Anotações Mongodb

- Mapa Mental
- SQL e NoSQL: trabalhando com bancos relacionais e não relacionais
- Banco de dados relacionais: conhecendo conceitos, terminologias e ferramentas
- O que é NoSql
- Banco de Dados na Nuvem: conhecendo as soluções AWS, Azure e GCP

Para exercitar você pode utilizar uma base de dados própria, ou utilizar uma base de dados disponibilizadas em plataformas online, como a kaggle, que é uma comunidade de online de cientistas de dados, onde você pode explorar, analisar e compartilhar dados.

MongoDB é um projeto open source com distribuição gratuita para Linux, Mac e Windows. Ele está disponível em duas edições de servidor: Community e Enterprise.

A versão Community edition é a edição gratuita do MongoDB disponível para fins de avaliação e desenvolvimento. Com a versão Community edition, pode-se criar diversos bancos de dados e coleções, além de tudo, trabalhar com diversos documentos e realizar diversas análises dos dados.

Já a versão Enterprise edition é a edição comercial do MongoDB. Ela inclui recursos adicionais não disponibilizados na versão gratuita, como recursos avançados de segurança, assistência especializada, ferramentas poderosas, entre outros.

Além das duas versões de servidores, o MongoDB também disponibiliza a sua versão hospedada na nuvem, o MongoDB Atlas. Você pode experimentar o Atlas gratuitamente com um cluster de 512 MB disponibilizado pela MongoDB.

Neste curso, vamos trabalhar com a versão Community edition, que supre todas as nossas necessidades durante os nossos estudos.

Como interface gráfica utilize o MongoDB Compass

Criando o banco de dados

- Para abrir o shell do mongodb execute:

```
mongosh
```

- Para visualizar as bases de dados existentes:

```
show databases
```

- Para criar/usar uma base de dados:

```
use {nome_base}
```

Por padrão, o MongoDB já vem com três bancos de dados criados, o Admin, config e local. Além desses, nós temos a liberdade de criar novos bancos de dados. Porém, precisamos seguir algumas restrições:

- Diferenciação de maiúsculas e minúsculas do nome do banco de dados

Os nomes de banco de dados diferenciam maiúsculas de minúsculas no MongoDB. Por exemplo: se o banco de dados AluraDB já existir, o MongoDB retornará um erro se você tentar criar um banco de dados chamado AluraDB.

- Restrições sobre nomes de banco de dados para Windows

Para implantações do MongoDB em sistemas operacionais Windows, os nomes de banco de dados não podem conter nenhum dos seguintes caracteres:

/. "$*<>:|?

- Restrições sobre nomes de banco de dados para sistemas Unix e Linux

Para implantações do MongoDB em em sistemas operacionais Unix e Linux, os nomes de banco de dados não podem conter nenhum dos seguintes caracteres:

/. "$

Além disso, os nomes de banco de dados não podem conter o caractere nulo.

- Comprimento dos nomes de banco de dados

Os nomes de banco de dados não podem estar vazios e devem ter menos de 64 caracteres.

Criando coleções

Coleções no mongoDB são análogas às tabelas em bancos de dados relacionais. É necessário, via shell, criar ao menos uma coleção em uma base de dados para que ambos sejam exibidos no MongoDB Compass.

- Para criar uma coleção:

```
db.createCollection("{nome_colecao}")
```

Assim como para criar um banco de dados, também existem restrições para se criar uma coleção aqui no MongoDB, que são:

- Os nomes das coleções devem começar com um sublinhado ou um caractere de letra.
- Não podem:
    - Conter o $.
    - Ser uma string vazia (por exemplo "",).

- Conter o caractere nulo.
- Começar com o system.prefixo. (Reservado para uso interno).

Excluindo banco de dados e coleções

- Para excluir uma coleção execute:

```
db.{nome_colecao}.drop()
```

- Para excluir uma base de dados é preciso estar conectado a ela, em seguida execute o comando:

```
db.dropDatabase()
```

Criando documentos

O MongoDB armazena registros de dados como documentos BSON, que é uma representação binária de documentos JSON.

O valor de um campo em um documento pode ser qualquer um dos tipos de dados BSON, incluindo outros documentos, matrizes e matrizes de documentos, então vamos conhecer alguns desses tipos de dados?

NULL: armazena valores nulos;

Boolean: pode armazenar valores true ou falso;

Number: número com sinal que pode ter uma notação com E exponencial;

Inteiro: pode armazenar o tipo de dados inteiro em duas formas, inteiro assinado de 32 bits e inteiro assinado de 64 bits;

String: uma sequência de um ou mais caracteres Unicode;

Object: um array não ordenado com itens do tipo chave/valor, também conhecidos como documentos aninhados;

Array: armazena uma lista ordenada de qualquer tipo, criada usando colchetes e com cada elemento separado por vírgulas;

ObjectId: identificador único de um registro do MongoDB;

Date(): retorna a data atual em formato de string; e

New Date() e **ISODate()**: retornam um objeto de data.

Esses são apenas alguns tipos de dados que podemos trabalhar no MongoDB, a título de complemento, você pode acessar também a documentação do MongoDB sobre BSON types.

Assim, como para criar um banco de dados e coleções, também existem restrições ao se criar um documento:

- O nome do campo _id é reservado para uso como chave primária. Seu valor deve ser único na coleção, é imutável e pode ser de qualquer tipo que não seja um array.
- Os nomes dos campos não podem conter o caractere NULL.

Os documentos BSON, também possuem restrições de tamanho:

- O tamanho máximo de um documento BSON é 16 megabytes.
- Para inserir um documento na coleção da base de dados:

```
db.series.insertOne({"chave":"valor", "chave":"valor"})
```

- Para inserir vários documentos na coleção da base de dados:

```
db.series.insertMany([{"chave":"valor", "chave":"valor"},{"chave":"valor",
"chave":"valor"}])
```

Realizando consultas

- Informar 0 quando ao realizar uma projeção não quiser retornar um campo, exemplo:

```
{id: 0}
```

- Para consultar em ordem dos registros com sort, informar 1 no campo a ser ordenado, exemplo:

```
{"Nome": 1}
```

- Ainda na consulta o mongoDB tem o campos de **Max Time MS** para informar o timeout da consulta, também há o **Limit** para trazer a quantidade de documentos a serem retornados, e o **Skip** para informar até quantos documentos podem ser pulados durante a consulta

FILTER: utilizado para especificar qual será a condição que os documentos devem atender para serem retornados na busca.

PROJECT: utilizado para especificar quais campos serão ou não retornados na consulta.

```
Ao Informar o nome do campo e informar 0, todos os campos, exceto os campos
especificados no campo project, são retornados. Se o campo receber o valor
de 1, ele será retornado na consulta. O campo _id é retornado por padrão, a
menos que este seja especificado no campo project e definido como 0.
```

SORT: especifica a ordem de classificação dos documentos retornados.

```
Para especificar a ordem crescente de um campo, defina o campo como 1.
Para especificar a ordem decrescente de um campo, defina o campo como -1.
```

MAX TIME MS: define o limite de tempo cumulativo em milissegundos para processar as operações da barra de consulta. Se o limite de tempo for atingido antes da conclusão da operação, o Compass interrompe a operação.

COLLATION: utilizado para especificar regras específicas do idioma para comparação de textos, como regras para letras maiúsculas ou minúsculas, acentos, entre outros.

SKIP: especifica quantos documentos devem ser ignorados antes de retornar o conjunto de resultados.

LIMIT: especifica o número máximo de documentos a serem retornados.

Comparando dados e realizando consultas

O MongoDB nos fornece alguns operadores para a manipulação dos dados e que podemos conhecer através da documentação, disponível neste link. Por exemplo, os operadores de comparação, que são bem semelhantes aos operadores de comparação na linguagem SQL. A diferença é que na linguagem SQL, fazemos as comparações utilizando símbolos. No MongoDB, usamos um comando específico.

Exemplo:

```
{"Ano de lançamento": {$in: [2019, 2020]}}
```

O MongoDB disponibiliza um único método para realizarmos consultas por linha de comando, o Find, que tem a seguinte estrutura:

```
db.users.find(                          ← collection
   { age: { $gt: 18  } },               ← query criteria
     { name: 1, address: 1 }            ← projection
).limit(5)                              ← cursor modifier
```

Precisamos indicar a coleção, "collection", que estamos realizando nas nossas buscas, então, informamos "db.users", que, no caso, seria a nossa coleção de "Séries", seguido de ".find" e parênteses. Dentro dos parênteses, abrimos o primeiro conjunto de chaves e especificamos o filtro, o critério para a coleção, "query criteria", isto é, o que os documentos precisam atender para serem retornados na busca.

Em seguida, adicionamos outro conjunto de chaves com a projeção, "projection", onde especificamos quais campos serão, ou não, retornados na busca. Para finalizar, temos o "cursor modifier", onde limitamos quantos documentos serão retornados na busca.

Já conhecemos a estrutura do comando "Find". Agora voltaremos para a linha de comando e realizaremos nossas buscas. Quando abrimos a linha de comando, por padrão, o MongoDB vem conectado ao banco Test. Precisamos nos conectar novamente ao Alura_Series e executar as buscas.

Para retornarmos todos os documentos que existem na coleção "Séries", vamos executar a seguinte consulta: db.series, que se refere à coleção, .find(), que é o método. Não é necessário abrir chaves, pois não passaremos nenhum filtro ou projeção.

```
db.series.find()
```

Quando esse código é executado na linha de comando, o resultado são todos os 100 documentos que existem na nossa coleção "Séries". Também podemos filtrar os documentos que serão retornados. Para isso, executaremos o comando: db.series.find().

Como informaremos um critério, desta vez abriremos chaves. Porém, Como não vamos especificar nenhuma projeção, não é necessário abrir um segundo conjunto de chaves, basta o primeiro, onde definiremos o campo, "Ano de lançamento", e o valor, "2018". Na nossa busca, devem ser retornados apenas documentos com ano de lançamento igual a 2018.

```
db.series.find({"Ano de lançamento": 2018})
```

E se quiséssemos projetar quais campos desses documentos poderíamos retornar? Vamos executar a consulta: db.series.find(). Quando desejamos especificar só a projeção, precisamos referenciá-la com um primeiro conjunto de chaves vazio, que é onde indicaríamos o filtro, db.series.find({}, }. Então, nós não faremos nenhum filtro, todos os documentos devem ser retornados, por isso o conjunto de chaves vazio.

No segundo conjunto de chaves, após a vírgula, diremos quais campos não queremos retornar. O primeiro campo é de Série, com valor 1, porque deve ser retornado. O segundo campo é de Ano de lançamento, com valor 1, porque também deve ser retornado. O terceiro campo é id_ com valor zero, porque não deve ser retornado.

```
db.series.find({},{"Série":1, "Ano de lançamento": 1, "_id":0})
```

Ao executarmos, serão retornados os campos "Série" e o "Ano de lançamento". Também poderíamos utilizar operadores de comparação para passarmos mais de um valor no filtro. Para isso, vamos passar db.series.find(). Como o filtro fica no primeiro conjunto de chaves e não faremos nenhuma projeção, não é necessário abrir um segundo conjunto de chaves.

Portanto, no primeiro e único conjunto de chaves, informaremos o campo, "Ano de lançamento", e abriremos outro conjunto de chaves dentro do filtro, e utilizaremos o operador "$in": {$in: [2019, 2020}. Entre colchetes, informaremos os anos de lançamento 2019 e 2020.

```
db.series.find({"Ano de lançamento": {$in: [2019,2020]}})
```

Agora, o nosso retorno apresenta os documentos com ano de lançamento 2019 e 2020. Também podemos estipular um limite, isto é, limitar a quantidade de documentos que serão retornados na busca. Para isso,

informaremos o comando db.series.find(), sem nada nos parênteses, pois não especificaremos nenhum filtro ou projeção de quais campos não devem ser retornados. Fora do find(), passaremos .limit(5).

```
db.series.find().limit(5)
```

Agora, vamos ordenar a forma como nossos documentos serão retornados. Para isso, precisamos executar a seguinte consulta:

```
db.series.find().sort({"Série":1})
```

Então, começamos pelo padrão, db.series.find(). Nós não vamos informar nenhum critério ou projeção, logo, não é necessário passar colchetes. Fora do find(), passamos o .sort() que é um modificador.

Dentro dos parênteses, abriremos chaves e informaremos por qual campo desejamos ordenar o resultado. Neste caso, é pelo campo Série. Por fim, passaremos o valor 1, determinando que os documentos devem ser ordenados de forma crescente. Se quiséssemos uma ordenação decrescente, passaríamos o valor -1.

O resultado é que todos os documentos foram retornados, mas ordenados de forma crescente a partir de sua descrição, que é o campo "Série". Agora, vamos retornar apenas documentos com valor maior que um número determinado. Para isso, utilizaremos o operador de comparação "$gte".

```
db.series.find({"Temporadas disponíveis": {$gte: 4}})
```

Então, db.series.find(). Dentro dos parênteses, abriremos chaves e passaremos o campo Temporadas disponíveis. Ainda dentro das chaves, abriremos mais um par de chaves e informaremos o operador $gte: 4. Com isso, estamos determinando que o retorno seja de documentos com quantidade de temporadas disponíveis maior ou igual a 4.

Também podemos fazer o inverso com o operador "$ne", que vai retornar apenas documentos com valores diferentes do que o que passaremos para ele.

```
db.series.find({"Ano de lançamento": {$ne: 2020}})
```

Executamos a consulta db.series.find(). Nos parênteses, abriremos um conjunto de chaves e passaremos o campo. Ainda dentro das chaves, abriremos mais um conjunto de chaves com o operador e o valor, $ne: 2020. Todos os documentos com o ano de lançamento diferente de 2020 serão retornados na busca.

É possível também fazer buscas utilizando, como critério, os dados que estão armazenados no array.

```
db.series.find({"Genero": {$all:["Ação", "Comédia"]}})
```

Portanto, db.series.find(), passamos chaves e informamos o campo, que é o Genero e utilizaremos o operador "$all" em um novo conjunto de chaves, seguido de dois pontos, colchetes e os valores "Ação" e "Comédia".

Com esse comando, estamos especificando que todos os documentos de campo Genero que sejam array e tenham os valores "Ação" e "Comédia" ou "Comédia" e "Ação" devem ser retornados na busca.

Nós aprendemos a realizar buscas tanto pelo MongoDB Compass, quanto pela linha de comando, utilizando o método .find()

QUERY: especificamos a condição que os nossos documentos devem atender para serem retornados na nossa consulta.

PROJECTION: especificamos quais campos devem ou não ser retornados na nossa consulta.

Atualizando dados

O MongoDB disponibiliza dois métodos para a atualização de documentos por linha de comando. Um deles é o updateOne, que atualiza um único documento por vez. O outro método é o updateMany, que atualiza vários documentos ao mesmo tempo.

Além destes, há o ReplaceOne. Logo abaixo do vídeo, você pode acessar um Para saber mais explicando sobre esse método e sobre a sua estrutura.

A estrutura do comando de update, é:

```
db.users.updateMany(                  ← collection
  { age:  { $lt: 18 } },              ← update filter
    { $set:  { status: "reject" } }   ← update action
)
```

Precisamos especificar a coleção, collection, então vamos informar db., seguido da coleção e do método que, neste caso, é o updateMany() ou updateOne().

O comando é dividido em duas partes: primeiro, especificamos qual será o filtro, isto é, a condição que o documento precisa atender para que o parâmetro seja atualizado; depois, passamos o operador de atualização que, neste caso, é o operador $set e a ação, o que será atualizado no documento.

Antes de atualizarmos os documentos, vamos fazer uma busca na coleção para obtermos informações sobre a série "Grimm". Ela foi atualizada, saiu mais uma temporada, e precisamos atualizá-la. Para isso, utilizaremos o método .find e passaremos o filtro que é "Série": "Grimm".

```
db.series.find({"Série": "Grimm"})
```

A linha de comando, por padrão, vem conectada ao banco test. Já usamos o comando use Alura_Series e nos conectamos ao nosso banco de dados Alura_Series, portanto, basta colar o código e executá-lo. Como

retorno, receberemos as informações sobre a série Grimm: ano de lançamento, 2012; Temporadas disponíveis, 5; Linguagem, inglês; e o gênero, que é drama, ação e aventura.

```
test> use Alura_Series
switched to db Alura_Series
Alura_Series> db.series.find({"Série": "Grimm"})
[
 {
  _id: ObjectId("62f19a1daf663aad19e5e751"),
  'Série': 'Grimm',
  'Ano de lançamento': 2012,
  'Temporadas disponíveis': 5,
   Linguagem: 'Inglês',
   Genero: ['Drama', 'Ação', 'Aventura']
 }
]
```

A série tem mais uma temporada disponível, logo, é necessário atualizar essa informação no banco de dados. Para isso, utilizaremos o método updateOne(). Informaremos a coleção e o método, db.series.updateone(). Dentro dos parênteses, abriremos chaves e, na primeira parte do comando, especificaremos o filtro, que será "Série": "Grimm".

Na segunda parte, passaremos, entre chaves, o operador $set, abrimos chaves outra vez e passamos o campo e o valor, "Temporadas disponíveis": 6. Esse será o novo valor do campo Temporadas disponíveis na série.

```
db.series.updateOne({"Série": "Grimm"},{$set: {"Temporadas disponíveis":
6}})
```

Vamos executar o código. Temos um matchedCount: 1, portanto, conseguimos fazer a modificação. Se dermos find(), isto é, db.series.find({"Série": "Grimm"}), o número de temporadas disponíveis agora é 6.

Também poderíamos utilizar os métodos updateOne() e updateMany() na criação de campos que não existem nos documentos. Avaliando a série Grimm, notamos que ela não tem o campo de classificação. Podemos usar o método updateOne() para criá-lo. Então, vamos executar o seguinte comando:

```
db.series.updateOne({"Série": "Grimm"},{$set: {"Classificação": "16+"}})
```

Mais uma vez, precisamos informar db.series.updateOne(), passando a condição, isto é, o filtro, que será "Série": "Grimm". Na segunda parte, informamos o operador de atualização, já que estamos atualizando o documento, ao invés de informarmos o campo que já existe, informaremos o campo que desejamos criar. Neste caso, o campo Classificação, que terá o valor 16+.

Mais uma vez, tivemos correspondência e a informação foi modificada. Dano .find() novamente, visualizaremos o novo campo criado no documento, que é de 'Classificação': '16+'.

```
test> use Alura_Series
switched to db Alura_Series
Alura_Series> db.series.find({"Série": "Grimm"})
[
 {
  _id: ObjectId("62f19a1daf663aad19e5e751"),
  'Série': 'Grimm',
  'Ano de lançamento': 2012,
  'Temporadas disponíveis': 5,
   Linguagem: 'Inglês',
   Genero: ['Drama', 'Ação', 'Aventura']
     'Classificação':'16+'
 }
]
```

Também podemos atulizar mais de um documento ao mesmo tempo utilizando o método updateMany(). Antes, buscaremos informações sobre duas séries que existem na nossa coleção e que precisam ser atualizadas. Para isso, faremos db.series.find() e espeficaremos onde será o campo filtro. No caso, será em Série.

Nós utilizaremos o operador de comparação $in, já que queremos trazer a informação de mais de uma série ao mesmo tempo. Entre colchetes, passaremos a descrição das séries: Four More Shots Please e Fleabag.

```
db.series.find({"Série": {$in: ["Four More Shots Please", "Fleabag"]}})
```

O resultado são as informações das duas séries. Nenhuma delas possui o campo de classificação, assim como a série Grimm. Podemos inserir o campo nas duas séries ao mesmo tempo, já que as duas possuem o mesmo valor de classificação. Para isso, vamos informar db.series, que é a coleção, seguido do método updateMany().

Na primeira parte do comando, vamos especificar qual será o campo onde faremos o filtro, neste caso, o campo Série. Utilizaremos mais uma vez o operador $in, porque queremos que as duas séries sejam atualizadas. Novamente, passaremos a descrição dessas duas séries.

Também passaremos de novo o operador de atualização $set e informaremos o campo que desejamos criar e seu valor, "Classificação": "18+". As duas séries têm o mesmo valor de classificação.

```
db.series.updateMany({"Série":{$in:["Four More Shots Please", "Fleabag"]}},
{$set: {"Classificação": "18+"}})
```

Agora dois documentos corresponderam, matchedCount: 2, portanto, dois valores foram atualizados. Executando o .find(), teremos tanto a série "Fleabag", como a "Four More Shots Please" com o campo de classificação.

Além dos métodos UpdateOne e o UpdateMany, o MongoDB também disponibiliza o método ReplaceOne, que é utilizado para substituir um único documento na coleção com base no filtro.

O ReplaceOne possui a seguinte estrutura:

```
db.collection.replaceOne(
   <filter>,
   <replacement> )
```

Filter: especificamos a condição para a atualização.

Replacement: especificamos o documento que será substituído.

Excluindo dados

O MongoDB disponibiliza botões para atualização dos dados e um deles serve para a remoção dos documentos. Quando acessamos o MongoDB e passamos o mouse pelas coleções, na parte superior direita de cada uma delas, encontramos um botão para editar, outro para clonar, outro para copiar e um último para deletar documentos. Seu símbolo é uma lixeira.

Ao apertarmos esse botão, nosso documento fica com uma caixa de seleção vermelha. Na parte inferior do documento selecionado, há um botão de cancelar a ação, "Cancel", e outro para confirmar a remoção, "Delete".

Para remover documentos por linha de comando, o MongoDB disponibiliza dois métodos, o deleteOne, que remove um documento da coleção por ver e o deleteMany, que remove vários documentos ao mesmo tempo. Eles são bastante semelhantes aos métodos que utilizamos anteriormente, como o update.

A estrutura do delete é ainda mais simples do que a dos outros métodos.

```
db.users.deleteMany(     ← collection
   { status: "reject" }  ← delete filter
)
```

Precisamos especificar a coleção e informar o filtro. Precisamos de muito cuidado ao usar o delete. Se não passarmos o filtro, isto é, se informarmos o delete em branco, todos os documentos da coleção serão excluídos do banco de dados, portanto, perderemos todos os dados.

Já conhecemos a estrutura do código e agora retornaremos à linha de comando, onde removeremos algumas séries que não fazem mais parte do catálogo da Alura_Series. Começaremos confirmando as informações da série com find() e buscaremos por uma série específica.

```
Alura_Series> db.series.find({"Série": "The Boys"})
```

Fazendo isso, receberemos, como retorno, as informações da série The Boys, mas, ela vai sair do catálogo, portanto, precisamos removê-la do banco de dados. Para isso, utilizaremos o método deleteOne().

```
db.series.deleteOne({"Série": "The Boys"})
```

```
{ acknowleged: true, deleteCount: 1 }
```

O código foi executado com sucesso e um documento foi removido da base de dados. Se buscarmos outra vez a série The Boys, não encontraremos, pois ela não existe mais.

Além de removermos um documento por vez, podemos remover vários que correspondam ao filtro. Por exemplo, é possível remover da base de dados da Alura Séries todas as séries com apenas uma temporada disponível. Para isso, podemos executar o comando db.series.deleteMany(). Dentro dos parênteses, abriremos chaves e passaremos o filtro "Temporadas disponíveis": 1.

```
db.series.deleteMany({"Temporadas disponíveis": 1})
```

```
{ acknowleged: true, deletedCount: 29 }
```

Foram deletadas da base de dados da Alura Séries 29 documentos, ou seja, todas as séries com apenas uma temporada foram removidas. Para conferir, basta passar o find({}) vazio. Agora, de fato, não temos nenhuma série com números de temporadas disponíveis igual a 1.

Se passarmos o deleteMany() vazio, removeremos todos os documentos da coleção.

```
db.series.deleteMany({})
```