MongoDB

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande

Agenda

1. Modelagem de Dados
   - Único Documento
   - Referências
   - Considerações adicionais sobre modelagem de dados
2. Especificar validação de JSON schema
   - Testes
   - Informações adicionais
3. MongoDB: *Aggregation*
   - Como funciona o *pipeline* de *aggregation*?
   - Exemplo

Modelagem De Dados

Para vincular dados relacionados, você pode:

1. Incorpore dados relacionados em um **único documento**.
2. Armazene dados relacionados em uma coleção separada e acesse-os com uma **referência**.

Único Documento

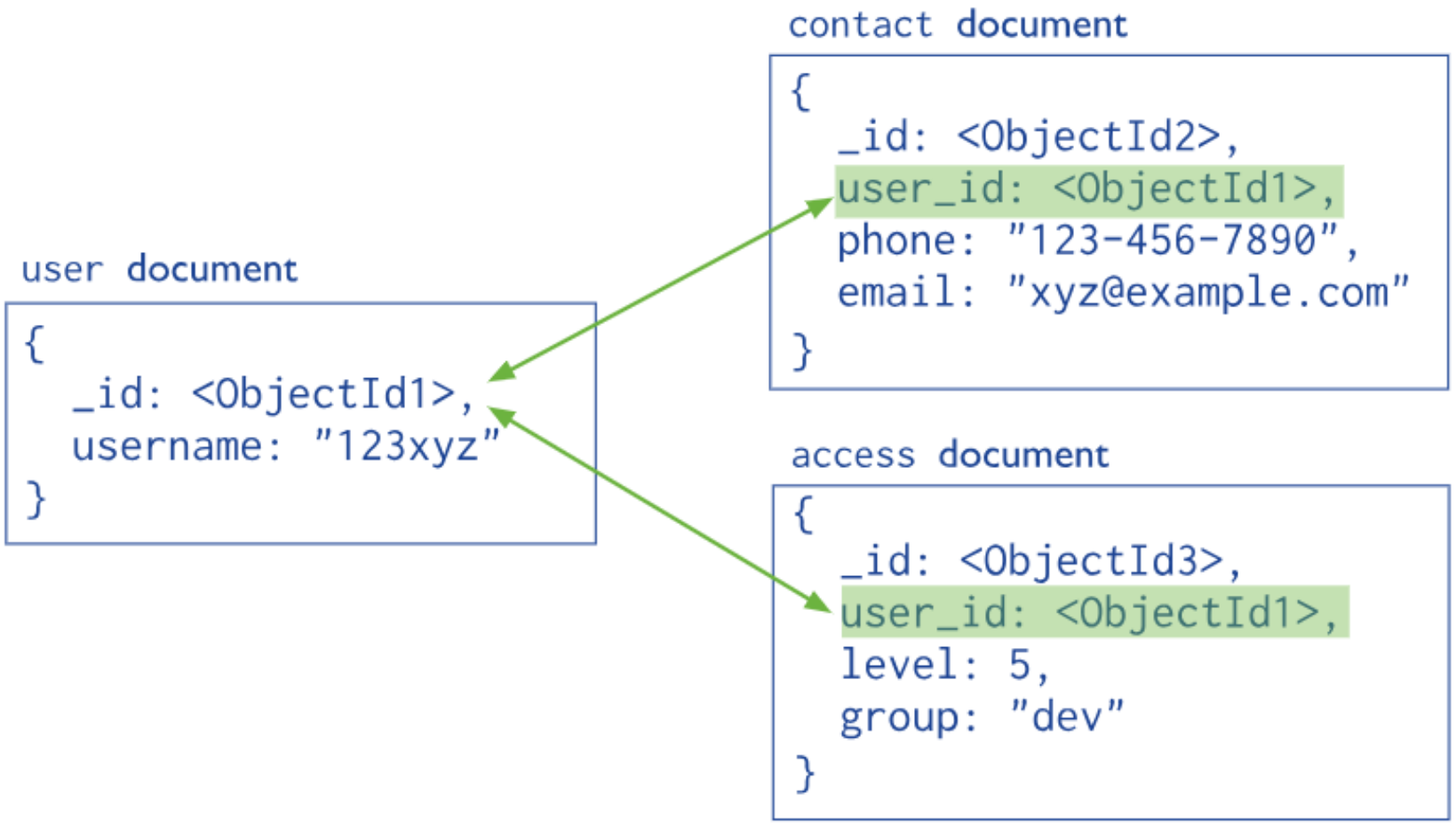
- Documentos incorporados armazenam dados relacionados em uma única estrutura de documento.
- Um documento pode conter *arrays* e subdocumentos com dados relacionados.
- Esses modelos de dados **denormalizados** permitem que os aplicativos recuperem dados relacionados em uma única operação do banco de dados.

Único Documento

Referências

- As referências armazenam relacionamentos entre dados ao incluir links, chamados de referências, de um documento para outro.
  - Por exemplo, um campo customerId em uma collection orders indica uma referência a um documento em uma collection customers.
- Os aplicativos podem resolver essas referências para acessar os dados relacionados. Em termos gerais, estes são modelos de dados **normalizados**.

Único Documento



Considerações adicionais sobre modelagem de dados

- Ao incorporar dados relacionados em um único documento, você pode duplicar dados entre duas collections.
- A duplicação de dados permite que o seu aplicativo consulte informações relacionadas a várias entidades em uma única query, separando logicamente as entidades no seu modelo.

Considerações adicionais sobre modelagem de dados

- Por exemplo, uma collection products armazena as cinco análises mais recentes em um documento de produto.
- Essas avaliações também são armazenadas em uma collection reviews , que contém todas as avaliações de produtos.

Quando uma nova revisão é escrita, ocorrem as seguintes gravações:

- A avaliação é inserida na collection reviews .
- A array de avaliações recentes na coleção products é atualizada com pop e push.

Considerações adicionais sobre modelagem de dados

- Se os dados duplicados não forem atualizados com frequência, o trabalho adicional necessário para manter as duas coleções consistentes será mínimo.
- No entanto, se os dados duplicados forem atualizados com frequência, usar uma referência para vincular dados relacionados pode ser uma abordagem melhor.

Antes de duplicar dados, considere os seguintes fatores:

- Com que frequência os dados duplicados precisam ser atualizados.
- O benefício de desempenho das leituras quando os dados são duplicados.

Especificar validação de JSON schema

Neste exemplo, você cria uma coleta **students** com regras de validação e observa os resultados após tentar inserir um documento inválido.

Especificar validação de JSON schema

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Student Object Validation",
         required: [ "address", "major", "name", "year" ],
         properties: {
            name: {
               bsonType: "string",
               description: "'name' must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "'year' must be an integer in [ 2017, 3017 ] and is requi
            },
            gpa: {
               bsonType: [ "double" ],
               description: "'gpa' must be a double if the field exists"
            }
         }
      }
   }
} )
```

Testes

Execute o seguinte comando. A operação de inserção falha, pois **gpa** é um número inteiro quando o **validator** exige um **double**.

```
db.students.insertOne( {
    name: "Alice",
    year: Int32( 2019 ),
    major: "History",
    gpa: Int32(3),
    address: {
        city: "NYC",
        street: "33rd Street"
    }
} )
```

Testes

```
MongoServerError: Document failed validation

Additional information: {
  failingDocumentId: ObjectId("630d093a931191850b40d0a9"),
  details: {
    operatorName: '$jsonSchema',
    title: 'Student Object Validation',
    schemaRulesNotSatisfied: [
      {
        operatorName: 'properties',
        propertiesNotSatisfied: [
          {
            propertyName: 'gpa',
            description: "'gpa' must be a double if the field exists",
            details: [
              {
                operatorName: 'bsonType',
                specifiedAs: { bsonType: [ 'double' ] },
                reason: 'type did not match',
                consideredValue: 3,
                consideredType: 'int'
              }
            ]
          }
        ]
      }
    ]
  }
}
```

Testes

Se você alterar o valor do campo **gpa** para um tipo **double**, a operação de inserção será bem-sucedida. Execute o seguinte comando para inserir o documento válido:

```
db.students.insertOne( {
    name: "Alice",
    year: NumberInt(2019),
    major: "History",
    gpa: Double(3.0),
    address: {
        city: "NYC",
        street: "33rd Street"
    }
} )
```

Testes

```
db.students.find()
```

OCULTAR SAÍDA

```
[
   {
      _id: ObjectId("62bb413014b92d148400f7a5"),
      name: 'Alice',
      year: 2019,
      major: 'History',
      gpa: 3,
      address: { city: 'NYC', street: '33rd Street' }
   }
]
```

Informações adicionais

Você pode combinar a validação de **JSON schema** com a validação do operador de **query**. Por exemplo, considere uma **collection sales** com esta validação de esquema:

```
db.createCollection("sales", {
  validator: {
    "$and": [
      // Validation with query operators
      {
        "$expr": {
          "$lt": ["$lineItems.discountedPrice", "$lineItems.price"]
        }
      },
      // Validation with JSON Schema
      {
        "$jsonSchema": {
          "properties": {
            "items": { "bsonType": "array" }
          }
        }
      }
    ]
  }
})
```

MongoDB: *Aggregation*

O *aggregation framework* do MongoDB permite que você processe coleções de documentos de dados e retorne resultados computados. Você pode usar o *aggregation* para:

- **Agrupar dados:** Somar vendas por região, contar usuários ativos, etc.
- **Filtrar dados:** Selecionar apenas os documentos que atendem a certos critérios.
- **Transformar dados:** Renomear campos, calcular novos valores, criar *arrays*, etc.
- **Unir dados:** Combinar documentos de diferentes coleções (como um JOIN em SQL).
- **Realizar cálculos complexos:** Calcular médias, desvios padrão, etc.

Como funciona o *pipeline* de *aggregation*?

- O *aggregation framework* usa um conceito de *pipeline*, que é uma sequência de estágios.
- Cada estágio transforma os documentos à medida que eles passam pelo *pipeline*.
  - A saída de um estágio se torna a entrada para o próximo estágio.

Como funciona o *pipeline* de *aggregation*?

Alguns dos estágios mais comuns incluem:

- **match:** Filtra os documentos para passar apenas aqueles que correspondem aos critérios especificados para o próximo estágio.
  - É como a cláusula WHERE em SQL
- **group:** Agrupa os documentos de entrada por um campo especificado e aplica uma função de agregação para cada grupo (por exemplo, *sum*, *avg*, *min*, *max*, *count*).
  - É semelhante à cláusula GROUP BY em SQL, juntamente com funções agregadas.

Como funciona o *pipeline* de *aggregation*?

- **project:** Reformata os documentos. Você pode adicionar novos campos, remover campos existentes ou redefinir os valores dos campos.
  - É como a cláusula SELECT em SQL, permitindo que você escolha e transforme os campos.
- **sort:** Reordena os documentos de entrada por um ou mais campos especificados.
  - É como a cláusula ORDER BY em SQL.
- **limit:** Restringe o número de documentos que passam para o próximo estágio.
  - É como a cláusula LIMIT em SQL.

Como funciona o *pipeline* de *aggregation*?

- **unwind:** Desconstrói um campo de array para criar documentos separados para cada elemento do array.
- **lookup:** Realiza uma junção com outra coleção no mesmo banco de dados para combinar documentos. É semelhante ao JOIN em SQL.
- **out:** Escreve os resultados do pipeline para uma coleção especificada.

Exemplo

Imagine uma coleção chamada pedidos com documentos como este:

JSON

```
{ "_id": 1, "cliente": "Maria", "produto": "Camiseta", "preco": 25.00 }
{ "_id": 2, "cliente": "João", "produto": "Calça", "preco": 75.00 }
{ "_id": 3, "cliente": "Maria", "produto": "Tênis", "preco": 120.00 }
{ "_id": 4, "cliente": "Pedro", "produto": "Camiseta", "preco": 25.00 }
```

Observação

Os pipelines de agregação executados com o método **db.collection.aggregate()** não modificam documentos em uma coleção, a menos que o pipeline contenha um estágio **merge** ou **out**.

Como funciona

Se quisermos saber o total gasto por cada cliente, podemos usar o seguinte pipeline de *aggregation*:

JavaScript

```
db.pedidos.aggregate([
  { $group: { _id: "$cliente", totalGasto: { $sum: "$preco" } } }
])
```

1. **group:** Agrupamos os documentos pelo campo cliente
2. Para cada grupo (cada cliente), calculamos a soma (**sum**) dos valores do campo *preco* e armazenamos o resultado em um novo campo chamado *totalGasto*.

Exemplo

O resultado dessa operação seria algo como:

JSON

```
{ "_id": "Maria", "totalGasto": 145.00 }
{ "_id": "João", "totalGasto": 75.00 }
{ "_id": "Pedro", "totalGasto": 25.00 }
```

Exemplo

```
db.orders.insertMany( [
   { _id: 0, name: "Pepperoni", size: "small", price: 19,
     quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },
   { _id: 1, name: "Pepperoni", size: "medium", price: 20,
     quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" ) },
   { _id: 2, name: "Pepperoni", size: "large", price: 21,
     quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" ) },
   { _id: 3, name: "Cheese", size: "small", price: 12,
     quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z" ) },
   { _id: 4, name: "Cheese", size: "medium", price: 13,
     quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z" ) },
   { _id: 5, name: "Cheese", size: "large", price: 14,
     quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" ) },
   { _id: 6, name: "Vegan", size: "small", price: 17,
     quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" ) },
   { _id: 7, name: "Vegan", size: "medium", price: 18,
     quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }
] )
```

Exemplo

```
db.orders.aggregate( [

   // Stage 1: Filter pizza order documents by pizza size
   {
      $match: { size: "medium" }
   },

   // Stage 2: Group remaining documents by pizza name and calculate total
   {
      $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }
   }

] )
```

Exemplo

```
db.orders.aggregate( [

   // Stage 1: Filter pizza order documents by date range
   {
      $match:
      {
         "date": { $gte: new ISODate( "2020-01-30" ), $lt: new ISODate( "2022-01
      }
   },

   // Stage 2: Group remaining documents by date and calculate results
   {
      $group:
      {
         _id: { $dateToString: { format: "%Y-%m-%d", date: "$date" } },
         totalOrderValue: { $sum: { $multiply: [ "$price", "$quantity" ] } },
         averageOrderQuantity: { $avg: "$quantity" }
      }
   },

   // Stage 3: Sort documents by totalOrderValue in descending order
   {
      $sort: { totalOrderValue: -1 }
   }
```

MongoDB

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande