PRÁTICA MongoDB

COMANDOS MongoDB

```
use people;
```
cria e/ou seleciona o banco de dados "nome_do_banco" para uso

```
db.createCollection("users");
```
cria uma coleção chamada "users" no banco de dados atual

```
db.users.insert({name: "Bob", age: 32});
```
Esse é um método usado para inserir um documento JSON na coleção "users". O documento inserido aqui é {name: "Bob", age: 32}.

```
db.users.insert({name: "Alice", age: 28});
db.users.insert({name: "Charlie", age: 25});
```

```
db.users.insertMany([
  { name: "Caio", age: 28 },
  { name: "Angelise", age: 32 },
  { name: "Sérgio", age: 25 },
  { name: "Rosalina", age: 30 }
]);
```

```
//Inserir vários registros
var registros = [
  { name: "Alice", age: 28 },
  { name: "Bob", age: 32 },
  { name: "Charlie", age: 25 },
  { name: "David", age: 30 },
  { name: "Ella", age: 22 },
  { name: "Frank", age: 40 },
  { name: "Grace", age: 29 },
  { name: "Henry", age: 35 },
  { name: "Ivy", age: 27 },
  { name: "Jack", age: 31 },
  { name: "Kate", age: 28 },
  { name: "Leo", age: 26 },
  { name: "Mia", age: 24 },
  { name: "Noah", age: 33 },
  { name: "Olivia", age: 29 },
  { name: "Peter", age: 37 },
  { name: "Quinn", age: 23 },
  { name: "Ryan", age: 30 },
```

```
  { name: "Sara", age: 28 },
  { name: "Thomas", age: 39 },
  { name: "Uma", age: 21 },
  { name: "Victor", age: 34 },
  { name: "Wendy", age: 36 },
  { name: "Xander", age: 32 },
  { name: "Yara", age: 25 },
  { name: "Zane", age: 28 },
  { name: "Alex", age: 30 },
  { name: "Bella", age: 27 },
  { name: "Caleb", age: 29 },
  { name: "Diana", age: 31 }
];

db.users.insertMany(registros);
```

```
db.users.find();
```

db.users: Isso se refere à coleção **"users"** no banco de dados atualmente selecionado.

.find(): Isso é um método utilizado para realizar consultas em uma coleção. Quando usado sem argumentos, como neste caso, o método **find()** retorna todos os documentos da coleção.

```
db.users.find({}, {name: 1, age: 1, _id:0});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({}, {name: 1, age: 1, _id: 0}): Isso é um método de consulta usado para buscar documentos na coleção "users". Os argumentos dentro dos parênteses especificam o critério de filtro e a projeção dos campos.

O primeiro objeto **{}** é o critério de filtro. Neste caso, está vazio, o que significa que não há critério de filtro específico, e todos os documentos serão retornados.

O segundo objeto **{name: 1, age: 1, _id: 0}** é um objeto de projeção que define quais campos você deseja incluir (1) ou excluir (0) na saída. **name** e **age** têm o valor 1, o que significa que você deseja incluir esses campos na saída, enquanto **_id** tem o valor 0, o que significa que você deseja excluir o campo _id da saída.

```
db.users.find({age: 33}, {name: 1, age: 1, _id:0});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({age: 33}, {name: 1, age: 1, _id: 0}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. Os argumentos dentro dos parênteses especificam o critério de filtro e a projeção dos campos.

O primeiro objeto **{age: 33}** é o critério de filtro. Isso significa que apenas os documentos que tenham o campo "age" com o valor igual a 33 serão retornados.

O segundo objeto **{name: 1, age: 1, _id: 0}** é um objeto de projeção que define quais campos você deseja incluir (1) ou excluir (0) na saída. **name** e **age** têm o valor 1, o que significa que você deseja incluir esses campos na saída, enquanto **_id** tem o valor **0**, o que significa que você deseja excluir o campo **_id** da saída.

db.users.find({age: {$gt: 33}});

db.users: Isso se refere à coleção "**users**" no banco de dados atualmente selecionado.

.find({age: {$gt: 33}}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{age: {$gt: 33}}** é o critério de filtro. O operador **$gt** significa "**maior que**". Portanto, esse critério de filtro está buscando documentos em que o valor do campo "age" seja maior que 33.

db.users.find({age: {$lte: 33}});

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({age: {$lte: 33}}): Isso é um método de consulta usado para buscar documentos na coleção "users". O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{age: {$lte: 33}}** é o critério de filtro. O operador **$lte** significa "**menor ou igual a**". Portanto, esse critério de filtro está buscando documentos em que o valor do campo "age" seja menor ou igual a 33.

db.users.find({age: {$gt: 33, $lt: 40}});

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({age: {$gt: 33, $lt: 40}}): Isso é um método de consulta usado para buscar documentos na coleção "**users**". O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{age: {$gt: 33, $lt: 40}}** é o critério de filtro. Os operadores **$gt** e **$lt** significam **"maior que"** e **"menor que"**, respectivamente. Portanto, esse critério de filtro está buscando documentos em que o valor do campo **"age"** esteja entre maior que 33 e menor que 40.

db.users.find({age: 32, name: “Bob”});

db.users: Isso se refere à coleção **"users"** no banco de dados atualmente selecionado.

.find({age: 32, name: "Bob"}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{age: 32, name: "Bob"}** é o critério de filtro. Ele busca documentos onde o valor do campo **"age"** seja igual a 32 e o valor do campo **"name"** seja igual a "Bob".

db.users.find({$or:[{age:33}, {name: “Bob”}]});

db.users: Isso se refere à coleção **"users"** no banco de dados atualmente selecionado.

.find({$or:[{age:33}, {name: "Bob"}]}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{$or:[{age:33}, {name: "Bob"}]}** é uma expressão lógica de "ou". Isso significa que ele busca documentos onde o valor do campo "age" seja igual a 33 ou o valor do campo "name" seja igual a "Bob".

db.users.find({age: 33}).sort({name: 1});

db.users: Isso se refere à coleção **"users"** no banco de dados atualmente selecionado.

.find({age: 33}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. O argumento dentro dos parênteses especifica o critério de filtro, que no caso é encontrar documentos onde o campo "age" seja igual a 33.

.sort({name: 1}): Isso é um método usado para ordenar os resultados da consulta. O argumento dentro dos parênteses especifica o campo pelo qual você deseja ordenar e a direção da ordenação. No caso, você está ordenando pelo campo "name" em ordem alfabética ascendente (1).

db.users.find().sort({name: -1});

db.users: Isso se refere à coleção **"users"** no banco de dados atualmente selecionado.

.find(): Isso é um método de consulta usado para buscar todos os documentos na coleção **"users"**. Não há argumentos dentro dos parênteses, o que significa que você deseja buscar todos os documentos sem filtros específicos.

.sort({name: -1}): Isso é um método usado para ordenar os resultados da consulta. O argumento dentro dos parênteses especifica o campo pelo qual você deseja ordenar e a direção da ordenação. No caso, você está ordenando pelo campo "name" em ordem alfabética descendente (-1).

db.users.find({name: /Joe/});

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({name: /Joe/}): Isso é um método de consulta usado para buscar documentos na coleção **"users"**. O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{name: /Joe/}** é o critério de filtro. Ele utiliza uma expressão regular **/Joe/** como valor para o campo "name". Isso significa que ele busca documentos onde o valor do campo "name" contenha a sequência "Joe".

```
db.users.find({name: /^Joe/});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({name: /^Joe/}): Isso é um método de consulta usado para buscar documentos na coleção "users". O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{name: /^Joe/}** é o critério de filtro. Ele utiliza a expressão regular /^Joe/ como valor para o campo "name". Isso significa que ele busca documentos onde o valor do campo "name" começa com a sequência "Joe".

```
db.users.find({name: /Joe\^/});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({name: /Joe\^/}): Isso é um método de consulta usado para buscar documentos na coleção "users". O argumento dentro dos parênteses especifica o critério de filtro.

O objeto **{name: /Joe\^/}** é o critério de filtro. Ele utiliza a expressão regular **/Joe\^/** como valor para o campo "name". Isso significa que ele busca documentos onde o valor do campo "name" contenha a sequência "Joe^", considerando que o caractere "^" é escapado na expressão regular.

```
db.users.find().skip(20).limit(10);
```

db.users: isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find(): esse é um método de consulta usado para buscar todos os documentos na coleção "users". Não há argumentos dentro dos parênteses, o que significa que você deseja buscar todos os documentos sem filtros específicos.

.skip(20): esse é um método usado para pular um número específico de documentos na ordem de resultado da consulta. Neste caso, você está pulando os primeiros 20 documentos.

.limit(10): esse é um método usado para limitar o número de documentos retornados na consulta. Neste caso, você está limitando a consulta a retornar apenas 10 documentos após pular os primeiros 20.

```
db.users.findOne();
```

db.users: isso se refere à coleção "users" no banco de dados atualmente selecionado.

.findOne(): esse é um método de consulta usado para buscar o primeiro documento na coleção "users". Não há argumentos dentro dos parênteses, o que significa que você deseja buscar o primeiro documento sem filtros específicos.

```
db.users.distinct("name");
```

O comando **db.users.distinct("name")**; é usado para realizar uma operação de busca distintiva na coleção "users" do banco de dados atualmente selecionado no MongoDB. **Ele busca todos os valores únicos** presentes no campo "name" dentro dos documentos da coleção.

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.distinct("name"): Isso é um método usado para buscar os valores distintos no campo "name" da coleção "users". O argumento dentro dos parênteses especifica o campo pelo qual você deseja encontrar os valores únicos.

```
db.users.count();
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.count(): Isso é um método usado para contar o número total de documentos na coleção "users". Não há argumentos dentro dos parênteses, o que significa que você está contando todos os documentos na coleção.

```
db.users.find({age: {$gt: 30}}).count();
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({age: {$gt: 30}}): Isso é um método de consulta usado para buscar documentos na coleção "users" onde o campo "age" seja maior que 30.

.count(): Isso é um método usado para contar o número de documentos que correspondem aos critérios de filtro da consulta. Neste caso, ele conta quantos documentos na coleção "users" têm o campo "age" maior que 30.

```
db.users.find({age: {$exists: true}}).count();
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.find({age: {$exists: true}}): Isso é um método de consulta usado para buscar documentos na coleção "users" onde o campo "age" exista nos documentos, ou seja, o campo "age" esteja presente.

.count(): Isso é um método usado para contar o número de documentos que correspondem aos critérios de filtro da consulta. Neste caso, ele conta quantos documentos na coleção "users" possuem o campo "age" presente.

```
db.users.update({name: "Bob"}, {$set: {age: 33}}, {multi: true});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.update({name: "Bob"}, {$set: {age: 33}}, {multi: true}): Isso é um método de atualização usado para modificar os documentos na coleção "users".

O primeiro objeto **{name: "Bob"}** é o critério de filtro. Ele seleciona os documentos onde o valor do campo "name" seja igual a "Bob".

O segundo objeto **{$set: {age: 33}}** é o operador de atualização. Ele define o valor do campo "age" como 33 nos documentos selecionados. O operador **$set** é usado para atualizar campos específicos sem afetar outros campos.

O terceiro objeto **{multi: true}** é um parâmetro opcional que indica que a atualização deve ser aplicada a vários documentos que correspondam ao critério de filtro. Se o valor for **false** ou não for especificado, a atualização afetaria apenas o primeiro documento encontrado.

Em resumo, esse comando busca documentos na coleção "users" onde o campo "name" seja igual a "Bob" e atualiza o campo "age" para o valor 33 nesses documentos. A opção **{multi: true}** garante que a atualização seja aplicada a todos os documentos correspondentes.

```
db.users.update({name: "Bob"}, {$inc: {age: 2}}, {multi: true});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.update({name: "Bob"}, {$inc: {age: 2}}, {multi: true}): esse é um método de atualização usado para modificar os documentos na coleção "users".

O primeiro objeto **{name: "Bob"}** é o critério de filtro. Ele seleciona os documentos onde o valor do campo "name" seja igual a "Bob".

O segundo objeto **{$inc: {age: 2}}** é o operador de atualização. Ele utiliza o operador **$inc** para incrementar o valor do campo "age" em 2 unidades nos documentos selecionados. Por exemplo: Se Bob tinha 30 anos, após o incremento a idade passou a ser 32.

O terceiro objeto **{multi: true}** é um parâmetro opcional que indica que a atualização deve ser aplicada a vários documentos que correspondam ao critério de filtro. Se o valor for false ou não for especificado, a atualização afetaria apenas o primeiro documento encontrado.

O operador **$inc** é útil quando deseja modificar campos numéricos em documentos de forma incremental (aumentando ou diminuindo o valor). Ele é especialmente útil para operações como contadores ou atualizações baseadas em valores numéricos existentes.

```
db.users.remove({name: "Bob"});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.remove({name: "Bob"}): Isso é um método de remoção usado para excluir documentos da coleção "users".

O objeto **{name: "Bob"}** é o critério de filtro. Ele seleciona os documentos onde o valor do campo "name" seja igual a "Bob".

Após a execução desse comando, todos os documentos que têm o valor "Bob" no campo "name" serão removidos da coleção "users".

O método **.remove()** foi depreciado nas versões mais recentes do MongoDB. Em vez disso, é recomendado o uso do método **.deleteOne()** para remover um único documento ou do método **.deleteMany()** para remover vários documentos, dependendo das suas necessidades específicas.

```
db.users.ensureIndex({name: 1});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.ensureIndex({name: 1}): Este é um método utilizado para criar um índice na coleção "users". O argumento **{name: 1}** especifica qual campo deve ser indexado e em que ordem. No caso, **name: 1** indica que o campo "name" deve ser indexado em ordem ascendente.

Os índices no MongoDB são utilizados para otimizar as operações de consulta, permitindo que o banco de dados recupere dados de maneira mais eficiente. No entanto, como mencionado anteriormente, o método **ensureIndex()** foi substituído pelo método **createIndex()**, que oferece mais flexibilidade e opções para a criação de índices.

Para criar um índice usando o método createIndex(), você pode fazer o seguinte:

```
db.users.createIndex({ name: 1 });
```

```
db.users.ensureIndex({name: 1, age: -1});
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.ensureIndex({name: 1, age: -1}): Este é um método utilizado para criar um índice composto na coleção "users". O argumento **{name: 1, age: -1}** especifica quais campos devem ser indexados e em que ordem. **name: 1** indica que o campo "name" deve ser indexado em ordem ascendente, e **age: -1** indica que o campo "age" deve ser indexado em ordem descendente.

Os índices compostos são úteis quando você deseja otimizar consultas que envolvem múltiplos campos. No entanto, como mencionado anteriormente, o método **ensureIndex()** foi substituído pelo método **createIndex()**.

Para criar um índice composto usando o método createIndex(), você pode fazer o seguinte:

```
db.users.createIndex({ name: 1, age: -1 });
```

```
db.users.find({age: 32}).explain();
```

db.users: Isso se refere à coleção "users" no banco de dados atualmente selecionado.

.**find({age: 32})**: esse é um método de consulta usado para buscar documentos na coleção "users" onde o campo "age" seja igual a 32.

.explain(): Isso é um método usado para obter informações detalhadas sobre a execução da consulta. Quando chamado após a consulta **.find()**, ele retorna informações estatísticas e de desempenho sobre a execução da consulta.

http://s3.amazonaws.com/info-mongodb-com/sql_to_mongo.pdf

CONSULTAR OS BANCOS EXISTENTES

show dbs;

CRIAR UM BANCO CHAMADO SORTEIO

use sorteio;

CONSULTAR AS COLEÇÕES

show collections;

ADICIONAR REGISTROS

Para adicionar novos registros, usamos o comando insert :

db.<nome-da-collection>.insert({ <campo1>:<valor1>,

<campo2>:<valor2>,

...

});

Vamos inserir um sorteio:

```
db.megasena.insert(
{
"Concurso" : 99999,
"Data Sorteio" : "07/09/2016",
"1ª Dezena" : 1,
"2ª Dezena" : 2,
"3ª Dezena" : 3,
"4ª Dezena" : 4,
"5ª Dezena" : 5,
"6ª Dezena" : 6,
"Arrecadacao_Total" : 0,
"Ganhadores_Sena" : 0,
"Rateio_Sena" : 0,
"Ganhadores_Quina" : 1,
"Rateio_Quina" : "88000",
"Ganhadores_Quadra" : 55,
```

```
"Rateio_Quadra" : "76200",
"Acumulado" : "NAO",
"Valor_Acumulado" : 0,
"Estimativa_Prêmio" : 0,
"Acumulado_Mega_da_Virada" : 0
});
```

Além dos sorteios, anotaremos o CPF do ganhador, no formato número do concurso e CPF.

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

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

No MongoDB, além de não necessitarmos criar a *collection* para armazenar os dados, não precisamos alterá-la também caso desejemos adicionar ou remover colunas.

```
db.ganhadores.count();

db.ganhadores.insert({"Concurso":7878787,
"CPF":12345151410});

db.ganhadores.insert({"Concurso":99999,
"CPF":12345678900,
"Nome":"Laura Cardoso"});

db.ganhadores.find().pretty();
```

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, ainda 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, mostrando que qualquer valor pode ser inserido, desde que seja único.

```
db.valores.insert({"_id" : 111,"valor" : 1000});

db.valores.insert({"_id" : "importante","valor" : 2000});
```

```
db.valores.insert({"valor" : 3000});

db.valores.find();
```

Note que, no primeiro *insert* , inserimos um número; no segundo, um texto; e no terceiro, nada. Logo, o MongoDB cria implicitamente um valor único de forma automática. Considerando a *collection* valores, vamos cadastrar um novo valor com taxa de 15%:

```
db.valores.insert({"valor":4000,taxa:"15%"});
```

Agora na mesma *collection* temos registros com e sem o atributo taxa. Para listar os registros que possuem a nova coluna devemos usar a opção *$exists*:

```
db.valores.find({taxa:{$exists:true}});
```

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 passíveis de se associarem aos 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("xxxxxxxxx")});
```

Existe, uma opção que se assemelha ao comando SQL, que é o 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("6352923a17565d7ce362bf0e")});
```

```
db.ganhadores.update(
{"_id" : ObjectId("6352923a17565d7ce362bf0e")},
{ $set: {"Nome":"Zé do caixão"}});
```

Outro ponto do comando update que é importante saber é que, por padrão, ele atualiza apenas o primeiro registro que obedecer ao critério de busca especificado. Para que ele altere todos os registros, é preciso adicionar mais um parâmetro booleano, o *multi* , que por padrão é false .

Por que existe essa opção? Quem nunca fez um *UPDATE* em uma base relacional e, esquecendo-se da cláusula *WHERE* , detonou todos os dados da tabela? Pensando nisso, sem nenhuma cláusula, no MongoDB o padrão é atualizar um único registro/ documento. Já nas bases relacionais, o padrão é atualizar todos os registros.

Neste primeiro exemplo, apenas uma linha é alterada:

```
db.ganhadores.find();

db.ganhadores.update({},{ $set: {"CPF": 55555555555 }});

db.ganhadores.find();
```

Agora, alterando multi para true , todas as linhas são modificadas.

```
db.ganhadores.update({},
{ $set: {"CPF": 55555555555 }},{multi:true});

db.ganhadores.find();
```

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": 33333333333 }},
{multi:0,upsert:0});

db.ganhadores.find();
```

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 , fazemos:

```
db.ganhadores.find({"CPF" : 55555555555}).count();

db.ganhadores.remove({"CPF" : 55555555555});

db.ganhadores.count();
```

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

```
db.ganhadores.remove({});

db.ganhadores.count();
```

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:

```
db.collectionNovaNaoPrecisaCriarAntes.insert(
{"uma coluna":"só"});

db.collectionNovaNaoPrecisaCriarAntes.count();

db.collectionNovaNaoPrecisaCriarAntes.find();

db.collectionNovaNaoPrecisaCriarAntes.drop();
```

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 removê-la em todos os documentos, e não apenas no primeiro. Veja um exemplo:

```
db.messages.update( {},
{ $unset : { titulo: 1 }},
false,true);
```

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

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

Veja um exemplo:

```
db.messages.update( {},
{ $rename :
{ "mailboxx" : "mailbox" }},
false,true);
```

VALIDAÇÃO DOS DADOS

A partir da versão 3.2, foi acrescentada a opção de adicionar um sistema de validação (validator) dentro de uma collection. Em nosso exemplo, vamos adicionar uma validação que verifica se uma coluna CPF existe:

```
db.runCommand({
collMod:"ganhadores",
validator: {
"CPF":{$exists:true}
}});
```

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

```
db.ganhadores.insert({"nome":"Sortudo","CPF":123456});
```

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

```
db.ganhadores.insert({"nome":"Sortudo sem CPF"});
```

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

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

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

```
db.ganhadores.insert({"nome":"Sortudo","CPF":123,
"email":"email@gmail.com"});
```

No e-mail com domínio correto, sem problemas:

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

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

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

Um registro válido é inserido sem problemas:

```
db.ganhadores.insert({"nome":"Sortudo",
"CPF":"W123W","email":"email@boaglio.com"});
```

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

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

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

---

PRINCIPAIS COMANDOS

Operações de Banco de Dados:

use nome_do_banco: Seleciona um banco de dados para ser usado.

show dbs: Lista todos os bancos de dados disponíveis.

db.dropDatabase(): Apaga o banco de dados atualmente selecionado.

Operações de Coleção:

db.createCollection("nome_da_colecao"): Cria uma nova coleção.

show collections: Lista todas as coleções no banco de dados atual.

db.nome_da_colecao.drop(): Remove a coleção especificada.

Operações de Documento:

db.nome_da_colecao.insert(documento): Insere um novo documento na coleção.

db.nome_da_colecao.find(query): Busca documentos na coleção com base em um critério de consulta.

db.nome_da_colecao.update(query, update): Atualiza documentos na coleção com base em um critério de consulta e uma operação de atualização.

db.nome_da_colecao.remove(query): Remove documentos da coleção com base em um critério de consulta.

Outras Operações:

Ctrl + L: Limpar a tela

db.nome_da_colecao.count(query): Conta o número de documentos que correspondem ao critério de consulta.

db.nome_da_colecao.aggregate(pipeline): Realiza operações de agregação nos documentos da coleção.

db.nome_da_colecao.ensureIndex({campo: 1}): Cria um índice na coleção para melhorar o desempenho de consultas.

db.nome_da_colecao.explain().find(query): Retorna informações de execução da consulta.

Material de Estudos

- MongoDB Documentation

Link: https://www.mongodb.com/docs/develop-applications/

- MongoDB CRUD Tutoriais

Link: https://www.mongodb.com/docs/v3.0/tutorial/insert-documents/

- Curso M001: MongoDB Basics
  Link: https://university.mongodb.com/mercury/M001/2022_October_11/overview

- Curso M320: Data Modeling
  Link: https://university.mongodb.com/courses/M320/about

- MongoDB operador condicional
  Link: http://www.w3big.com/pt/mongodb/mongodb-operators.html#gsc.tab=0

- JDoodle – online mongo Terminal
  Link: https://www.jdoodle.com/online-mongodb-terminal/

- Mapping SQL to MongoDB
  Link: http://s3.amazonaws.com/info-mongodb-com/sql_to_mongo.pdf