

**Objetivos:**

- I. Conceitos básicos sobre o MongoDB;
- II. Instalação do MongoDB e MongoDB Shell;
- III. Acesso ao MongoDB Shell;
- IV. Coleções no MongoDB;
- V. Inserir documentos na coleção;
- VI. Ler documentos da coleção;
- VII. Operadores;
- VIII. Atualizar documentos na coleção;
- IX. Excluir documentos da coleção;
- X. Operadores de agregação.

**i. Conceitos básicos sobre o MongoDB**

O MongoDB é um BD, mas não é um Sistema de Gerenciamento de Banco de Dados Relacional (SGBD-R) tradicional. Ele pertence à categoria de BD NoSQL (Not Only SQL). A principal diferença entre BD NoSQL, como o MongoDB, e BD relacionais é a forma como eles armazenam e organizam os dados:

- Modelo de dados NoSQL: o MongoDB utiliza um modelo de dados NoSQL baseado em **documentos**, onde os dados são armazenados em documentos BSON (Binary JSON - formato binário JSON-like);
- Esquema dinâmico: ao contrário dos BD relacionais, o MongoDB permite esquemas dinâmicos, o que significa que os documentos em uma coleção podem ter campos diferentes sem a necessidade de um esquema fixo, como nas tabelas de um SGBD-R;
- Flexibilidade: a estrutura flexível de documentos do MongoDB permite uma modelagem de dados dinâmica e adaptável, facilitando a evolução do esquema de dados ao longo do tempo, ou seja, podemos adicionar novos campos em parte dos documentos da coleção sem a necessidade de mudar a estrutura de toda a coleção. Em oposição, num SGBD-R teríamos de mudar a estrutura da tabela para acomodar um novo campo em alguns registros (linhas);
- Consultas baseadas em documentos: as consultas no MongoDB são feitas utilizando a sintaxe de consulta de documentos BSON, o que facilita a interação com os dados. No SGBD-R são feitas usando a linguagem SQL.

No MongoDB, os dados são organizados hierarquicamente da seguinte forma:

- BD:
  - É a unidade mais alta de armazenamento de dados, análogo a um BD no SGBD-R;
  - No MongoDB podem existir vários bancos e eles são independentes entre si;
  - Um BD pode conter várias coleções.
- Coleção (collection):
  - Uma coleção é análoga a uma tabela no SGBD-R;
  - Uma coleção é um grupo de documentos;

- As tabelas de um SGBD-R definem um esquema fixo para todos os registros, mas as coleções não impõem um esquema fixo aos documentos.
- Documento (document):
  - Um documento é uma unidade básica de dados no MongoDB e é representado em formato BSON (Binary JSON -formato binário JSON-like);
  - Um documento é análogo a uma linha (registro) de uma tabela do SGBD-R;
  - Enquanto todos os registros de uma tabela no SGBD-R compartilham o mesmo esquema (estrutura de colunas), os documentos de uma coleção não compartilham o mesmo esquema (propriedades do JSON). Desta forma, não se tem um esquema fixo;
  - Um documento no MongoDB é composto por campos e valores. Cada campo contém um valor associado, e estes campos podem ser de diferentes tipos de dados, como strings, inteiros, arrays, objetos, entre outros.

## ii. Instalação do MongoDB e MongoDB Shell

Para instalar o MongoDB sugere-se fazer o download da versão Community (gratuita) <https://www.mongodb.com/try/download/community>. O vídeo <https://www.youtube.com/watch?v=l4HeaNRi8f8> pode ajudar na instalação.

O MongoDB Compass é uma interface gráfica de usuário (GUI), instalada juntamente com o MongoDB, que facilita as tarefas administrativas, tais como, visualizar dados, criar bancos e collections (coleções) e gerenciar permissões de usuários.

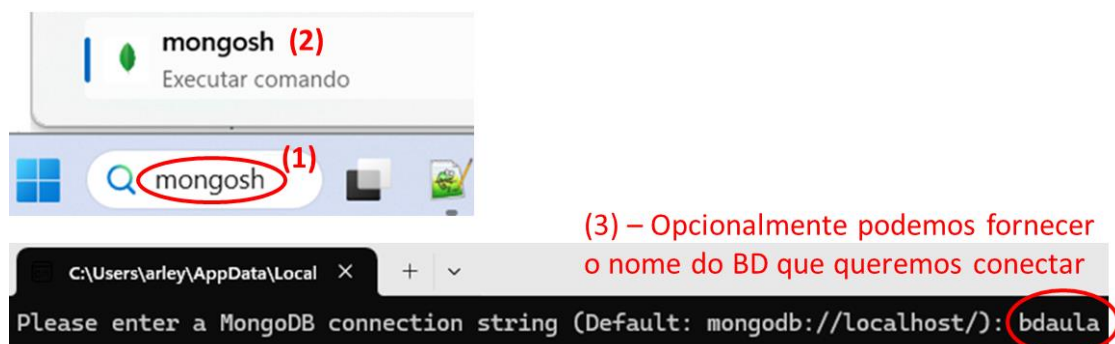
Sugere-se instalar também o MongoDB Shell. Ele é uma interface de linha de comando (CLI) interativa que permite aos usuários interagir diretamente com o MongoDB sem a necessidade de uma aplicação intermediária. Ele fornece um ambiente de shell para executar comandos e consultas no MongoDB.

Para instalar o MongoDB Shell no Windows, faça o download do [package MSI](https://www.mongodb.com/try/download/shell) disponível em <https://www.mongodb.com/try/download/shell>.

## iii. Acesso ao MongoDB Shell

O comando **mongosh** é usado para abrir um shell (terminal) do MongoDB:

- Digite **mongosh** na barra de pesquisas do Windows (1) para abrir o shell de conexão com o MongoDB. Neste shell podemos indicar o BD que queremos conectar (3), neste exemplo, fornecemos o **bdaula**, ou deixar em branco para conectarmos ao BD **test**. O BD **test** existe por padrão no MongoDB. Ele é usado frequentemente para testes e exemplos, mas não é comumente utilizado em aplicações de produção.



- O comando **use**, seguido pelo nome do BD (4), é usado para especificar o BD que queremos submeter os nossos comandos. O comando **use** criará o BD se ele não existir. No exemplo a seguir, o **bdaula** foi criado usando o comando **use bdaula**, ou seja, o **bdaula** não existia antes;

```

mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSe
lectionTimeoutMS=2000&appName=mongosh+2.2.3
Please enter a MongoDB connection string (Default: mongodb://localhost/):

Current Mongosh Log ID: 661af5c3a79d3014ea16c9b4
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSe
lectionTimeoutMS=2000&appName=mongosh+2.2.3
Using MongoDB:      7.0.4
Using Mongosh:      2.2.3

For mongosh info see: https://docs.mongodb.com/mongosh-shell/

-----
The server generated these startup warnings when booting
2024-04-13T11:08:15.186-03:00: Access control is not enabled for the database.
Read and write access to data and configuration is unrestricted
-----
(4) – O comando use para especificar o BD que queremos conectar
test> use bdaula
switched to db bdaula
bdaula> show databases (5) – O comando show databases lista os BD no MongoDB
admin                48.00 KiB
bdmeteorologico      16.98 MiB
config                84.00 KiB
local                120.00 KiB
bdaula>

```

- O comando **show databases** lista os BDs no MongoDB. Observe na figura anterior que o **bdaula** não foi listado. É importante notar que o BD só será efetivamente criado quando for inserido o primeiro documento em alguma coleção desse BD. O comando **use** não cria fisicamente o BD até que um documento seja inserido;
- Podemos acessar o MongoDB Shell também a partir do prompt de comando do Windows (CMD), digite **mongosh** ou **mongosh** seguido pelo nome do BD para estabelecer a conexão com o BD **test** ou BD fornecido.

No contexto do MongoDB Shell, **db** é um objeto global que representa o BD em uso. No exemplo a seguir, quando usamos **db** estamos nos referindo ao **bdaula**.

```

bdaula> db
bdaula

```

#### iv. Coleções no MongoDB

Para criar uma coleção no MongoDB, precisamos apenas inserir um documento nessa coleção usando o método **insertOne()** ou **insertMany()**. A coleção especificada será criada se ela não existir.

Observação: nos exemplos a seguir usaremos a coleção **users** com os campos **\_id**, **nome** e **idade** e **genero**, assim como está representada. O campo **\_id** é um identificador único que é automaticamente atribuído a cada documento em uma coleção no MongoDB. Ele serve como uma chave primária para os documentos na coleção e garante a unicidade de cada registro.

users
_id: ObjectId
nome: string
idade: number
genero: string

No comando `db.users.insertOne`:

- `db` é a variável global que possui a referência para o BD atual, ou seja, o `bdaula`;
- `users` é a coleção que queremos inserir o documento. Essa coleção pertence ao `bdaula`, ou seja, o BD selecionado. É importante notar que a coleção `users` será criada, se ela não existir no `bdaula`;
- `insertOne` é o método chamado a partir da coleção `users`, ou seja, o documento será inserido na coleção `users`. O método `insertOne` recebe um objeto JSON contendo os campos do documento:
  - `nome: "Maria"` é um campo do documento com a chave `name` e o valor `"Maria"`;
  - `idade: 21` é outro campo do documento com a chave `idade` e o valor `21`;
  - `genero: "F"` é um campo do documento com a chave `genero` e o valor `"F"`.

```
db.users.insertOne({  
  nome: "Maria",  
  idade: 21,  
  genero: "F"  
})
```

```
bdaula> db.users.insertOne({  
...   nome: "Maria",  
...   idade: 21,  
...   genero: "F",  
... })  
{  
  acknowledged: true,  
  insertedId: ObjectId('661c30ca51aa770f9e16c9d5')  
}
```

O campo `_id` é um campo especial em um documento MongoDB que serve como identificador único para esse documento dentro de uma coleção. Este campo é automaticamente gerado pelo MongoDB para cada novo documento que é inserido em uma coleção, a menos que um `_id` específico seja fornecido pelo usuário durante a inserção.

O tipo `ObjectId` é o tipo padrão usado pelo MongoDB para `_id` e é um identificador único de 12 bytes, geralmente representados em 24 dígitos hexadecimais. O valor do campo `_id` será gerado automaticamente pelo MongoDB se não fornecermos esse campo ao inserir um novo documento.

O campo `_id` é automaticamente indexado pelo MongoDB para otimizar as operações de busca e garantir a sua unicidade.

Assim como é mostrado ao lado, é importante notar que o BD foi efetivamente criado após termos inserido o primeiro documento em uma coleção do `bdaula`.

```
bdaula> show databases  
admin                40.00 KiB  
bdaula                8.00 KiB  
bdmeteorologico      16.98 MiB  
config               60.00 KiB  
local                120.00 KiB
```

Os comandos `show tables` e `show collections` são usados para listar as coleções do BD atual.

```
bdaula> show tables  
users  
bdaula> show collections  
users
```

O método `drop` é usado para remover a coleção do BD (<https://www.mongodb.com/docs/manual/reference/method/db.collection.drop>).

```
bdaula> db.users.drop()  
true  
bdaula> show collections
```

O comando `db.dropDatabase()` remove o BD em uso. No exemplo ao lado foi removido o `bdaula`;

```
bdaula> db.dropDatabase()
{ ok: 1, dropped: 'bdaula' }

bdaula> show databases
admin          40.00 KiB
bdmeteorologico 16.98 MiB
config         96.00 KiB
local          128.00 KiB
```

#### v. Inserir documentos na coleção

Assim como destacado no item anterior, as operações de criação (create) ou inserção (insert) adicionam novos documentos a uma coleção. Se a coleção ainda não existir, as operações de inserção criarão a coleção.

O MongoDB provê os seguintes métodos `insertOne()` e `insertMany()` para inserir documentos em uma coleção.

- `insertOne()` recebe um objeto JSON com os campos e valores do documento a ser inserido;
- `insertMany()` recebe um array onde cada elemento é um objeto JSON com os campos e valores do documento a ser inserido.

```
db.users.insertOne({
  nome: "Maria",
  idade: 21,
  genero: "F"
})
```

```
bdaula> db.users.insertOne({
...   nome: "Maria",
...   idade: 21,
...   genero: "F",
... })
{
  acknowledged: true,
  insertedId: ObjectId('661c30ca51aa770f9e16c9d5')
```

```
db.users.insertMany([
  {
    nome: "Pedro",
    idade: 25,
    genero: "M"
  },
  {
    nome: "Ana",
    idade: 20,
    genero: "F"
  }
])
```

```
bdaula> db.users.insertMany([
...   {
...     nome: "Pedro",
...     idade: 25,
...     genero: "M"
...   },
...   {
...     nome: "Ana",
...     idade: 20,
...     genero: "F"
...   }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('661c330451aa770f9e16c9d6'),
    '1': ObjectId('661c330451aa770f9e16c9d7')
  }
}
```

#### vi. Ler documentos da coleção

Para reproduzir os exemplos recomenda-se copiar os comandos do arquivo `exemplo.txt` e colar no shell do MongoDB Shell. Serão inseridos 11 documentos na coleção `users`.

As operações de leitura recuperam documentos de uma coleção, ou seja, consultam uma coleção em busca de documentos. O MongoDB fornece os seguintes métodos para ler documentos de uma coleção:

- `find(query, projection, options)` seleciona documentos em uma coleção ou view e retorna um cursor para os documentos selecionados. O método `find` recebe os seguintes parâmetros:
  - `query` (opcional): é um objeto que especifica os critérios de filtragem para a consulta. Se nenhum critério de seleção for fornecido, o método `find()` retornará todos os documentos da coleção;
  - `projection` (opcional): é um objeto que especifica quais campos devem ser incluídos ou excluídos no resultado da consulta;
  - `options` (opcional): é um objeto que especifica opções adicionais para controlar o comportamento da consulta. Entre outras opções temos: **sort** (ordenação), **limit** (número de documentos no resultado) e **skip** (salta um número especificado de documentos antes de começar a retornar os resultados).
- `findOne(query, projection, options)` seleciona documentos em uma coleção ou view e retorna um cursor para os documentos selecionados. Se vários documentos satisfazem a query, o método `findOne` retorna o 1º documento.

**Consulta básica:** este comando retornará um array com todos os documentos da coleção `users`:

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

O método `find` não recebeu parâmetros, desta forma, o parâmetro `query` ficou vazio. Se fosse em um SGBD relacional o método `find` estaria executando o comando SQL:

```
select * from users
```

```
bdaula> db.users.find()
[
  {
    _id: ObjectId('661ffd7ea1007c13a616c9b7'),
    nome: 'Pedro',
    idade: 25,
    genero: 'M'
  },
  {
    _id: ObjectId('661ffd7ea1007c13a616c9b8'),
    nome: 'Ana',
    idade: 20,
    genero: 'F'
  },
]
```

**Consulta com critérios de seleção:** este comando retornará um array com todos os documentos da coleção `users` onde o campo `genero` é igual a `F` e o campo `idade` é igual a `20`:

```
db.users.find({
  genero:"F",
  idade:20
})
```

O método `find` recebe um objeto JSON com os critérios de filtragem. Se fosse em um SGBD relacional o método `find` estaria executando o comando SQL:

```
select *
from users
where genero = "F" and idade = 20
```

```
bdaula> db.users.find({
...   genero:"F",
...   idade:20
... })
[
  {
    _id: ObjectId('661ffd7ea1007c13a616c9b8'),
    nome: 'Ana',
    idade: 20,
    genero: 'F'
  },
  {
    _id: ObjectId('661ffd7ea1007c13a616c9bf'),
    nome: 'Irene',
    idade: 20,
    genero: 'F'
  }
]
```

**Consulta com projeção de campos:** este comando retornará um array com todos os documentos da coleção `users`, mas mostrará somente o campo `nome`:

```
db.users.find(
  {},
  { nome:true, _id:false }
)
```

O método `find` recebe um objeto JSON com os critérios de filtragem. Se fosse em um SGBD relacional o método `find` estaria executando o comando SQL:

```
select nome
from users;
```

Observe que os documentos do resultado não terão os campos se fornecermos na projeção um campo que não existe em nenhum documento da coleção.

```
bdaula> db.users.find(
...   {},
...   { nome:true, _id:false }
... )
[
  { nome: 'Pedro' },
  { nome: 'Ana' },
  { nome: 'Maria' },
  { nome: 'Lucas' },
  { nome: 'João' },
  { nome: 'Renata' },
  { nome: 'Paulo' },
  { nome: 'Bruna' },
  { nome: 'Irene' },
  { nome: 'Roberto' },
  { nome: 'Yuri' }
]
```

```
bdaula> db.users.find(
...   {},
...   { teste:true, _id:false }
... )
[
  {}, {}, {}, {}, {},
  {}, {}, {}, {}, {},
  {}
]
```

Na projeção, os valores `true` e `false` indicam, respectivamente, que o campo deverá ser incluído ou não incluído na resposta. O campo `_id` (chave primária) é obrigatório na resposta, por isso, ele estará nos documentos da resposta mesmo se não colocarmos ele na projeção.

```
db.users.find(
  {},
  { nome:true }
```

)

**Consulta com opções:** este comando retornará um array com documentos da coleção `users` ordenados pelo campo `idade`, ignorando os 2 primeiros documentos e mostrando 4 documentos:

```
db.users.find(
  {},
  {
    nome:true,
    idade:true,
    _id:false
  },
  {
    sort:{ idade:1 },
    skip: 2,
    limit: 4
  })
```



```
bdau\> db.users.find(
... {},
... {
...   nome:true,
...   idade:true,
...   _id:false
... },
... {
...   sort:{ idade:1 },
...   skip: 2,
...   limit: 4
... })
[
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'João', idade: 22 }
]
```

Se fosse em um SGBD relacional o exemplo estaria executando o seguinte comando SQL:

```
select nome, idade
from users
order by idade asc
limit 4
offset 2;
```

As opções possuem o seguinte significado na consulta:

- **sort:** especifica a ordenação dos documentos retornados. Pode fornecer critérios de ordenação para classificar os documentos com base em um ou mais campos. O valor deve ser um objeto onde as chaves são os campos a serem ordenados e os valores indicam a direção da ordenação (1 para ascendente e -1 para descendente).;
- **limit:** limita o número de documentos retornados pela consulta ao valor especificado. Útil para paginar resultados ou limitar o volume de dados retornados;
- **skip:** pula um número especificado de documentos antes de começar a retornar os resultados. Útil em combinação com limit para implementar paginação.

Para mais opções consulte <https://www.mongodb.com/docs/manual/reference/method/db.collection.find>

Na prática, o método `find` retorna um cursor que aponta para os documentos do resultado. Podemos aplicar os métodos `sort`, `skip` e `limit` no cursor e ter os mesmos resultados das propriedades `sort`, `skip` e `limit` passadas como opções na consulta. Ao lado tem-se um exemplo de uso dos métodos atachados ao cursor.

```
db.users.find(
  {},
  {
    nome:true,
    idade:true,
    _id:false
  })
.sort({idade:1}).skip(2).limit(4)
```



Os exemplos a seguir fazem uso do método `findOne`. A diferença básica entre os métodos `find` e `findOne`, é que o último retorna apenas o 1º documento selecionado pelos critérios da consulta.

**Consulta básica:** este comando retorna o 1º documento da coleção:

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

Se fosse em um SGBD relacional o exemplo seria equivalente ao comando SQL:

```
select * from users limit 1;
```

```
bdaula> db.users.findOne()
{
  _id: ObjectId('661ffd7ea1007c13a616c9b7'),
  nome: 'Pedro',
  idade: 25,
  genero: 'M'
}
```

**Consulta com critérios de seleção e projeção de campos:** este comando retorna o 1º documento da coleção que satisfaz os critérios de consulta:

```
db.users.findOne({
  genero: "F",
  idade: 20
},
{ nome: true, _id: false })
```

```
bdaula> db.users.findOne({
...   genero: "F",
...   idade: 20
... },
... { nome: true, _id: false })
{ nome: 'Ana' }
```

Se fosse em um SGBD relacional o exemplo seria equivalente ao comando SQL:

```
select nome
from users
where genero = "F" and idade = 20
limit 1;
```

**Consulta com opções:** este comando retorna o 1º documento da coleção que satisfaz os critérios de consulta e as opções `sort` e `skip`:

```
db.users.findOne(
{},
{
  nome: true,
  idade: true,
  _id: false
},
{
  sort: { nome: 1 },
  skip: 2,
})
```

```
bdaula> db.users.findOne(
... {},
... {
...   nome: true,
...   idade: true,
...   _id: false
... },
... {
...   sort: { nome: 1 },
...   skip: 2,
... })
{ nome: 'Irene', idade: 20 }
```

O método `find` pode ser usado no MongoDB Compass. Siga os passos a seguir para reproduzir o comando ao lado no Compass:

- 1) Certifique-se de estar na coleção `users`;
- 2) Use a aba `Documents` para visualizar os documentos da coleção. O número `11` indica que a coleção `users` possui `11` documentos;
- 3) Expanda a `Options` para ter acesso aos demais parâmetros `projection` e `options` do método `find`;
- 4) Parâmetro `query` do método `find`;
- 5) Parâmetro `projection` do método `find`;
- 6) Propriedade `sort` do parâmetro `options` do método `find`;
- 7) Propriedade `skip` do parâmetro `options` do método `find`;
- 8) Propriedade `limit` do parâmetro `options` do método `find`;
- 9) Clique para submeter o comando ao MongoDB;
- 10) Documentos retornados pelo comando.

```
db.users.find(
  { idade: { $lt: 25 } },
  { nome: 1, idade: 1, _id: 0 },
  {
    sort: { idade: 1 },
    limit: 1,
    skip: 2
  }
)
```

The screenshot shows the MongoDB Compass interface with the following elements labeled with red numbers:

- (1) The `users` collection selected in the left sidebar.
- (2) The `Documents` tab selected at the top.
- (3) The `Options` dropdown menu expanded on the right.
- (4) The query field containing `{idade:{$lt:25}}`.
- (5) The projection field containing `{nome:1,idade:1,_id:0}`.
- (6) The sort field containing `{idade:1}`.
- (7) The skip field containing the value `2`.
- (8) The limit field containing the value `1`.
- (9) The `Find` button.
- (10) The resulting document: `{ "nome": "Maria", "idade": 21 }`.

## vii. Operadores

Os operadores de comparação são usados para realizar consultas mais específicas e filtrar documentos com base em critérios de comparação. Aqui estão alguns dos operadores de comparação mais usados no parâmetro `query` dos métodos `find` e `findOne`.

### Operadores de igualdade:

- `$eq`: comparador “igual a”. No comando a seguir serão exibidos os documentos que possuem o valor `20` no campo `idade` e o valor `F` no campo `genero`. Observe que o operador `$eq` fica em um objeto `{ $eq: 20 }`.

```
db.users.find(
  {
    idade: {$eq:20},
    genero: {$eq:"F"}
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```

```
bdaula> db.users.find(
...   {
...     idade: {$eq:20},
...     genero: {$eq:"F"}
...   },
...   {nome:true, idade:true, _id:false}
... )
[ { nome: 'Ana', idade: 20 }, { nome: 'Irene', idade: 20 } ]
```

Este comando é equivalente ao comando SQL:

```
select nome, idade
from users
where idade = 20 and genero = "F";
```

- **\$ne**: comparador “diferente de”. No comando a seguir serão exibidos os documentos que **não** possuem o valor 20 no campo **idade** e que possuem o valor **F** no campo **genero**.

```
db.users.find(
  {
    idade: {$ne:20},
    genero: {$eq:"F"}
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```

Este comando é equivalente ao comando SQL:

```
select nome, idade
from users
where idade != 20 and genero = "F";
```

```
bdaula> db.users.find(
...   {
...     idade: {$ne:20},
...     genero: {$eq:"F"}
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Maria', idade: 21 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Bruna', idade: 27 }
]
```

### Operadores de comparação:

- **\$gt**, **\$gte**, **\$lt**, **\$lte**: comparadores “maior que”, “maior ou igual a”, “menor que” e “menor ou igual a”, respectivamente. No comando a seguir serão exibidos os documentos que possuem o valor do campo **idade** no intervalo [22,24].

```
db.users.find(
  {
    idade: {$gte:22, $lte:24}
  },
  {
    nome:true,

```

```
bdaula> db.users.find(
...   {
...     idade: {$gte:22, $lte:24}
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 }
]
```

```

        idade:true,
        _id:false
    }
)

```

Este comando é equivalente ao comando SQL:

```

select nome, idade
from users
where idade >= 22 and idade <= 24;

```

### Operadores de intervalo:

- **\$in**: corresponde a qualquer um dos valores especificados em um array. No exemplo a seguir serão retornados os documentos que possuem o campo **nome** igual a um dos valores fornecidos no array.

```

db.users.find(
{
    nome:{$in:["Maria","João","Bruna"]}
},
{
    nome:true,
    idade:true,
    _id:false
}
)

```

```

bdaula> db.users.find(
...   {
...     nome:{$in:["Maria","João","Bruna"]}
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Maria', idade: 21 },
  { nome: 'João', idade: 22 },
  { nome: 'Bruna', idade: 27 }
]

```

Este comando é equivalente ao comando SQL:

```

select nome, idade
from users
where nome in ('Maria', 'João', 'Bruna');

```

- **\$nin**: não corresponde a nenhum dos valores especificados em um array. No exemplo a seguir serão retornados os documentos que **não** possuem o campo **nome** igual a um dos valores fornecidos no array.

```

db.users.find(
{
    nome:{$nin:["Maria","João","Bruna"]}
},
{
    nome:true,
    idade:true,
    _id:false
}
)

```

```

bdaula> db.users.find(
...   {
...     nome:{$nin:["Maria","João","Bruna"]}
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]

```

)

Este comando é equivalente ao comando SQL:

```
select nome, idade
from users
where nome not in ('Maria', 'João', 'Bruna');
```

### Operador de expressão regular:

- **\$regex**: seleciona os documentos cujos valores correspondem a expressão regular especificada. No exemplo a seguir serão retornados os documentos que possuem a letra **o** em qualquer parte do campo **nome**. O **i** indica que a expressão é insensível ao case da letra, ou seja, não diferencia maiúsculos de minúsculos. Observe que não tem aspas na expressão regular e a expressão é delimitada por barras.

```
db.users.find(
  {
    nome: { $regex: /o/i }
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```

```
bdaula> db.users.find(
...  {
...    nome: { $regex: /o/i }
...  },
...  {
...    nome:true,
...    idade:true,
...    _id:false
...  }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'João', idade: 22 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Roberto', idade: 21 }
]
```

### Operadores lógicos:

- **\$or**: permite buscar documentos que atendam a pelo menos uma das condições especificadas. Observe que o operador **\$or** recebe um array de JSON (colchetes vermelhos), onde cada objeto possui uma condição de teste (chaves azuis).

```
db.users.find(
  {
    $or: [
      { idade: { $lt: 22 } },
      { idade: { $gt: 26 } }
    ]
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```


```
bdaula> db.users.find(
...  {
...    $or: [
...      { idade: { $lt: 22 } },
...      { idade: { $gt: 26 } }
...    ]
...  },
...  {
...    nome:true,
...    idade:true,
...    _id:false
...  }
... )
[
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 }
]
```

Este comando é equivalente ao comando SQL:

```
select nome, idade
from users
where idade < 22 or idade > 26;
```

- **\$and**: permite buscar documentos que atendam a ambas as condições especificadas. Observe que o operador **\$and** recebe um array de JSON (colchetes vermelhos), onde cada objeto possui uma condição de teste (chaves azuis).

```
db.users.find(
  {
    $and: [
      { idade: { $gt: 22 } },
      { idade: { $lt: 26 } }
    ]
  },
  {
    nome: true,
    idade: true,
    _id: false
  }
)
```



```
bdau1a> db.users.find(
...   {
...     $and: [
...       { idade: { $gt: 22 } },
...       { idade: { $lt: 26 } }
...     ]
...   },
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 }
]
```

Este comando é equivalente ao comando SQL:

```
select nome, idade
from users
where idade > 22 and idade < 26;
```

No exemplo a seguir os operadores **\$or** e **\$and** foram combinados para reproduzir este comando SQL:

```
select nome, idade
from users
where (idade >= 21 and idade <= 23) or idade > 28;
```

```
db.users.find(
  {
    $or: [
      {
        $and: [
          { idade: { $gte: 21 } },
          { idade: { $lte: 23 } }
        ]
      },
      { idade: { $gt: 28 } }
    ]
  },
  {
    nome: true,
    idade: true,
    _id: false
  }
)
```

```
bdaula> db.users.find(
... {
...   $or: [
...     {
...       $and: [
...         { idade: { $gte: 21 } },
...         { idade: { $lte: 23 } }
...       ]
...     },
...     { idade: { $gt: 28 } }
...   ]
... },
... {
...   nome: true,
...   idade: true,
...   _id: false
... }
... )
[
  { nome: 'Maria', idade: 21 },
  { nome: 'João', idade: 22 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Roberto', idade: 21 }
]
```

- **\$nor**: une as cláusulas da consulta com um NOR lógico e retorna todos os documentos que não correspondem a ambas as cláusulas. No exemplo a seguir serão selecionados os documentos que possuem idades acima de 22 e abaixo de 26.

```
db.users.find(
  {
    $nor: [
      { idade: { $lte: 22 } },
      { idade: { $gte: 26 } }
    ]
  },
  {
    nome: true,
    idade: true,
    _id: false
  }
)
```

```
bdaula> db.users.find(
... {
...   $nor: [
...     { idade: { $lte: 22 } },
...     { idade: { $gte: 26 } }
...   ]
... },
... {
...   nome: true,
...   idade: true,
...   _id: false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Yuri' }
]
```

- **\$not**: inverte o efeito da expressão. No exemplo a seguir serão selecionados os documentos que possuem idades abaixo de 23.

```
db.users.find(
  {
    idade: {
      $not:{$gte:23}
    }
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```

```
bdaula> db.users.find(
... {
...   idade: {
...     $not:{$gte:23}
...   }
... },
... {
...   nome:true,
...   idade:true,
...   _id:false
... }
... )
[
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'João', idade: 22 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

#### Operador de existência:

- **\$exists**: seleciona os documentos que possuem o campo especificado. No exemplo a seguir serão selecionados os documentos que possuem o campo **idade**. Veja que o documento que possui o nome Yuri não possui o campo **idade**.

```
db.users.find(
  {
    idade:{ $exists: true }
  },
  {
    nome:true,
    idade:true,
    _id:false
  }
)
```

```
bdaula> db.users.find(
... {
...   idade:{ $exists: true }
... },
... {
...   nome:true,
...   idade:true,
...   _id:false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 }
]
```

#### Operador de tipo:



- **\$type**: seleciona o documento se o campo possui o tipo especificado. No exemplo a seguir serão selecionados os documentos que possuem o campo **idade** do tipo **number**. Veja que o documento que possui o nome Yuri não possui o campo **idade**.

```
db.users.find(
  {
    idade: { $type: "number" }
  },
  {
    nome: true,
    idade: true,
    _id: false
  }
)
```

```
bdaula> db.users.find(
...   {
...     idade: { $type: "number" }
...   },
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 }
]
```

Para mais detalhes sobre operadores <https://www.mongodb.com/docs/manual/reference/operator/query>

### viii. Atualizar documentos na coleção

A atualização de documentos pode ser feita através dos seguintes métodos, entre outros:

- **updateOne**(filter, update, options?): atualiza somente o 1º documento que corresponde ao filtro especificado;
- **updateMany**(filter, update, options?): atualiza todos os documentos que correspondem ao filtro especificado;
- **replaceOne**(filter, replacement, options?): substitui um único documento que corresponde ao filtro especificado pelo documento de substituição (replacement).

Recomenda-se executar os comandos do arquivo [exemplo.txt](#) antes de cada exemplo de update, para termos uma coleção com os dados originais.

**Operadores de atualização** (<https://www.mongodb.com/docs/manual/reference/operator/update/#update-operators-1>):

- **\$set**: é usado para atualizar os valores de um ou mais campos em um documento sem substituir todo o documento e pode ser utilizado também para criar um campo se ele não existir;
- **\$inc**: incrementa o valor do campo por um valor específico. Na prática faz uma soma no valor do campo;
- **\$mul**: multiplica o valor do campo por um valor específico. Na prática multiplica o valor do campo por um valor;
- **\$rename**: é usado para renomear um campo em um documento existente;
- **\$unset**: é usado para excluir um campo em um documento existente.

**Exemplo de `updateOne`:** atualiza a idade para 30 do 1º documento que possui `idade` maior que 25. O operador `$set` é usado para atualizar o campo `idade` para 30.

```
db.users.updateOne(
  {
    idade: { $gt: 25 }
  },
  {
    $set: { idade: 30 }
  }
)
```

```
bdaula> db.users.updateOne(
...   {
...     idade: { $gt: 25 }
...   },
...   {
...     $set: { idade: 30 }
...   }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Antes da atualização:

```
bdaula> db.users.find(
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após a atualização:

```
bdaula> db.users.find(
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 30 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

**Exemplo de `updateMany`:** subtrai 10 na idade de todos os documentos que possuem `idade` menor que 25. O operador `$inc` é usado para atualizar o campo `idade` usando como base o seu valor atual, ou seja, a nova `idade` será 10 anos menor que a `idade` atual.

```
db.users.updateMany(
  {
    idade: { $lt: 25 }
  },
  {
    $inc: { idade: -10 }
  }
)
```

```
bdaula> db.users.updateMany(
...   {
...     idade: { $lt: 25 }
...   },
...   {
...     $inc: { idade: -10 }
...   }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 7,
  modifiedCount: 7,
  upsertedCount: 0
}
```

Antes da atualização:

```
bdaula> db.users.find(
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após a atualização:

```
bdaula> db.users.find(
...   {},
...   {
...     nome: true,
...     idade: true,
...     _id: false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 10 },
  { nome: 'Maria', idade: 11 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 12 },
  { nome: 'Renata', idade: 14 },
  { nome: 'Paulo', idade: 13 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 10 },
  { nome: 'Roberto', idade: 11 },
  { nome: 'Yuri' }
]
```

**Exemplo de `replaceOne`:** substitui o documento que possui o nome João pelo documento fornecido.

```
db.users.replaceOne(
  {
    nome: "João"
  },
  {
    nome: "José",
    idade: 44,
    genero: "M"
  }
)
```

```
bdaula> db.users.replaceOne(
...   {
...     nome: "João"
...   },
...   {
...     nome: "José",
...     idade: 44,
...     genero: "M"
...   }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Antes da atualização:

```
bdaula> db.users.find(
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após a atualização:

```
bdaula> db.users.find(
...   {},
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'José', idade: 44 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

**Exemplo de \$rename:** renomeia o campo `idade` para `age` dos documentos que satisfazem a query.

```
db.users.updateMany(
  {
    idade: { $lt: 25 }
  },
  {
    $rename: { "idade": "age" }
  }
)
```

```
bdaula> db.users.updateMany(
... {
...   idade: { $lt: 25 }
... },
... {
...   $rename: { "idade": "age" }
... }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 7,
  modifiedCount: 7,
  upsertedCount: 0
}
```

Antes da atualização:

```
bdaula> db.users.find(
... {
... },
... {
...   nome: true,
...   idade: true,
...   _id: false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após a atualização:

```
bdaula> db.users.find(
... {},
... {
...   nome: true,
...   idade: true,
...   age: true,
...   _id: false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', age: 20 },
  { nome: 'Maria', age: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', age: 22 },
  { nome: 'Renata', age: 24 },
  { nome: 'Paulo', age: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', age: 20 },
  { nome: 'Roberto', age: 21 },
  { nome: 'Yuri' }
]
```

**Exemplo de \$set:** o operador `$set` pode ser utilizado também para criar um campo se ele não existir. No exemplo a seguir foi adicionado o campo `peso` em todos os documentos da coleção.

```
db.users.updateMany(
  {},
  {
    $set: { peso: 70 }
  }
)
```

```
bdaula> db.users.updateMany(
... {},
... {
...   $set: { peso: 70 }
... }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 11,
  modifiedCount: 11,
  upsertedCount: 0
}
```

Antes da atualização:

```
bdaula> db.users.find(
... {
... },
... {
...   nome:true,
...   idade:true,
...   _id:false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após a atualização:

```
bdaula> db.users.find(
... {},
... {
...   nome:true,
...   idade:true,
...   peso:true,
...   _id:false
... }
... )
[
  { nome: 'Pedro', idade: 25, peso: 70 },
  { nome: 'Ana', idade: 20, peso: 70 },
  { nome: 'Maria', idade: 21, peso: 70 },
  { nome: 'Lucas', idade: 28, peso: 70 },
  { nome: 'João', idade: 22, peso: 70 },
  { nome: 'Renata', idade: 24, peso: 70 },
  { nome: 'Paulo', idade: 23, peso: 70 },
  { nome: 'Bruna', idade: 27, peso: 70 },
  { nome: 'Irene', idade: 20, peso: 70 },
  { nome: 'Roberto', idade: 21, peso: 70 },
  { nome: 'Yuri', peso: 70 }
]
```

**Exemplo de \$unset:** o operador `$unset` será usado para remover o campo `idade` dos documentos que possuem `idade` acima de 25. O valor associado ao campo que desejamos remover deve ser uma string vazia `""`.

O motivo de usar uma string vazia `""` no `$unset` é uma convenção do MongoDB.

```
db.users.updateMany(
  {
    idade: { $gt: 25 }
  },
  {
    $unset: { "idade": "" }
  }
)
```

```
bdaula> db.users.updateMany(
... {
...   idade: { $gt: 25 }
... },
... {
...   $unset: { "idade": "" }
... }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
```

Antes da atualização:

Após a atualização:

```
bdaula> db.users.find(
... {
... },
... {
...   nome:true,
...   idade:true,
...   _id:false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

```
bdaula> db.users.find(
... {},
... {
...   nome:true,
...   idade:true,
...   _id:false
... }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas' },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna' },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

#### ix. Excluir documentos da coleção

Os métodos a seguir são usados para remover documentos de uma coleção:

- `deleteOne(filter, options?)`: remove somente o 1º documento que corresponde ao filtro especificado;
- `deleteMany(filter, options?)`: remove todos os documentos que correspondem ao filtro especificado.

Recomenda-se executar os comandos do arquivo [exemplo.txt](#) antes de cada exemplo de update, para termos uma coleção com os dados originais.

**Exemplo de `deleteOne`:** remove o 1º documento que possui `idade` acima de 25. Veja que será excluído o documento que possui o nome Lucas.

```
db.users.deleteOne(
  {
    idade: { $gt: 25 }
  }
)
```

Antes da atualização:

```
bdaula> db.users.deleteOne(
... {
...   idade: { $gt: 25 }
... }
... )
{ acknowledged: true, deletedCount: 1 }
```

Após remover:



```
bdaula> db.users.find(
...   {
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Exemplo de **deleteMany**: remove todos os documentos que possuem **idade** abaixo de 25.

```
db.users.deleteMany(
  {
    idade: { $lt: 25 }
  }
)
```

```
bdaula> db.users.deleteMany(
...   {
...     idade: { $lt: 25 }
...   }
... )
{ acknowledged: true, deletedCount: 7 }
```

Antes da atualização:

```
bdaula> db.users.find(
...   {
...   },
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Ana', idade: 20 },
  { nome: 'Maria', idade: 21 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'João', idade: 22 },
  { nome: 'Renata', idade: 24 },
  { nome: 'Paulo', idade: 23 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Irene', idade: 20 },
  { nome: 'Roberto', idade: 21 },
  { nome: 'Yuri' }
]
```

Após remover:

```
bdaula> db.users.find(
...   {},
...   {
...     nome:true,
...     idade:true,
...     _id:false
...   }
... )
[
  { nome: 'Pedro', idade: 25 },
  { nome: 'Lucas', idade: 28 },
  { nome: 'Bruna', idade: 27 },
  { nome: 'Yuri' }
]
```



## x. Operadores de agregação

O método `aggregate()` é usado para executar operações de agregação em documentos de uma coleção.

O `aggregate()` aceita um array de estágios de um pipeline que processam e transformam os documentos conforme especificado, produzindo um conjunto de resultados. Cada etapa do pipeline realiza uma operação específica nos documentos de entrada e passa os resultados para a próxima etapa – é necessário manter a sequência fornecida a seguir no pipeline.

Os estágios mais comuns em um pipeline de agregação incluem:

1. `$match`: filtra os documentos;
2. `$group`: agrupa os documentos com base em uma chave de agrupamento e realiza operações de agregação nos documentos agrupados. Semelhante ao termo *group by* nos comandos SQL;
3. `$sort`: ordena os documentos. Semelhante ao termo *order by* nos comandos SQL;
4. `$project`: indica os campos dos documentos de saída;
5. `$limit`: limita o número de documentos de saída. Semelhante ao termo *limit* nos comandos SQL;
6. `$skip`: pula um número específico de documentos. Semelhante ao termo *offset* nos comandos SQL;

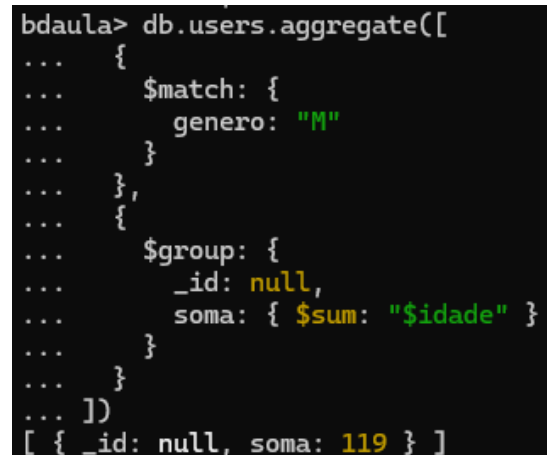
O resultado é um conjunto de documentos transformados conforme os estágios do pipeline.

O método `aggregate()` é especialmente útil para realizar cálculos de agregação complexos, como calcular médias (`$avg`), mínimos (`$min`), máximos (`$max`), somas/contar (`$sum`) e outros tipos de análises de dados. Ele oferece uma flexibilidade significativa para manipular e transformar dados diretamente no BD.

**Exemplo de `$match` e `$group`:** o método `aggregate` recebe um array com os estágios `match` e `group`, nesta ordem, pois a ordem dos estágios influencia na resposta. O comando soma a `idade` de todos os documentos. Significado dos termos:

- `$match`: filtra os documentos que possuem o campo `genero` igual a `M`;
- `$group`: agrupa os documentos seguindo as operações de agregação:
  - `_id: null` é obrigatório. Indica o campo que queremos agrupar. `null` significa que estamos agrupando todos os documentos em um único conjunto;
  - `"$idade"` é o campo que estamos usando para calcular o somatório.

```
db.users.aggregate([
  {
    $match: {
      genero: "M"
    }
  },
  {
    $group: {
      _id: null,
      soma: { $sum: "$idade" }
    }
  }
])
```



```
bdau1a> db.users.aggregate([
...   {
...     $match: {
...       genero: "M"
...     }
...   },
...   {
...     $group: {
...       _id: null,
...       soma: { $sum: "$idade" }
...     }
...   }
... ])
[ { _id: null, soma: 119 } ]
```

É necessário seguir a ordem dos estágios no pipeline de

1)

agregação. O comando a seguir não tem retorno pelo fato de não existir **genero** na resposta do agrupamento.

```
bdau1a> db.users.aggregate([
...   {
...     $group: {
...       _id: null,
...       soma: { $sum: "$idade" }
...     }
...   },
...   {
...     $match: {
...       genero: "M"
...     }
...   }
... ])
```

No exemplo anterior o estágio **\$match** precisa vir antes do estágio **\$group**, pelo fato do filtro **genero = "M"** ser aplicado antes da operação de agrupamento (group). Porém, a ordem precisa ser invertida quando o estágio **\$match** faz uma filtragem usando o resultado de um agrupamento. No exemplo a seguir, o estágio **\$match** é aplicado em dois momentos:

- Antes do estágio **\$group**: ele faz o papel de where (no comando SQL), ou seja, o filtro não pode usar campos calculados no estágio **\$group**;
- Após o estágio **\$group**: ele faz o papel de having (no comando SQL), ou seja, o filtro só pode usar campos calculados no estágio **\$group**. No exemplo a seguir o filtro **quantidade = 4** faz uso do campo **quantidade** que é calculado no estágio **\$group**.

```
db.users.aggregate([
{
  $match: {
    idade: { $lt: 25 }
  }
},
{
  $group: {
    _id: "$genero",
    quantidade: { $sum: 1 }
  }
},
{
  $match: {
    quantidade: { $eq: 4 }
  }
},
{
  $project: {
    sexo: "$_id",
    quantidade: 1,
  }
}]
```

```
bdau1a> db.users.aggregate([
...   {
...     $match: {
...       idade: { $lt: 25 }
...     }
...   },
...   {
...     $group: {
...       _id: "$genero",
...       quantidade: { $sum: 1 }
...     }
...   },
...   {
...     $match: {
...       quantidade: { $eq: 4 }
...     }
...   },
...   {
...     $project: {
...       sexo: "$_id",
...       quantidade: 1,
...       _id: 0
...     }
...   }
... ])
[ { quantidade: 4, sexo: 'F' } ]
```

```

    _id: 0
  }
}
1)

```

O comando anterior é equivalente ao seguinte comando SQL. Observe que o estágio `$match` faz o papel de `where` (`$match` antes do agrupamento) e `having` (`$match` após o agrupamento) no comando SQL:

```

select sexo, quantidade
from users
where idade < 25
group by genero
having quantidade = 5;

```

**Exemplo de `$group` e `$sort`:** o método `aggregate` recebe um array com os estágios `group` e `sort`, nesta ordem, para agrupar os documentos pelo campo `genero` e depois calcular a média no campo `idade`. O resultado terá dois documentos, visto que a coleção possui documentos com os gêneros `F` e `M`. Significado dos termos:

- `$match`: filtra os documentos que possuem o campo `genero` igual a `M`;
- `$group`: agrupa os documentos seguindo as operações de agregação:
  - `_id: "$genero"` significa que estamos agrupando todos os documentos pelo valor do campo `genero`;
  - `"$idade"` é o campo que estamos usando para calcular a média.

```

db.users.aggregate([
  {
    $group: {
      _id: "$genero",
      media: { $avg: "$idade" }
    }
  },
  {
    $sort: {
      media: -1
    }
  }
])

```

```

bdaula> db.users.aggregate([
...   {
...     $group: {
...       _id: "$genero",
...       media: { $avg: "$idade" }
...     }
...   },
...   {
...     $sort: {
...       media: -1
...     }
...   }
... ])
[ { _id: 'M', media: 23.8 }, { _id: 'F', media: 22.4 } ]

```

**Exemplo de `$project`:** o exemplo a seguir usamos o estágio `$project` para renomear o campo `_id` para `sexo` no documento de saída. O estágio `$project` é adicionado após o estágio `$sort`. Significado dos termos:

- `sexo: "$_id"`: renomeamos o campo `_id` para `sexo`;
- `soma: 1`: o valor `1` ou `true` indica que mantemos o campo `soma` no documento de saída;
- `_id: 0`: o valor `0` ou `false` indica que excluímos o campo `_id` no documento de saída.

```
db.users.aggregate([
  {
    $group: {
      _id: "$genero",
      soma: { $sum: "$idade" }
    }
  },
  {
    $sort: {
      soma: -1
    }
  },
  {
    $project: {
      sexo: "$_id",
      soma: 1,
      _id: 0
    }
  }
])
```

```
bdaula> db.users.aggregate([
...   {
...     $group: {
...       _id: "$genero",
...       soma: { $sum: "$idade" }
...     }
...   },
...   {
...     $sort: {
...       soma: -1
...     }
...   },
...   {
...     $project: {
...       sexo: "$_id",
...       soma: 1,
...       _id: 0
...     }
...   }
... ])
[ { soma: 119, sexo: 'M' }, { soma: 112, sexo: 'F' } ]
```

## Exercícios

Execute os comandos do arquivo [exercicio.txt](#) para criar a coleção **products** e inserir os documentos.

```
bdaula> db.products.countDocuments()
12
```

**Exercício 1** – Fazer um comando para listar os campos **nome** e **valor** de todos os documentos da coleção **products**. Apresente o resultado ordenado pelo campo **nome**.

Dicas:

- Use o método `find(query, projection, options);`
- Deixe o objeto `query` vazio (sem propriedades) para não termos filtros na consulta e assim listar todos os documentos da coleção;
- Forneça os campos **nome** e **valor** no objeto `projection` e retire o campo obrigatório `_id`;
- Forneça a propriedade `sort` no objeto `options`.

```
[
  { nome: 'Arroz', valor: 22.8 },
  { nome: 'Açúcar', valor: 19.8 },
  { nome: 'Bolacha', valor: 3.9 },
  { nome: 'Detergente', valor: 5.9 },
  { nome: 'Farinha de trigo', valor: 3.2 },
  { nome: 'Feijão', valor: 7.5 },
  { nome: 'Macarrão', valor: 4.9 },
  { nome: 'Pasta de dente', valor: 8.5 },
  { nome: 'Sabonete', valor: 2.9 },
  { nome: 'Sabão', valor: 2.5 },
  { nome: 'Sal', valor: 3.1 },
  { nome: 'Óleo de soja', valor: 7.5 }
]
```

**Exercício 2** – Fazer um comando para listar os campos `nome` e `valor` dos documentos da coleção `products` que são da categoria `"Higiene pessoal"`. Apresente o resultado ordenado pelo campo `nome` e convertido para maiúsculo.

```
[
  { nome: 'PASTA DE DENTE', valor: 8.5 },
  { nome: 'SABONETE', valor: 2.9 }
]
```

Dicas:

- Use o método `find(query, projection, options)`;
- Coloque no objeto `query` a propriedade `categoria` com valor `"Higiene pessoal"`;
- Forneça os campos `nome` e `valor` no objeto `projection` e retire o campo obrigatório `_id`. Será necessário substituir `nome:true` por `nome:{$toUpper: "$nome"}` para projetar o campo `nome` com as letras maiúsculas;
- Forneça a propriedade `sort` no objeto `options`.

Detalhes do operador `toUpper` <https://www.mongodb.com/docs/manual/reference/operator/aggregation/toUpper>

**Exercício 3** – Fazer um comando para listar os campos `nome` e `quantidade` dos documentos da coleção `products` que possuem `quantidade > 30`. Apresente o resultado em ordem decrescente de `quantidade`, altere o nome do campo `quantidade` para `unidades` e limite o resultado para mostrar somente os 2 primeiros documentos.

```
[ { nome: 'Açúcar', unidades: 54 }, { nome: 'Sabão', unidades: 45 } ]
```

Dicas:

- Use o método `find(query, projection, options)`;
- Coloque no objeto `query` a propriedade `quantidade` com o operador de comparação `$gt`;
- Forneça os campos `nome` e `quantidade` no objeto `projection` e retire o campo obrigatório `_id`. Será necessário substituir `quantidade:true` por `unidades: "$quantidade"` para renomear o campo na projeção;
- Forneça as propriedades `sort` (-1 para ordem decrescente) e `limit` no objeto `options`.

**Exercício 4** – Fazer um comando para listar os campos `nome` e `quantidade` dos documentos da coleção `products`

```
[ { nome: 'Macarrão', quantidade: 42 } ]
```

que possuem `quantidade > 30` e são da categoria `"Alimentação"`. Apresente o resultado em ordem decrescente de `quantidade` e mostre somente o 2º documento.

Dicas:

- Use o método `find(query, projection, options)`;
- Coloque no objeto `query` a propriedade `quantidade` com o operador de comparação `$gt` e a propriedade `categoria` com valor `"Alimentação"`;
- Forneça os campos `nome` e `quantidade` no objeto `projection` e retire o campo obrigatório `_id`;
- Forneça as propriedades `sort` (-1 para ordem decrescente) e `limit` no objeto `options`.

**Exercício 5** – Fazer um comando para listar os campos `nome`, `valor` e `quantidade` dos documentos da coleção `products` que possuem `quantidade > 40` ou `valor < 4`. Apresente o resultado ordenado pelo campo `nome`.

```
[
  { nome: 'Açúcar', valor: 19.8, quantidade: 54 },
  { nome: 'Bolacha', valor: 3.9, quantidade: 18 },
  { nome: 'Farinha de trigo', valor: 3.2, quantidade: 11 },
  { nome: 'Macarrão', valor: 4.9, quantidade: 42 },
  { nome: 'Sabonete', valor: 2.9, quantidade: 34 },
  { nome: 'Sabão', valor: 2.5, quantidade: 45 },
  { nome: 'Sal', valor: 3.1, quantidade: 21 }
]
```

Dicas:

- Use o método `find(query, projection, options)`;
- Coloque no objeto `query` o operador lógico `$or`. Esse operador receberá um array com dois objetos, um para comparar o campo `quantidade` e outro campo para comparar o campo `valor`;
- Forneça os campos `nome`, `quantidade` e `valor` no objeto `projection` e retire o campo obrigatório `_id`;
- Forneça a propriedade `sort` (1 para ordem crescente) no objeto `options`.

**Exercício 6** – Fazer um comando para listar os documentos cuja a `quantidade` é menor que o `minimo`. Apresente o resultado ordenado pelo `nome` e mostre (projete) somente os campos `nome`, `quantidade` e `minimo`.

```
[
  { nome: 'Arroz', quantidade: 12, minimo: 15 },
  { nome: 'Farinha de trigo', quantidade: 11, minimo: 15 },
  { nome: 'Óleo de soja', quantidade: 18, minimo: 20 }
]
```

Dicas:

- Use o método `find(query, projection, options)`;
- Coloque no objeto `query` o operador de avaliação de expressão JavaScript `$where`. A expressão passada para o operador `$where` será aplicada em cada documento da coleção. Nessa expressão o documento atual pode ser referenciado usando as palavras reservadas `this` ou `obj`. Desta forma, podemos referenciar o campo `quantidade` usando `this.quantidade` ou `obj.quantidade` na expressão a ser avaliada pelo `$where` (<https://www.mongodb.com/docs/manual/reference/operator/query/where/#mongodb-query-op-where>);
- Forneça os campos `nome`, `quantidade` e `minimo` no objeto `projection` e retire o campo obrigatório `_id`;
- Forneça a propriedade `sort` no objeto `options`.

Observação:

- O comando a seguir não funciona. A expressão `$lt: "$minimo"` não funciona pelo fato do operador `$lt` não poder usar a referência direta para outro campo no mesmo documento.

```
db.products.find({ quantidade: { $lt: "$minimo" } })
```

**Exercício 7** – Fazer um comando para listar a quantidade de documentos por categoria. Apresente o resultado em ordem decrescente de quantidade de documentos e campo `quantidade` de documentos renomeado para `total`.

```
[
  { total: 8, categoria: 'Alimentação' },
  { total: 2, categoria: 'Limpeza doméstica' },
  { total: 2, categoria: 'Higiene pessoal' }
]
```

Dicas:

- Use o método `aggregate([{$group}, {$sort}, {$project}])` recebendo um array com os estágios `$group`, `$sort` e `$project`. Cada estágio precisa estar em um objeto, ou seja, delimitados por chaves;
- No estágio `$group`:
  - A chave `_id` recebe o campo usado para agrupar, neste caso será a `$categoria`;
  - Use o operador de agregação `$sum:1` para contar a quantidade de documentos em cada subconjunto criado pelo agrupamento `_id`;
- No estágio `$sort`, use o campo `total` para ordenar os documentos pelo campo criado no agrupamento;
- No estágio `$project`, especifique os campos que serão projetados no resultado.

**Exercício 8** – Alterar o comando do Exercício 7 para incluir o valor médio dos produtos por categoria.

```
[
  { total: 8, media: 9.0875, categoria: 'Alimentação' },
  { total: 2, media: 4.2, categoria: 'Limpeza doméstica' },
  { total: 2, media: 5.7, categoria: 'Higiene pessoal' }
]
```

Dicas:

- Adicione o operador `$avg` no estágio `$group`. Lembre-se que o nome do campo `$valor` precisa ser precedido por `$`;
- Adicione o campo de agregação no estágio `$project`.

**Exercício 9** – Alterar o comando do exercício 7 para listar o total de unidades de produtos de cada categoria.

```
[
  { total: 210, categoria: 'Alimentação' },
  { total: 63, categoria: 'Limpeza doméstica' },
  { total: 56, categoria: 'Higiene pessoal' }
]
```

Dicas:

- No estágio `$group`, use o operador de agregação de `$sum:"$categoria"`. Neste caso, o operador de agregação `$sum` somará o valor do campo `categoria` de cada documento. No Exercício 7 a operação `$sum:1` foi usada para contar a quantidade de documentos.

Recomenda-se executar os comandos do arquivo `exercicio.txt` antes de cada exercício de update e delete, para termos uma coleção com os dados originais.

**Exercício 10** – Alterar o comando do Exercício 7 para listar somente as categorias que possuem 2 produtos.

```
[
  { total: 2, categoria: 'Higiene pessoal' },
  { total: 2, categoria: 'Limpeza doméstica' }
]
```

Dicas:

- Inclua o estágio `$match` no método `aggregate([{$group}, {$match}, {$sort}, {$project}])`;
- O estágio `$match` veio após o estágio `$group` pelo fato de precisarmos comparar usando o campo `total`, que foi calculado no estágio `$group`.

**Exercício 11** – Fazer um comando para aumentar em 10% o valor de todos os produtos da categoria Alimentação.

#### Antes da atualização

```
bdaula> db.products.find(
... {},
... {
...   nome:true,
...   valor:true,
...   _id:false
... })
[
  { nome: 'Feijão', valor: 7.5 },
  { nome: 'Arroz', valor: 22.8 },
  { nome: 'Sal', valor: 3.1 },
  { nome: 'Bolacha', valor: 3.9 },
  { nome: 'Macarrão', valor: 4.9 },
  { nome: 'Sabão', valor: 2.5 },
  { nome: 'Açúcar', valor: 19.8 },
  { nome: 'Farinha de trigo', valor: 3.2 },
  { nome: 'Óleo de soja', valor: 7.5 },
  { nome: 'Sabonete', valor: 2.9 },
  { nome: 'Pasta de dente', valor: 8.5 },
  { nome: 'Detergente', valor: 5.9 }
]
```

#### Após a atualização

```
bdaula> db.products.find(
... {},
... {
...   nome:true,
...   valor:true,
...   _id:false
... })
[
  { nome: 'Feijão', valor: 8.25 },
  { nome: 'Arroz', valor: 25.080000000000002 },
  { nome: 'Sal', valor: 3.4100000000000006 },
  { nome: 'Bolacha', valor: 4.29 },
  { nome: 'Macarrão', valor: 5.390000000000001 },
  { nome: 'Sabão', valor: 2.5 },
  { nome: 'Açúcar', valor: 21.78 },
  { nome: 'Farinha de trigo', valor: 3.5200000000000005 },
  { nome: 'Óleo de soja', valor: 8.25 },
  { nome: 'Sabonete', valor: 2.9 },
  { nome: 'Pasta de dente', valor: 8.5 },
  { nome: 'Detergente', valor: 5.9 }
]
```

#### Dicas:

- Use o método `updateMany(query, update);`
- Coloque no objeto `query` a propriedade `categoria` com o valor `"Alimentação"` para filtrar somente os documentos que possuem o campo `categoria` com o valor `"Alimentação"`;
- No objeto `update`, use o operador `$mul` para multiplicar por `1.1` o valor do campo `valor`.

**Exercício 12** – Fazer um comando para adicionar R\$2 no valor de cada produto que custa mais de R\$10.

#### Antes da atualização

```
bdaula> db.products.find(
... {},
... {
...   nome:true,
...   valor:true,
...   _id:false
... })
[
  { nome: 'Feijão', valor: 7.5 },
  { nome: 'Arroz', valor: 22.8 },
  { nome: 'Sal', valor: 3.1 },
  { nome: 'Bolacha', valor: 3.9 },
  { nome: 'Macarrão', valor: 4.9 },
  { nome: 'Sabão', valor: 2.5 },
  { nome: 'Açúcar', valor: 19.8 },
  { nome: 'Farinha de trigo', valor: 3.2 },
  { nome: 'Óleo de soja', valor: 7.5 },
  { nome: 'Sabonete', valor: 2.9 },
  { nome: 'Pasta de dente', valor: 8.5 },
  { nome: 'Detergente', valor: 5.9 }
]
```

#### Após a atualização

```
bdaula> db.products.find(
... {},
... {
...   nome:true,
...   valor:true,
...   _id:false
... })
[
  { nome: 'Feijão', valor: 7.5 },
  { nome: 'Arroz', valor: 24.8 },
  { nome: 'Sal', valor: 3.1 },
  { nome: 'Bolacha', valor: 3.9 },
  { nome: 'Macarrão', valor: 4.9 },
  { nome: 'Sabão', valor: 2.5 },
  { nome: 'Açúcar', valor: 21.8 },
  { nome: 'Farinha de trigo', valor: 3.2 },
  { nome: 'Óleo de soja', valor: 7.5 },
  { nome: 'Sabonete', valor: 2.9 },
  { nome: 'Pasta de dente', valor: 8.5 },
  { nome: 'Detergente', valor: 5.9 }
]
```

#### Dicas:

- Use o método `updateMany(query, update);`
- Coloque no objeto `query` a propriedade `valor` com o operador de comparação `$gt`;
- No objeto `update`, use o operador `$inc` para somar 2 no valor do campo `valor`.



**Exercício 13** – Fazer um comando para converter para maiúsculo o conteúdo do campo `nome`.

Dicas:

- Use o método `updateMany(query, update)`. O parâmetro `update` precisará receber um array quando usarmos o operador `$toUpper`;
- No array passado como parâmetro para o `update` precisamos definir um objeto com a propriedade `$set`;
- A propriedade `$set` precisa ter um objeto com a propriedade `nome: { $toUpper: "$nome" }`, onde `nome` é o campo a ser atualizado e `{ $toUpper: "$nome" }` significa que o valor será o campo `nome` convertido para maiúsculo.

```
bdau1a> db.products.find(
...   {},
...   {
...     nome:true,
...     _id:false
...   }
... )
[
  { nome: 'FEIJÃO' },
  { nome: 'ARROZ' },
  { nome: 'SAL' },
  { nome: 'BOLACHA' },
  { nome: 'MACARRÃO' },
  { nome: 'SABÃO' },
  { nome: 'AÇÚCAR' },
  { nome: 'FARINHA DE TRIGO' },
  { nome: 'ÓLEO DE SOJA' },
  { nome: 'SABONETE' },
  { nome: 'PASTA DE DENTE' },
  { nome: 'DETERGENTE' }
]
```

**Exercício 14** – Fazer um comando para adicionar o campo `total` nos documentos da coleção `products`. O campo `total` deverá receber o resultado da multiplicação dos campos `valor` e `quantidade`.

```
bdau1a> db.products.find(
...   {},
...   {nome:true,valor:true,quantidade:true,total:true,_id:false}
... )
[
  { nome: 'Feijão', valor: 7.5, quantidade: 34, total: 255 },
  { nome: 'Arroz', valor: 22.8, quantidade: 12, total: 273.6 },
  { nome: 'Sal', valor: 3.1, quantidade: 21, total: 65.100000000000001 },
  { nome: 'Bolacha', valor: 3.9, quantidade: 18, total: 70.2 },
  { nome: 'Macarrão', valor: 4.9, quantidade: 42, total: 205.8 },
  { nome: 'Sabão', valor: 2.5, quantidade: 45, total: 112.5 },
  { nome: 'Açúcar', valor: 19.8, quantidade: 54, total: 1069.2 },
  { nome: 'Farinha de trigo', valor: 3.2, quantidade: 11, total: 35.2 },
  { nome: 'Óleo de soja', valor: 7.5, quantidade: 18, total: 135 },
  { nome: 'Sabonete', valor: 2.9, quantidade: 34, total: 98.6 },
  { nome: 'Pasta de dente', valor: 8.5, quantidade: 22, total: 187 },
  { nome: 'Detergente', valor: 5.9, quantidade: 18, total: 106.2 }
]
```

Dicas:

- Use o método `updateMany(query, update)`. O parâmetro `update` precisará receber um array quando usarmos o operador no `$multiply`;
- No array passado como parâmetro para o `update` precisamos definir um objeto com a propriedade `$set`. O operador `$set` cria o campo se ele não existir;
- A propriedade `$set` precisa ter um objeto com a propriedade `total: { $multiply: ["$valor", "$quantidade"] }`. O operador `$multiply` multiplica os valores passados no array.

Detalhes do operador `$multiply` <https://www.mongodb.com/docs/manual/reference/operator/aggregation/multiply>

**Exercício 15** – Fazer um comando para adicionar o campo `data` nos documentos da coleção `products`. O campo `data` deverá receber `ISODate("2024-01-02")`.

```
bdau1a> db.products.find(
...   {},
...   {nome:true,data:true,_id:false}
... )
[
  { nome: 'Feijão', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Arroz', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Sal', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Bolacha', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Macarrão', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Sabão' },
  { nome: 'Açúcar', data: ISODate('2024-01-02T00:00:00.000Z') },
  {
    nome: 'Farinha de trigo',
    data: ISODate('2024-01-02T00:00:00.000Z')
  },
  { nome: 'Óleo de soja', data: ISODate('2024-01-02T00:00:00.000Z') },
  { nome: 'Sabonete' },
  { nome: 'Pasta de dente' },
  { nome: 'Detergente' }
]
```

Dicas:

- Use o método `updateMany(query, update)`;
- Use o operador `$set` para criar o campo `data`;
- Observação: o parâmetro `update` não precisará receber um array, já que não estamos usando um operador não suportado pelo método `updateMany`. A lista de operadores suportados pelo `update` está disponível em <https://www.mongodb.com/docs/manual/reference/operator/update>.