

MongoDB

Construa novas aplicações com
novas tecnologias



Casa do
Código

FERNANDO BOAGLIO

© 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.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

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

Agradecimentos

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

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!

Prefácio

Por que construir aplicações novas com tecnologia antiga?

É impressionante como aprendemos o que bancos de dados relacionais são e o que 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

Esse 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, 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 e FAQ

Foram criados dois apêndices focados em instalação: o apêndice A, que é 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, por exemplo, se ele suporta transações ou quais as grandes empresas que o usam.

Código-fonte

O código-fonte deste livro está disponível no endereço <https://github.com/boaglio/mongodb-casadocodigo>, onde foram criadas tags para cada um dos capítulos, para facilitar a compreensão da evolução do nosso sistema de filmes.

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	9
3	MongoDB básico	11
3.1	Conceitos	11
3.2	Acessando o MongoDB	13
3.3	Exemplo da Mega-Sena	14
3.4	Buscar registros	16
3.5	Adicionar registros	21
3.6	Atualizar registros	26
3.7	Remover registros	32
3.8	Criar e remover collections	33
3.9	Alterando uma coluna de uma collection	34
3.10	Melhorando as buscas	35
3.11	Capped Collection	39
3.12	Próximos passos	40
4	Schema design	41
4.1	Relacionando uma collection para muitas	43
4.2	Relacionando muitas collection para muitas	45

4.3	Tudo em uma collection	45
4.4	Schema design na prática	45
4.5	Protótipo	46
4.6	Sistema Meus filmes relacional	48
4.7	Sistema Meus filmes no MongoDB	50
4.8	Próximos passos	51
5	Conversando com MongoDB	53
5.1	O sistema de seriados	53
5.2	Seriados em PHP	54
5.3	Java	62
5.4	Play Framework	69
5.5	Ruby on Rails	71
5.6	Node.js	73
5.7	Qt	75
5.8	Próximos passos	77
6	Migrando o seu banco de dados	79
6.1	IMDB simplificado	79
6.2	Migrando de um banco de dados relacional	84
6.3	Migrando para nuvem	86
6.4	Próximos passos	95
7	Buscas avançadas	97
7.1	Operadores de comparação	98
7.2	Operador distinct	99
7.3	Expressões regulares	100
7.4	Operadores lógicos	100
7.5	Operadores unários	101
7.6	Operador estilo LIKE	102
7.7	Incrementando valores	106
7.8	Próximos passos	107

8 Busca geoespacial	109
8.1 O banco de dados	109
8.2 Usando o sistema web	112
8.3 Entendo o sistema web	113
8.4 Indo além	115
8.5 Próximos passos	115
9 Aggregation Framework	117
9.1 Por que não usar Map Reduce	117
9.2 Explorando o Aggregation Framework	119
9.3 Próximos passos	126
10 Aumentando a performance	127
10.1 Criar um índice	129
10.2 Listar os índices criados	130
10.3 Remover um índice criado	130
10.4 Índice textual	131
10.5 Criar índice em background	133
10.6 Próximos passos	134
11 MongoDB para administradores	135
11.1 Ajuste de performance	135
11.2 Gerenciando espaço em disco	136
11.3 Autenticação	137
11.4 Programas externos	140
11.5 Backup	142
11.6 Restore	144
11.7 Exibir operações rodando	146
11.8 Próximos passos	146
12 MongoDB em cluster	149
12.1 Alta disponibilidade	149
12.2 Testando dois replica sets	150
12.3 Particionamento	152
12.4 Próximos passos	159

13 Continue seus estudos	161
14 Apêndice A. Instalando MongoDB	163
15 Apêndice B. Robomongo	177
16 Apêndice C. Perguntas e respostas	193

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 vai mudar suas regras tão cedo, portanto, desenhamos o problema em cima das regras

(ou limitações) relacionais e a partir deles criamos uma camada de manipulação desses dados pela nossa aplicação, seja ela desktop, web ou mobile.

1.1 O SISTEMA NA MANEIRA TRADICIONAL

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 1.1.

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`.

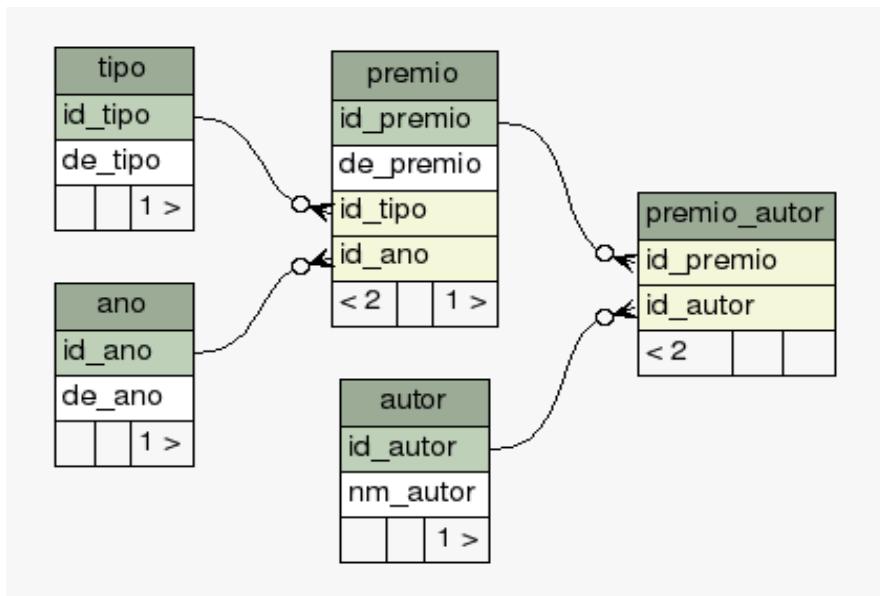


Fig. 1.1: Modelo relacional dos ganhadores

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 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) e nem transações, além de não ter as limitações de uma tabela relacional. Dentro de uma coluna, você pode ter um array, 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;
- 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 [16](#).

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

CAPÍTULO 2

JSON veio para ficar

Com o crescimento de serviços no começo do século, o tráfego de informações também aumentou, e 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, onde 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:

```
{  
    "país": "Brasil",  
    "população": 201011823,  
    "PIB total em trilhões de dólares": 2.422,  
    "faz fronteira com": [  
        "Argentina",  
        "Bolívia",  
        "Colômbia",  
        "Guiana Francesa",  
        "Guiana",  
        "Paraguai",  
        "Peru",  
        "Suriname",  
        "Uruguai",  
        "Venezuela"  
    ],  
    "cidades": {  
        "capital" : "Brasília",  
        "mais populosa": "São Paulo"  
    }  
}
```

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.

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 o site <http://jsonprettyprint.com/>.

2.1 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- o que é JSON;
- os tipos existentes no JSON;
- a sintaxe do 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 [14](#).

3.1 CONCEITOS

O MongoDB é um *document database* (banco de dados de documentos), mas não são os documentos da família Microsoft, mas 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 podem 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 3.1 mostra uma *replica set* rs1 que contém dois databases, e cada um deles possui duas collections.

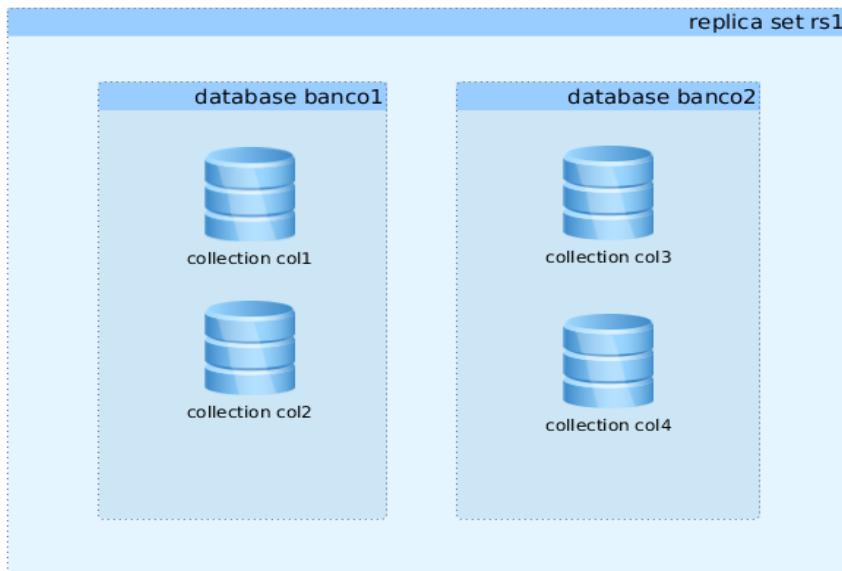


Fig. 3.1: Conceitos básicos

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

A figura 12.2 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.

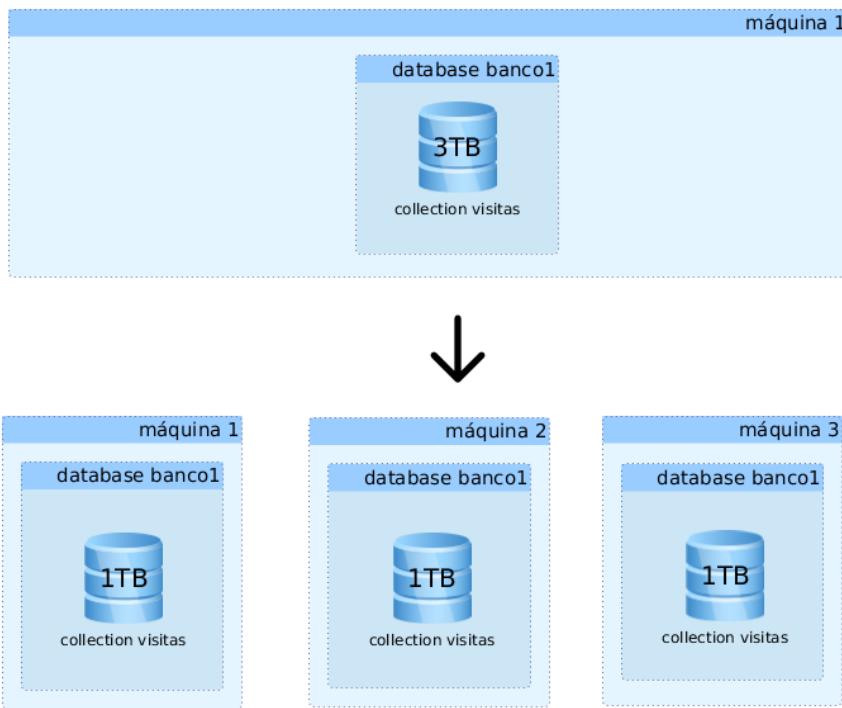


Fig. 3.2: Sharding

3.2 ACESSANDO O MONGODB

O acesso ao MongoDB pode ser feito via console (executável `mongo`) ou via Robomongo (para instalar, consulte [15](#)).

Qualquer comando funcionará nos dois, mas o Robomongo é mais amigável, além de possuir 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>;
```

Exemplo:

```
db.colecao1.count();
```

3.3 EXEMPLO DA MEGA-SENA

No site da Caixa, estão disponíveis para download todos os resultados da Mega-Sena em formato HTML (<http://www1.caixa.gov.br/loterias/loterias/megasena/download.asp>) .

Copiando o HTML, colando em uma planilha e gravando no formato CSV, podemos facilmente importar os valores para o MongoDB.

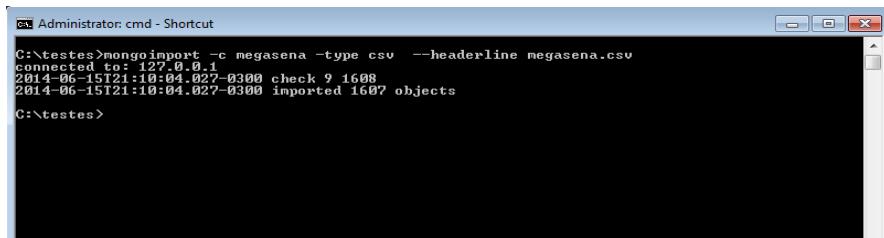
Baixe o arquivo `megasena.csv` (<https://github.com/boaglio/mongodb-casadocodigo/blob/master/capitulo-03/megasena.csv>) , copie para um diretório de testes e execute o comando:

```
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 3.3.



A screenshot of a Windows Command Prompt window titled "Administrator: cmd - Shortcut". The command entered is "mongoimport -c megasena -type csv --headerline megasena.csv". The output shows the connection to the database and the successful import of 1607 objects. The command prompt then returns to the directory "C:\testes".

```
C:\Administrator: cmd - Shortcut  
C:\testes>mongoimport -c megasena -type csv --headerline megasena.csv  
connected to: 127.0.0.1  
2014-06-15T21:10:04.027-0300 check 9 1608  
2014-06-15T21:10:04.027-0300 imported 1607 objects  
C:\testes>
```

Fig. 3.3: Importar dados da Mega-Sena

Em Linux, o resultado é parecido com esse. Note que aparece o total de documentos (objetos) importados: 1607 registros dentro da collection `megasena`.

```
fb@cascao > mongoimport  
  -c megasena
```

```
-type csv  
--headerline megasena.csv  
connected to: 127.0.0.1  
2014-06-15T21:10:04.027-0300 check 9 1608  
2014-06-15T21:10:04.027-0300 imported 1607 objects
```

Em um banco relacional tradicional, para qualquer carga de dados é necessário criar a estrutura (a tabela) que receberá os dados. No MongoDB não precisa 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: 2.6.1  
connecting to: test  
> db.megasena.count();  
1607  
> 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 3.4).

Key	Value	Type
ns	{ 14 fields }	Object
count	1607	Int32
size	797072	Int32
avgObjSize	496	Int32
storageSize	2793472	Int32
numExtents	5	Int32
nindexes	1	Int32
lastExtentSize	2097152	Int32
paddingFactor	1.000000	Double
systemFlags	1	Int32
userFlags	1	Int32
totalIndexSize	65408	Int32
indexSizes	{ 1 fields }	Object
ok	1.000000	Double

Fig. 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("427539a3425c3245a124b211"), "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("427539a3425c3245a124b211"),
    "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
}
```

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`.

Esse campo 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("427539a3425c3245a124b211"), "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. Para resolver esse problema, basta adicionar no comando o sufixo `.pretty()`:

```
> db.megasena.find({"Concurso":73}).pretty();
{
    "_id" : ObjectId("427539a3425c3245a124b211"),
    "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("427539a3425c3245a124b200"),
```

```

"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_Quina" : 512,
"Rateio_Quina" : "3790,28", "Ganhadores_Quadra" : 21452,
"Rateio_Quadra" : "90,2", "Acumulado" : "NÃO",
"Valor_Acumulado" : 0, "Estimativa_Prêmio" : 0,
"Acumulado_Mega_da_Virada" : 0 }
{ "_id": ObjectId("427539a3425c3245a124b500"),
"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_Quina" : 954, "Rateio_Quina" : "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` por padrão exibe todas as colunas. 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 em dois formatos: `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("427539a3425c3245a124b200"),
  "Concurso" : 233 }
{ "_id": ObjectId("427539a3425c3245a124b500"),
  "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_Sena"=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" : "19/06/2014",  
"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,
"19/06/2014",
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 double);
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("427539a3425c3245a124b550"),
```

```
"Concurso" : 99999,  
"CPF" : 12345678900  
}  
{  
    "_id" : ObjectId("427539a3425c3245a124b320"),  
    "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("427539a3425c3245a124b220"), "valor" : 3000 }
```

Note que no primeiro `insert` inserimos um número; no segundo, um texto; e no terceiro, nada, portanto, implicitamente o MongoDB cria 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("427539a3425c3245a124b320")});  
{ "_id" : ObjectId("427539a3425c3245a124b320"),  
  "Concurso" : 99999,  
  "CPF" : 12345678900,  
  "Nome" : "Coffin Joe" }  
>  
>db.ganhadores.update(  
  {"_id" : ObjectId("427539a3425c3245a124b320")},  
  {"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("427539a3425c3245a124b320")});  
{ "_id" : ObjectId("427539a3425c3245a124b320"),  
  "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("427539a3425c3245a124b320")},  
  {"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 = "427539a3425c3245a124b320";
```

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("427539a3425c3245a124b320")});  
  { "_id": ObjectId("427539a3425c3245a124b320"),  
    "Concurso" : 99999,  
    "CPF" : 12345678900,  
    "Nome" : "Coffin Joe" }
```

```
>
> db.ganhadores.update(
{ "_id" : ObjectId("427539a3425c3245a124b320")},
           { $set: {"Nome":"Zé do caixão"}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0,"nModified" : 1 })
>
> db.ganhadores.find(
{ "_id" : ObjectId("427539a3425c3245a124b320")});
{ "_id" : ObjectId("427539a3425c3245a124b320"),
  "Concurso" : 99999,
  "CPF" : 12345678900,
  "Nome" : "Zé do caixão" }
>
```

Portanto, em comparação, temos:

- MongoDB:

```
db.ganhadores.update(
{ "_id" : ObjectId("427539a3425c3245a124b320")},
           { $set: {"Nome":"Zé do caixão"}});
```

- MySQL:

```
update ganhadores
set nome = 'Zé do caixão'
where _id = "427539a3425c3245a124b320";
```

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, enquanto nas bases relacionais o padrão é atualizar todos os registros.

Neste primeiro exemplo, apenas uma linha é alterada:

```
> db.ganhadores.find();
{ "_id" : ObjectId("427539a3425c3245a124b550"),
  "Concurso" : 99999,
  "CPF" : 12345678900 }
{ "_id" : ObjectId("427539a3425c3245a124b320"),
  "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("427539a3425c3245a124b550"),
  "Concurso" : 99999,
  "CPF" : 555555555555 }
{ "_id" : ObjectId("427539a3425c3245a124b320"),
  "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("427539a3425c3245a124b550"),
  "Concurso" : 99999,
  "CPF" : 555555555555 }
{ "_id" : ObjectId("427539a3425c3245a124b320"),
  "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, cadastra? 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": 333333333333 }},  
{multi:0,upsert:0});  
WriteResult({ "nMatched" : 0, "nUpserted" : 0,"nModified" : 0 })  
>  
> db.ganhadores.find();  
{ "_id" : ObjectId("427539a3425c3245a124b550"),  
  "Concurso" : 99999,  
  "CPF" : 555555555555 }  
{ "_id" : ObjectId("427539a3425c3245a124b320"),  
  "Concurso" : 99999,  
  "CPF" : 555555555555,  
  "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": 333333333333 }},  
                      {multi:0,upsert:1});  
  
WriteResult({  
    "nMatched" : 0,  
    "nUpgraded" : 1,  
    "nModified" : 0,  
    "_id" : ObjectId("427539a3425c3245a124b222")  
})  
>  
> db.ganhadores.find();  
{ "_id" : ObjectId("427539a3425c3245a124b550"),  
  "Concurso" : 99999,  
  "CPF" : 555555555555 }  
{ "_id" : ObjectId("427539a3425c3245a124b320"),  
  "Concurso" : 99999,  
  "CPF" : 555555555555,  
  "Nome" : "Zé do caixão" }  
{ "_id" : ObjectId("427539a3425c3245a124b222"),  
  "Nome" : "Mula sem cabeça",  
  "CPF" : 333333333333 }  
>
```

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 = 33333333333
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 33333333333:

```
> db.ganhadores.count();
3
> db.ganhadores.find({"CPF" : 33333333333}).count();
1
> db.ganhadores.remove({ "CPF" : 33333333333 });
WriteResult({ "nRemoved" : 1 })
> db.ganhadores.count();
2
```

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

- MongoDB:

```
db.ganhadores.remove({ "CPF" : 33333333333 });
```

- MySQL:

```
delete from ganhadores
where CPF=33333333333;
```

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: 2.6.1
connecting to: test
> db.collectionNovaNaoPrecisaCriarAntes.insert(
{"uma coluna":"só"}); WriteResult({ "nInserted" : 1 })
> db.collectionNovaNaoPrecisaCriarAntes.count();
1
> db.collectionNovaNaoPrecisaCriarAntes.find();
{ "_id" : ObjectId("427539a3425c3245a124b212"),
  "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.

Por 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);
```

Por exemplo:

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

3.10 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>}})
```

Exemplo:

```
fb@cascao ~ > mongo  
MongoDB shell version: 2.6.1  
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 quantos ganhadores têm joe em seu nome, temos o resultado:

```
> db.ganhadores.find({"Nome":/joe/}).count();  
0  
> db.ganhadores.find({"Nome":/Joe/}).count();  
1
```

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'}})  
  
fb@cascao ~ > mongo  
MongoDB shell version: 2.6.1  
connecting to: test  
> db.ganhadores.find({"Nome":/joe/i}).count();  
1  
> db.ganhadores.find({  
  "Nome":{$regex:'joe', $options:'i'}}).count();  
1  
> exit  
bye
```

A lista contém outros operadores de busca, consulte na figura 3.5 e no site oficial do MongoDB.

- `$gt` maior que (*greater-than*)
- `$gte` igual ou maior que (*greater-than or equal to*)
- `$lt` menor que (*less-than*)
- `$lte` igual ou menor 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]}

Fig. 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.

3.11 CAPPED COLLECTION

O MongoDB possui um recurso bem interessante, as *capped collections* (collections “tampadas”), que 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("427539a3425c3245a124b333"),
  "nome" : "teste 3" }
{ "_id" : ObjectId("427539a3425c3245a124b369"),
  "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.

3.12 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`;
- 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 utilizado.

A normalização evita a redundância de dados armazenados, mas 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 e somente depois as collections são organizadas.

Em um modelo relacional, é comum separar tudo em tabelas, já no MongoDB separa-se por “entidades”, onde os todos dados necessários (ou quase) estão juntos.

Lembre-se 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, 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;
```

Analisando 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"]});
```

E 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 [16](#).

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 quase meio milhão de filmes cadastrados.

4.5 PROTÓTIPO

Nosso protótipo do sistema “Meus filmes” tem como objetivo exibir filmes e atores.

Na tela inicial 4.1 é possível fazer uma busca por somente filmes, somente atores ou ambos.



Fig. 4.1: Protótipo do Meus filmes página inicial

Saindo da tela inicial , podemos detalhar um ator, como é exibido na 4.2. 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

Fig. 4.2: Protótipo do Meus filmes detalhe de ator

Podemos detalhar um filme, como é exibido na 4.3. Além do nome do filme, são exibidas a quantidade de votos, a nota média e a lista de categorias de filme, diretores e atores.

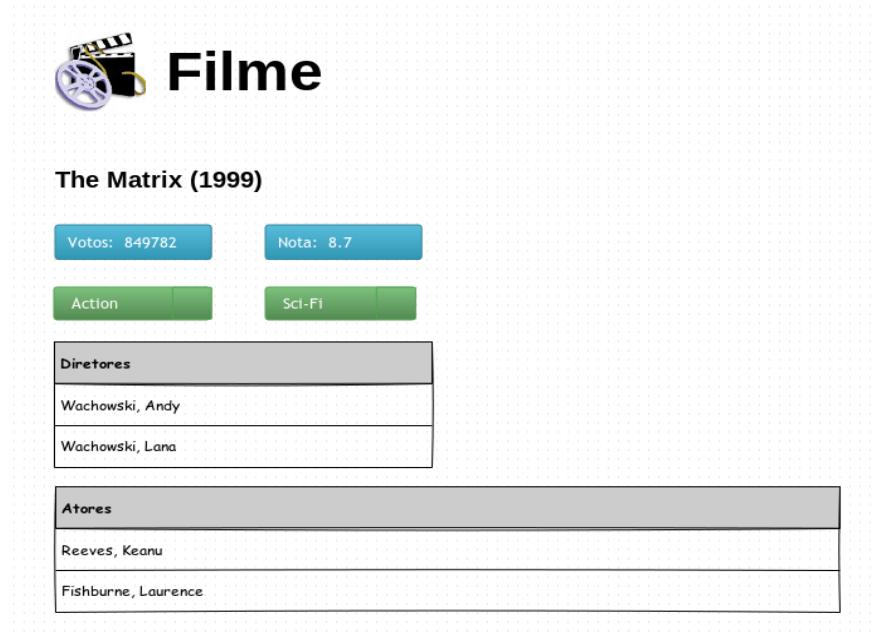


Fig. 4.3: Protótipo do Meus filmes detalhe de filme

4.6 SISTEMA MEUS FILMES RELACIONAL

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

- `actors` atores
- `movies` filmes
- `directors` diretores
- `genres` generos dos filmes
- `movies2actors` tabela associativa de filmes e atores
- `ratings` notas dos filmes
- `movies2directors` tabela associativa de filmes e diretores

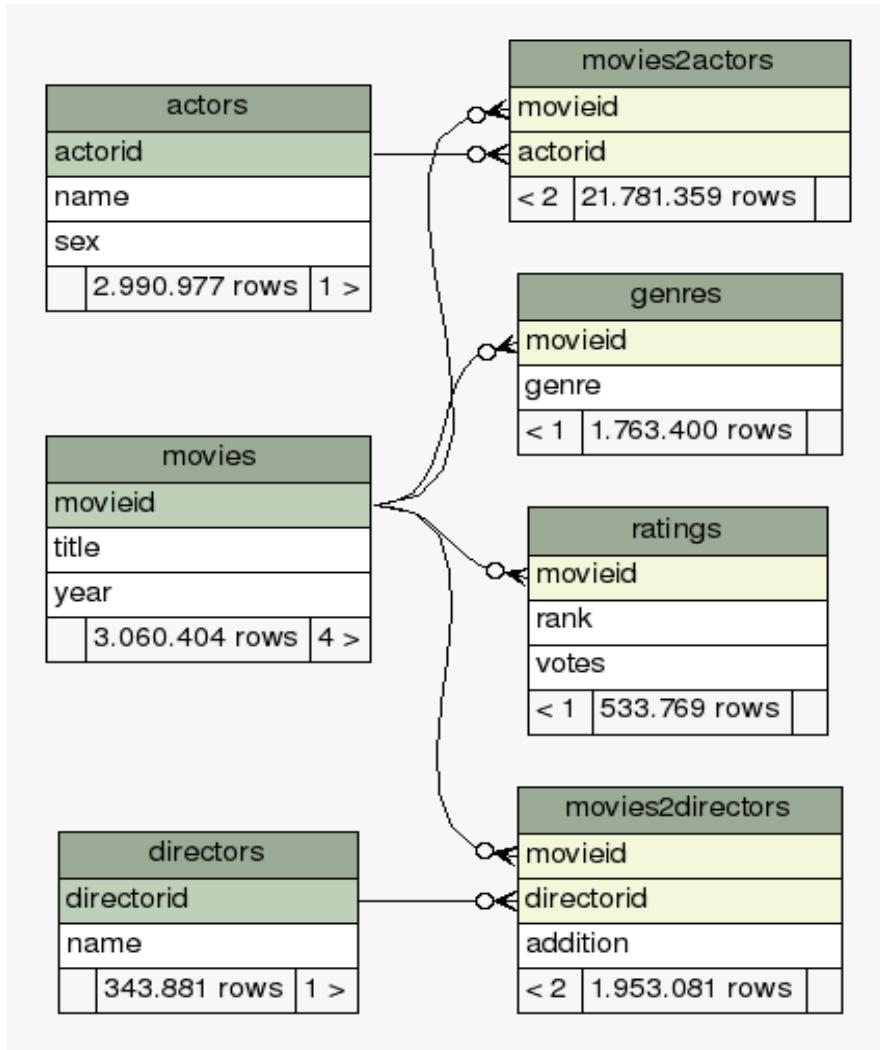


Fig. 4.4: Modelo relacional de meus filmes

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

4.7 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:12344000,  
    title: "The Matrix",  
    year: 1999,  
    rank: 8.7,  
    votes: 456400,  
    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"}  
    ]})
```

4.8 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- a avaliar a aplicação antes de montar o *schema*;
- a representar collections em relacionamento um para muitos;
- a 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 faremos 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. Através 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("427539a3425c3245a124b399"),  
    "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/mongo>

A instalação correta exibirá nas informações do PHP o driver do MongoDB instalado conforme a figura 5.1.

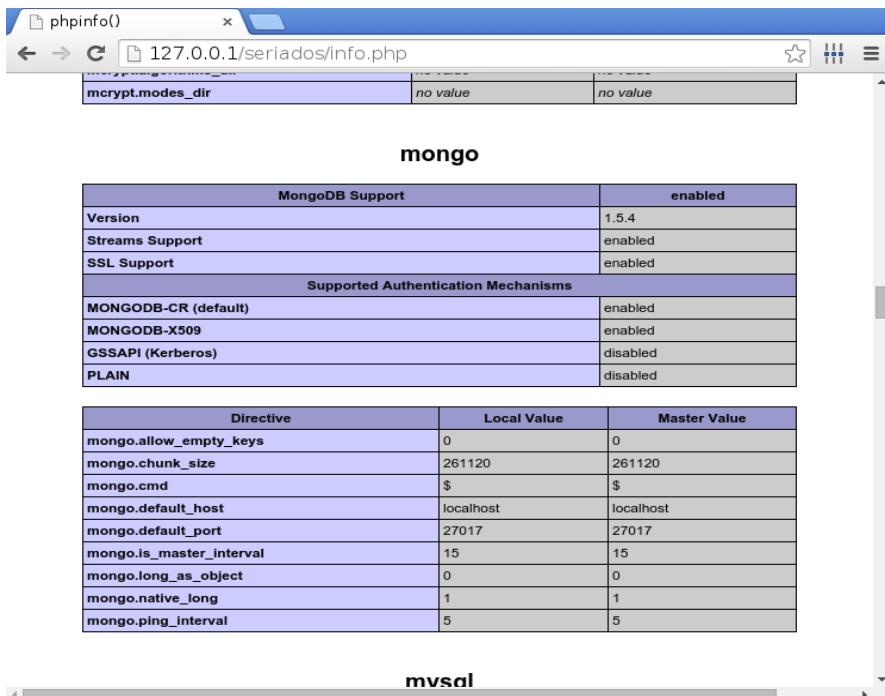


Fig. 5.1: phpInfo com MongoDB driver

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:

```
<?php
$conexao = new MongoClient();
```

Em seguida, escolhemos a collection desejada utilizando a sintaxe:

```
[variavel] -> [nome-do-banco-de-dados] -> [nome-da-collection]
```

```
$collection = $conexao->test->seriados;
```

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

```
$cursor = $collection->find();  
?>
```

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 5.2.



Meus seriados preferidos

Chaves	Carga Pesada	Breaking Bad
novo seriado		

Fig. 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` e inicialmente abrimos a conexão no MongoDB escolhendo a collection desejada:

```
$conexao = new MongoClient();
$collection = $conexao->test->seriados;
```

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 `nome` e a lista de personagens:

```
$documento = array("nome" => $_POST['nome'],  
                   "personagens" => $personagens);
```

Finalmente o documento é inserido através do método `insert`:

```
$collection->insert($documento);
```

O resultado obtido é exibido na figura [5.3](#).



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

Fig. 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 através da variável `$_GET`.

A página de detalhe do seriado inicia abrindo uma conexão no MongoDB

da mesma maneira que as páginas anteriores:

```
$conexao = new MongoClient();
$collection = $conexao->test->seriados;
```

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'];
$cursor = $collection->find( array('_id' => new MongoId($id)));
```

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

```
$documento= $cursor->getNext();
```

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']);
$documento = array( "nome" => $_POST['nome'],
    "personagens" => $personagens);
```

Usamos o método `update` para alterar o documento, enviando como parâmetros: o identificador único, a variável `documento`, e a opção `upsert` ativa, que fará um cadastro se não encontrar o registro que está sendo alterado:

```
$collection->update(array('_id' =>
    new MongoId($id)), $documento, array('upsert'=>true));
```

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

```
$collection->remove(array('_id' => new MongoId($id)));
```

O resultado obtido da tela de alteração é exibido na figura 5.4.



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

Fig. 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>.

Pré-requisitos:

- Servidor Apache Tomcat <http://tomcat.apache.org/>
- Java <http://java.sun.com/>
- 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 `DBCursor`, que retorna uma linha/ documento (`DBObject`) através de um objeto `DBObject`.

```
public List<Seriado> findAll() {  
  
    List<Seriado> seriados = new ArrayList<Seriado>();  
    DBCursor cursor = seriadosCollection.find();  
  
    while (cursor.hasNext()) {  
  
        DBObject resultElement = cursor.next();  
    }  
}
```

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

```
Map<?,?> resultElementMap = resultElement.toMap();  
Seriado seriado = new Seriado();  
seriado.setId((ObjectId) resultElementMap.get("_id"));  
seriado.setNome((String) resultElementMap.get("nome"));  
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` através da 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="<%=\n        dao.findAll()\n    %>">  
    <a class="alert alert-success"  
        href="detalhe.jsp?id=<c:out value="${seriado.id}" />">  
        <c:out value="${seriado.nome}" />  
    </a>  
</c:forEach>
```

```
</a>
</c:forEach>
```

Um novo seriado

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

Essa aplicação 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* usado 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 sobreescrivar 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 `seriado` 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, onde 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/>), quem tem como objetivo oferecer comandos no Java semelhantes aos do Mongo shell.

Se uma busca no Mongo shell for:

```
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 bem 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>.

A aplicação em Play tem quase tudo do 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 o figura [5.5](#).



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](#)

Fig. 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 5.6.

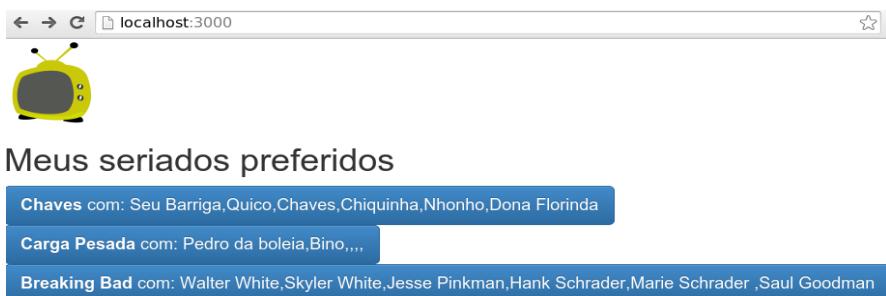


Fig. 5.6: MongoDB com Node.js

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 utiliza 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)
```

```
    }  
}
```

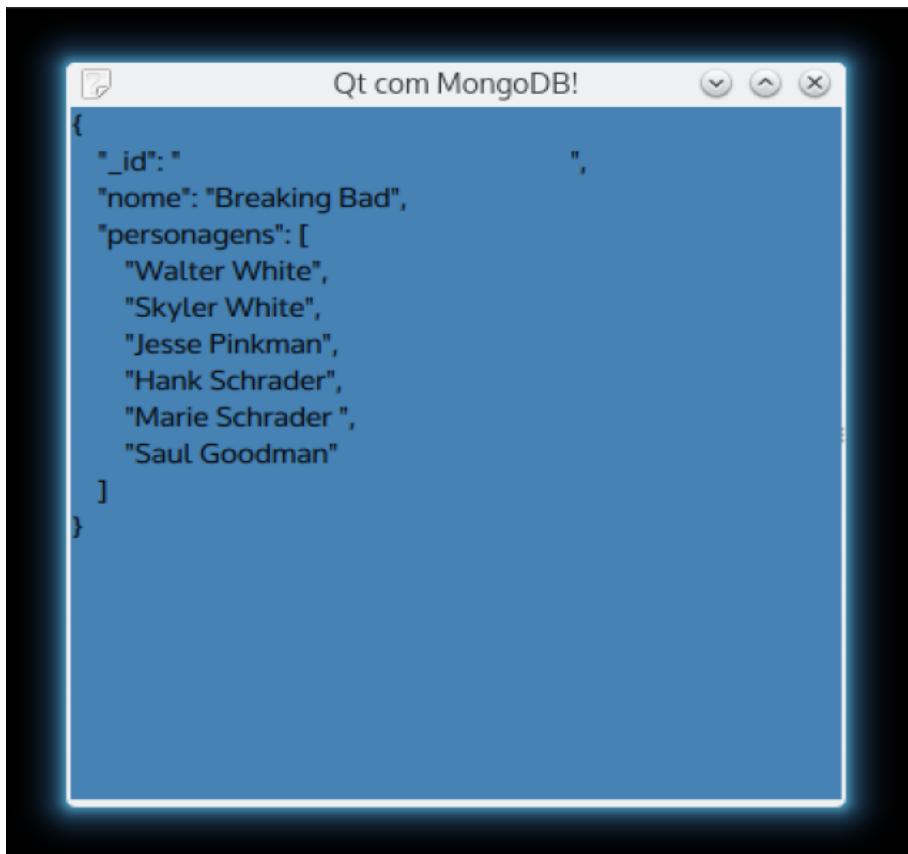


Fig. 5.7: MongoDB com Qt

O resultado é a figura 5.7.

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.

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 e seus atores e diretores.

O interessante é que eles disponibilizam os dados gratuitamente para baixar (<ftp://ftp.fu-berlin.de/pub/misc/movies/database/>) , e 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, o que significa que no MongoDB elas devem ficar em uma única uma collection.

Sua estrutura deverá ficar como nesse exemplo:

```
{
  "_id": NumberLong(12345678),
  "titulo": "TRON (1982)",
  "ano": "1982",
  "nota": 6.8,
  "votos": NumberLong(12210),
  "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, basicamente ela 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 depois remove a collection de filmes (se ela já existir):

```
try {
    mongoClient = new MongoClient(
        new MongoClientURI("mongodb://localhost")
    );
} catch (UnknownHostException e) {
    e.printStackTrace();
}
blogDatabase = mongoClient.getDB("test");
filmesCollection = blogDatabase.getCollection("filmes");
filmesCollection.drop();
```

Em seguida, 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`:

```
BasicDBObject document = new BasicDBObject();
document.put("_id", movieid);
document.put("titulo", title);
document.put("ano", year);
document.put("nota", rank);
document.put("votos", votes);
document.put("categorias",
            mysqlDAO.getCategorias(Long.valueOf(movieid)));
document.put("diretores",
            mysqlDAO.getDiretores(Long.valueOf(movieid))');
```

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

```
List<BasicDBObject> atoresMongoDB =
    new ArrayList<BasicDBObject>();
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) {
    BasicDBObject atorMongoDB = new BasicDBObject();
    atorMongoDB.put("nome", ator.getNome());
    atorMongoDB.put("sexo", ator.getSexo());
    atoresMongoDB.add(atorMongoDB);
}
```

A lista de documentos de atores é inserida ao documento:

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

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

```
filmesCollection.insert(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 (<http://www.mongodb.com/partners/cloud>) , mas vamos nos focar 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 6.1.

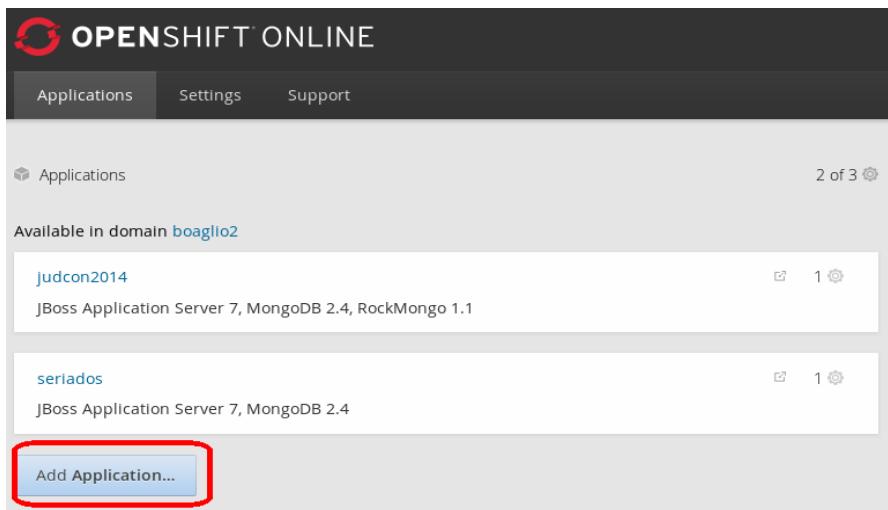


Fig. 6.1: OpenShift lista de aplicações

Diversas opções serão oferecidas, mas escolheremos o JBoss Application Server 7 Cartridge, conforme a figura 6.2.

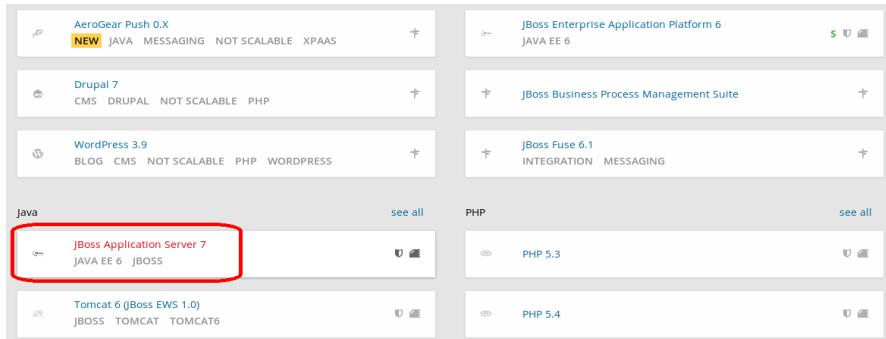


Fig. 6.2: OpenShift lista de componentes

Depois de escolher o servidor, é oferecida a opção de escolher a URL pública, onde informamos o valor de `seriados`. Veja a figura 6.3 .

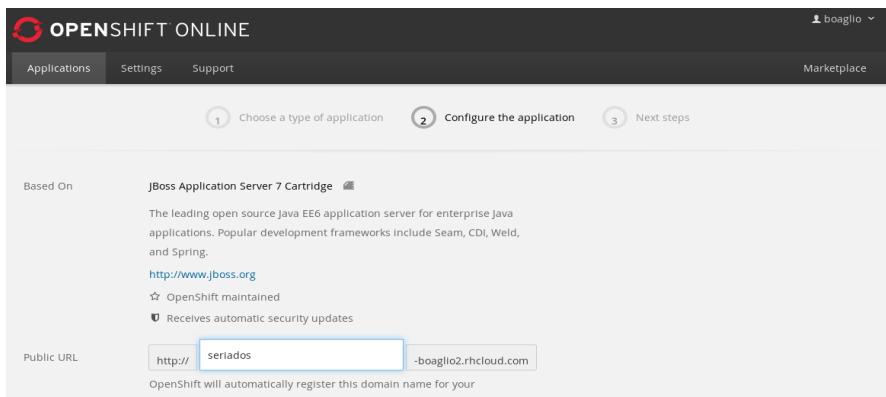


Fig. 6.3: OpenShift registrando aplicação

Uma vez criada a aplicação de `seriados`, adicionaremos o componente do MongoDB (figura 6.4).

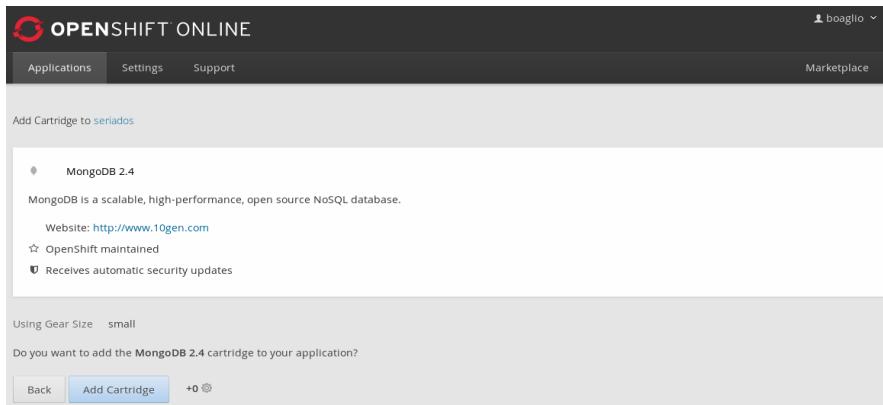


Fig. 6.4: OpenShift adicionando MongoDB

Agora é informada a senha de acesso (figura 6.5).

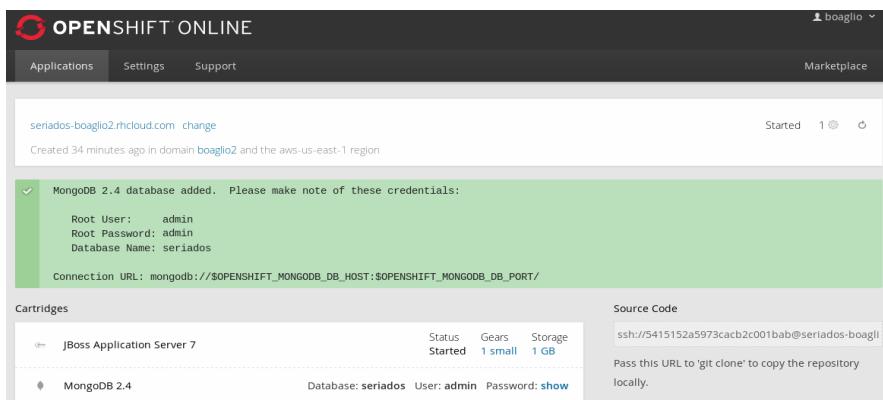


Fig. 6.5: OpenShift MongoDB adicionado

Para administrar o MongoDB remotamente, adicionamos o componente RockMongo (figura 6.6).

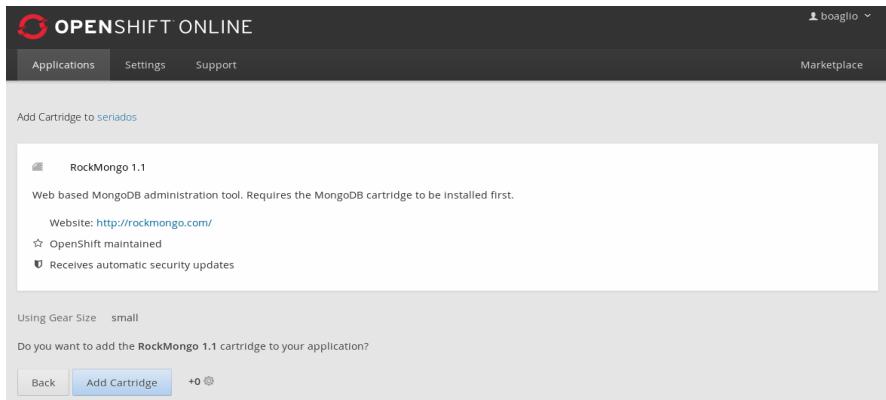


Fig. 6.6: OpenShift adicionando RockMongo

Depois de adicionado, é informada a senha de acesso (figura 6.7).

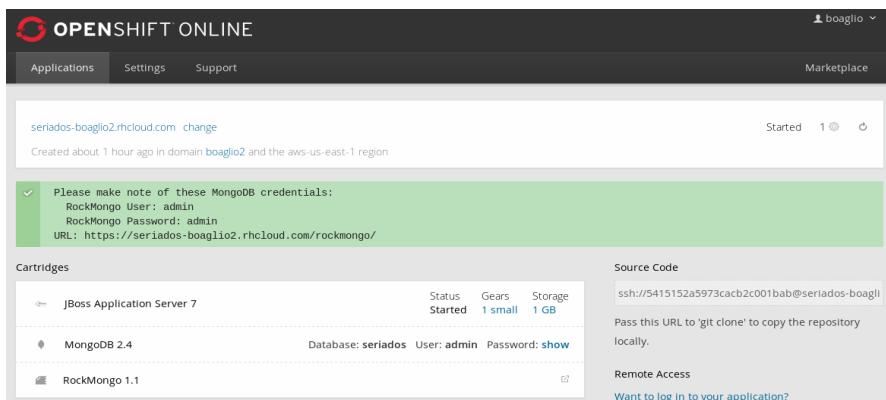


Fig. 6.7: OpenShift RockMongo adicionado

Conseguimos acessar o RockMongo com a senha fornecida (figura 6.8).

The screenshot shows the RockMongo login interface. It has two main sections: 'Log-in:' and 'Non-Admin users:'. In the 'Log-in:' section, the 'Host' field is set to 'Localhost' with a dropdown arrow. The 'UserName:' field contains 'admin' and the 'Password:' field contains a masked password. Below these, under 'Non-Admin users:', the 'DB Name(s)' field contains 'seriados'. At the bottom left is a link 'More »' and a large blue button labeled 'Login and Rock'. Below the button, there is a message: 'Powered by RockMongo v1.1.7, check out new version here. [Donate 10 dollars]'.

Fig. 6.8: OpenShift entrar no RockMongo

Ao entrar no RockMongo, visualizamos todos as collections (figura 6.9).

The screenshot shows the OpenShift MongoDB interface. On the left, there's a sidebar with 'Servidor' and 'Overview' buttons, and a tree view showing the 'seriados' database with collections: keyword, openshift (1), system.indexes (3), system.users (1), and a 'Create' button. The main panel is titled 'Bancos » seriados' and 'Estatísticas'. It displays 'Database Statistics' with the following data:

Database Statistics ({dbStats:1})	
Size	0
Is Empty?	No
Collections	3 collections: [Drop All] [Clear All]
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
Data File Version	{ "major": NumberInt(4), "minor": NumberInt(5) }

Fig. 6.9: OpenShift visão geral

Através dele, podemos importar uma base local com a opção `import` (figura 6.10).

The screenshot shows the 'Importar' (Import) page for the 'seriados' database. The left sidebar is identical to Fig. 6.9. The right panel has a title 'Bancos » seriados » Importar'. It contains sections for '.js file exported with RockMongo:' and '.json file exported with mongoexport:'. Under '.js', there's a 'JS File:' field with 'Selecionar arquivo...' and a note 'Nenhum arquivo selecionado.' Below it is an 'Importar' button. Under '.json', there's an 'Import to collection name:' field with 'seriados', a 'JSON File:' field with 'Selecionar arquivo...', and a note 'seriados.json'. Below it is another 'Importar' button.

Fig. 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 6.11).

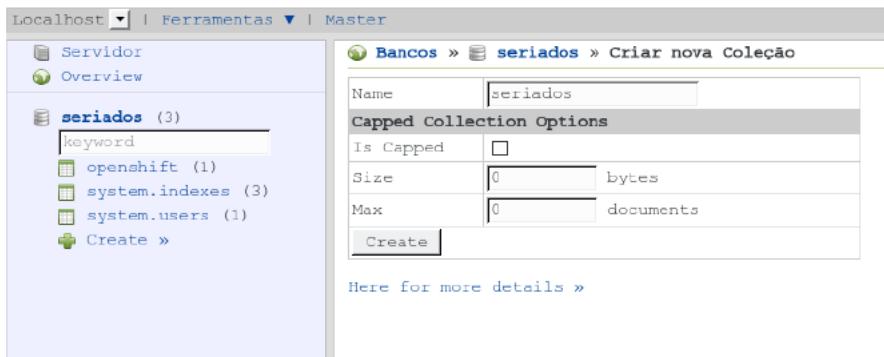


Fig. 6.11: OpenShift criar collection

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

Para tal, basta clonarmos o repositório remoto com git:

```
git clone
```

```
ssh://12345@seriados-boaglio2.rhcloud.com/~/git/seriados.git/
cd seriados/
```

Por exemplo:

```
> git clone
ssh://12345@seriados-boaglio2.rhcloud.com/~/git/seriados.git/Cloning into
'seriados'...
remote: Counting objects: 34, done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 34 (delta 2), reused 34 (delta 2)
Receiving objects: 100% (34/34), 29.27 KiB | 0 bytes/s, done. Resolving deltas:
100% (2/2), done.
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 12c2311] 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 47843432
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
```

```
342387468..96a1231 master -> master
```

Acessamos a aplicação no endereço <http://seriados-boaglio2.rhcloud.com/>, obtendo um resultado semelhante à figura 6.12.

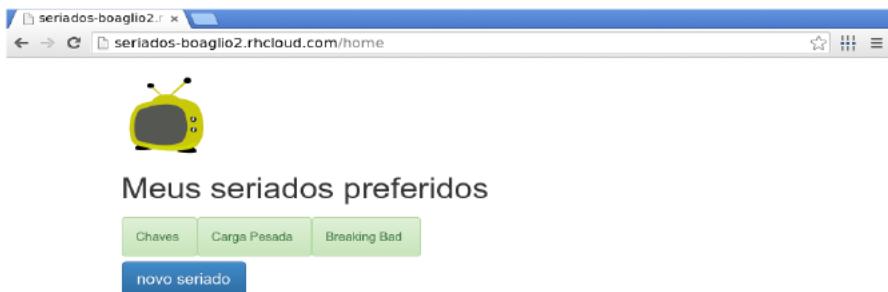


Fig. 6.12: OpenShift aplicação de seriados

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;
- a 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 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();
```

7.1 OPERADORES DE COMPARAÇÃO

No nosso exemplo, temos o operador `greater than` (ou maior 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:

- `$gt`: maior que
- `$gte`: maior ou igual que
- `$lt`: menor que
- `$lte`: menor ou igual 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()
3
```

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":{$ne:7}}).count()
1607
> db.megasena.find({"Ganhadores_Sena":{$in:[7]}}).count()
1
> db.megasena.find().count()
1608
```

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

```
> db.megasena.find().count()
1608
> db.megasena.find({"Ganhadores_Sena":{$nin:[5,6,7]}}).count()
1605
> db.megasena.find({"Ganhadores_Sena":{$in:[5,6,7]}}).count()
3
```

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>)
```

Exemplo da quantidade de ganhadores da Mega-Sena:

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

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`, através da sintaxe:

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

Exemplo:

```
db.megasena.find({ "Acumulado": { $regex: /^\\d.*$/ } })
{
    "_id": ObjectId("427539a3425c3245a124b555"),
    "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();  
1609
```

```
> db.megasena.find({"obs":{$exists:true}}).count();
1
> db.megasena.find({"obs":{$exists:false}}).count();
1608
```

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 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);
```

Exemplo:

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

7.7 INCREMENTANDO VALORES

É comum utilizarmos 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> }})
```

Exemplo de dar um aumento de R\$500 a um funcionário:

```
db.funcionarios.update(
  {"_id" : ObjectId("427539a3425c3245a124b712")},
  { $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):

```
<derruba-serviço-no-MongoDB>
cd <diretorio-do-projeto>/dump/
mongorestore --drop
    --dbpath <diretorio-de-dados-do-mongodb>
    --db test
    dump/test/
```

Exemplo:

```
service mongodb stop
cd /home/fb/workspace/projeto/dump/
mongorestore --drop
    --dbpath /var/lib/mongodb
    --db test
    dump/test/
```

O nosso banco de dados possui 25704 cidades cadastradas, cada um deles com o documento `loc` com coordenadas de latitude (Y) e longitude (X).

Exemplo de busca pela cidade de Orlando no estado da Flórida:

```
db.zipcodes.find({ "city" : "ORLANDO", "state" : "FL"})
{
    "_id" : ObjectId("427539a3425c3245a124c222"),
    "city" : "ORLANDO",
    "state" : "FL",
    "loc" : {
        "x" : 81.408162,
        "y" : 28.487102
    }
}
```

Buscando a cidade de Miami, também na Flórida:

```
db.zipcodes.find({ "city" : "MIAMI", "state" : "FL"})
{
    "_id" : ObjectId("427539a3425c3245a124c729"),
    "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](#), 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:

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

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

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

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 8.1.

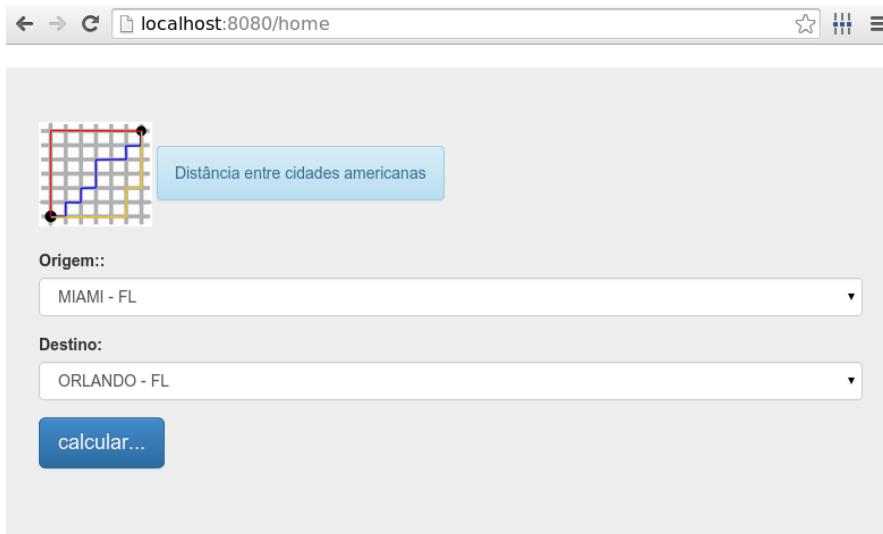


Fig. 8.1: Calculando a distância entre Miami e Orlando

Em seguida, exibe o mapa das duas cidades e a distância entre elas e suas cidades vizinhas, como mostra a figura 8.2.

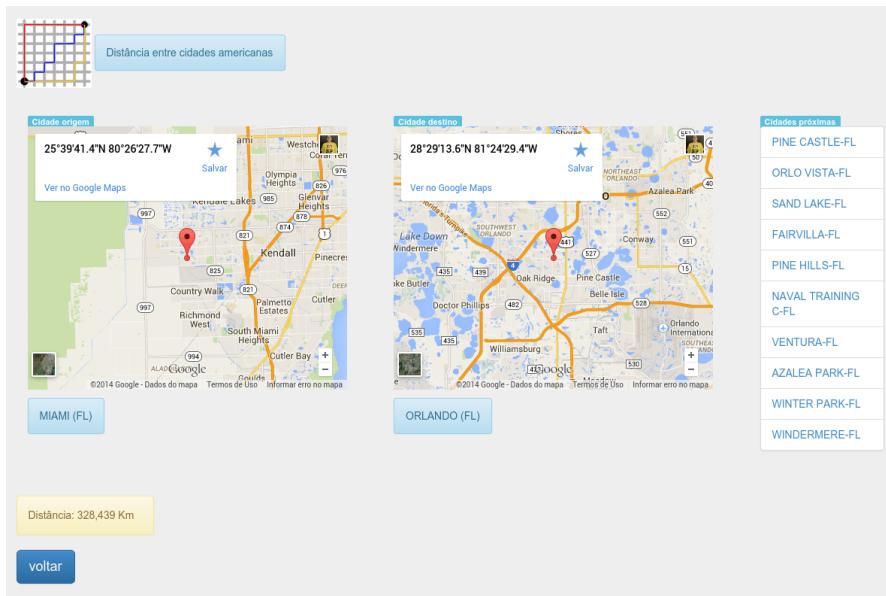


Fig. 8.2: Exibindo o resultado da distância entre Miami e Orlando

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`, e 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, onde 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);
```

8.4 INDO ALÉM

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

8.5 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- a criar índice geoespacial em campo com coordenadas;
- a 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.

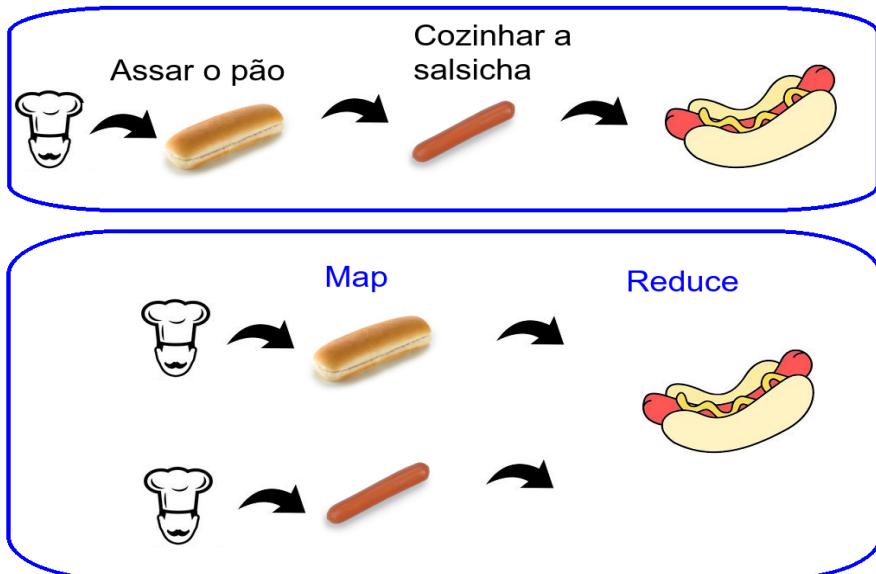


Fig. 9.1: Fazendo hot dog com e sem map reduce

Na figura 9.1, vemos duas maneiras de fazer um hot dog: 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 algumas 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, mas se o objetivo for cinco mil unidades, com certeza a segunda opção é a mais eficiente.

Vamos para um exemplo na nossa collection dos sorteios.

Inicialmente definimos a função de mapeamento da quantidade de ganhadores agrupado pelos *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"})
```

Fazendo uma consulta na collection criada, percebemos que em todos os sorteios em que acumulou (SIM), ninguém ganhou.

Já nos sorteios em que não acumulou (NÃO), 533 pessoas ganharam a Mega-Sena.

```
> db.ganhadores.find();  
{ "_id" : "NÃO", "value" : 533 }  
{ "_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" : 533 }
```

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" : 1607,  
"soma" : 533, "avg" : 0.3316739265712508 }
```

Pelo resultado, percebemos que além dos 533 ganhadores temos ao todo 1607 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( { $group :  
  { _id : <nome-do-campo> ,  
    <nome-do-campo>:{  
      <função-de-grupo>:"$<nome-do-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" : 533 }  
{ "_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 que zero:

```
db.megasena.aggregate({ $group: {
    _id: "$Acumulado",
    soma: { $sum: "$Ganhadores_Sena" }
} },
{ $match : { soma : { $gt : 0 }}}),
{ "_id" : "NÃO", "soma" : 533 }
```

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":5, "acumulado":"NÃO" }, "soma" : 10 }
{"_id":{ "ganhadores_sena":4, "acumulado":"NÃO" }, "soma" : 32 }
{"_id":{ "ganhadores_sena":3, "acumulado":"NÃO" }, "soma" : 54 }
{"_id":{ "ganhadores_sena":2, "acumulado":"NÃO" }, "soma" : 136 }
{"_id":{ "ganhadores_sena":1, "acumulado":"NÃO" }, "soma" : 279 }
{"_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" : 533 }
```

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 [5](#).

```
> db.filmes.stats();
{
  ...
  "count" : 511010,
  "nindexes" : 1,
  "indexSizes" : {
    "_id_" : 2142354
  },
}
```

```
...  
}
```

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.

Novamente vamos destacar apenas o que interessa:

```
> db.filmes.find({"ano" :"1999"}).explain();  
{  
  ...  
  "cursor" : "BasicCursor",  
  "n" : 8097,  
  "nscanned" : 511010,  
  "millis" : 198,  
  ...  
}
```

Isso significa que, pela informação do campo `cursor`, nenhum índice foi usado na busca. O campo `n` representa a quantidade de documentos retornados da busca, ou seja, a collection possui 8097 filmes do ano de 1999.

O campo `nscanned` 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 `millis`. Neste caso, foi 198 milissegundos.

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();  
{  
  ...  
  "cursor" : "BtreeCursor ano_1",  
  "n" : 8097,  
  "nscanned" : 8097,  
  "millis" : 18,  
  ...  
}
```

Veja um exemplo de criação de um índice composto pelos campos `ano` com ordem crescente e `nota` com ordem decrescente:

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

Inicialmente verificamos pelo cursor que foi usado o nosso índice do campo `ano`, e a quantidade de registros retornados continua 8097.

Entretanto, o campo `nscanned` retornou 8097 em vez de 511010, e pelo campo `millis` vemos que o tempo caiu de 198 para 18 milissegundos.

10.2 LISTAR OS ÍNDICES CRIADOS

Para listar os índices criados, usamos a sintaxe:

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

Por exemplo:

```
> db.filmes.getIndexes();
[
{
  "v" : 1,
  "key" : { "_id" : 1 },
  "name" : "_id_",
  "ns" : "test.filmes"
},
{
  "v" : 1,
  "key" : { "ano" : 1 },
  "name" : "ano_1",
  "ns" : "test.filmes"
},
]
```

10.3 REMOVER UM ÍNDICE CRIADO

Para remover um índice existente, usamos a sintaxe:

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

Vamos resolver nosso índice criado:

```
> db.filmes.dropIndex("ano_1");
{ "nIndexesWas" : 2, "ok" : 1 }
> db.filmes.getIndexes();
[
{
  "v" : 1,
  "key" : { "_id" : 1 },
  "name" : "_id_",
}
```

```
"ns" : "test.filmes"
}
]
```

O índice do campo `_id` não pode ser removido. 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("427539a3425c3245a124b729"),
  "texto" : "Eu gosto de São Paulo"
}
> db.textos.find( { $text: { $search: "gostar" } } );
{ "_id": ObjectId("427539a3425c3245a124b729"),
  "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("427539a3425c3245a124b729"),
  "texto" : "Eu gosto de São Paulo"
}
{ "_id": ObjectId("427539a3425c3245a124b730"),
  "texto" : "Eu gosto de São Paulo e Rio Claro" }
```

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("427539a3425c3245a124b729"),
  "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"} );
```

Na nossa collection de textos temos:

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

```
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
}
```

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.

10.6 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- a fazer o *explain* de uma consulta ;
- a analisar o resultado de um *explain*;
- a criar e remover índices simples e compostos;
- a criar índices de busca textual;
- a 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 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.

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.

11.2 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.

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. Entretanto, depois de executado, ele não libera espaço em disco.

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

Exemplo:

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

11.3 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/mongodb.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 no 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" }
    ]
}
)

```

11.4 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 11.1 temos o exemplo da base `test` em uso.

locked db	idx	miss %	qr qw	ar aw	netIn	netOut	conn	time
todo@0.0%	0	0 0	0 0	62b	5k	3	16:30:17	
..@0.8%	0	0 0	0 0	567b	5k	3	16:30:18	
todo@0.0%	0	0 0	0 0	62b	5k	3	16:30:19	
..@0.7%	0	0 0	0 0	49k	14k	3	16:30:20	
..@0.8%	0	0 0	0 0	101k	19k	3	16:30:21	
..@0.8%	0	0 0	0 0	79k	19k	3	16:30:22	
..@1.2%	0	0 0	0 0	90k	21k	3	16:30:23	
..@0.8%	0	0 0	0 0	94k	19k	3	16:30:24	
test@0.3%	0	0 0	0 0	77k	15k	3	16:30:25	
todo@0.0%	0	0 0	0 0	62b	5k	3	16:30:26	
locked db	idx	miss %	qr qw	ar aw	netIn	netOut	conn	time
todo@0.0%	0	0 0	0 0	62b	5k	3	16:30:27	

Fig. 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 11.2 temos o exemplo da collection `filmes` da base `test` em uso.

ns	total	read	write
<code>test.filmes</code>	4ms	0ms	4ms
<code>local.startup_log</code>	0ms	0ms	0ms
<code>local.system.indexes</code>	0ms	0ms	0ms
<code>local.system.namespaces</code>	0ms	0ms	0ms
<code>local.system.replset</code>	0ms	0ms	0ms
<code>m101.funnynumbers</code>	0ms	0ms	0ms
<code>m101.hw1</code>	0ms	0ms	0ms
<code>m101.profile</code>	0ms	0ms	0ms

Fig. 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.

11.5 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 <diretório-de-backup>
cd <diretório-de-backup>
mongodump --dbpath <diretório-de-dados-do-mongodb>
<sobe-serviço-do-MongoDB>
```

Exemplo em Linux:

```
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 em Linux:

```
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
```

11.6 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 <diretorio-de-backup>
mongorestore --dbpath <diretorio-de-dados-do-mongodb> dump
```

Exemplo no Linux:

```
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 no Linux 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
```

11.7 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"  
     ...  
   }]  
}
```

Se necessário, é possível derrubar um processo desses com o comando `db.killOp`:

```
mongo> db.killOp(123)  
{ "info" : "attempting to kill op" }
```

11.8 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.

CAPÍTULO 12

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

Ter os seus dados replicados em diferentes lugares (chamados **nós**) e se um algum nó cair, outro assumir no lugar, isso é o que chamamos de alta disponibilidade; a sua aplicação não deixa de funcionar.

Essa arquitetura é chamada de *replica set* (ou conjunto de servidores replicados), onde podemos ter entre 2 e 12 servidores (mas o mínimo sugerido é 3).

Na figura 12.1, 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).

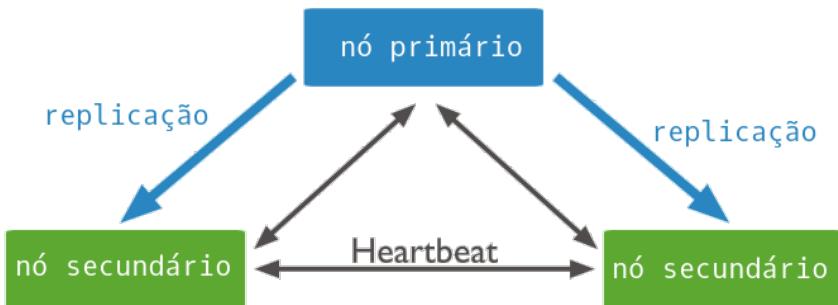


Fig. 12.1: MongoDB *replica set* com três nós

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 `rso`:

```
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:

```
fb@cascao ~ > mongo --port 27017  
MongoDB shell version: 2.6.1  
connecting to: 127.0.0.1:27017/test
```

```
> rs.initiate()  
{  
    "info2" : "no configuration explicitly specified  
              -- making one",  
    "me" : "cascao:27017",  
    "info" : "Config now saved locally.  
             Should come online in about a minute.",  
    "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.

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 12.2 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.

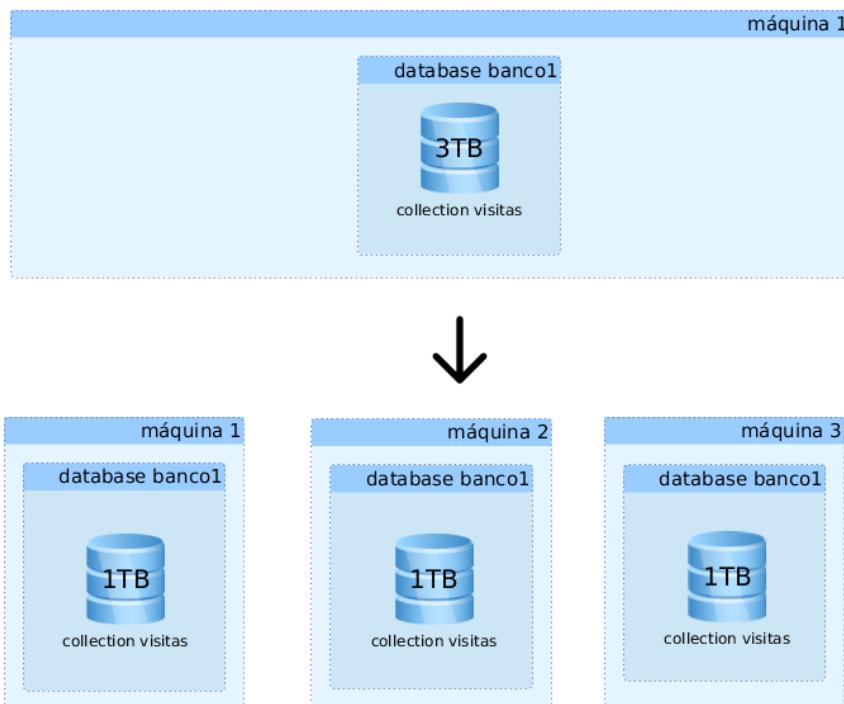


Fig. 12.2: Sharding

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 irá 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 12.3. 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`.

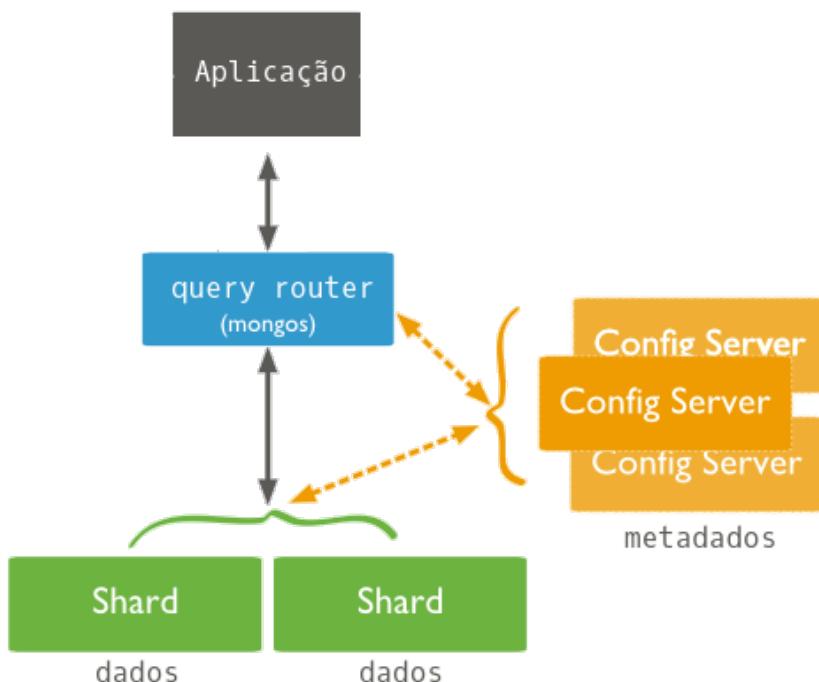


Fig. 12.3: MongoDB com particionamento (sharding)

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

O primeiro passo é criar um config server na porta 27020, com essas configurações:

```
root@cascao /etc > cat config_db.conf
fork=true
dbpath=/var/lib/mongodb/dbs/config_db
logpath=/var/log/mongodb/config_db.log
logappend=true
port=27020
```

Em seguida, criamos o diretório do banco de dados e subimos o serviço apontando para o arquivo de configuração config_db.conf:

```
root@cascao > mkdir /var/lib/mongodb/dbs/config_db
root@cascao > mongod --configsvr --config /etc/config_db.conf
```

```
about to fork child process,  
waiting until server is ready for connections.
```

```
forked process: 8080
```

```
    child process started successfully, parent exiting
```

Exemplo de query routing instance

O segundo passo é criar uma query routing instance, que será acessada pela aplicação e ficará também na porta 27020, com essas configurações:

```
root@cascao /etc > cat mongos.conf  
fork = true  
port = 27017  
configdb = cascao:27020  
logpath=/var/log/mongodb/mongos.log
```

Desta vez, iniciamos o serviço com o executável mongos:

```
root@cascao > mongos --config /etc/mongos.conf  
2014-10-21T01:31:11.222-0200 warning: running with 1 config  
server should be done only for testing purposes and is not  
recommended for production  
about to fork child process, waiting until server is ready  
for connections.  
forked process: 12345  
child process started successfully, parent exiting
```

Com isso, até o momento temos dois processos do MongoDB rodando:

```
root@cascao /etc [ 1:35:55]> ps -ef | grep mongo  
root      8080  mongod --configsvr --config /etc/config_db.conf  
root     12345  mongos --config /etc/mongos.conf
```

Exemplo de sharding instances

Finalmente, vamos criar os shards, começando com o primeiro no dbpath padrão e na porta 30001:

```
root@cascao > mongod --fork --port 30001
                  --dbpath /var/lib/mongodb
                  --logpath /var/log/mongodb/shard1.log
about to fork child process, waiting until server is ready
for connections.
forked process: 18330
child process started successfully, parent exiting
```

Criaremos agora o segundo em um novo dbpath e na porta 30002:

```
root@cascao > mkdir /var/lib/mongodb2
root@cascao > mongod --fork --port 30002
                  --dbpath /var/lib/mongodb2
                  --logpath /var/log/mongodb/shard2.log
about to fork child process, waiting until server is ready
for connections.
forked process: 18445
```

Temos quatro processos do MongoDB no ar:

```
root@cascao > ps -ef | grep mongo
root      8080      mongod --configsvr --config ...
root     12345      mongos --config /etc/mongos.conf
root     18330      mongod --fork --port 30001 ...
root     18445      mongod --fork --port 30002 ...
```

Depois de criados os shardings, precisamos adicioná-los ao *cluster*.

Conectamos ao query routing instance e adicionamos com o comando sh.addShard:

```
root@cascao > mongo --port 27017
MongoDB shell version: 2.6.5
connecting to: test
mongos>
mongos> sh.addShard( "cascao:30001");
{ "shardAdded" : "shard0000", "ok" : 1 }
mongos> sh.addShard( "cascao:30002");
{ "shardAdded" : "shard0001", "ok" : 1 }
```

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(12313112),
  "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" : {
    "cascao:30001" : {
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
    }
  },
  "ok" : 1
}
```

Em seguida, vamos distribuir (particionar) esses dados:

```
mongos> sh.shardCollection("test.filmes", {"ano": "hashed"});  
{ "collectionsharded" : "test.filmes", "ok" : 1 }  
mongos> db.filmes.count();  
425568
```

Finalmente, vamos identificar como os dados estão distribuídos:

```
mongos> db.filmes.getShardDistribution();  
  
Shard shard0000 at cascao:30001  
data : 224.63MiB docs : 271032 chunks : 7  
estimated data per chunk : 32.09MiB  
estimated docs per chunk : 38718  
  
Shard shard0001 at cascao:30002  
data : 201.98MiB docs : 239978 chunks : 7  
estimated data per chunk : 28.85MiB  
estimated docs per chunk : 34282  
  
Totals  
data : 426.62MiB docs : 511010 chunks : 14  
Shard shard0000 contains 52.65% data, 53.03%  
docs in cluster, avg obj size on shard : 869B  
Shard shard0001 contains 47.34% data, 46.96%  
docs in cluster, avg obj size on shard : 882B
```

Observe que a distribuição foi uniforme: ficou aproximadamente metade (224.63 e 201.98) em cada shard.

Em um ambiente de produção, ao contrário desse exemplo, em cada um dos shards teríamos um *replica set* para garantir a alta disponibilidade dos dados.

12.4 PRÓXIMOS PASSOS

Certifique-se de que aprendeu:

- a criar *clusters* com *replica set*;

- a partitionar dados com *sharding*;
- a 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>

E acompanhe os principais blogs:

- blog oficial <http://blog.mongodb.org/>

- blog que centraliza notícias relacionadas ao MongoDB <http://planet.mongodb.org/>
- blog sobre MongoDB e outras bases NoSQL <https://blog.compose.io>

CAPÍTULO 14

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 e, 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 nem instala como serviço, 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 (antiga 10gen):

```
root@perola:~# apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80
--recv 777CEC20
Executing: gpg --ignore-time-conflict --no-options
--no-default-keyring --secret-keyring /tmp/tmp.werwes2234 --trustdb-
name /etc/apt/trustdb.gpg
--keyring /etc/apt/trusted.gpg
--primary-keyring /etc/apt/trusted.gpg
--keyserver hkp://keyserver.ubuntu.com:80 --recv 777CEC20
gpg: requisitando chave 777CEC20 de servidor hkp -
      keyserver.ubuntu.com
gpg: chave 777CEC20: chave pública "Richard Kreuter
      <richard@10gen.com>" importada
gpg: ultimamente não encontradas chaves confiáveis
gpg: Número total processado: 1
gpg:                      importados: 1  (RSA: 1)
```

Em seguida, adicionamos o repositório do MongoDB aos existentes:

```
root@perola:~# echo 'deb
http://downloads-distro.mongodb.org/repo/
ubuntu-upstart dist 10gen' |
tee /etc/apt/sources.list.d/mongodb.list
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen
```

Atualizamos o nosso repositório local:

```
root@perola:~# apt-get update
Atingido http://br.archive.ubuntu.com precise Release.gpg
Obter:1 http://br.archive.ubuntu.com precise-updates
      Release.gpg [198 B]
Atingido http://br.archive.ubuntu.com precise Release
Obter:2 http://br.archive.ubuntu.com precise-updates
```

Release [49,6 kB]

...

E finalmente instalamos:

```
root@perola:~# apt-get install mongodb-org
Lendo listas de pacotes... Pronto
Construindo árvore de dependências
Lendo informação de estado... Pronto
Os pacotes extra a seguir serão instalados:
  mongodb-org-mongos mongodb-org-server mongodb-org-shell
  mongodb-org-tools
```

Instalação em Windows

Vamos baixar a versão para Windows pelo site <http://www.mongodb.org/downloads> e executar:

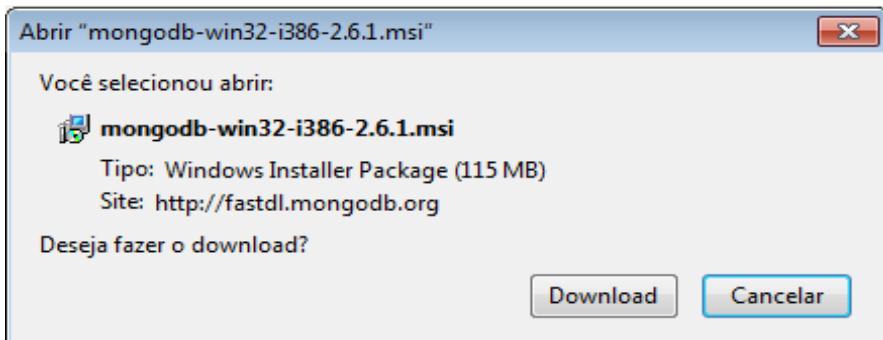


Fig. 14.1: Instalação de MongoDB para Windows 32 bits

Iniciamos a instalação clicando em **Next**:



Fig. 14.2: Início da instalação

Aceitamos os termos de uso também:

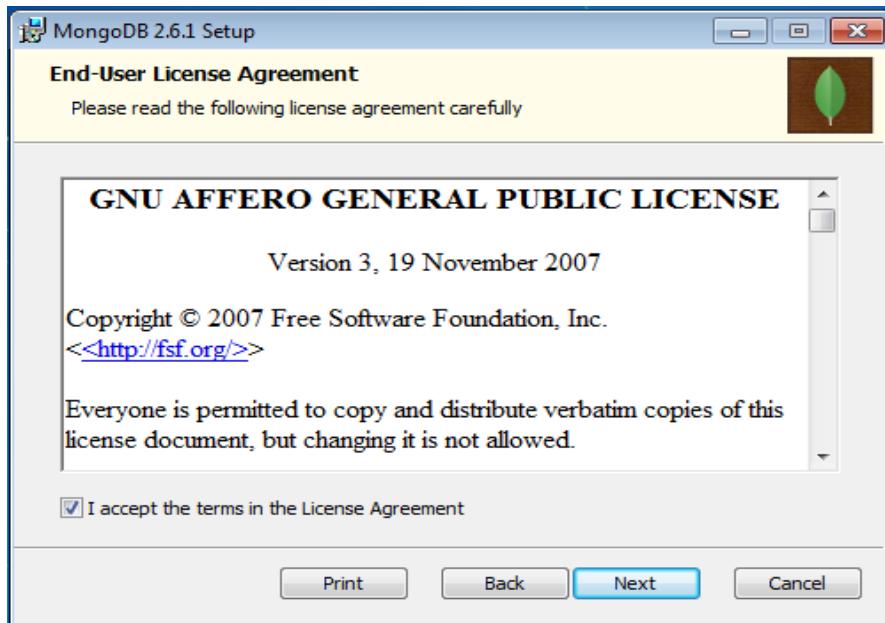


Fig. 14.3: Termos de uso

Selecionamos a opção de instalação completa:

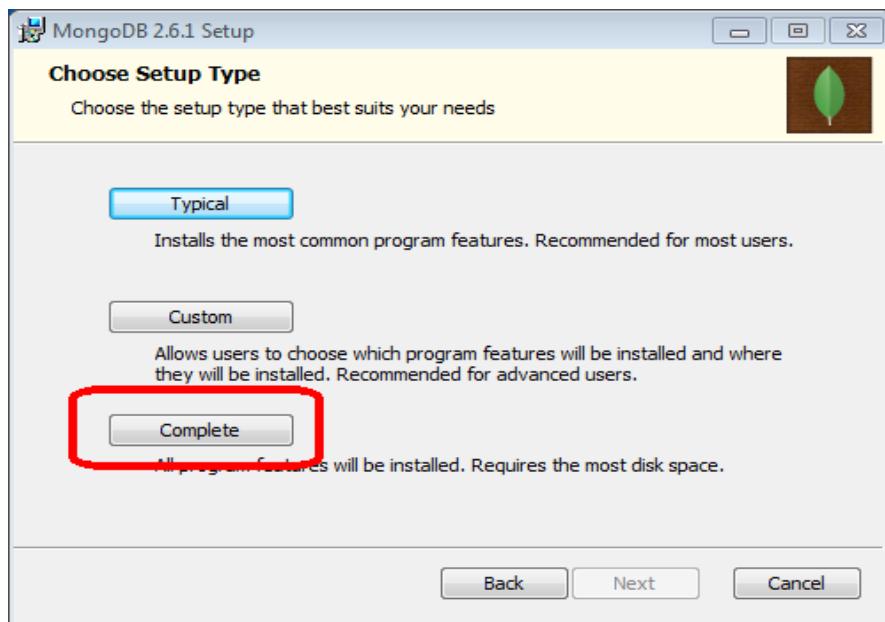


Fig. 14.4: Escolhendo o tipo de instalação

Finalmente, clicamos em `Install` para instalar:

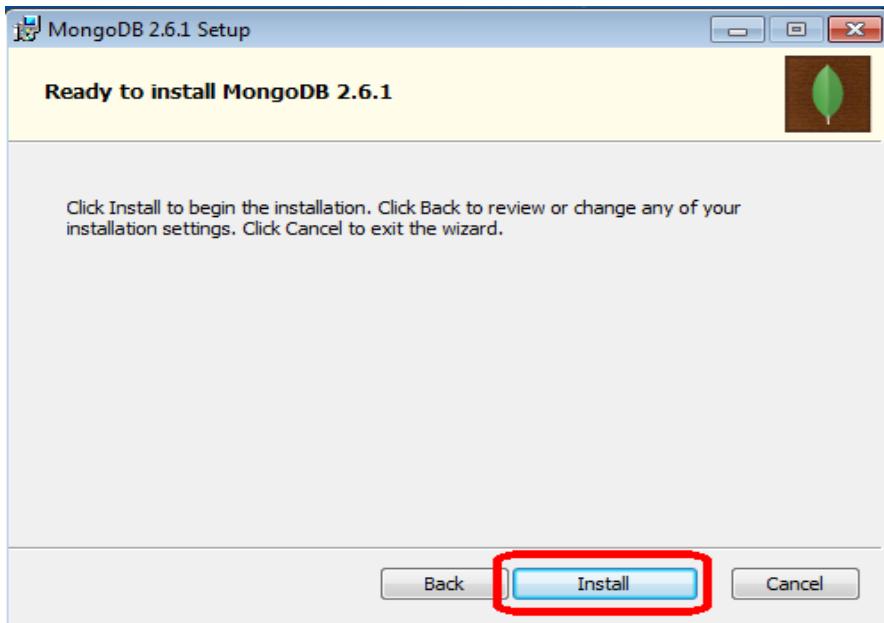


Fig. 14.5: Copiando os arquivos com o instalador

A instalação será concluída em poucos instantes:

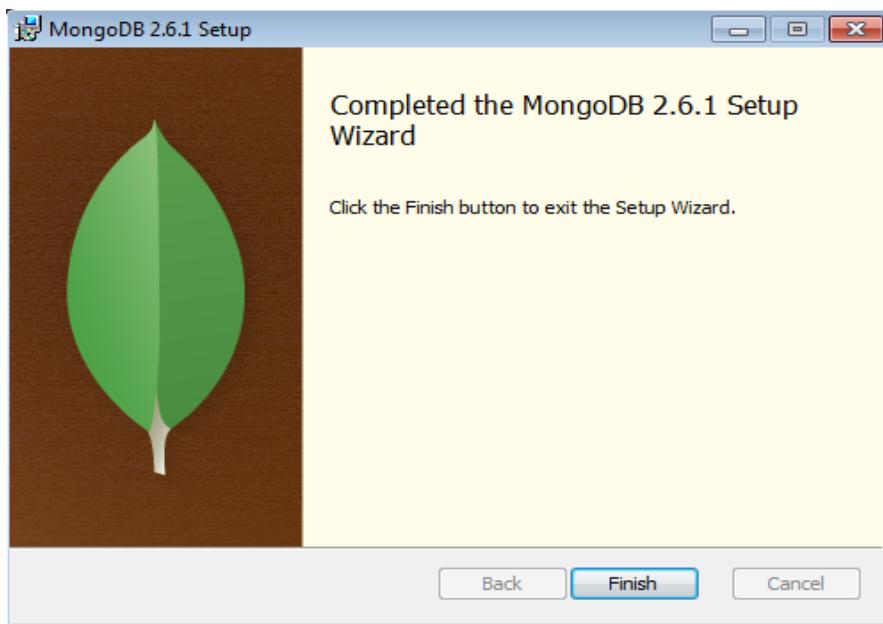


Fig. 14.6: Finalizando a instalação

Vamos adicionar os executáveis do MongoDB ao PATH do Windows. Para isso, acessamos o painel de controle e modificamos algumas variáveis de ambiente:

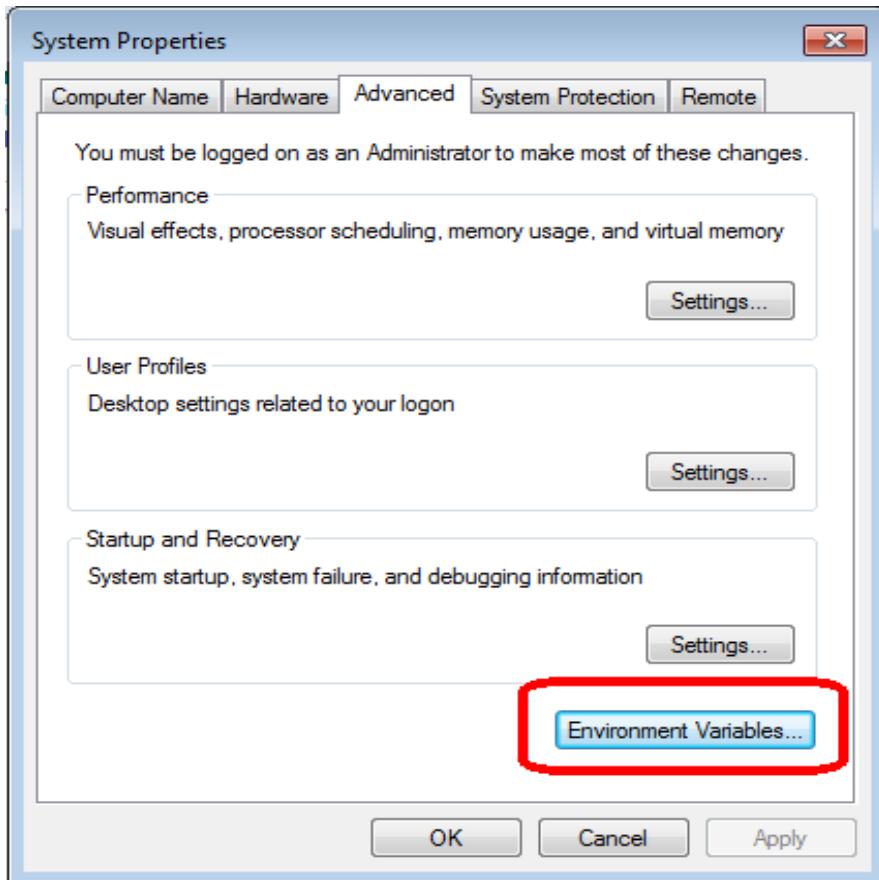


Fig. 14.7: Alterando variáveis de ambiente

Clicando na opção New, criamos uma nova variável MONGODB_HOME que contém o valor de C:\Program Files\MongoDB 2.6 Standard\.

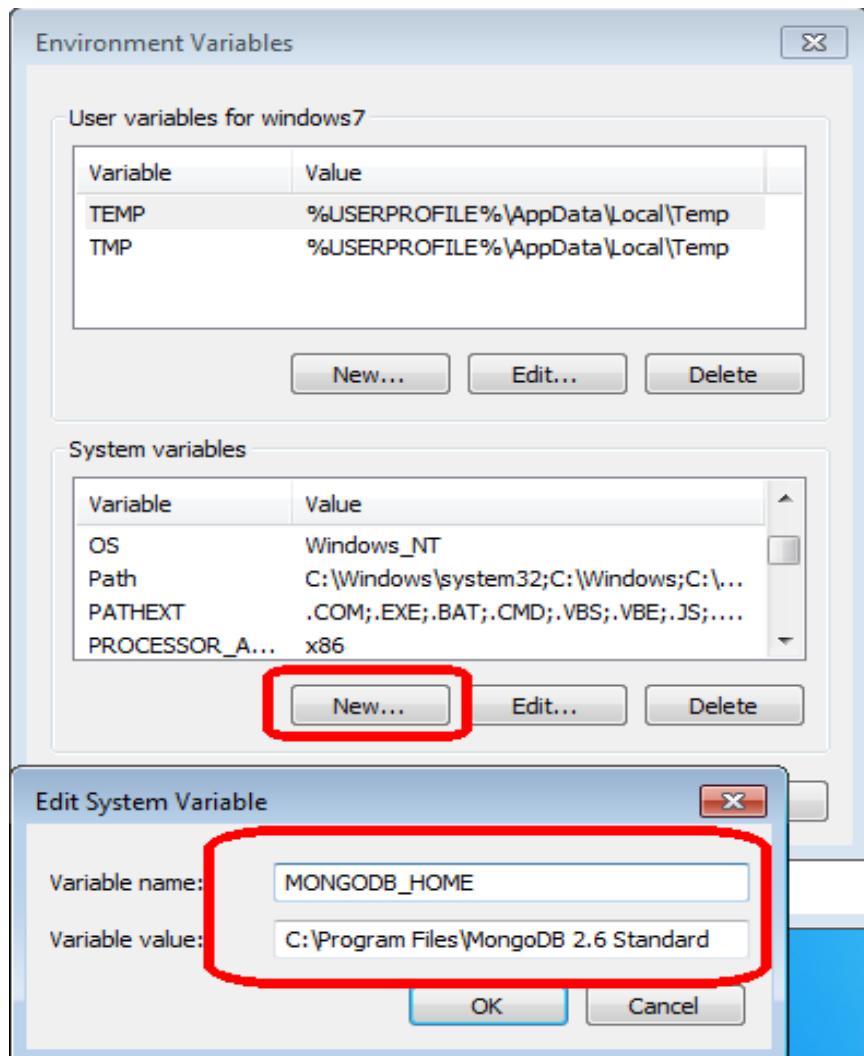


Fig. 14.8: Alterando variáveis de ambiente

Em seguida, editamos a variável de ambiente PATH e adicionamos ao final %MONGODB_HOME\BIN:

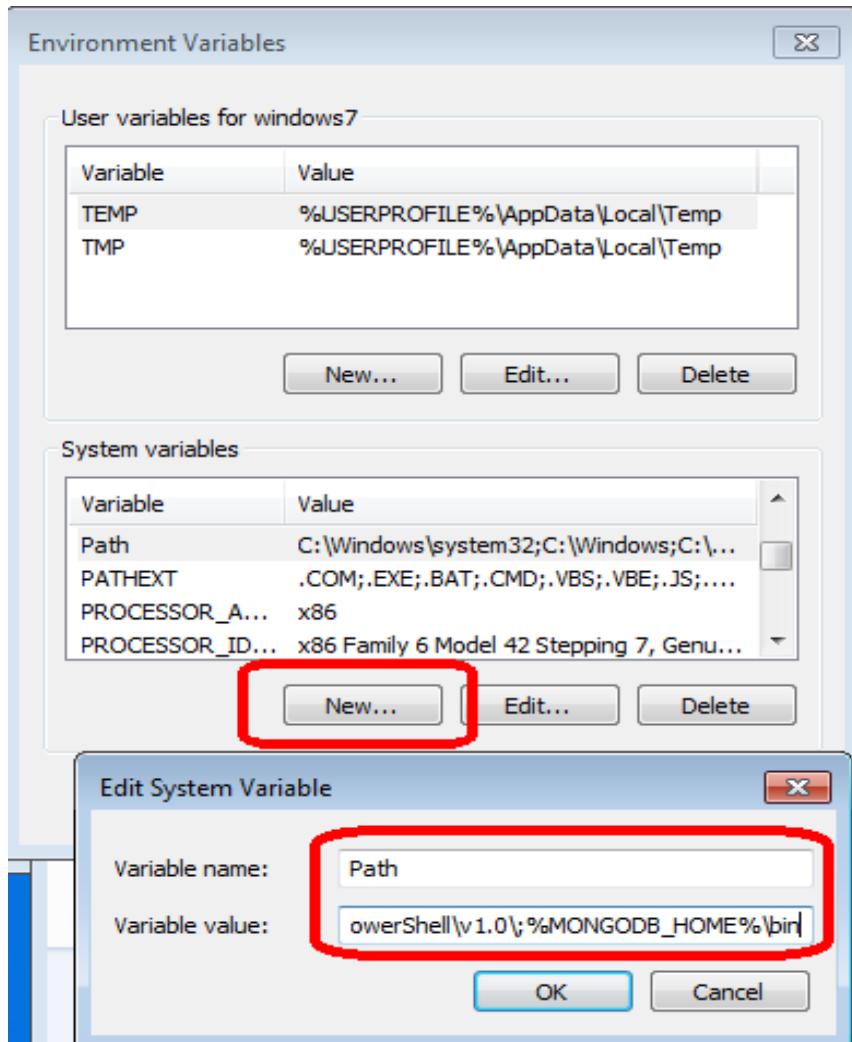


Fig. 14.9: Alterando variáveis de ambiente

Chamamos o prompt de comando para instalar o serviço. Iniciamos criando os diretórios dos dados com:

```
mkdir c:\data\db  
mkdir c:\data\log
```

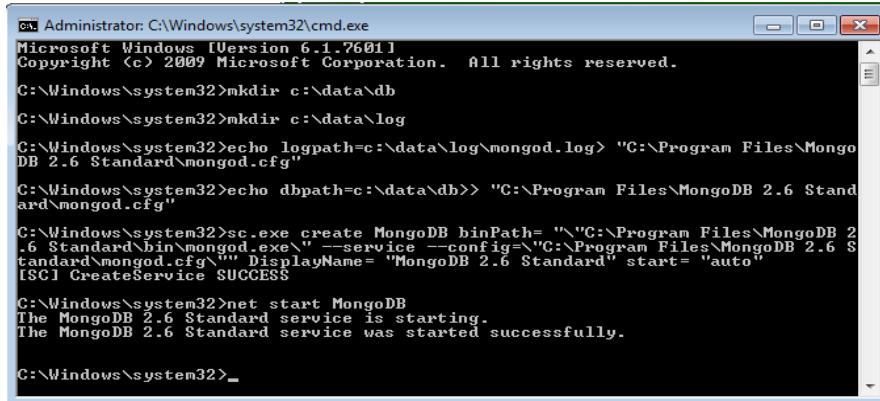
Criamos o arquivo de configuração:

```
echo logpath=c:\data\log\mongod.log>
"C:\Program Files\MongoDB 2.6
Standard\mongod.cfg"
echo dbpath=c:\data\db>> "C:\Program Files\MongoDB 2.6
Standard\mongod.cfg"
```

E finalmente criamos o serviço:

```
sc.exe create MongoDB binPath= "\"C:\Program Files\MongoDB 2.6
Standard\bin\mongod.exe\" --service
--config=\"C:\Program Files\MongoDB
2.6 Standard\mongod.cfg\"" DisplayName= "MongoDB 2.6 Standard"
start= "auto"
```

O resultado será semelhante a esse:



The screenshot shows a Windows Command Prompt window titled 'Administrator: C:\Windows\system32\cmd.exe'. The window displays the following command-line session:

```
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>mkdir c:\data\db
C:\Windows\system32>mkdir c:\data\log
C:\Windows\system32>echo logpath=c:\data\log\mongod.log > "C:\Program Files\MongoDB 2.6 Standard\mongod.cfg"
C:\Windows\system32>echo dbpath=c:\data\db>> "C:\Program Files\MongoDB 2.6 Standard\mongod.cfg"
C:\Windows\system32>sc.exe create MongoDB binPath= "\"C:\Program Files\MongoDB 2.6 Standard\bin\mongod.exe\" --service --config=\"C:\Program Files\MongoDB 2.6 Standard\mongod.cfg\"" DisplayName= "MongoDB 2.6 Standard" start= "auto"
[SC] CreateService SUCCESS

C:\Windows\system32>net start MongoDB
The MongoDB 2.6 Standard service is starting.
The MongoDB 2.6 Standard service was started successfully.

C:\Windows\system32>_
```

Fig. 14.10: Instalando o MongoDB como serviço do Windows

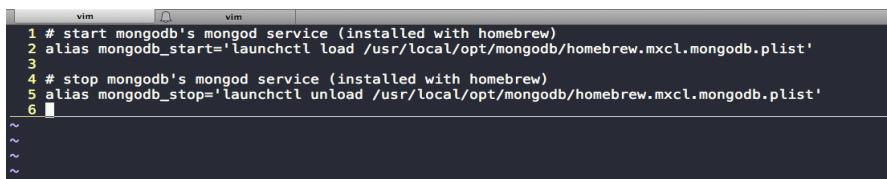
Instalação em Mac OS X

A instalação é feita através do Homebrew:

```
valholl:~ leandropincini$ brew install mongodb
==> Downloading https://downloads.sf.net/project/machomebrew/Bottles/mongodb-2.6.1.mavericks.bottle.tar.gz
######################################################################## 100.0%
=> Pouring mongodb-2.6.1.mavericks.bottle.tar.gz
=> Caveats
To have launchd start mongodb at login:
  ln -s /usr/local/opt/mongodb/*.plist ~/Library/LaunchAgents
Then to load mongodb now:
  launchctl load ~/Library/LaunchAgents/homebrew.mxcl.mongodb.plist
Or, if you don't want/need launchctl, you can just run:
  mongod --config /usr/local/etc/mongod.conf
=> Summary
  /usr/local/Cellar/mongodb/2.6.1: 17 files, 317M
valholl:~ leandropincini$
```

Fig. 14.11: Instalando o MongoDB pelo Homebrew no Mac OS X

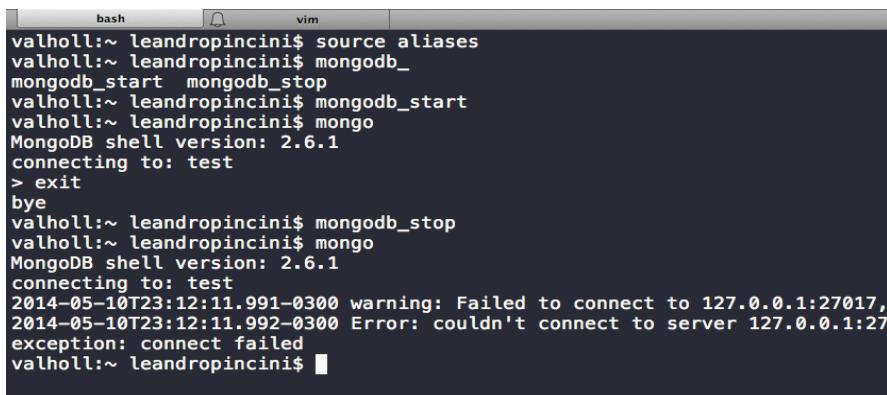
Criamos um *alias* para o MongoDB instalar como um serviço:



```
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
```

Fig. 14.12: Instalando o MongoDB como serviço

Em seguida, usamos `mongodb_start` / `mongodb_stop`:



```
bash vim vim
valholl:~ leandropincini$ source aliases
valholl:~ leandropincini$ mongodb_
mongodb_start mongodb_stop
valholl:~ leandropincini$ mongodb_start
valholl:~ leandropincini$ mongo
MongoDB shell version: 2.6.1
connecting to: test
> exit
bye
valholl:~ leandropincini$ mongodb_stop
valholl:~ leandropincini$ mongo
MongoDB shell version: 2.6.1
connecting to: test
2014-05-10T23:12:11.991-0300 warning: Failed to connect to 127.0.0.1:27017,
2014-05-10T23:12:11.992-0300 Error: couldn't connect to server 127.0.0.1:27
exception: connect failed
valholl:~ leandropincini$
```

Fig. 14.13: Manipulando serviços do MongoDB no Mac OS X

Testando a instalação

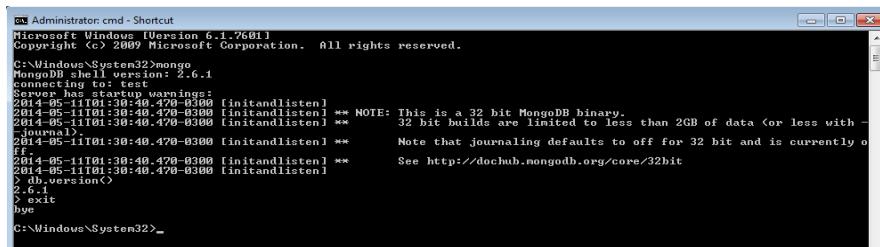
Com o MongoDB no ar, conectamos e digitamos o comando para informar a versão instalada:

```
db.version()
```

O resultado será semelhante a esse em Linux e Mac OS X:

```
fb@cascao ~ > mongo
MongoDB shell version: 2.6.1
connecting to: test
> db.version()
2.6.1
> exit
bye
fb@cascao ~ >
```

E a esse em Windows:



A screenshot of a Windows Command Prompt window titled "Administrator: cmd - Shortcut". The window shows the output of a MongoDB session. The text is as follows:

```
Administrator: cmd - Shortcut
Microsoft Windows [Version 6.1.7601]
Copyright © 2009 Microsoft Corporation. All rights reserved.

C:\Windows\System32>mongo
MongoDB shell version: 2.6.1
connecting to: test
Server has startup warnings:
2014-05-11T01:30:40.470-0300 [initandlisten] ** Note: This is a 32 bit MongoDB binary.
2014-05-11T01:30:40.470-0300 [initandlisten] ** 32 bit builds are limited to less than 2GB of data (or less with --journal).
2014-05-11T01:30:40.470-0300 [initandlisten] ** Note that journaling defaults to off for 32 bit and is currently off.
2014-05-11T01:30:40.470-0300 [initandlisten] ** See http://dochub.mongodb.org/core/32bit
> db.version()
2.6.1
> exit
bye
C:\Windows\System32>
```

Fig. 14.14: Exibindo a versão do MongoDB no Windows 7

CAPÍTULO 15

Apêndice B. Robomongo

Existem diversos clientes desenvolvidos para MongoDB, dentre os quais se sobressai o RoboMongo, uma excelente ferramenta open source que possui a mesma engine em JavaScript que o 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.

A instalação do Robomongo é bem simples e não precisa de nenhuma opção especial, por esse motivo só ilustraremos a instalação em Windows.

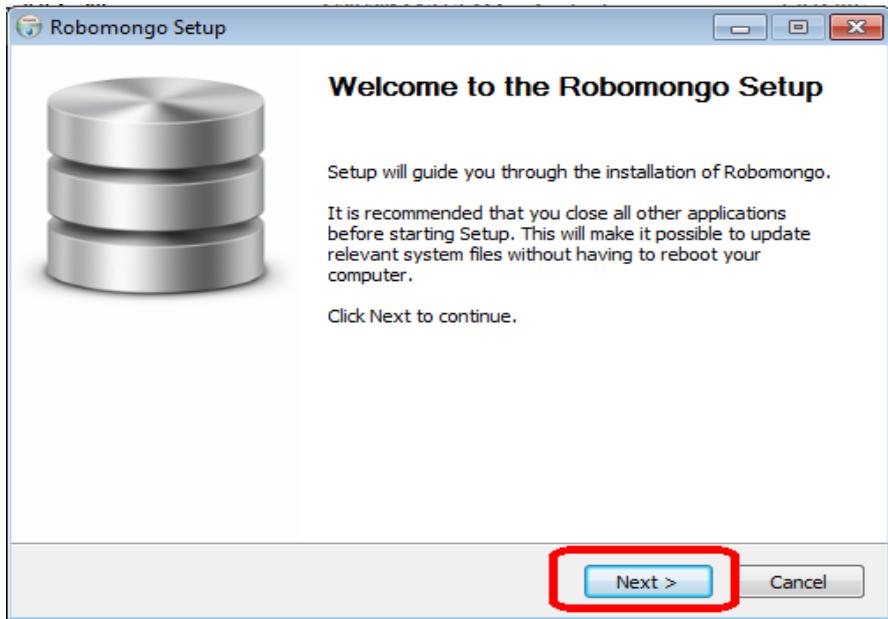


Fig. 15.1: Início da instalação

São exibidos os termos de uso da licença open source (GPL) para concordar:

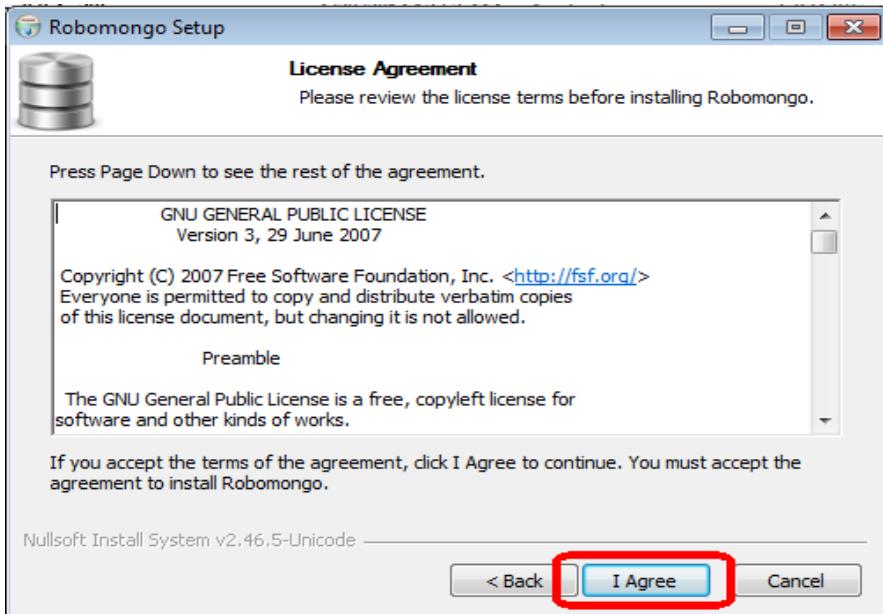


Fig. 15.2: Termos de uso do Robomongo

Em seguida, é exibido o diretório de instalação:

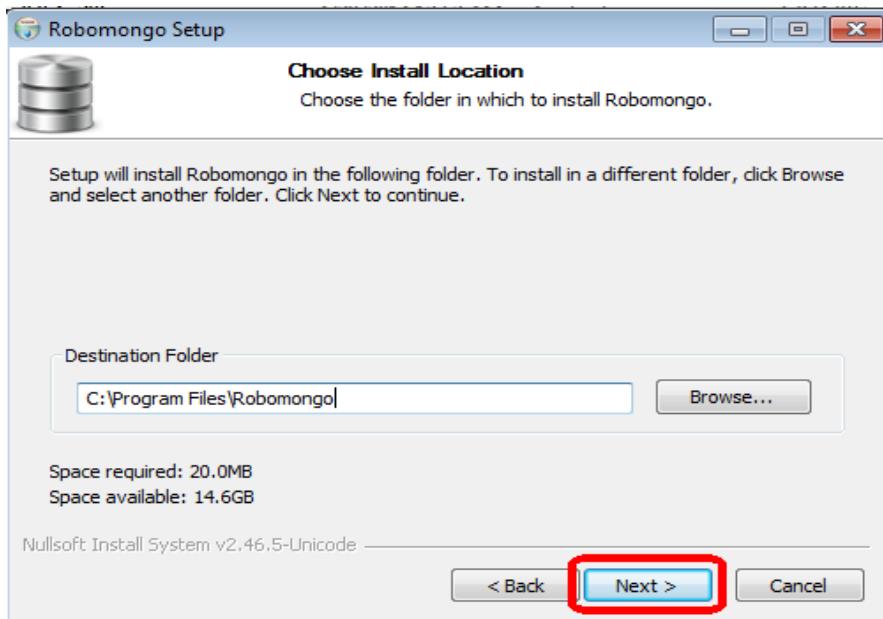


Fig. 15.3: Escolhendo o diretório de instalação

Depois é informado a opção de pasta de menu de inicialização que será criado:

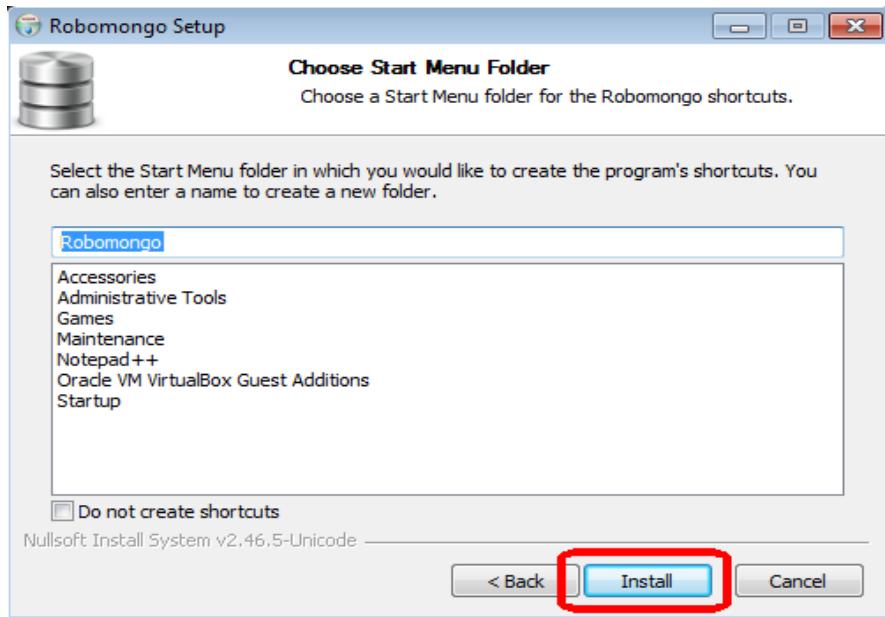


Fig. 15.4: Opção de pasta de menu de inicialização

Finalizada a instalação, é dada a opção de iniciar o Robomongo:

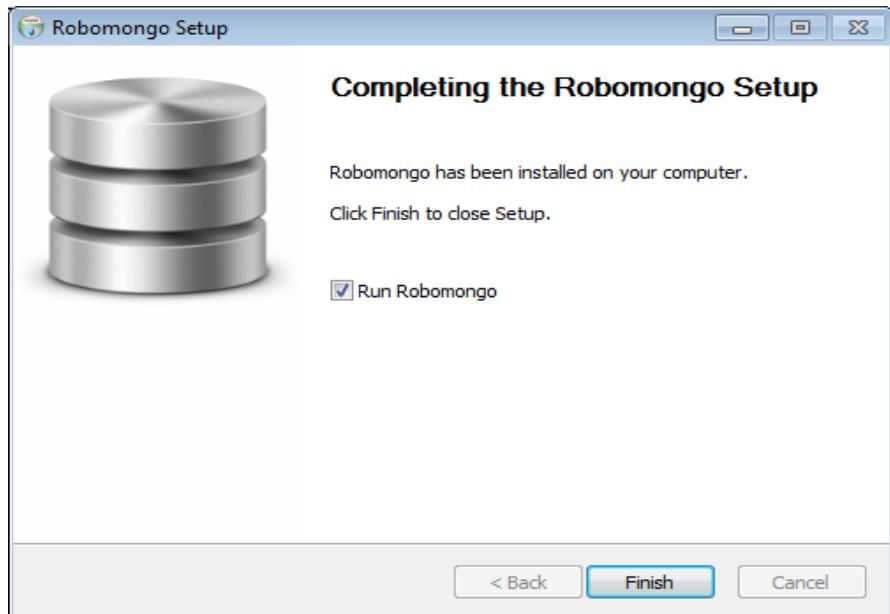


Fig. 15.5: Final da instalação

Configuração

Ao iniciar o Robomongo, é preciso configurar pelo menos uma conexão ao servidor. Isso é feito clicando na opção `Create`:

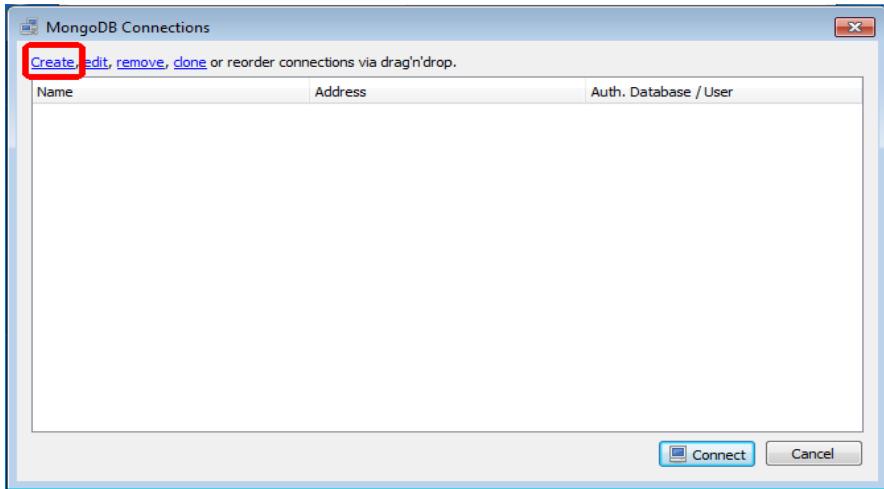


Fig. 15.6: Configurando uma conexão ao servidor

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`:

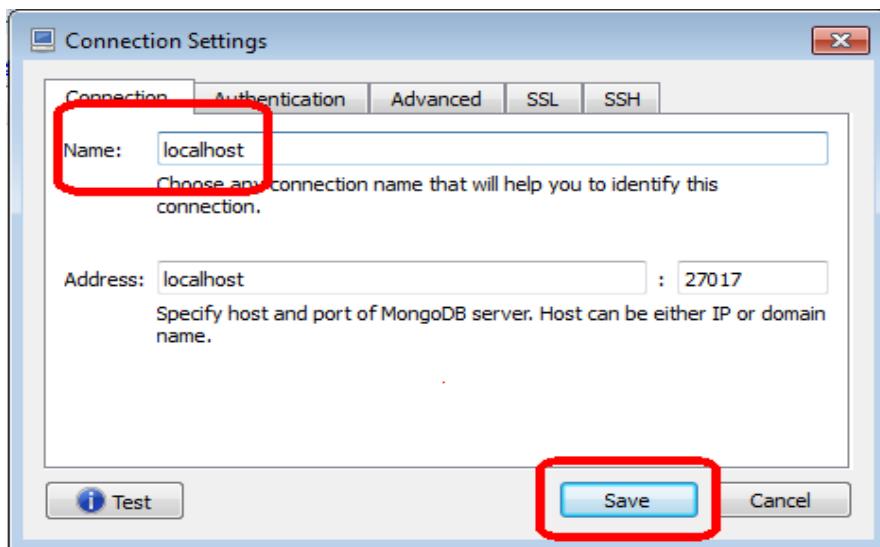


Fig. 15.7: Cadastrando uma conexão local

Depois de criada a conexão, podemos conectar clicando em Connect:

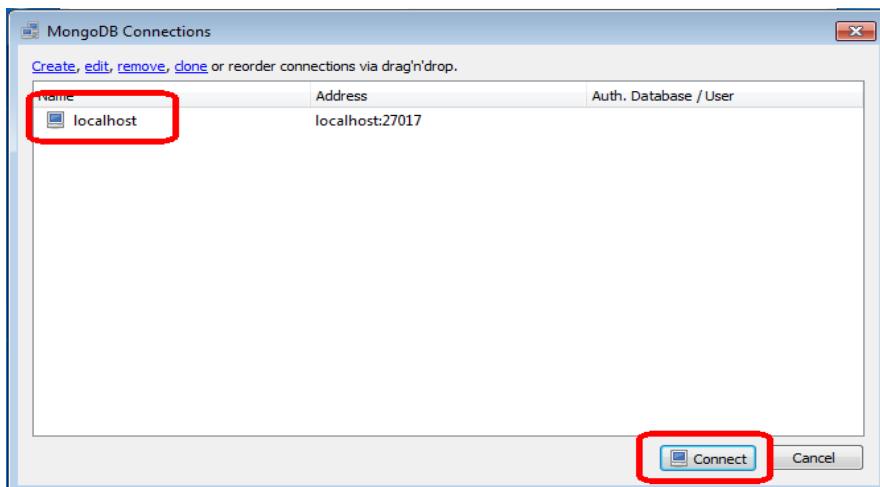


Fig. 15.8: Conectando localmente ao MongoDB

Temos algumas opções nativas do Robomongo, como por exemplo exibir informações do servidor na opção Server Status:

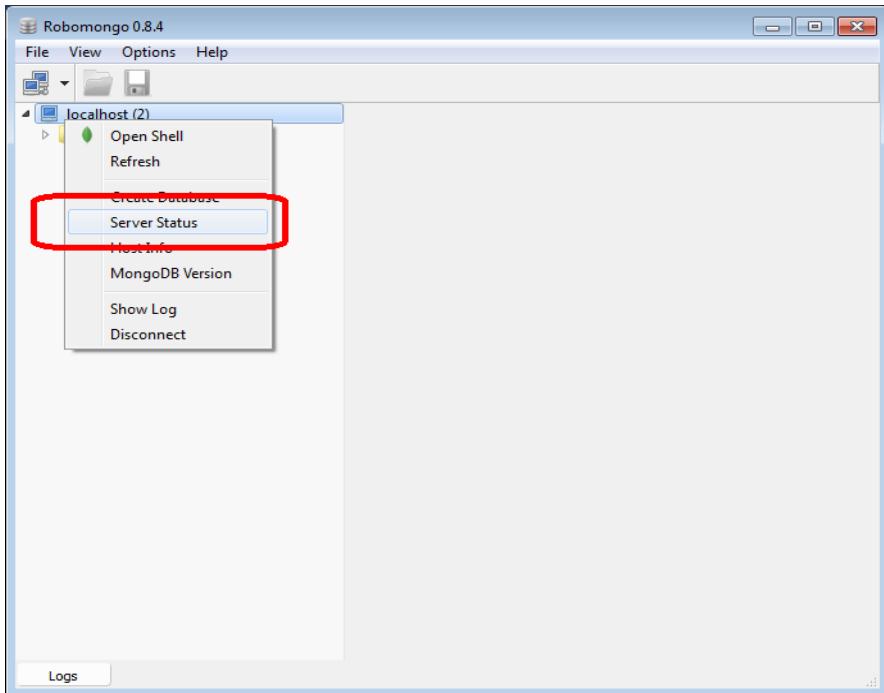


Fig. 15.9: Exibindo informações do servidor

O resultado da situação do servidor é exibido de maneira amigável:

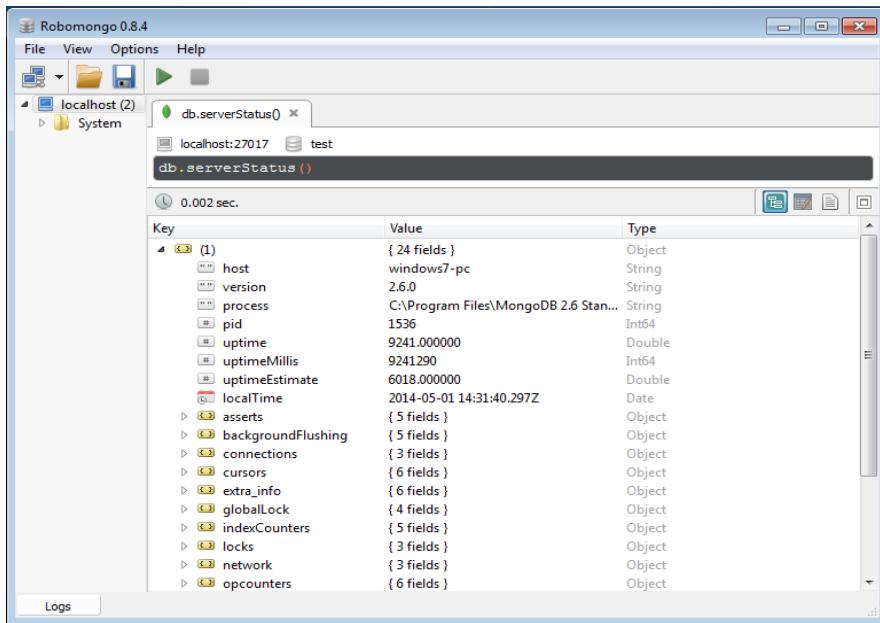


Fig. 15.10: Resultado da situação do servidor

Podemos também visualizar os logs do servidor com `Control + L`, que aparecem na janela inferior:

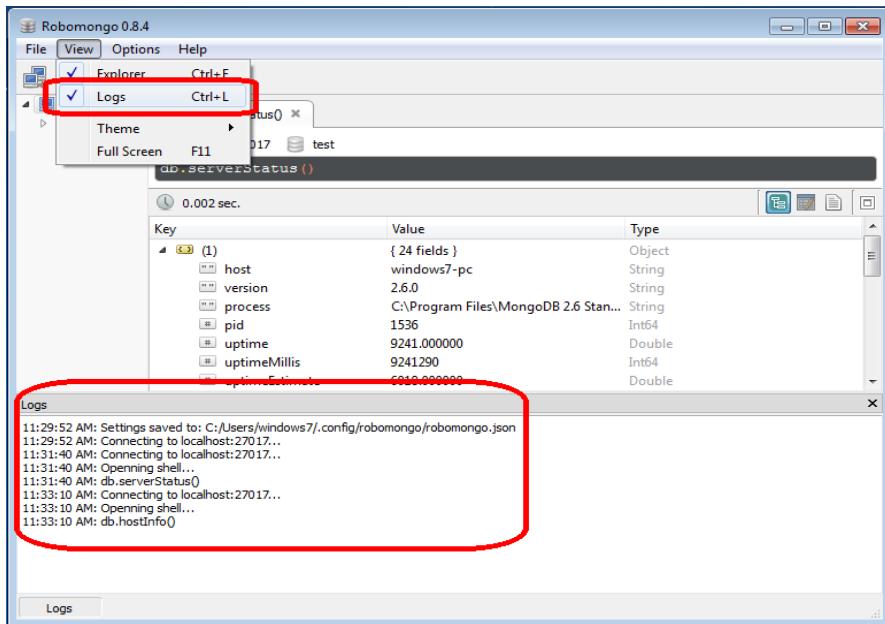


Fig. 15.11: Visualizando os logs do servidor

Além disso, conseguimos navegar pelas collections existentes, e visualizamos o seu conteúdo com a opção View Documents:

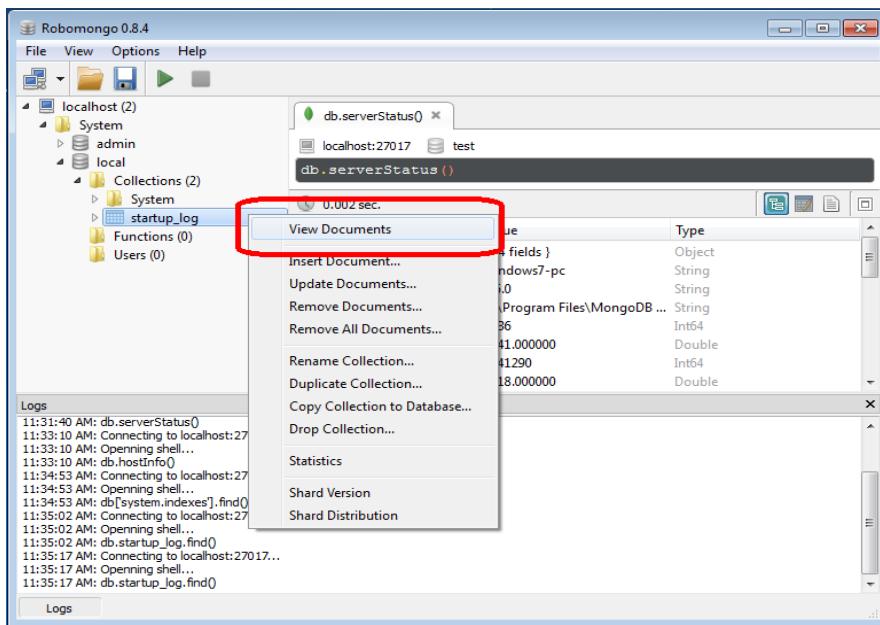


Fig. 15.12: Ativando a opção de exibir o conteúdo de uma collection

Temos também o comando executado exibido em uma nova aba:

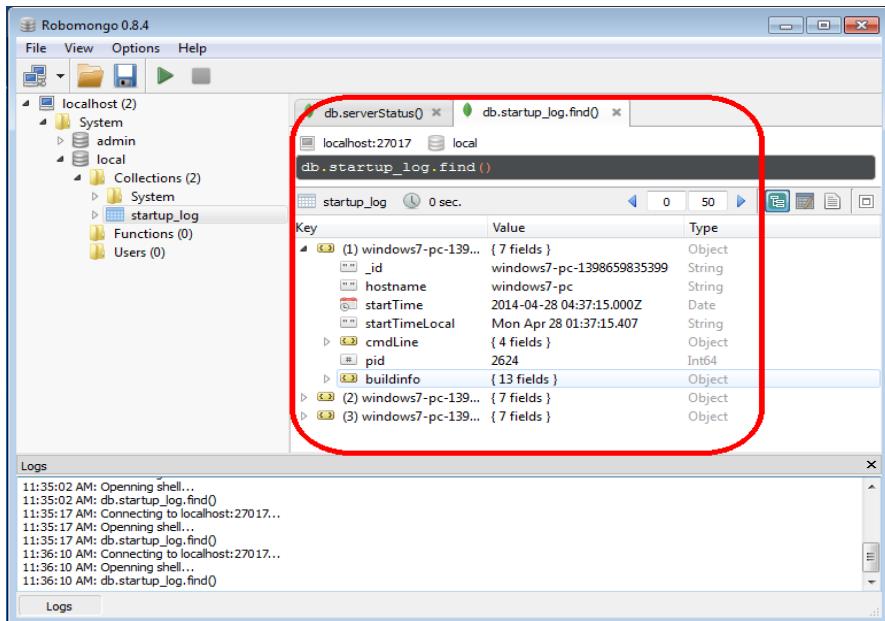
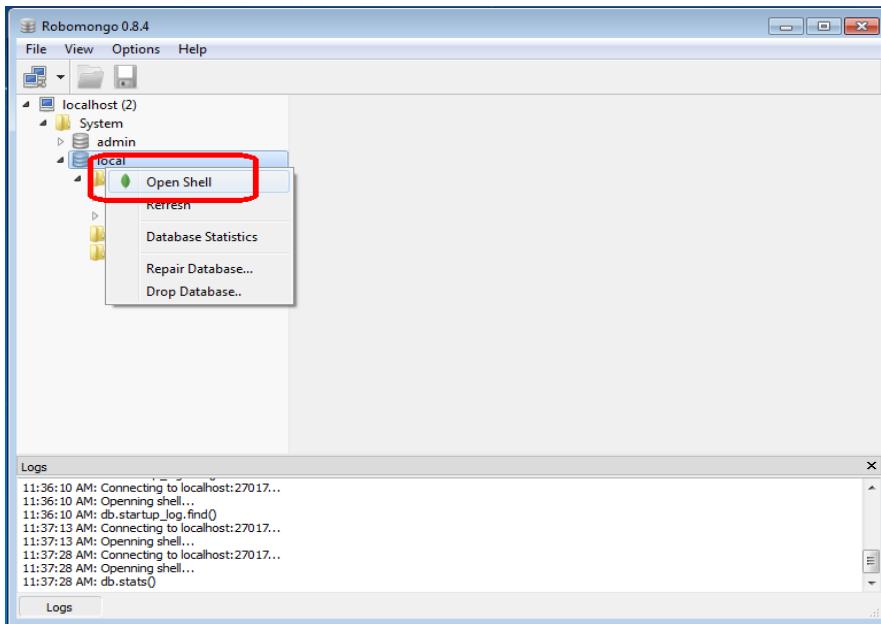


Fig. 15.13: Conteúdo de uma collection exibido em uma nova aba

E conseguimos abrir um novo terminal com a opção Open Shell:



Usando o terminal para exibir as collections:

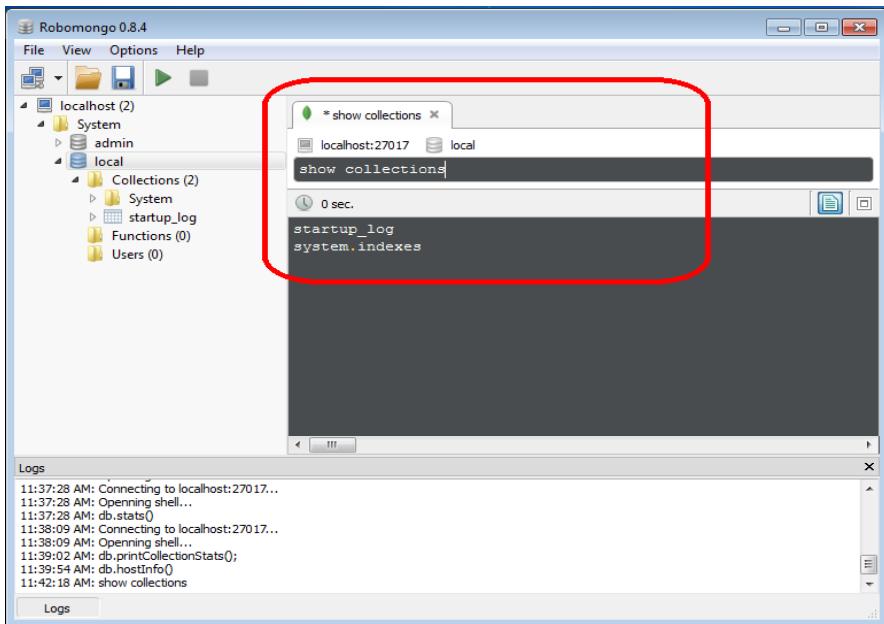


Fig. 15.15: Usando o terminal para exibir as collections

Um recurso muito bom é o autocomplete do terminal, que automaticamente exibe ao usuário as opções que ele pode usar:

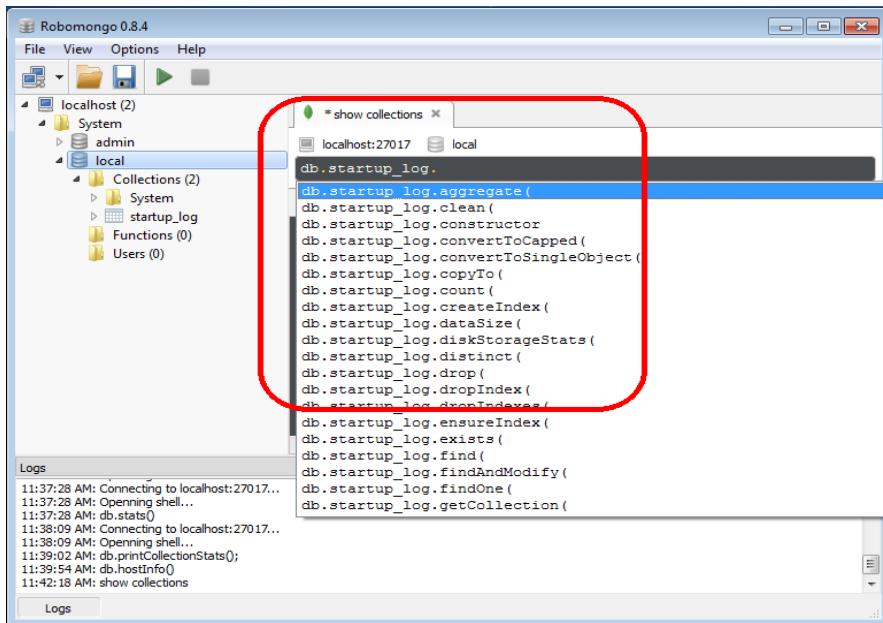


Fig. 15.16: Exemplo de autocomplete no terminal do Robomongo

CAPÍTULO 16

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/customers>. Seguem alguns dos mais famosos no Brasil: IBM, Foursquare, Bosch, Cisco, eBay, McAfee, Microsoft, MTV Networks, Telefónica 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*, 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.

O MongoDB possui índices?

Sim, com MongoDB conseguimos criar índices simples e compostos (mais de um campo), inclusive para arrays.

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 [12](#).

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, mas 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()`.