Objetivos:

I. Modelagem de dados no MongoDB;
II. Modelo de dados incorporados;
III. Modelo de dados relacionados;
IV. CRUD de dados incorporados;
V. CRUD de dados relacionados.

i. Modelagem de dados no MongoDB

Modelagem de dados refere-se à organização dos dados dentro de um BD e às relações entre entidades relacionadas. Os dados no MongoDB possuem um modelo de esquema flexível, o que significa:

- Documentos dentro de uma única coleção não são obrigados a terem o mesmo conjunto de campos;
- O tipo de dados de um campo pode diferir entre documentos dentro de uma coleção;
- Geralmente, os documentos em uma coleção compartilham uma estrutura semelhante. Para garantir consistência no modelo de dados podemos criar regras de validação de esquema.

Casos de uso de um modelo de dados flexível

O modelo de dados flexível nos permite organizar os dados para atender às necessidades de uma aplicação. O MongoDB é um BD orientado a documentos, o que significa que podemos incorporar dados relacionados em campos de objeto e array.

Um esquema flexível é útil nos seguintes cenários:

- Considere uma empresa que rastreia em qual departamento cada funcionário trabalha. Podemos incorporar informações do departamento dentro da coleção de funcionários para retornar informações relevantes em uma única consulta;
- Um aplicativo de comércio eletrônico mostra as cinco avaliações mais recentes ao exibir um produto. Podemos armazenar as avaliações recentes na mesma coleção dos dados do produto e armazenar avaliações mais antigas em uma coleção separada, porque as avaliações mais antigas não são acessadas com tanta frequência;
- Uma loja de roupas precisa criar um aplicativo para um catálogo de produtos. Diferentes produtos têm diferentes atributos e, portanto, usam campos de documento diferentes. No entanto, podemos armazenar todos os produtos na mesma coleção.

Design de esquema: diferenças entre BD Relacionais e de documentos

Ao projetar um esquema para um BD orientado a documentos como o MongoDB, existem algumas diferenças importantes em relação aos BDs relacionais que devem ser consideradas.

Comportamento de BD Relacional	Comportamento de BD de Documento
Devemos determinar o esquema de uma tabela antes de inserir os dados.	O esquema pode mudar ao longo do tempo conforme as necessidades da aplicação se alteram.
Frequentemente, é necessário unir dados de várias tabelas para retornar os dados necessários	O modelo de dados flexível permite armazenar dados de forma a corresponder à maneira como a aplicação necessita dos dados e evitar joins. Evitar joins entre várias coleções melhora o desempenho e

pela aplicação.	reduz a carga de trabalho de implementação.

Vinculação de dados relacionados

Ao projetar o modelo de dados no MongoDB, considere a estrutura dos documentos e as formas como a aplicação utilizará dados de entidades relacionadas. Para vincular dados relacionados, podemos optar por:

- Modelo de dados incorporados;
- Modelo de dados relacionados.

Todo o texto anterior foi retirado de https://www.mongodb.com/docs/manual/data-modeling.

ii. Modelo de dados incorporados

O termo "incorporado" refere-se à inclusão de dados dentro de outros dados, como subdocumentos em um documento principal. Esse modelo é considerado desnormalizado, enquanto o "modelo de dados relacionados" é considerado normalizado.

Documentos incorporados armazenam dados relacionados em uma única estrutura de documento, onde o documento principal possui campos contendo subdocumentos ou array de subdocumentos. No exemplo a seguir os telefones do usuário estão em subdocumentos embutidos no array do campo `telefones`:

Comando para criar a coleção `users`:

```
db.users.drop()
db.users.insertMany([
  {
    nome: "Pedro",
    idade: 25,
    genero: "M",
    telefones: [
      {
        numero: "12988776655",
        operadora: "TIM"
      },
      {
        numero: "12991234455",
        operadora: "Vivo"
      }
    ]
  },
  {
    nome: "Ana",
    idade: 20,
    genero: "F",
    telefones: [
      {
        numero: "12911223344",
        operadora: "Claro"
      },
```

Comando para listar os documentos da coleção `users`:

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

```
        {
          numero: "12991234455",
          operadora: "Vivo"
        }
      ]
    }
])
```

Vantagens do modelo de dados incorporados:

- Desempenho: recuperar dados incorporados geralmente requer menos operações de leitura e menos junções, o que pode resultar em tempos de resposta mais rápidos e comandos mais simples;
- Consistência: os dados relacionados são armazenados juntos em um único documento, garantindo a consistência dos dados. No exemplo anterior os telefones estão em um campo do documento, desta forma, se excluirmos o documento que possui o nome Pedro todos os seus telefones serão excluídos, ou seja, não corre o risco de deixarmos telefones sem "dono" (referência para um usuário);
- Facilidade de leitura: é mais fácil e mais rápido recuperar todos os dados relacionados de um documento único. No exemplo anterior basta o comando `db.users.find()` sem filtros ou junções para recuperar todos os documentos e subdocumentos da coleção;
- Escalabilidade: é simples adicionar novos campos com subdocumentos.

Desvantagens do modelo de dados incorporados:

- Duplicação de dados: os dados podem ser duplicados em vários documentos, o que pode levar a inconsistências se os dados não forem atualizados corretamente. No exemplo anterior o documente sinalizado em amarelo repete em dois documentos, desta forma, para atualizarmos a `operadora` de `Vivo` para `Tim` teríamos de fazer um update em dois documentos;
- Limite de tamanho do documento: o tamanho máximo de um documento no MongoDB, nesse momento o limite é de 16 MB, o que pode ser uma limitação se os subdocumentos crescerem significativamente (https://www.mongodb.com/docs/manual/reference/limits/#bson-documents);
- Complexidade de atualização: atualizar dados em subdocumentos pode ser mais complexo e demorado.

Para muitos casos de uso no MongoDB, o modelo de dados desnormalizado é o mais adequado. Para entender um pouco mais sobre as vantagens e desvantagens de incorporar documentos, consulte (https://www.mongodb.com/docs/manual/data-modeling/concepts/embedding-vs-references).

iii. Modelo de dados relacionados

O termo "relacionado" descreve a estrutura em que os dados em uma coleção fazem referência aos dados em outra coleção, estabelecendo assim uma relação entre elas.

Para estabelecer uma relação entre documentos de diferentes coleções, armazenamos o `_id` de um documento de uma coleção em um campo de referência no documento de outra coleção. Isso cria uma ligação entre os documentos, permitindo que possamos recuperar informações relacionadas por meio de consultas.

No exemplo a seguir a relação foi estabelecida através dos `_id` armazenados no campo `telefones`. Desta forma, a coleção `users` possui as referências para os documentos da coleção `phones`.

Esse modelo é considerado normalizado, enquanto o modelo de dados incorporado é considerado desnormalizado.

```
bdaula> db.phones.find()
[
  {
    _id: ObjectId('6625c42d37788b2c7416c9bc'),
    numero: '12988776655',
    operadora: 'TIM'
  },
  {
    _id: ObjectId('6625c42d37788b2c7416c9bd'),
    numero: '12991234455',
    operadora: 'Vivo'
  }
]
```

```
bdaula> db.users.find()
[
  {
    _id: ObjectId('6625c59237788b2c7416c9c0'),
    nome: 'Pedro',
    idade: 25,
    genero: 'M',
    telefones: [
      ObjectId('6625c42d37788b2c7416c9bc'),
      ObjectId('6625c42d37788b2c7416c9bd')
    ]
  }
]
```

Vantagens:

- Conservação de espaço: a referência para um documento de outra coleção reduz a duplicação de dados e o uso de espaço de armazenamento;
- Flexibilidade: facilita a atualização de informações relacionadas, pois as mudanças precisam ser feitas em um único lugar;
- Escalabilidade: pode lidar com grandes volumes de dados relacionados de forma eficiente.

Desvantagens:

- Desempenho: geralmente requer mais operações de leitura e junções, o que pode resultar em tempos de resposta mais lentos.
- Complexidade de consulta: requer múltiplas operações de leitura e possivelmente junções, tornando as consultas mais complexas e potencialmente mais lentas.
- Consistência: a necessidade de manter a consistência entre documentos relacionados pode ser desafiadora e requerer cuidados adicionais.

Para aprender um pouco mais sobre as vantagens e desvantagens de usar referências para documentos, consulte (https://www.mongodb.com/docs/manual/data-modeling/concepts/embedding-vs-references).

iv. CRUD de dados incorporados

> Recomenda-se executar os comandos a seguir antes de cada exemplo de CRUD, para termos a coleção `users` com os dados originais.

Comando para criar a coleção `users`:

Comando para listar os documentos da coleção `users`:

```
db.users.drop()
db.users.insertMany([
  {
    nome: "Pedro",
    idade: 25,
    genero: "M",
    telefones: [
      {
        numero: "12988776655",
        operadora: "TIM"
      },
      {
        numero: "12991234455",
        operadora: "Vivo"
      },
      {
        numero: "12922002299",
        operadora: "Vivo"
      }
    ]
  },
  {
    nome: "Ana",
    idade: 20,
    genero: "F",
    telefones: [
      {
        numero: "12911223344",
        operadora: "Claro"
      },
      {
        numero: "12991234455",
        operadora: "Vivo"
      }
    ]
  }
])
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991234455', operadora: 'Vivo' },
      { numero: '12922002299', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

Adicionar subdocumento

O operador `$push` adiciona elementos ao array. Como o campo `telefones` possui um array de documentos, então podemos utilizar o operador `$push` para adicionar um elemento nesse array. Veja que fizemos uma operação de update no documento que possui `nome:"Pedro"` e nessa operação de update inserimos o elemento sinalizado em verde no campo `telefones`.

```
db.users.updateOne(
  { nome: "Pedro" },
  {
    $push: {
      telefones: {
        numero: "12933228899",
        operadora: "TIM"
      }
    }
  }
)
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991234455', operadora: 'Vivo' },
      { numero: '12922002299', operadora: 'Vivo' },
      { numero: '12933228899', operadora: 'TIM' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

O operador `$push` precisa do operador `$each` para inserir mais de um elemento ao array.

O operador `$sort` ordena os elementos do array, porém ele requer o uso do operador `$each`. No exemplo a seguir serão inseridos dois documentos no campo `telefones` e os elementos do array serão ordenados pelo campo `operadora`.

```
db.users.updateOne(
  { nome: "Pedro" },
  {
    $push: {
      telefones: {
        $each: [
          {
            numero: "11955443322",
            operadora: "Claro"
          },
          {
            numero: "11911009900",
            operadora: "Vivo"
          }
        ],
        $sort: {operadora:1}
      }
    }
  }
)
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '11955443322', operadora: 'Claro' },
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991234455', operadora: 'Vivo' },
      { numero: '12922002299', operadora: 'Vivo' },
      { numero: '11911009900', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

Para mais detalhes sobre o operador $push acesse https://www.mongodb.com/docs/manual/reference/operator/update/push.

Remover subdocumento

O operador `$pull` remove elementos do array. No exemplo a seguir serão removidos todos os documentos do array `telefones` que satisfazem a condição `operadora:"Vivo`.

```
db.users.updateOne(
  { nome: "Pedro" },
  {
    $pull: {
      telefones: {
        operadora: "Vivo"
      }
    }
  }
)
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [ { numero: '12988776655', operadora: 'TIM' } ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

Para mais detalhes sobre o operador $push acesse https://www.mongodb.com/docs/manual/reference/operator/update/pull.

Atualizar subdocumento

Para atualizar um campo temos de usar o operador `$set` e para atualizar o campo de um subdocumentos temos de informar o caminho começando no campo do documento "externo" seguido pelo operador de posição `$` e o campo do subdocumento. No exemplo a seguir:

- `{"telefones.operadora":"Vivo"}`: é a parte do filtro (query). Identifica o documento cujo subdocumento `telefones` contém o campo `operadora` com o valor `"Vivo"`;
- `$set:{"telefones.$.numero":"12991230000"}`: o operador `$set` é usado para atualizar o valor do campo especificado. Neste caso, somente o campo `numero` do 1º subdocumento que corresponde à condição é atualizado para `"12991230000"`.

O operador **$** representa o identificador do 1º elemento que corresponde à condição de consulta dentro de um array, ou seja, mesmo que a subcoleção `telefones` tenha mais de um subdocumento da operadora Vivo, eles não serão atualizados. Veja que o número "12922002299" não foi atualizado.

```
db.users.updateMany(
  { "telefones.operadora": "Vivo" },
  {
    $set: {
      "telefones.$.numero": "12991230000"
    }
  }
)
```

```
bdaula> db.users.find(
...   {},
...   {
...     nome: 1,
...     telefones: 1,
...     _id: 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991230000', operadora: 'Vivo' },
      { numero: '12922002299', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991230000', operadora: 'Vivo' }
    ]
  }
]
```

Para atualizar todos os subdocumentos de um array que satisfazem a condição de filtragem, temos de utilizar um índice de posição no operador de posição. No exemplo a seguir:

- `$set`: é o operador de atualização que define como os campos dos documentos selecionados serão modificados;
- `"telefones.$[elemento].numero"`: o operador `$[elemento]` indica os subdocumentos a serem atualizados. O `elemento` é uma variável que recebe o valor fornecido pelo `arrayFilters`;
- `{arrayFilters:[{"elemento.operadora":"Vivo"}]}`: o parâmetro `arrayFilters` permite especificar condições para a seleção de elementos de array para atualização. Neste exemplo, estamos dizendo que queremos atualizar apenas os subdocumentos onde a operadora é `"Vivo"`.

```
db.users.updateMany(
  { },
  {
    $set: {
        "telefones.$[elemento].numero": "12991230000"
    }
  },
  { arrayFilters: [{"elemento.operadora": "Vivo"}] }
)
```

```
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991230000', operadora: 'Vivo' },
      { numero: '12991230000', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [
      { numero: '12911223344', operadora: 'Claro' },
      { numero: '12991230000', operadora: 'Vivo' }
    ]
  }
]
```

Para mais detalhes sobre o operador $set acesse https://www.mongodb.com/docs/manual/reference/operator/update/set.

Ler subdocumentos

Para ler campos de subdocumentos temos de usar a notação `"campoDoDocumento.campoDoSubdocumento"`. Observe que é necessário envolver por aspas quando existir o ponto para acessar subdocumento `"campo.campo"`:

```
find(query, projection, options)
```

O comando a seguir aplica o filtro `operadora: "Vivo"` em cada elemento do array `telefones`:

- O operador `$elemMatch` verifica se pelo menos um subdocumento dentro do array `telefones` tem o campo `operadora` igual a `"Vivo"`;
- A projeção `telefones.$` retornará apenas o 1º subdocumento que atende ao critério.

```
db.users.find(
  { telefones: { $elemMatch: { operadora: "Vivo" } } },
  {
    "telefones.$": 1,
    "nome": 1,
    "_id": 0
  }
)
```

Observação: a ordenação é aplicada aos documentos principais, e não aos elementos individuais dentro de arrays de subdocumentos.

```
bdaula> db.users.find(
...   { telefones: { $elemMatch: { operadora: "Vivo" } } },
...   {
...     "telefones.$": 1,
...     "nome": 1,
...     "_id": 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [ { numero: '12991234455', operadora: 'Vivo' } ]
  },
  {
    nome: 'Ana',
    telefones: [ { numero: '12991234455', operadora: 'Vivo' } ]
  }
]
```

Observação: se incluirmos o campo `operadora` na projeção. O parâmetro `query` deixará de ser aplicado e serão retornados os subdocumentos que possuem as demais operadoras.

```
bdaula> db.users.find(
...   { telefones: { $elemMatch: { operadora: "Vivo" } } },
...   {
...     "telefones.operadora": 1,
...     "nome": 1,
...     "_id": 0
...   }
... )
[
  {
    nome: 'Pedro',
    telefones: [
      { operadora: 'TIM' },
      { operadora: 'Vivo' },
      { operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [ { operadora: 'Claro' }, { operadora: 'Vivo' } ]
  }
]
```

Para obter todos os subdocumentos que satisfazem ao filtro, podemos usar a projeção condicional com o operador `$filter` no método de agregação, já que o método `find()` por si só não suporta essa funcionalidade.

Neste exemplo, o estágio `$match` filtra os documentos onde pelo menos um subdocumento em `telefones` possui `operadora` igual a `"Vivo"`. Em seguida, o estágio `$project` usa o operador `$filter` para projetar apenas os subdocumentos que satisfazem ao critério de filtro.

```
bdaula> db.users.aggregate([
...   {
...     $match: {
...       "telefones.operadora": "Vivo"
...     }
...   },
...   {
...     $project: {
...       nome: 1,
...       telefones: {
...         $filter: {
...           input: "$telefones",
...           as: "telefone",
...           cond: { $eq: ["$$telefone.operadora", "Vivo"] }
...         }
...       },
...       _id: 0
...     }
...   }
... ])
```

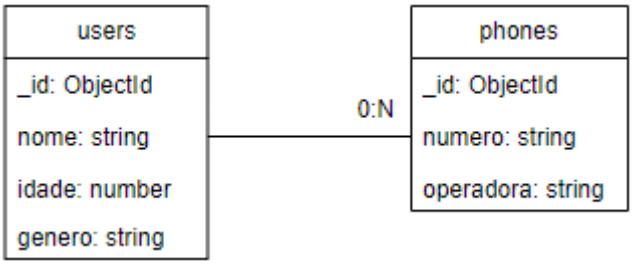
```
[
  {
    nome: 'Pedro',
    telefones: [
      { numero: '12991234455', operadora: 'Vivo' },
      { numero: '12922002299', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    telefones: [ { numero: '12991234455', operadora: 'Vivo' } ]
  }
]
```

```
db.users.aggregate([
  {
    $match: {
      "telefones.operadora": "Vivo"
    }
  },
  {
    $project: {
      nome: 1,
      telefones: {
        $filter: {
          input: "$telefones",
          as: "telefone",
          cond: { $eq: ["$$telefone.operadora", "Vivo"] }
        }
```

```
            },
            _id: 0
          }
        }
    ])
```

v. CRUD de dados relacionados

Utilize os comandos a seguir para criar as coleções `phones` e `users`. Veja que um usuário pode ter vários telefones, a relação acontece no campo `telefones` dos documentos da coleção `users`, ou seja, um documento na coleção `phones` não tem conhecimento da sua relação com algum documento da coleção `users`.



Comando para criar a coleção `phones`:

```
db.phones.drop()
db.phones.insertMany([
  {
    _id: ObjectId("662ae8e634e0be1da016c9b0"),
    numero: "12988776655",
    operadora: "TIM"
  },
  {
    _id: ObjectId("662ae8e634e0be1da016c9b1"),
    numero: "12991234455",
    operadora: "Vivo"
  },
  {
    _id: ObjectId("662ae8e634e0be1da016c9b2"),
    numero: "12922002299",
    operadora: "Vivo"
  },
  {
    _id: ObjectId("662ae8e634e0be1da016c9b3"),
    numero: "12911223344",
    operadora: "Claro"
  }
])
```

Comando para criar a coleção `users`:

```
db.users.drop()
db.users.insertMany([
  {
    nome: "Pedro",
    idade: 25,
    genero: "M",
    telefones: [
      ObjectId("662ae8e634e0be1da016c9b0"),
      ObjectId("662ae8e634e0be1da016c9b1"),
      ObjectId("662ae8e634e0be1da016c9b2")
    ]
  },
  {
    nome: "Ana",
    idade: 20,
    genero: "F",
    telefones: [
      ObjectId("662ae8e634e0be1da016c9b3"),
      ObjectId("662ae8e634e0be1da016c9b1")
    ]
  }
])
```

As operações para escrever, atualizar e remover documentos são individuais em cada coleção, ou seja, para inserir um documento na coleção `users`, antes teremos de inserir os telefones na coleção `phones` e obter os `_id` a serem colocados no array `telefones`.

Da mesma forma, se formos excluir um usuário, teremos de fazer uma operação na coleção `users` e outra na coleção `phones`.

Porém, para recuperar os documentos de duas coleções teremos de fazer um join nos documentos relacionados.

Para recuperar os documentos da coleção `phones` a partir de uma consulta na coleção `users` precisaremos usar o método `aggregate` com o estágio `$lookup`.

Para realizar uma correspondência de igualdade entre um campo dos documentos de entrada com um campo dos documentos da coleção relacionada. O estágio `$lookup` recebe um objeto com a seguinte sintaxe:

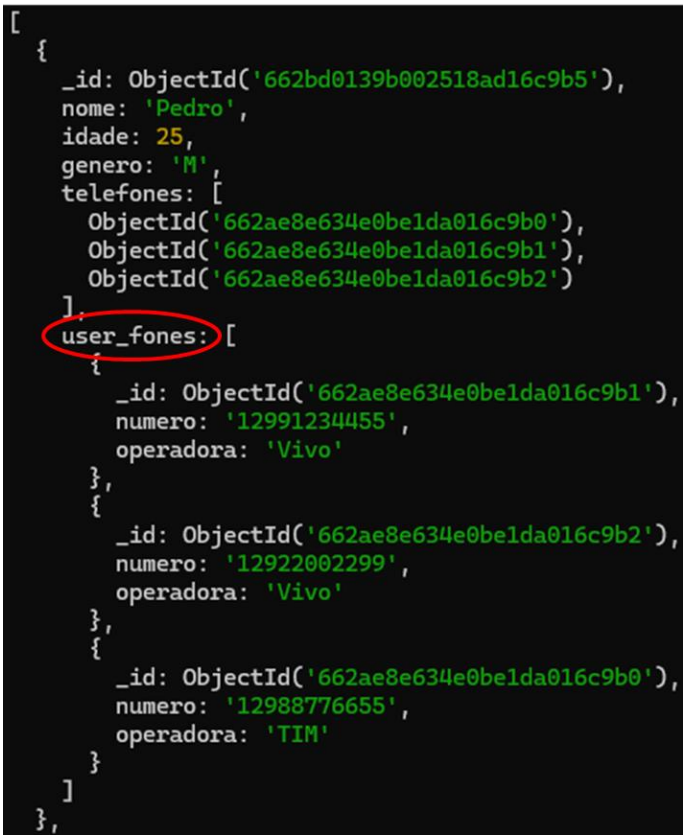
```
{
   $lookup:
     {
       from: <especifica a coleção para fazer a junção (join)>,
       localField: <campo no documento de entrada usado para fazer a junção>,
       foreignField: <campo no documento relacionado usado para fazer a junção>,
       as: <nome do campo a ser criado com o resultado do lookup>
     }
}
```

Observação: são obrigatórios todos os campos do objeto {`from`, `localField`, `foreignField`, `as`} passados para o estágio `$lookup`.

O comando a seguir usa o estágio `$lookup` para fazer a junção dos documentos da coleção `users` com os documentos da coleção `phones`. Observe que o campo `as` foi usado para criar o campo `user_phones` no resultado.

```
db.users.aggregate([
  {
    $lookup: {
      from: "phones",
      localField: "telefones",
      foreignField: "_id",
      as: "user_fones"
    }
  }
])
```

Por motivo de espaço, o print mostra apenas parte do resultado.



No estágio `$project` do pipeline do método `aggregate` definimos os campos a serem projetados no resultado. Veja que renomeamos o campo `user_fones` para `fones` na projeção.

```
db.users.aggregate([
  {
    $lookup: {
      from: "phones",
      localField: "telefones",
      foreignField: "_id",
      as: "user_fones"
    }
  },
  {
    $project: {
      nome: 1,
      fones: "$user_fones",
      _id: 0
    }
  }
])
```

Por motivo de espaço, o print mostra apenas parte do resultado.

```
[
  {
    nome: 'Pedro',
    fones: [
      {
        _id: ObjectId('662ae8e634e0be1da016c9b1'),
        numero: '12991234455',
        operadora: 'Vivo'
      },
      {
        _id: ObjectId('662ae8e634e0be1da016c9b2'),
        numero: '12922002299',
        operadora: 'Vivo'
      },
      {
        _id: ObjectId('662ae8e634e0be1da016c9b0'),
        numero: '12988776655',
        operadora: 'TIM'
      }
    ]
  },
```

Para projetar apenas os campos `numero` e `operadora` do subdocumento temos de usar a notação `"campoDoDocumento.campoDoSubdocumento"`.

```
db.users.aggregate([
  {
    $lookup: {
      from: "phones",
      localField: "telefones",
      foreignField: "_id",
      as: "user_fones"
    }
  },
  {
    $project: {
      nome: 1,
      "user_fones.numero": 1,
      "user_fones.operadora": 1,
      _id: 0
    }
  }
])
```

```
[
  {
    nome: 'Pedro',
    user_fones: [
      { numero: '12922002299', operadora: 'Vivo' },
      { numero: '12988776655', operadora: 'TIM' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  },
  {
    nome: 'Ana',
    user_fones: [
      { numero: '129112223344', operadora: 'Claro' },
      { numero: '12991234455', operadora: 'Vivo' }
    ]
  }
]
```

Para mais detalhes sobre o estágio $lookup acesse https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup.

Exercícios

Execute os comandos do arquivo exercicio-parte1.txt para criar a coleção `ufs` e inserir os documentos. Essas coleções serão usadas nos Exercício 1 a 8.

```
bdaula> db.ufs.countDocuments()
27
```

Os dados de densidade demográfica, do Censo de 2022, foram obtidos na Wikipédia (https://pt.wikipedia.org/wiki/Lista_de_unidades_federativas_do_Brasil_por_densidade_demográfica).

Exercício 1 – Fazer um comando para listar os campos `nome` do estado e `nome` da região. Apresente o resultado ordenado pelo campo `nome` do estado. A seguir tem-se parte dos 27 documentos do resultado.

```
[
  { nome: 'Acre', regiao: { nome: 'Norte' } },
  { nome: 'Alagoas', regiao: { nome: 'Nordeste' } },
  { nome: 'Amapá', regiao: { nome: 'Norte' } },
  { nome: 'Amazonas', regiao: { nome: 'Norte' } },
  { nome: 'Bahia', regiao: { nome: 'Nordeste' } },
  { nome: 'Ceará', regiao: { nome: 'Nordeste' } },
  { nome: 'Distrito Federal', regiao: { nome: 'Centro-Oeste' } },
  { nome: 'Espírito Santo', regiao: { nome: 'Sudeste' } },
  { nome: 'Goiás', regiao: { nome: 'Centro-Oeste' } },
```

Dicas:

- Use o método `find(query, projection, options);`
- Forneça os campos `nome` e `"regiao.nome"` no objeto `projection` e retire o campo obrigatório `_id`. Observe que é necessário envolver o nome dos campos por aspas quando acessamos o campo de um subdocumento;
- Forneça a propriedade `sort` no objeto `options`.

Exercício 2 – Alterar o comando do Exercício 1 para apresentar o resultado em ordem decrescente de `nome` da região e `nome` do estado. A seguir tem-se parte dos 27 documentos do resultado.

```
[
  { nome: 'Santa Catarina', regiao: { nome: 'Sul' } },
  { nome: 'Rio Grande do Sul', regiao: { nome: 'Sul' } },
  { nome: 'Paraná', regiao: { nome: 'Sul' } },
  { nome: 'São Paulo', regiao: { nome: 'Sudeste' } },
  { nome: 'Rio de Janeiro', regiao: { nome: 'Sudeste' } },
  { nome: 'Minas Gerais', regiao: { nome: 'Sudeste' } },
  { nome: 'Espírito Santo', regiao: { nome: 'Sudeste' } },
  { nome: 'Tocantins', regiao: { nome: 'Norte' } },
  { nome: 'Roraima', regiao: { nome: 'Norte' } },
  { nome: 'Rondônia', regiao: { nome: 'Norte' } },
  { nome: 'Pará', regiao: { nome: 'Norte' } },
```

Dica:

- Será necessário incluir o campo `"regiao.nome"` no objeto `options`.

Exercício 3 – Fazer um comando para contar a quantidade de estados por região. Apresente o resultado ordenado pela quantidade de estados por região.

```
[
  { total: 9, regiao: 'Nordeste' },
  { total: 7, regiao: 'Norte' },
  { total: 4, regiao: 'Sudeste' },
  { total: 4, regiao: 'Centro-Oeste' },
  { total: 3, regiao: 'Sul' }
]
```

Dicas:

- Use o método `aggregate([{$group}, {$sort}, {$project}])` com os estágios `$group`, `$sort` e `$project`. Cada estágio precisa estar em um objeto, ou seja, delimitados por chaves;
- No estágio `$group`:
  - A chave `_id` recebe o campo usado para agrupar, neste caso será `"regiao.nome"`;
  - Use o operador de agregação `$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 4 – Alterar o comando do Exercício 3 para listar somente o 1º documento do resultado.

```
[ { total: 9, regiao: 'Nordeste' } ]
```

Dica:

- Adicione o estágio `$limit` no pipeline do método `aggregate([{$group}, {$sort}, {$project}, {$limit}])`.

Exercício 5 – Alterar o comando do Exercício 3 para listar a densidade média por região. Apresente o resultado ordenado pela média.

```
[
  { media: 171.70499999999998, regiao: 'Sudeste' },
  { media: 135.005, regiao: 'Centro-Oeste' },
  { media: 62.10555555555556, regiao: 'Nordeste' },
  { media: 59.92333333333334, regiao: 'Sul' },
  { media: 5.017142857142857, regiao: 'Norte' }
]
```

Dicas:

- Use o operador `$avg` para calcular a média;
- O operador `$avg` deve receber o campo `densidade`. Lembre-se que o nome do campo precisa ser precedido por `$`.

Exercício 6 – Fazer o comando para alterar a sigla da região norte para NO.

```
[
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } },
  { regiao: { sigla: 'NO', nome: 'Norte' } }
]
```

Dicas:

- Use o método `updateMany(query, update);`

- Coloque no objeto `query` a propriedade para selecionar apenas os subdocumentos que são da região Norte;
- No objeto `update`, use o operador `$set` para alterar o campo `sigla` do subdocumento `regiao`.

Exercício 7 – Fazer o comando para adicionar o campo id com valor 5 nos documentos da região centro-oeste.

```
[
  { regiao: { sigla: 'CO', nome: 'Centro-Oeste', id: 5 } },
  { regiao: { sigla: 'CO', nome: 'Centro-Oeste', id: 5 } },
  { regiao: { sigla: 'CO', nome: 'Centro-Oeste', id: 5 } },
  { regiao: { sigla: 'CO', nome: 'Centro-Oeste', id: 5 } }
]
```

Dicas:

- Use o método `updateMany(query, update);`
- Coloque no objeto `query` a propriedade para selecionar apenas os subdocumentos que são da região Centro-Oeste;
- No objeto `update`, use o operador `$set` para criar o campo `id` do subdocumento `regiao`.

Exercício 8 – Fazer o comando para remover os campos `id` e `sigla` dos documentos da região centro-oeste.

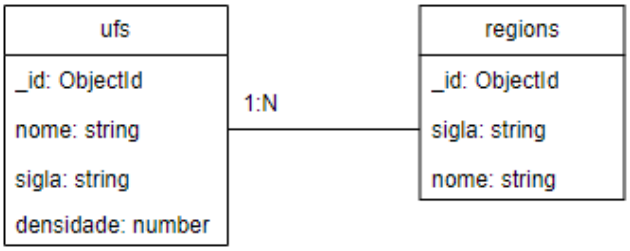
```
[
  { regiao: { nome: 'Centro-Oeste' } },
  { regiao: { nome: 'Centro-Oeste' } },
  { regiao: { nome: 'Centro-Oeste' } },
  { regiao: { nome: 'Centro-Oeste' } }
]
```

Dicas:

- Use o método `updateMany(query, update);`
- Coloque no objeto `query` a propriedade para selecionar apenas os subdocumentos que são da região Centro-Oeste;
- No objeto `update`, use o operador `$unset` para remover os campos `id` e `sigla` do subdocumento `regiao` (https://www.mongodb.com/pt-br/docs/manual/reference/operator/update/unset).

Execute os comandos do arquivo exercicio-parte2.txt para criar a coleção `regions` e `ufs` e inserir os documentos. Essas coleções serão usadas nos exercício 9 a 13.

```
bdaula> db.ufs.countDocuments()
27
bdaula> db.regions.countDocuments()
5
```



Exercício 9 – Fazer um comando para listar os campos `nome` do estado e `nome` da região. Apresente o resultado ordenado pelo campo `nome` do estado. A seguir tem-se parte dos 27 documentos do resultado.

```
[
  { estado: 'Acre', regiao: 'Norte' },
  { estado: 'Alagoas', regiao: 'Nordeste' },
  { estado: 'Amapá', regiao: 'Norte' },
  { estado: 'Amazonas', regiao: 'Norte' },
  { estado: 'Bahia', regiao: 'Nordeste' },
  { estado: 'Ceará', regiao: 'Nordeste' },
  { estado: 'Distrito Federal', regiao: 'Centro-Oeste' },
  { estado: 'Espírito Santo', regiao: 'Sudeste' },
  { estado: 'Goiás', regiao: 'Centro-Oeste' },
```

Dicas:

- Use o método `aggregate([{$lookup}, {$unwind}, {$project}, {$sort}])` com os estágios `$group`, `$sort`, `$project` e `$sort`. Cada estágio precisa estar em um objeto, ou seja, delimitados por chaves;
- O estágio `$unwind` é usado para "desconstruir" arrays nos documentos de entrada, criando uma cópia do documento para cada elemento do array. O resultado a seguir foi obtido sem o estágio `$unwind`, veja que o resultado do estágio `$lookup` está em um array.

```
[
  { estado: 'Acre', regiao: [ 'Norte' ] },
  { estado: 'Alagoas', regiao: [ 'Nordeste' ] },
  { estado: 'Amapá', regiao: [ 'Norte' ] },
  { estado: 'Amazonas', regiao: [ 'Norte' ] },
```

Para mais detalhes https://www.mongodb.com/pt-br/docs/manual/reference/operator/aggregation/unwind.

Exercício 10 – Alterar o comando do Exercício 9 para apresentar o resultado em ordem decrescente de `nome` da região e `nome` do estado. A seguir tem-se parte dos 27 documentos do resultado.

```
[
  { estado: 'Santa Catarina', regiao: 'Sul' },
  { estado: 'Rio Grande do Sul', regiao: 'Sul' },
  { estado: 'Paraná', regiao: 'Sul' },
  { estado: 'São Paulo', regiao: 'Sudeste' },
  { estado: 'Rio de Janeiro', regiao: 'Sudeste' },
  { estado: 'Minas Gerais', regiao: 'Sudeste' },
  { estado: 'Espírito Santo', regiao: 'Sudeste' },
  { estado: 'Tocantins', regiao: 'Norte' },
  { estado: 'Roraima', regiao: 'Norte' },
  { estado: 'Rondônia', regiao: 'Norte' },
  { estado: 'Pará', regiao: 'Norte' },
```

Dica:

- Será necessário incluir o campo `regiao` no objeto do estágio `$sort`.

Exercício 11 – Fazer um comando para contar a quantidade de estados por região. Apresente o resultado ordenado pela quantidade de estados por região.

```
[
  { total: 9, regiao: 'Nordeste' },
  { total: 7, regiao: 'Norte' },
  { total: 4, regiao: 'Sudeste' },
  { total: 4, regiao: 'Centro-Oeste' },
  { total: 3, regiao: 'Sul' }
]
```

Dicas:

- Use o método `aggregate([{$lookup}, {$unwind}, {$group}, {$project}, {$sort}])` com os estágios `$group`, `$sort`, `{$group}`, `$project` e `$sort`. Cada estágio precisa estar em um objeto, ou seja, delimitados por chaves;
- No estágio `$lookup`: será necessário setar o apelido do resultado da consulta usando o campo `as`, por exemplo, para `uf_regiao`. Esse apelido será usado nos estágios `$unwind` e `{$group}`;
- No estágio `$group`:
  - A chave `_id` recebe o campo usado para agrupar, neste caso será `"$uf_regiao.nome"`;
  - Use o operador de agregação `$sum:1` para contar a quantidade de documentos em cada subconjunto criado pelo agrupamento `_id`.
- No estágio `$project`, especifique os campos que serão projetados no resultado;
- No estágio `$sort`, use o campo `total` para ordenar os documentos pelo campo criado no agrupamento.

Exercício 12 – Alterar o comando do Exercício 11 para listar somente o 1º documento do resultado.

```
[ { total: 9, regiao: 'Nordeste' } ]
```

Dica:

- Adicione o estágio `$limit` no pipeline do método `aggregate([{$lookup}, {$unwind}, {$group}, {$project}, {$sort}, {$limit}])`.

Exercício 13 – Alterar o comando do Exercício 11 para listar a densidade média por região. Apresente o resultado ordenado pela média.

```
[
  { media: 171.70499999999998, regiao: 'Sudeste' },
  { media: 135.005, regiao: 'Centro-Oeste' },
  { media: 62.10555555555556, regiao: 'Nordeste' },
  { media: 59.92333333333334, regiao: 'Sul' },
  { media: 5.017142857142857, regiao: 'Norte' }
]
```

Dicas:

- Use o operador `$avg` para calcular a média no estágio `$group`;
- O operador `$avg` deve receber o campo `densidade`. Lembre-se que o nome do campo precisa ser precedido por `$`.