum/#!forum/br-mongodb>.
- Participe do grupo brasileiro do *nossa livro de MongoDB* — <http://forum.casadocodigo.com.br>.

E acompanhe os principais blogs:

- Blog oficial — <http://blog.mongodb.org>
- Blog sobre MongoDB e outras bases NoSQL — <https://blog.compose.io>

APÊNDICE A — INSTALANDO MONGODB

O MongoDB é um excelente banco de dados NoSQL, mas está em sua fase NERD, em que tudo se faz com linha de comando. Apesar da excelente performance e estabilidade, por enquanto não apresenta preocupação com a interface gráfica.

A sua instalação é bem simples. Mas, ao contrário da maioria dos fabricantes, na plataforma Windows, ele não instala como serviço. Então, será preciso fazer este ajuste manualmente. A instalação em Linux atualmente é a mais completa, pois instala e configura como serviço.

A documentação completa está no site: <http://docs.mongodb.org/manual/installation/>. Vamos resumir alguns passos em seguida.

Instalação em Ubuntu Linux

Inicialmente, adicionamos as chaves de criptografia para garantir a instalação de pacotes oficiais gerados pela MongoDB:

```
# sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80  
--recv EA312927  
Executing: /tmp/tmp.tnmM5pMWFY/gpg.1.sh --keyserver
```

```
hkp://keyserver.ubuntu.com:80
--recv
EA312927
gpg: requesting key EA312927 from hkp server keyserver.ubuntu.com
gpg: key EA312927: public key "MongoDB 3.2 Release
      Signing Key <packaging@mongodb.com>" imported
gpg: Total number processed: 1
gpg:          imported: 1  (RSA: 1)
#
#
```

Em seguida, adicionamos o repositório do MongoDB aos existentes:

```
# echo "deb
http://repo.mongodb.org/apt/ubuntu
precise/mongodb-org/3.2
multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-3.2.list
#
```

Atualizamos o nosso repositório local:

```
# apt-get update
...
Get:7 http://repo.mongodb.org/apt/ubuntu
...
...
```

E finalmente instalamos:

```
# apt-get install -y mongodb-org
...
The following NEW packages will be installed:
  mongodb-org  mongodb-org-mongos
  mongodb-org-server  mongodb-org-shell
  mongodb-org-tools
...
Setting up mongodb-org-shell (3.2.9) ...
Setting up mongodb-org-server (3.2.9) ...
Setting up mongodb-org-mongos (3.2.9) ...
Setting up mongodb-org-tools (3.2.9) ...
Setting up mongodb-org (3.2.9) ...
#
#
```

Em versões recentes do Ubuntu, que o sistema de inicialização é o `Systemd`, precisamos criar um serviço manualmente para ele. Crie o arquivo `/etc/systemd/system/mongodb.service` com o conteúdo:

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
Documentation=https://docs.mongodb.org/manual

[Service]
User=mongodb
Group=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf

[Install]
WantedBy=multi-user.target
```

Para subir o serviço no `Systemd`, usamos:

```
# sudo systemctl start mongodb
```

Nas versões mais antigas do Ubuntu, o comando é:

```
# sudo service mongod start
```

Para verificarmos se o serviço está ativo, o comando é:

```
# sudo systemctl status mongodb
● mongodb.service - High-performance,
   schema-free document-oriented database
   Loaded: loaded
   Active: active (running)
     Docs: https://docs.mongodb.org/manual
 Main PID: 7964 (mongod)
  CGroup: /system.slice/mongodb.service
          └─7964 /usr/bin/mongod
                  --quiet --config /etc/mongod.conf
```

Observe que, no resultado do comando, temos a palavra

running , comprovando que o banco de dados está no ar. Para ativar o serviço no próximo boot, faça o seguinte:

```
# systemctl enable mongodb
Created symlink from
/etc/systemd/system/multi-user.target.wants/mongodb.service
to /etc/systemd/system/mongodb.service.
```

Para não ativar o serviço no próximo boot, seria:

```
# systemctl disable mongodb
Removed symlink
/etc/systemd/system/multi-user.target.wants/mongodb.service.
```

O arquivo de log do MongoDB será `/var/log/mongodb/mongod.log` .

Instalação em Windows

Vamos baixar a versão para Windows pelo site <http://www.mongodb.org/downloads> e executar:

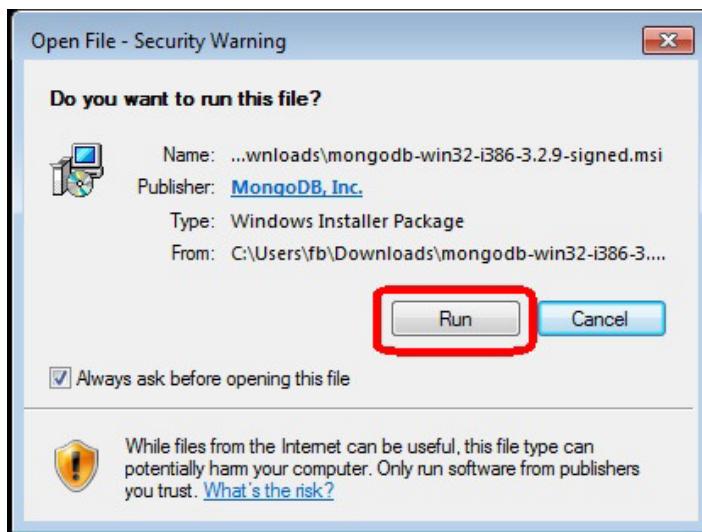


Figura 14.1: Instalação de MongoDB para Windows 32 bits

Iniciamos a instalação clicando em Next :

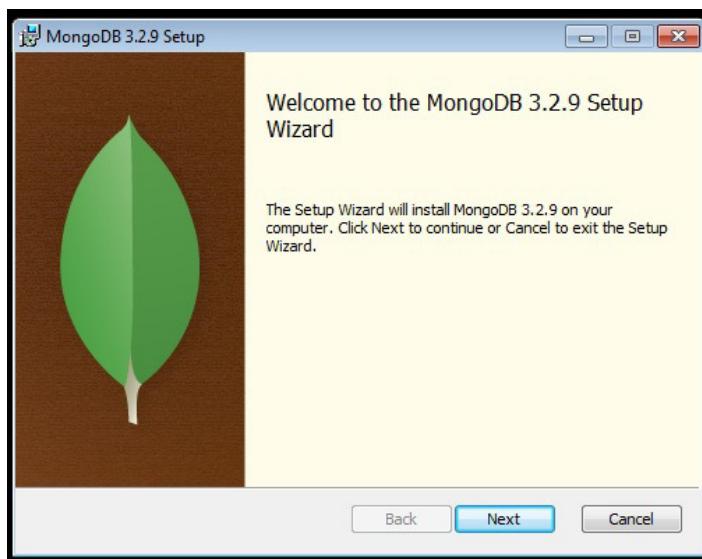


Figura 14.2: Início da instalação

Aceitamos os termos de uso também:



Figura 14.3: Termos de uso

Selecionamos a opção de instalação customizada (para facilitar o caminho do banco de dados):

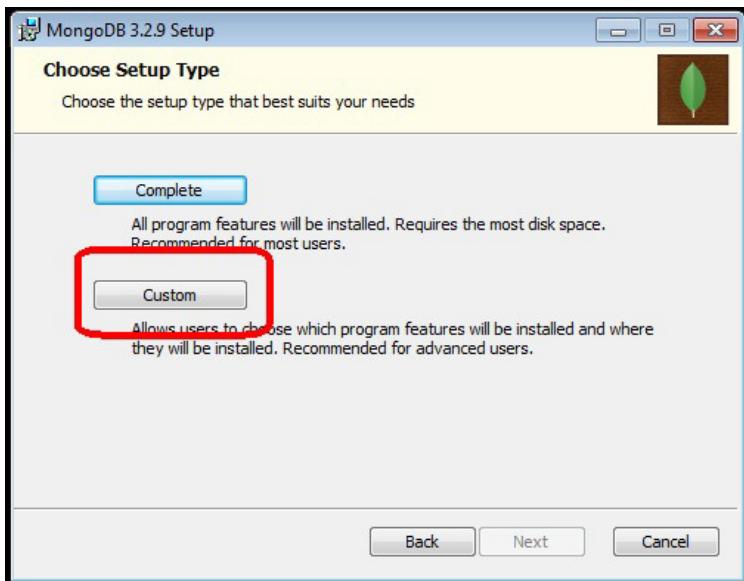


Figura 14.4: Escolhendo o tipo de instalação

Selecionamos a opção para alterar o caminho do banco de dados:

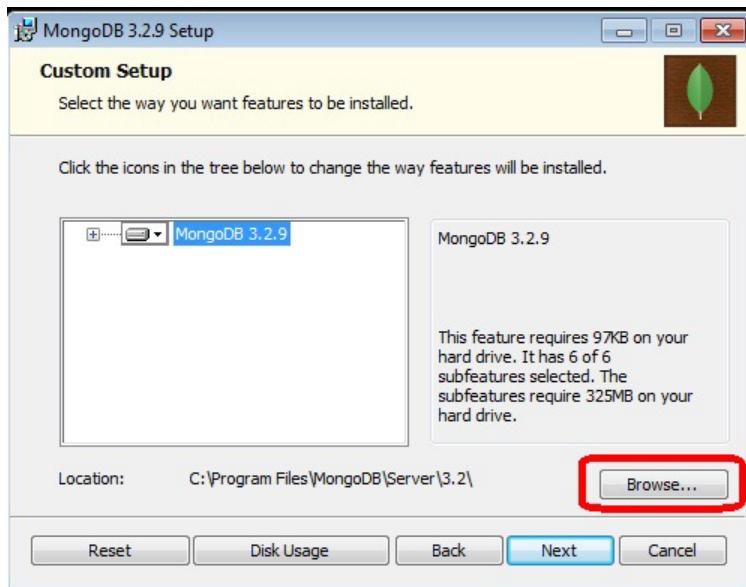


Figura 14.5: Alterando o caminho de instalação

Informamos o novo caminho para o banco de dados:

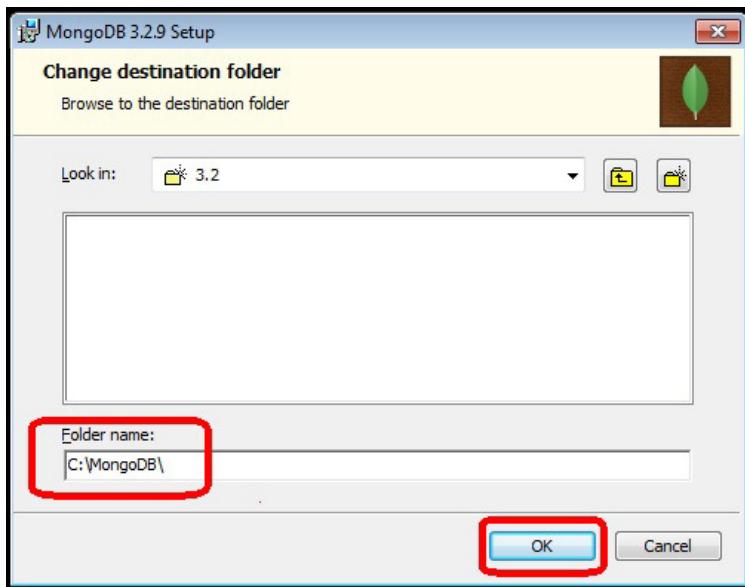


Figura 14.6: Escolhendo o tipo de instalação

Continuamos a instalação com o novo caminho:

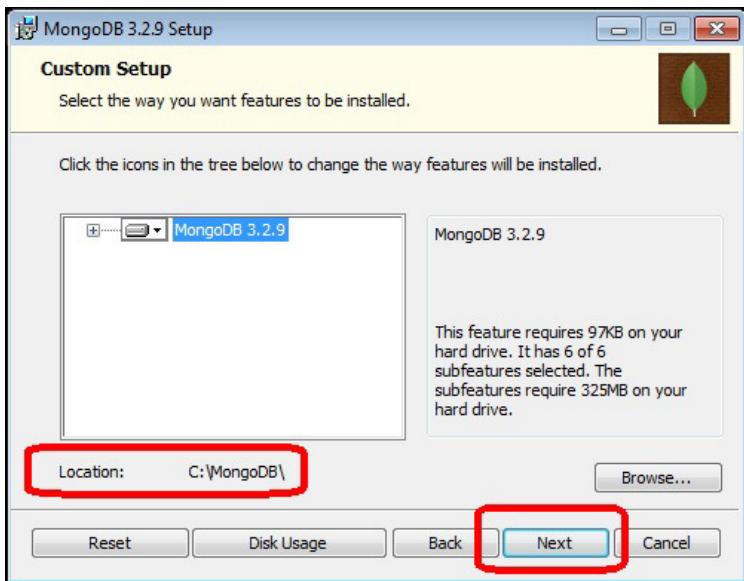


Figura 14.7: Escolhendo o tipo de instalação

Finalmente, clicamos em `Install` para instalar:

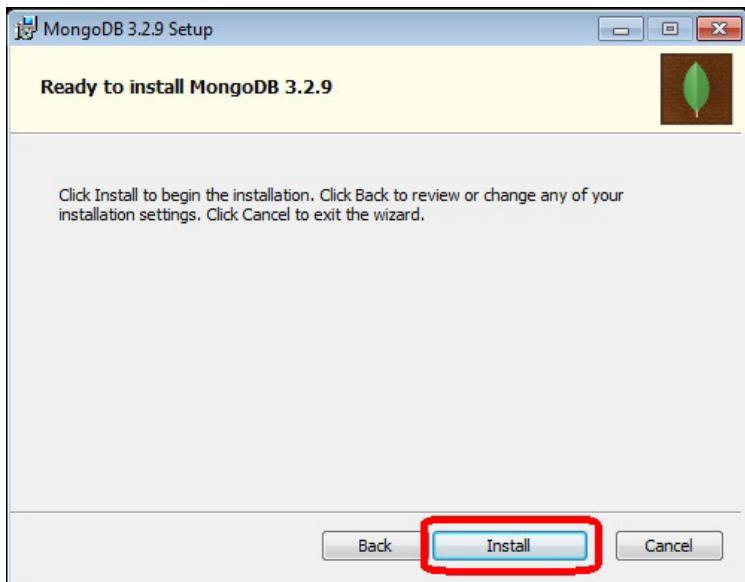


Figura 14.8: Copiando os arquivos com o instalador

A instalação será concluída em poucos instantes.



Figura 14.9: Finalizando a instalação

Vamos adicionar os executáveis do MongoDB ao PATH do Windows. Para isso, acessamos o painel de controle e modificamos algumas variáveis de ambiente:

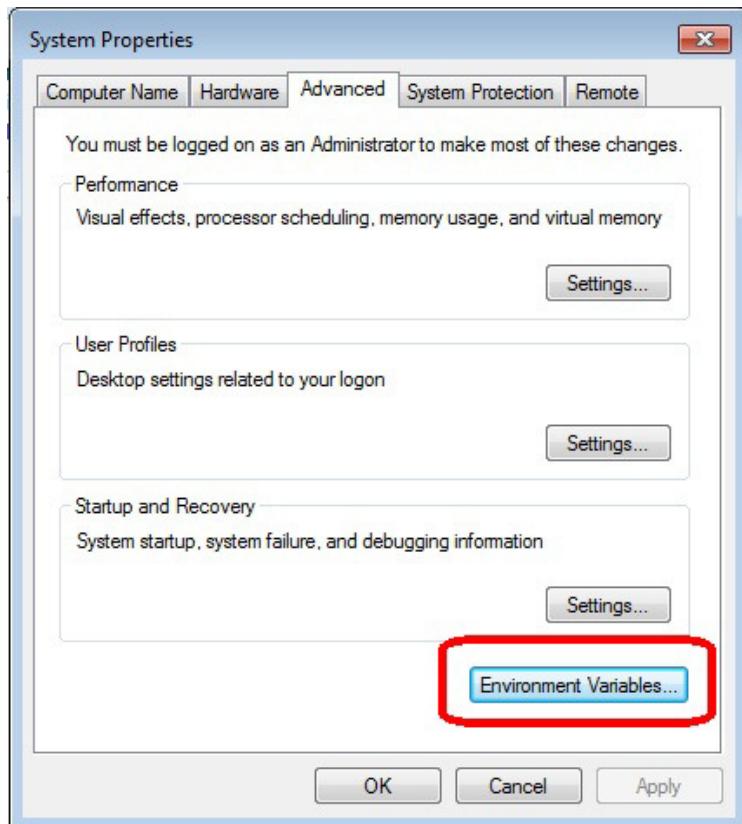


Figura 14.10: Alterando variáveis de ambiente

Clicando na opção `New`, criamos uma nova variável `MONGODB_HOME` que contém o valor de `C:\MongoDB\`.

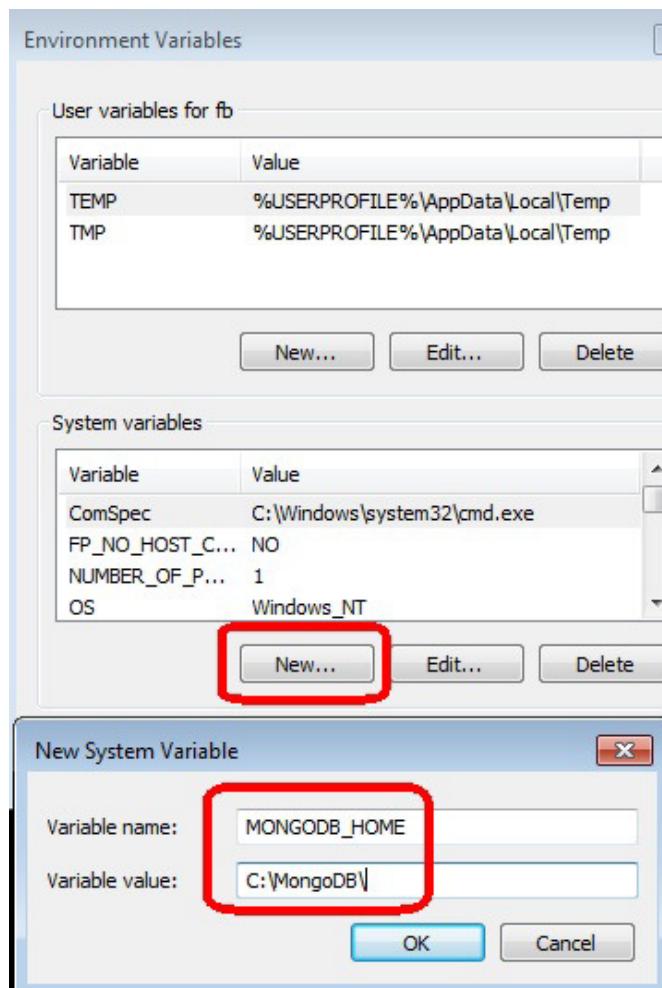


Figura 14.11: Alterando variáveis de ambiente

Em seguida, editamos a variável de ambiente PATH e adicionamos ao final %MONGODB_HOME\BIN :

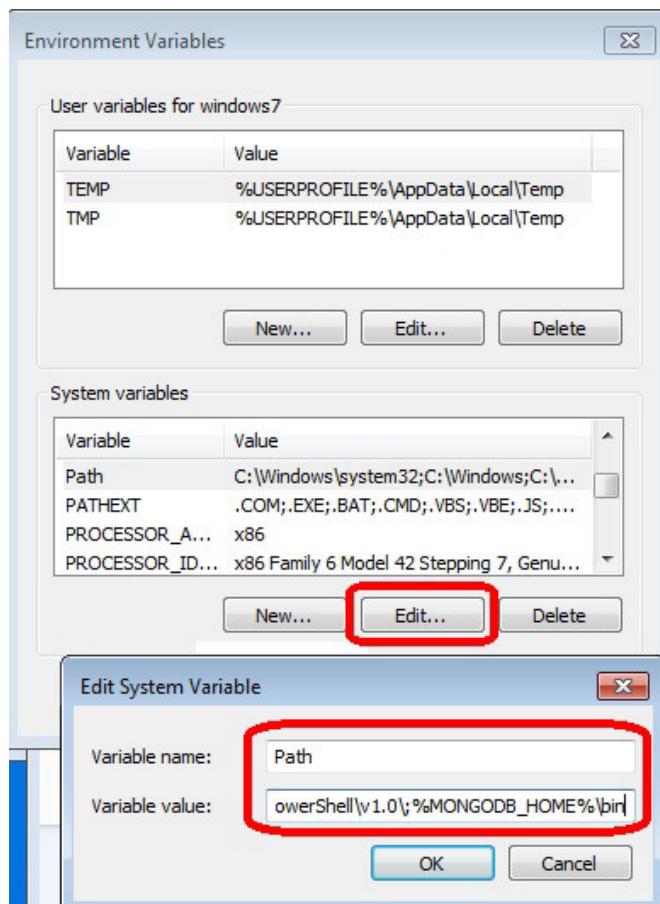


Figura 14.12: Alterando variáveis de ambiente

Chamamos o prompt de comando para instalar o serviço. Iniciamos criando os diretórios dos dados com:

```
mkdir c:\data\db  
mkdir c:\data\log
```

Com isso, conseguimos subir o banco de dados de maneira manual, apenas chamando o comando mongod sem nenhum

parâmetro. Entretanto, essa maneira não é muito prática. O interessante é configurar como serviço, para subir automaticamente junto com o Windows.

Inicialmente criamos o arquivo de configuração em C:\MongoDB\mongod.cfg , com o seguinte conteúdo:

```
logpath=c:\data\log\mongod.log  
dbpath=c:\data\db
```

E finalmente criamos o serviço:

```
sc.exe create MongoDB binPath=\""C:\MongoDB\bin\mongod.exe\" --se  
rvice  
--config=\"C:\MongoDB\mongod.cfg\"" DisplayName="MongoDB"  
start="auto"
```

O resultado será semelhante a esse:

```
C:\WINDOWS\system32>  
C:\WINDOWS\system32>sc.exe create MongoDB binPath=\""C:\MongoDB\b  
in\mongod.exe\"  
--service --config=\"C:\MongoDB\mongod.cfg\"" DisplayName="MongoDB"  
start="auto"  
[SC] CreateService ÉXITO
```

```
C:\WINDOWS\system32>
```

O arquivo de log do MongoDB será c:\data\log\mongod.log .

Instalação em Mac OS X

A instalação é feita através do Homebrew:

```
valholl:Homebrew leandropincini$ clear
valholl:Homebrew leandropincini$ cd
valholl:~ leandropincini$ clear
valholl:~ leandropincini$ brew install mongodb
⇒ Downloading https://homebrew.bintray.com/bottles/mongodb-3.2.9.mavericks.bottle.tar.gz
#####
⇒ Pouring mongodb-3.2.9.mavericks.bottle.tar.gz
⇒ Caveats
To have launchd start mongodb now and restart at login:
  brew services start mongodb
Or, if you don't want/need a background service you can just run:
  mongod --config /usr/local/etc/mongod.conf
⇒ Summary
  ↗ /usr/local/Cellar/mongodb/3.2.9: 17 files, 245.8M
valholl:~ leandropincini$
```

Figura 14.13: Instalando o MongoDB pelo Homebrew no Mac OS X

Criamos um *alias* para o MongoDB instalar como um serviço:

```
vim vim
1 # start mongodb's mongod service (installed with homebrew)
2 alias mongodb_start='launchctl load /usr/local/opt/mongodb/homebrew.mxcl.mongodb.plist'
3
4 # stop mongodb's mongod service (installed with homebrew)
5 alias mongodb_stop='launchctl unload /usr/local/opt/mongodb/homebrew.mxcl.mongodb.plist'
6
```

Figura 14.14: Instalando o MongoDB como serviço

Em seguida, usamos `mongodb_start` / `mongodb_stop` :

```
valholl:~ leandropincini$ source aliases
valholl:~ leandropincini$ mongodb_
mongodb_start mongodb_stop
valholl:~ leandropincini$ mongodb_start
valholl:~ leandropincini$ mongo
MongoDB shell version: 3.2.9
connecting to: test
> exit
bye
valholl:~ leandropincini$ mongodb_stop
valholl:~ leandropincini$ mongo
MongoDB shell version: 3.2.9
connecting to: test
2016-09-08T21:20:40.227-0300 W NETWORK  [thread1] Failed to connect to 127.0.0.1:27017, reason: errno:61 Connection refused
2016-09-08T21:20:40.228-0300 E QUERY  [thread1] Error: couldn't connect to server 127.0.0.1:27017, connection attempt failed :
connect@src/mongo/shell/mongo.js:229:14
@Connect():16
exception: connect failed
valholl:~ leandropincini$
```

Figura 14.15: Manipulando serviços do MongoDB no Mac OS X

O arquivo de log do MongoDB será /usr/local/var/log/mongodb/mongo.log .

Testando a instalação

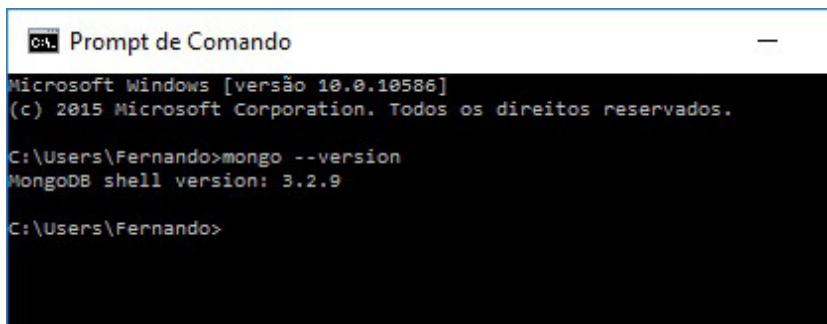
Com o MongoDB no ar, conectamos e digitamos o comando para informar a versão instalada:

```
db.version()
```

O resultado será semelhante a esse em Linux e Mac OS X:

```
fb@cascao ~ > mongo
MongoDB shell version: 3.2.9
connecting to: test
> db.version()
3.2.9
> exit
bye
fb@cascao ~ >
```

E a esse em Windows:



A screenshot of a Windows Command Prompt window titled "Prompt de Comando". The window shows the following text:
Microsoft Windows [versão 10.0.10586]
(c) 2015 Microsoft Corporation. Todos os direitos reservados.
C:\Users\Fernando>mongo --version
MongoDB shell version: 3.2.9
C:\Users\Fernando>

Figura 14.16: Exibindo a versão do MongoDB no Windows 7

Instalação em Docker

Docker é uma maneira virtualizada de usar diversos servidores

prontos, sendo um deles o MongoDB, disponível em https://hub.docker.com/_/mongo/. Ele é uma ótima maneira de você manter um ambiente sem se preocupar com instalação das coisas, e flexibiliza o upgrade de versão também.

Para usarmos a imagem oficial para testes, fazemos da seguinte maneira:

```
# docker run -p 27017:27017 -d mongo
```

Vamos entender os parâmetros:

- docker — executável do Docker;
- run — roda um contêiner de uma imagem;
- -p 27017:27017 — define que a porta que será usada no serviço, para mapear na porta 10000 , usamos 10000:27017 ;
- -d — roda um serviço em background (daemon);
- mongo — nome da imagem que será baixada do Docker hub e executada.

Com isso, já temos um serviço disponível funcionando, mas no momento que ele for desligado, todo o banco de dados será perdido. Para mantermos os dados para uso posterior, devemos informar ao docker o diretório externo dessa maneira:

```
# docker run -p 27017:27017 -v /opt/mongodb/data:/data/db -d mongo
```

Usando o novo parâmetro, mapeamos o diretório de dados para o diretório externo /opt/mongodb/data . Dessa maneira, temos um ambiente virtualizado gravando os dados externamente, apenas precisamos visualizar os logs.

Para ver os logs de qualquer imagem do Docker, em primeiro lugar identificamos o ID do contêiner com `ps`:

```
# docker ps
CONTAINER ID        IMAGE      ...
d0b0ae007579        mongo      ...
```

Em seguida, usamos o comando `logs` para visualizar os logs:

```
root@cascao ~]# docker logs d0b0ae007579 -f
CONTROL [initandlisten] MongoDB starting : pid=1
port=27017 dbpath=/data/db 64-bit host=d0b0ae007579
[initandlisten] Initializing full-time diagnostic data
capture with directory '/data/db/diagnostic.data'
...
```

APÊNDICE B — ROBOMONGO

Existem diversos clientes desenvolvidos para MongoDB, dentre os quais se sobressai o RoboMongo. Ele é uma excelente ferramenta open source que possui a mesma engine em JavaScript, cliente *shell* oficial. Isso significa que todos os comandos existentes funcionam também no Robomongo, mas com uma interface muito mais amigável.

Instalação

O Robomongo pode ser baixado em seu site <http://robomongo.org/> para Windows, Linux e Mac OS X. Sua instalação é bem simples e não precisa de nenhuma opção especial. Por esse motivo, só ilustraremos a instalação em Windows.



Figura 15.1: Início da instalação

São exibidos os termos de uso da licença open source (GPL) para concordar:



Figura 15.2: Termos de uso do Robomongo

Em seguida, é exibido o diretório de instalação:

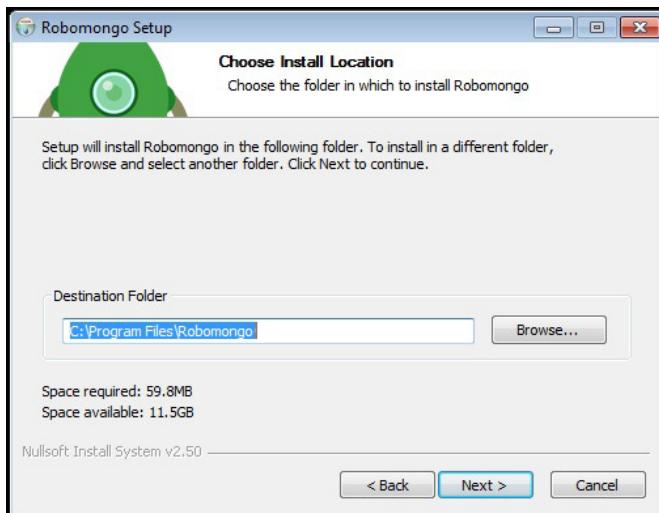


Figura 15.3: Escolhendo o diretório de instalação

Depois é informado a opção de pasta de menu de inicialização que será criado:



Figura 15.4: Opção de pasta de menu de inicialização

Finalizada a instalação, é dada a opção de iniciar o Robomongo:



Figura 15.5: Final da instalação

Configuração

Ao iniciar o Robomongo, é preciso configurar pelo menos uma conexão ao servidor. Isso é feito clicando na opção **Create** :

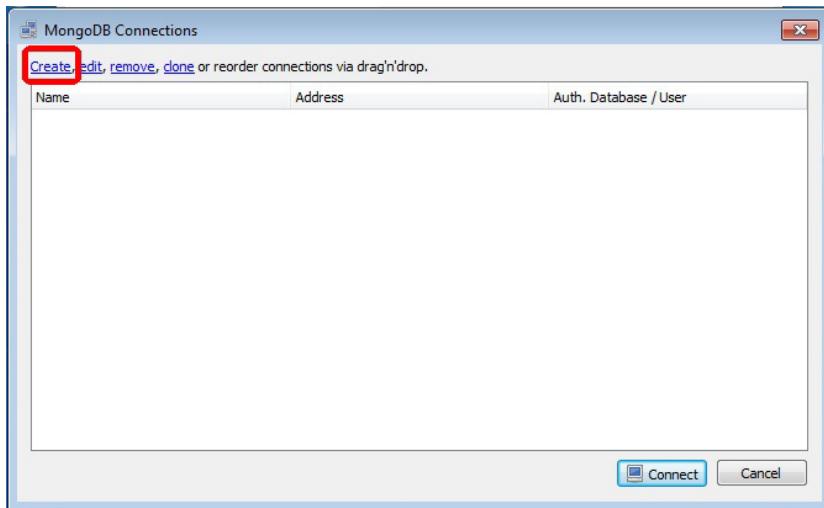


Figura 15.6: Configurando uma conexão ao servidor

Em seguida, informamos o nome da conexão. Como instalamos o MongoDB na mesma máquina, chamamos a conexão de `localhost`, que por padrão escuta requisições na porta 27017:

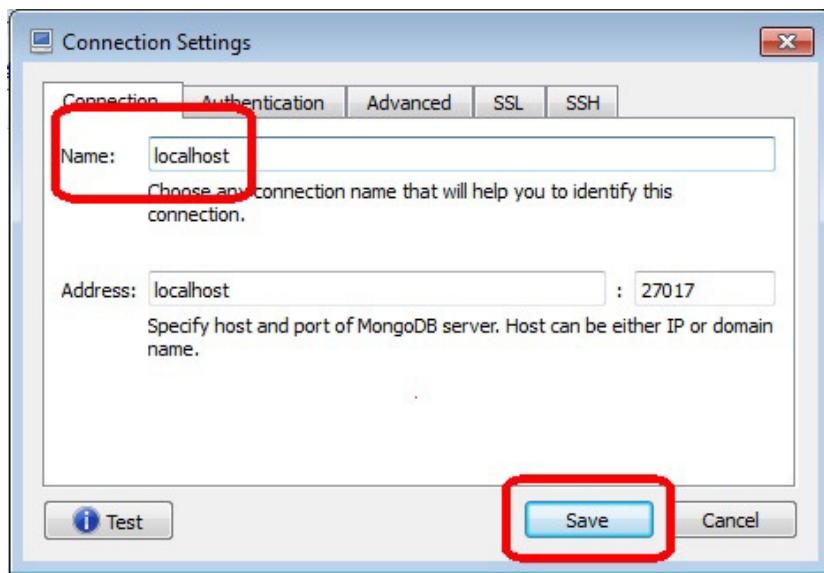


Figura 15.7: Cadastrando uma conexão local

Depois de criada a conexão, podemos conectar clicando em Connect :

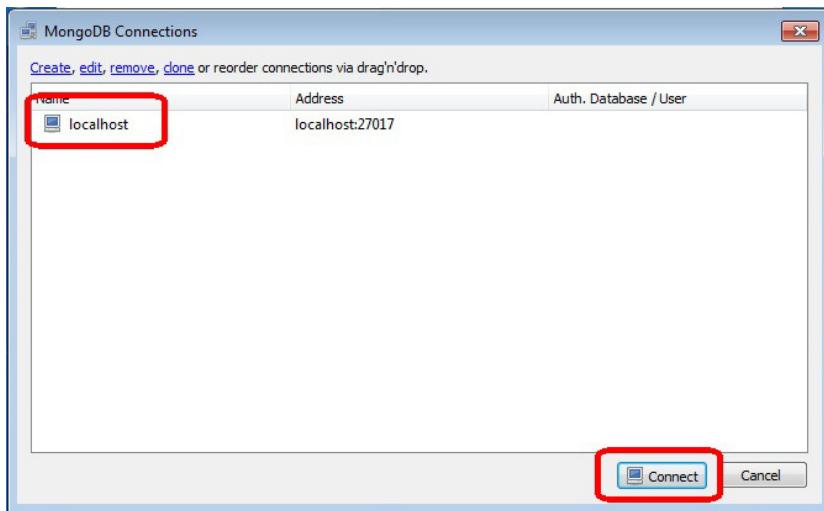


Figura 15.8: Conectando localmente ao MongoDB

Temos algumas opções nativas do Robomongo, como por exemplo, exibir informações do servidor na opção Server Status :

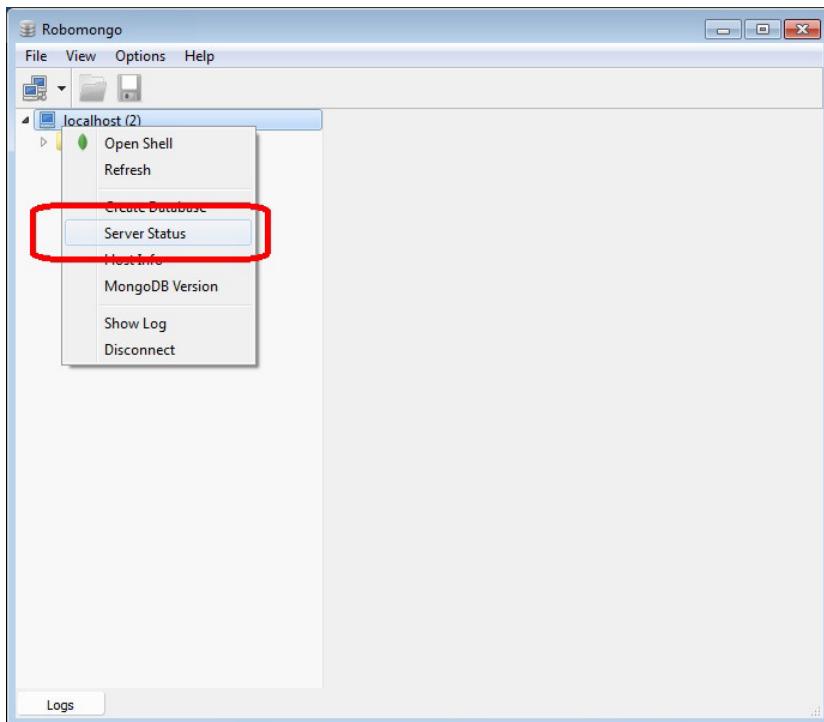


Figura 15.9: Exibindo informações do servidor

O resultado da situação do servidor é exibido de maneira amigável:

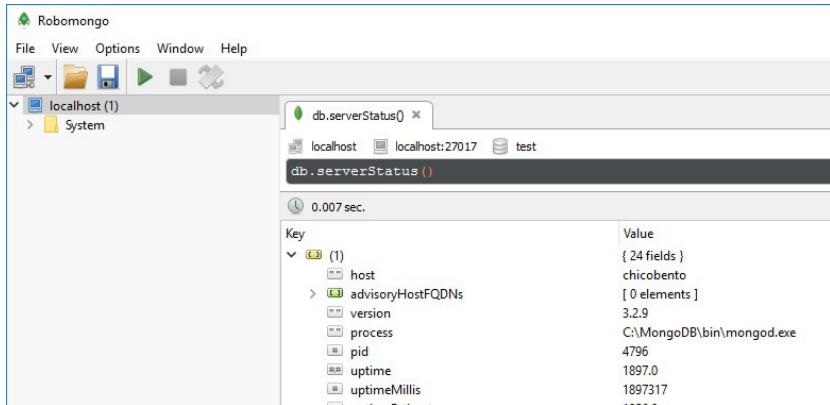


Figura 15.10: Resultado da situação do servidor

Podemos também visualizar os logs do servidor com `Control + L`, que aparecem na janela inferior:

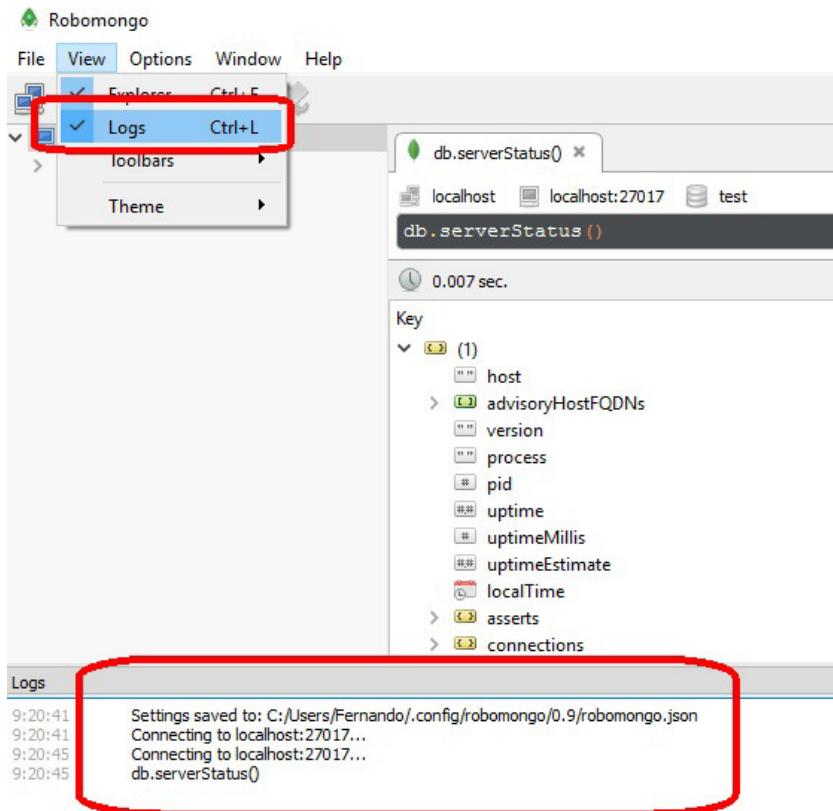


Figura 15.11: Visualizando os logs do servidor

Além disso, conseguimos navegar pelas collections existentes, e visualizamos o seu conteúdo com a opção `View Documents` :

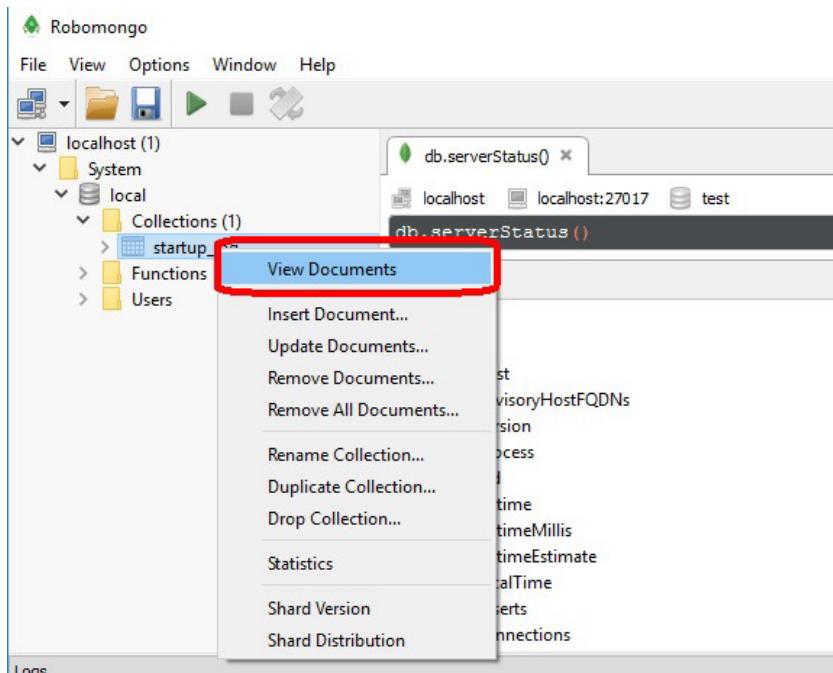


Figura 15.12: Ativando a opção de exibir o conteúdo de uma collection

Temos também o comando executado exibido em uma nova aba:

The screenshot shows the Robomongo interface. On the left, the database structure is displayed under 'localhost (1)'. A red box highlights the 'local' database. On the right, a new tab titled 'db.getCollection('startup_log').find({})' is open, showing the contents of the 'startup_log' collection. The results are presented in a table with 'Key' and 'Value' columns. Two documents are shown:

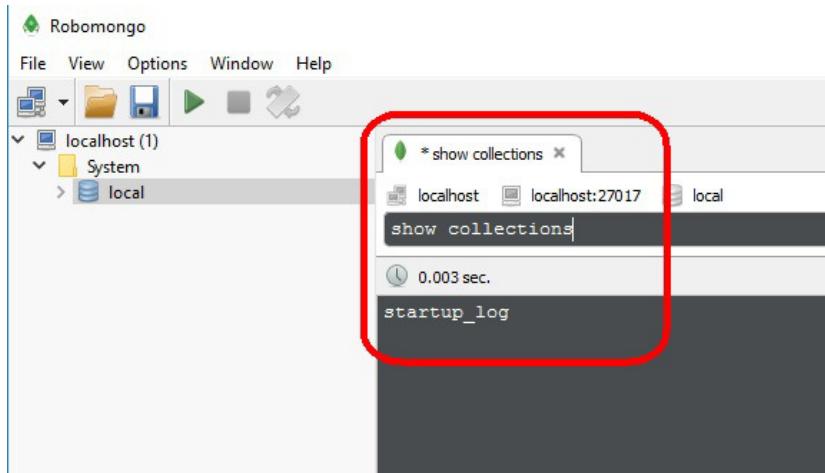
Key	Value
(1) chicobento-1473291229491	{ 7 fields } _id: chicobento-1473291229491 hostname: chicobento startTime: 2016-09-07 23:33:49.000Z startTimeLocal: Wed Sep 07 20:33:49.491 cmdLine: { 0 fields } pid: 7120 buildInfo: { 14 fields }
(2) chicobento-1473292149541	{ 7 fields }

Figura 15.13: Conteúdo de uma collection exibido em uma nova aba

E conseguimos abrir um novo terminal com a opção Open Shell :

The screenshot shows the Robomongo interface. On the left, the database structure is displayed under 'localhost (1)'. A red box highlights the 'local' database. A context menu is open over the 'local' database entry, listing several options: 'Open Shell' (which is highlighted), 'Refresh', 'Database Statistics', 'Repair Database...', and 'Drop Database...'. The rest of the interface is mostly empty.

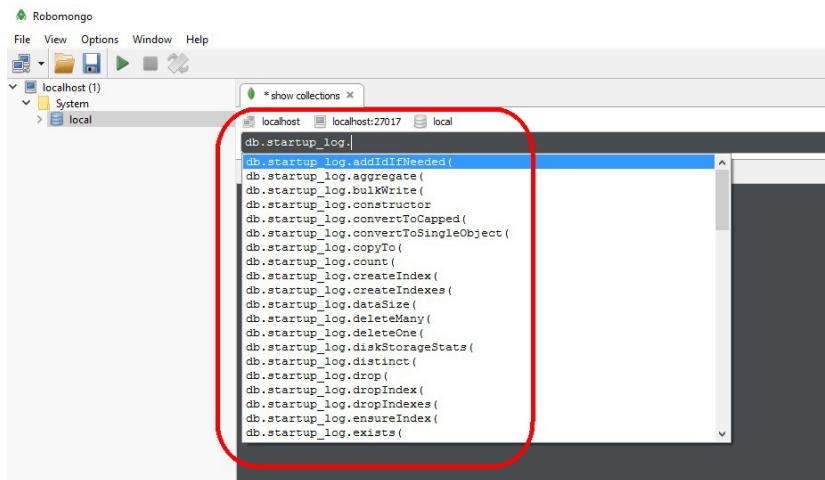
Usando o terminal para exibir as collections:



The screenshot shows the Robomongo interface. In the left sidebar, there's a tree view with 'localhost (1)' expanded, showing 'System' and 'local'. The main window has a title bar 'Robomongo' and a menu bar with 'File', 'View', 'Options', 'Window', and 'Help'. Below the menu is a toolbar with icons for file operations. A terminal window titled '* show collections *' is open, showing the command 'show collections' and its output 'startup_log'. The output area has a timestamp '0.003 sec.' and a clock icon. A red box highlights the terminal window.

Figura 15.15: Usando o terminal para exibir as collections

Um recurso muito bom é o autocomplete do terminal, que automaticamente exibe ao usuário as opções que ele pode usar:



The screenshot shows the Robomongo interface with the same setup as Figure 15.15. The terminal window now shows the command 'db.startup_log.|' followed by a list of methods: addIdIfNeeded(), aggregate(), bulkWrite(), constructor(), convertToCapped(), convertToSingleObject(), copyTo(), count(), createIndex(), createIndexes(), dataSize(), deleteMany(), deleteOne(), diskStorageStats(), distinct(), drop(), dropIndex(), dropIndexes(), ensureIndex(), and exists(). A red box highlights the list of methods.

Figura 15.16: Exemplo de autocomplete no terminal do Robomongo

APÊNDICE C — PERGUNTAS E RESPOSTAS

Não é preciso ler o livro inteiro para esclarecer algumas dúvidas simples. Vamos tentar aqui enumerar as questões mais comuns e esclarecê-las de uma vez por todas!

O que é NoSQL?

O NoSQL é um termo técnico para denominar um banco de dados que não é relacional. Normalmente, ele é do tipo banco de dados de documento, orientado a objetos, chave-valor ou de grafos. O MongoDB ocupa a primeira posição de banco de dados NoSQL de acordo com a pesquisa do site <http://db-engines.com/>.

De onde veio o nome MongoDB?

O nome veio da palavra *humongous*, que significa enorme, gigantesco, para dar a ideia de grande gerenciamento de dados.

Quem usa o MongoDB?

A lista completa com mais de 50 clientes está em <https://www.mongodb.com/who-uses-mongodb>. Seguem alguns

dos mais famosos no Brasil: IBM, Foursquare, Bosch, Cisco, eBay, McAfee, Microsoft, MTV Networks, Telefonica e The New York Times.

O MongoDB é um substituto para os bancos relacionais?

Não, ele não substitui um banco relacional, pois não possui transação ou *constraints* de referência, que quase todo sistema possui, mas ele pode ser um complemento de uma base relacional, servindo como cache, por exemplo.

Entretanto, se sua aplicação for desenhada adequadamente, ela pode usar inteiramente o MongoDB e não usar nenhuma base relacional.

O MongoDB possui constraints?

Não, o MongoDB cria um índice para cada collection, mas validações nos campos são esperadas que aconteçam na aplicação. Existe um tipo bem simples de validação, que verifica se um campo é obrigatório ou se obedece a uma expressão regular.

O MongoDB possui índices?

Sim, com MongoDB conseguimos criar índices simples e compostos (mais de um campo), inclusive para arrays. Existe também o poderoso índice de busca textual (full text search).

O MongoDB suporta transações?

Não.

O MongoDB suporta cluster?

Sim, os bancos NoSQL em geral suportam o ambiente de *cluster*, e o MongoDB não é exceção. Ele trabalha de maneira eficiente nessa arquitetura utilizando `replica set`. Para mais detalhes, consulte o *capítulo 12 — MongoDB em cluster*.

Qual o limite máximo de um registro / documento em uma collection do MongoDB? E quais são os outros limites?

O limite é de 16Mb de tamanho máximo, permitindo ter até 100 níveis de documentos aninhados. Para ter um comparativo, existem algumas versões na internet da Bíblia em formato texto que ocupam aproximadamente 4mb. Portanto, para ultrapassar o limite atual do MongoDB, um simples registro/ documento precisa ter mais texto do que quatro bíblias completas juntas.

Os nomes de campos, collections e databases podem ter até 123 bytes. Um collection pode ter até 64 índices, cada um deles pode conter entre 1 até 31 campos.

O tamanho máximo do banco de dados pode variar conforme o tipo de *file system* e o sistema operacional. Porém, a grosso modo é 4 terabytes para Windows e 54 terabytes para Linux. Consulte os limites restantes na documentação oficial <http://docs.mongodb.org/manual/reference/limits/>.

Como listar todos os bancos de dados existentes?

Utilize o comando `show databases` ou `show dbs`.

Como listar todos as collections de um banco de dados?

Utilize o comando `show collections` .

Como listar os comandos existentes?

Utilize o comando `help` e `db.listCommands()` .

Como alterar o cliente do MongoDB para que a busca com `find` traga mais/menos resultados do que a opção padrão (20)?

Antes de executar o comando, informe o novo valor com o comando seguinte, no exemplo 100:

```
DBQuery.shellBatchSize = 100;
```

Preciso fazer uma pesquisa para trazer os registros criados entre uma data e outra. Isso é possível?

Sim, existe a opção `created_on` que permite essa busca, e podemos usar com duas variáveis, conforme o exemplo:

```
var dt_ini = new Date(2016,1,1,10,0,0);
var dt_fim = new Date(2016,1,1,20,0,0);
db.collection.find({created_on: {$gte: dt_ini, $lt: dt_fim}});
```

Como fazer uma pesquisa para trazer os últimos cinco registros criados?

```
db.collection.find().sort({$natural: -1}).limit(5);
```

Como fazer uma pesquisa para trazer o último item de um array?

Supondo a collection:

```
db.seriados.insert({  
    "_id":4,  
    "nome":"Chaves",  
    "personagens": [  
        "Seu Barriga",  
        "Quico",  
        "Chaves",  
        "Chiquinha",  
        "Nhonho",  
        "Dona Florinda"]})
```

Usamos o comando `slice` (verbo fatiar) para trazer o último:

```
db.seriados.find({}, { personagens: {$slice: -1}})
```

O resultado seria:

```
{  
    "_id" : 4.0,  
    "nome" : "Chaves",  
    "personagens" : [  
        "Dona Florinda"  
    ]  
}
```

CAPÍTULO 17

APÊNDICE D — UPGRADE DA VERSÃO 2.6 PARA MONGODB 3.X

Em 2015, o MongoDB mudou para a versão 3 com várias melhorias, e a migração é bem fácil de fazer. Não precisa nem ao menos exportar os dados. Basicamente, instala-se a versão nova e atualiza-se o serviço.

Certamente, a versão 32 bits para Windows não é recomendada para ambiente de produção, já que é limitada em 2Gb de tamanho. Entretanto, ela é boa para pequenos testes e vamos mostrar como instalá-la mais adiante.

O que mudou na versão 3

Até as versões anteriores ao 3, o MongoDB possuía apenas uma implementação de armazenamento (*storage engine*), chamada MMAPv1. Essa implementação continua sendo o padrão da versão 3, sendo assim, é possível manter o ambiente funcionando sem manutenção alguma apenas com a atualização da versão.

Contudo, uma nova storage engine foi criada, a WiredTiger, que promete ser de 7 a 10 vezes mais rápida que a MMAPv1 e com

uma taxa de compressão de até 80%. Para tirar proveito dessas melhorias, é preciso migrar todos os bancos de dados existentes.

Isso pode ser feito facilmente rodando a sequência de comandos a seguir. Inicialmente, com o MongoDB 3 no ar, exportamos todos os dados com o comando:

```
mongodump
```

Em seguida, paramos o serviço e movemos os arquivos do banco atual para um diretório de backup.

```
service mongodb stop  
mv /var/lib/mongodb/* /bkp-mongodb/
```

Depois, alteramos o arquivo de configuração `/etc/mongodb.conf`, adicionando a linha `engine: wiredTiger` conforme o exemplo:

```
storage:  
  dbPath: "/var/lib/mongodb"  
  engine: wiredTiger
```

Depois da alteração, iniciamos o serviço do MongoDB:

```
service mongodb start
```

Agora, já com a storage engine WiredTiger rodando, restauramos o backup com o comando:

```
mongorestore
```

Depois de validar a sua base de dados restaurada, remova o diretório de backup com:

```
rm -rf /bkp-mongodb/
```

Seu ambiente está migrado e pronto para uso. Se desejar voltar

para a storage engine padrão (MMAPv1), basta derrubar o serviço, comentar o parâmetro adicionado do WiredTiger, mover os arquivos do diretório de backup para o diretório de origem e depois subir o serviço.

Download da versão 32 bits para Windows

O link oficial de download é <https://www.mongodb.org/downloads>. Para Windows, está disponível apenas a versão de 64 bits.

Felizmente existe uma versão de 32 bits que podemos usar para testes, mas ela não é 100% estável. Esta é RC (Release Candidate). O seu link de download é <https://www.mongodb.org/dl/win32/>, ou <http://downloads.mongodb.org/win32/mongodb-win32-i386-3.0.0-rc11-signed.msi>.

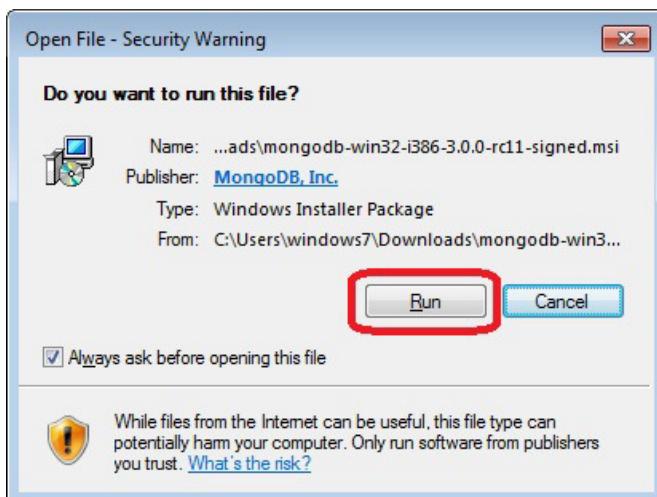


Figura 17.1: Instalação de MongoDB 3 para Windows 32 bits

A instalação é muito semelhante à da versão 2 (figura anterior e figura a seguir).

Instalação dos binários

Antes de iniciar, derrube o serviço do MongoDB 2.6 criado no *Apêndice A — Instalando MongoDB*.



Figura 17.2: Instalação de MongoDB 3 para Windows 32 bits

A estrutura de diretórios é diferente, portanto a versão 3.0 pode ser instalada junto com a 2.6 (figura seguinte):

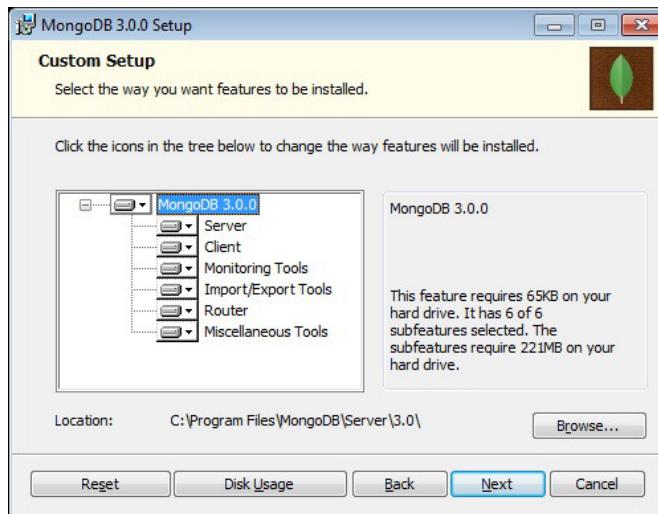


Figura 17.3: Diretórios do MongoDB 3

Em seguida, altere o valor da variável `MONGODB_HOME` para `C:\Program Files\MongoDB\Server\ 3.0` :

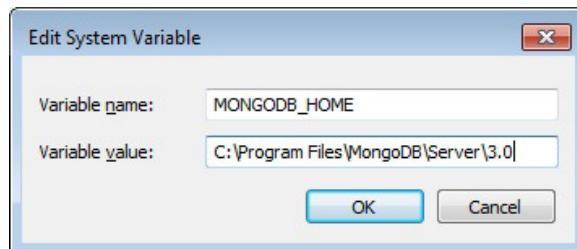


Figura 17.4: Alteração da variável MONGODB_HOME

Instalação do serviço do Windows

Copie o arquivo de configuração de `C:\Program Files\MongoDB 2.6 Standard\mongod.cfg` para `C:\Program Files\MongoDB\mongod.cfg`.

```
C:\Program Files\MongoDB>type mongod.cfg
logpath=c:\data\log\mongod.log
dbpath=c:\data\db
```

Em seguida, instale o serviço do MongoDB 3:

```
C:\Program Files\MongoDB>sc.exe
create MongoDB3
binPath= "\"C:\Program Files\MongoDB\Server\3.0\bin\mongod.exe\""
--service --config=\"C:\Program Files\MongoDB\mongod.cfg\""
DisplayName= "MongoDB 3.0"
start= "auto"
[SC] CreateService SUCCESS
```

E remova o serviço do MongoDB 2:

```
C:\Program Files\MongoDB>sc.exe delete MongoDB
[SC] DeleteService SUCCESS
```

Inicie o serviço do MongoDB 3 com:

```
C:\Program Files\MongoDB>net start MongoDB3
The MongoDB 3.0 service is starting.
The MongoDB 3.0 service was started successfully.
```

Com tudo no ar, vamos validar a atualização a seguir.

Testando o novo ambiente

Chame no console o mongo e verifique a versão instalada:

```
C:\Program Files\MongoDB>mongo
MongoDB shell version: 3.0.0-rc11
connecting to: test
>
> db.version();
3.0.0-rc11
```

Além disso, faça um count na collection megasena para conferir se os dados foram migrados corretamente:

```
> db.megasena.count();
```

```
1607
```

```
> exit
```

```
bye
```

```
C:\Program Files\MongoDB>
```

Seu ambiente está pronto e já rodando a versão 3!