Banco de Dados II

Módulo 3

Professor: Saulo Morais

Bancos de Dados Não Relacionais (NoSQL)

O que são?

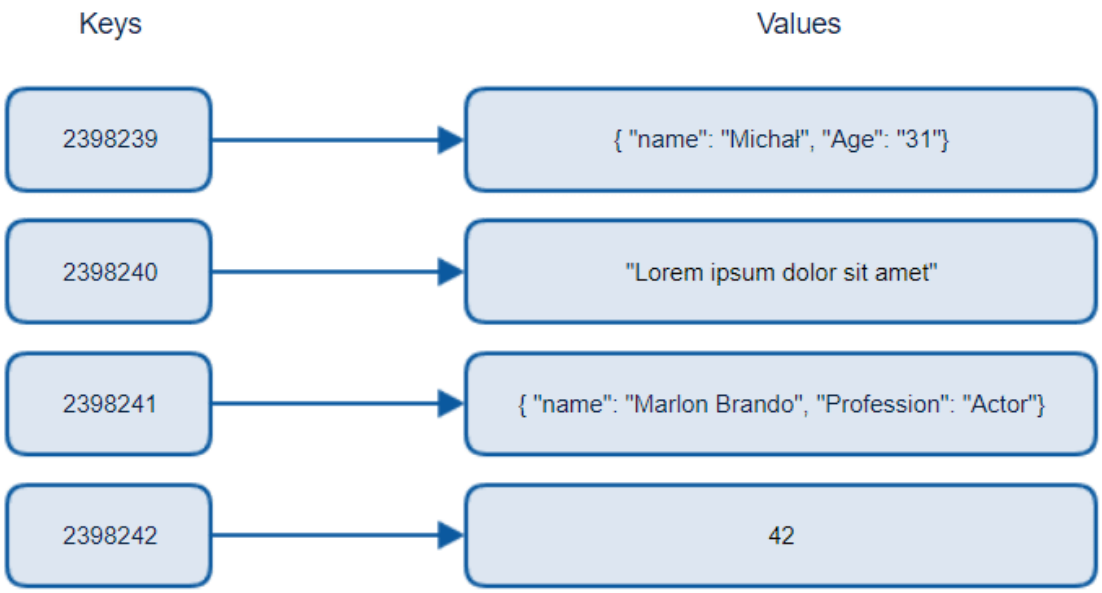
Bancos de dados NoSQL (Not Only SQL) são sistemas de gerenciamento de banco de dados que não seguem o modelo relacional tradicional, e não utilizam SQL como linguagem principal, embora alguns ofereçam suporte a consultas similares.

Eles surgiram para resolver limitações de escalabilidade, desempenho e flexibilidade enfrentadas por bancos de dados relacionais em aplicações modernas, especialmente web e mobile.

Tipos de bancos de dados NoSQL

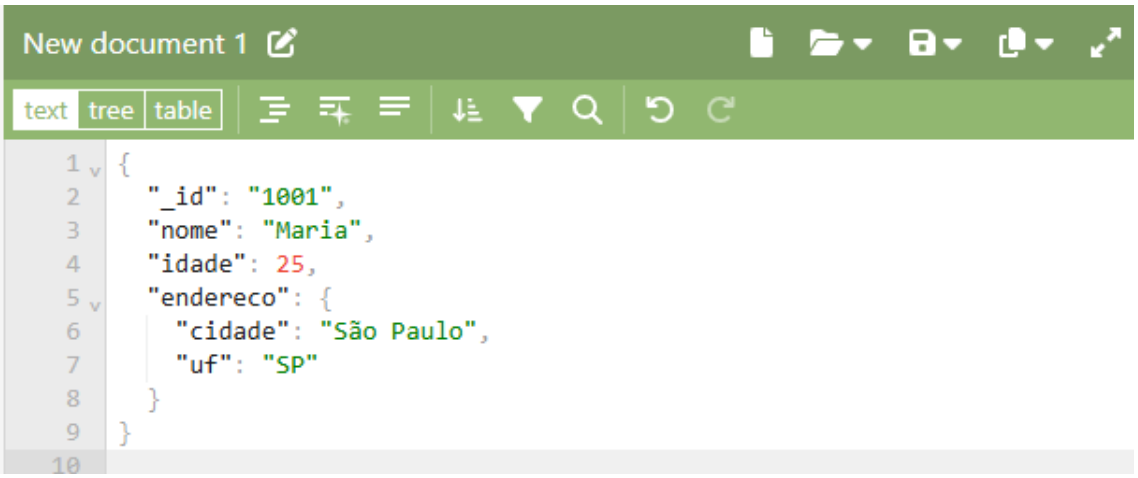
1. Chave-Valor (Key-Value)
    a. Armazenam dados como **pares chave-valor**, onde a chave é única.
    b. Simples, rápidos e ideais para cache e sessões.

Exemplo: Redis, Amazon DynamoDB



2. Documentos (Document-based)
    a. Armazenam dados em **documentos (geralmente JSON, BSON)**.
    b. Permitem estruturas complexas e aninhadas.
    c. Flexíveis quanto ao esquema.

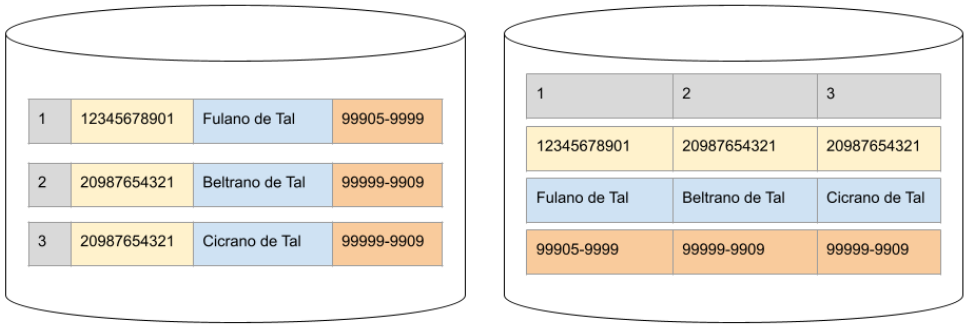
Exemplo: MongoDB, CouchDB



3. Colunar (Column-family)
    a. Armazenam dados por colunas (não por linhas).
    b. Muito usados em sistemas de **big data** e **análises em larga escala**.

Exemplo: Apache Cassandra, HBase

id	cpf	nome	telefone
1	12345678901	Fulano de Tal	99905-9999
2	20987654321	Beltrano de Tal	99999-9909
3	34567890123	Cicrano de Tal	98888-8888

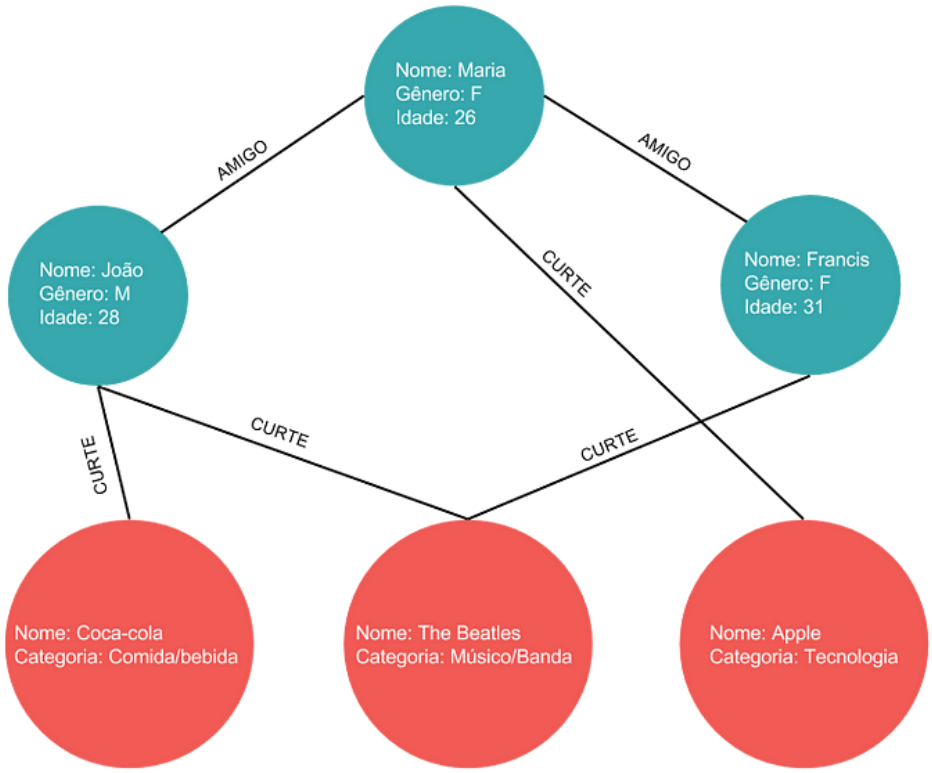


Banco de Dados Orientado a Linhas (modelo tradicional)

Banco de Dados Orientado a Colunas

4. Grafos (Graph-based)
    a. Armazenam dados como **nós e arestas**, representando relacionamentos complexos.
    b. Úteis para redes sociais, motores de recomendação, etc.

Exemplo: Neo4j, ArangoDB



O que é JSON?

Abreviação de JavaScript Object Notation, é um formato leve de troca de dados, muito utilizado em APIs e bancos de dados NoSQL, especialmente os do tipo documento, como o MongoDB.

Características

- Baseado em texto.
- Fácil de ler e escrever.
- Suporta estruturas aninhadas.
- Compatível com muitas linguagens de programação.

Estrutura JSON

```json
{
  "nome": "João",
  "idade": 30,
  "ativo": true,
  "interesses": ["música", "futebol"],
  "endereco": {
    "cidade": "Recife",
    "uf": "PE"
  }
}
```

Tipos de dados aceitos no JSON

Tipo	Exemplo
String	"nome": "Ana"
Número	"idade": 25
Booleano	"ativo": true
Array	"interesses": ["A", "B"]
Objeto	"endereco": { ... }
Null	"fone": null

Estrutura do JSON

1. **Objeto JSON:** Representado por chaves { }, contém pares chave: valor.

```json
{
  "nome": "Lucas",
  "idade": 35,
  "ativo": true
}
```

2. **Array JSON:** Representado por colchetes [ ], pode conter objetos ou valores simples.

```json
[
  "banana",
  "maçã",
  "laranja"
]
```

Ou uma lista de objetos:

```json
[
  { "nome": "Ana", "idade": 30 },
  { "nome": "Pedro", "idade": 22 }
]
```

Regras de Sintaxe JSON

- Chaves devem estar entre aspas duplas " ".
- O JSON não aceita comentários.
- Não pode haver vírgula após o último item de um objeto ou array.
- Os valores não podem ser funções, datas JS ou undefined (em JS puro).

Exemplos práticos

1. Objeto simples

```json
{
  "nome": "Carla",
  "idade": 29,
  "profissao": "Arquiteta"
}
```

2. Objeto com array e objeto aninhado

```json
{
  "nome": "João",
  "cursos": ["HTML", "CSS", "JavaScript"],
  "endereco": {
    "rua": "Rua A",
    "numero": 100
  },
  "ativo": true
}
```

Ferramentas para criar e validar JSON

https://jsoneditoronline.org

https://jsonlint.com/

https://jsonformatter.org/

https://code.visualstudio.com/

Exercícios

1. Veja o seguinte trecho de JSON e **encontre os erros de sintaxe**:

```
{
  nome: "Carlos",
  "idade": 30,
  "ativo": true,
  "cursos": ["HTML", "CSS", "JavaScript",]
}
```

2. Crie um JSON que represente um **aluno**, contendo as seguintes informações:

- nome (string)
- idade (número)
- matriculado (booleano)
- disciplinas (array com pelo menos 2 nomes)

3. Crie um JSON que representa um cliente, contendo as seguintes informações:

- nome (string)
- classificacao (array: Ruim, Bom, Ótimo)
- endereco (objeto contendo: cep, endereço, número, bairro, cidade, estado)

MongoDB

MongoDB é um banco de dados NoSQL orientado a documentos.
Em vez de armazenar os dados em tabelas e linhas (como nos bancos relacionais), ele armazena os dados em documentos no formato BSON (uma versão binária de JSON), dentro de coleções.

Conceitos principais

Termo MongoDB	Equivalente em SQL	Descrição
Database	Banco de dados	Conjunto de coleções
Collection	Tabela	Conjunto de documentos
Document	Linha (registro)	Dados armazenados em formato JSON/BSON
Field	Coluna	Um par chave-valor no documento

Instalação

1. Instalação local:
Baixe em https://www.mongodb.com/try/download/community

2. MongoDB Atlas (na nuvem):
Crie uma conta gratuita em https://www.mongodb.com/cloud/atlas

Interface de acesso

- MongoDB Shell – Linha de comando (mongosh)
- MongoDB Compass – Interface gráfica amigável
- Drivers – Para integrar com linguagens como JavaScript, Python, etc.

Baixar o MongoDB Shell

1. https://www.mongodb.com/try/download/shell
2. Após instalar, adicione o caminho ao mongosh na variável de ambiente PATH.

Sintaxe básica no mongosh

- Listar todos os bancos de dados:

show dbs

- Criar ou acessar um banco de dados:

  use loja

- Inserir um documento:

  db.produtos.insertOne({
   nome: "Teclado",
   preco: 120.90,
   categorias: ["Acessórios", "Informática"],
   estoque: 20
  })

- Inserir vários documentos:

  db.produtos.insertMany([
   { nome: "Mouse", preco: 80.0, estoque: 50 },
   { nome: "Monitor", preco: 600.0, estoque: 15 }
  ])

- Listar documentos:

  db.produtos.find()

- Listar collections de um banco de dados:

  use loja

  show collections

- Apagar collection

  use loja

  db.produtos.drop()

Operadores

- **Comparação**

$eq	Igual a	{ preco: { $eq: 100 } }

db.produtos.find({ preco: { $eq: 100 } })

$ne	Diferente de	{ preco: { $ne: 0 } }

db.produtos.find({ preco: { $ne: 0 } })

$gt	Maior que	{ preco: { $gt: 50 } }

db.produtos.find({ preco: { $gt: 50 } })

$gte	Maior ou igual	{ preco: { $gte: 10 } }

db.produtos.find({ preco: { $gte: 10 } })

$lt	Menor que	{ preco: { $lt: 200 } }

db.produtos.find({ preco: { $lt: 200 } })

$lte	Menor ou igual	{ preco: { $lte: 100 } }

db.produtos.find({ preco: { $lte: 100 } })

$in	Está dentro de uma lista	{ categoria: { $in: ["Eletrônicos", "Informática"] } }

db.produtos.find({ categoria: { $in: ["Eletrônicos", "Informática"] } })

$nin	Não está dentro na lista	{ categoria: { $nin: ["Eletrônicos", "Informática"] } }

db.produtos.find({ categoria: { $nin: ["Eletrônicos", "Informática"] } })

Exercícios

1. Crie um banco de dados com o nome de exercicios_comparacao (caso já tiver um banco com esse nome, você deverá apaga-lo e criar um novo) e adicione uma coleção de produtos com 5 documentos. A coleção deve ter os campos:
    a. nome: tipo string
    b. preco: tipo número
    c. categoria: tipo array com 3 categorias
    d. estoque: tipo número
2. Liste todos os produtos com preço maior que R$ 100.

3. Liste os produtos com estoque menor ou igual a 10 unidades
4. Encontre os produtos cuja categoria não seja "perifericos".
5. Liste os produtos com preço entre R$ 50 e R$ 200, inclusive.
6. Encontre todos os produtos que custam exatamente R$ 150.

- **Lógicos**

$and: Todas as condições devem ser verdadeiras

```
db.usuarios.find({
 $and: [
  { idade: { $gte: 18 } },
  { ativo: true }
 ]
})
```

$or: Pelo menos uma condição deve ser verdadeira

```
db.usuarios.find({
 $or: [
  { cidade: "Nova Serrana" },
  { cidade: "Pará de Minas" }
 ]
})
```

$not: Nega a condição (inverte o resultado)

```
db.usuarios.find({
 idade: { $not: { $eq: 30 } }
})
```

$nor: Nenhuma condição deve ser verdadeira

```
db.usuarios.find({
 $nor: [
  { cidade: "Curitiba" },
  { idade: { $lt: 18 } }
 ]
})
```

Combinação de operadores ($and e $or)

```
db.usuarios.find({
 $and: [
  { idade: { $gte: 30, $lte: 50 } },
  {
   $or: [
    { cidade: "Belo Horizonte" },
    { ativo: true }
   ]
```

```
  }
 ]
})
```

Exercícios

1. Crie um banco de dados com o nome de exercicio_operadores (caso já tiver um banco com esse nome, você deverá apagá-lo e criar um) e adicione uma coleção de usuarios com 5 documentos, onde todos devem ter idade maior de 18 anos e pelo menos 2 usuários devem ser de Nova Serrana. A coleção deve ter os campos:
    a. nome: tipo string
    b. idade: tipo número
    c. cidade: tipo string
    d. ativo: tipo booleano
2. Liste os usuários que têm **idade maior ou igual a 18** e estão **ativos**.
3. Encontre os usuários que moram em **"Nova Serrana"** ou têm **menos de 25 anos**.
4. Liste os usuários que **não** estão ativos.
5. Encontre os usuários que **não moram em "Nova Serrana" nem** têm mais de 40 anos.
6. Encontre os usuários que:
    a. **têm idade entre 30 e 50**, **e**
    b. **moram fora de "Nova Serrana"** e **estão ativos**.

Apagar documentos

- deleteOne(): Apaga apenas o primeiro documento que atende ao critério

  ```
  db.produtos.deleteOne( { _id: ObjectId("563237a41a4d68582c2509da") } )
  ```

  ```
  db.usuarios.deleteOne({ nome: "João" })
  ```

- deleteMany(): Apaga todos os documentos que atendem ao critério

  ```
  db.usuarios.deleteMany({ status: "inativo" })
  ```

- Apagar todos os documentos de uma coleção (sem apagar a coleção)

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

- Apagar a coleção inteira: Apaga **todos os documentos** e **remove a coleção** do banco de dados. Esta ação **não pode ser desfeita**.

  ```
  db.usuarios.drop()
  ```

Atualizar documentos

- updateOne(): atualiza campos específicos de **um único documento**, que atender ao critério. Caso o filtro, trouxer mais de um documento, a alteração será feita somente no primeiro.

```
// json clientes
{
 "nome": "Dante",
 "idade": 2,
 "email": "dante@email.com",
 "ativo": true
}

// alterar a idade do cliente Dante para 3
db.customer.updateOne(
 {name: {$eq: "Dante"}},
 {$set: {idade: 3}}
)
```

- replaceOne(): Substitui **todo o documento** por outro (mantendo apenas o _id)

```
//json usuarios
{
   "_id": {
    "$oid": "6835c1c712fb929069f8c41b"
   },
   "name": "Saulo Morais Lara",
   "email": "saulo@abilityonline.com.br"
 }

// substituir o documento conforme critério
db.usuarios.replaceOne(
 {nome: {$eq: "Saulo Morais Lara"}},
 {
   nome: "Saulo Morais Lara",
   email: "saulo@abilityonline.com.br",
   endereco: "Rua Teste, 100"
 }
)
```

- updateMany(): Atualiza **vários documentos** que correspondem ao filtro, alterando apenas os campos desejados.

```
// json
[
 {
   "_id": {
    "$oid": "6835ca43b90d341c91cbea9d"
   },
   "nome": "Marcela",
```

```
    "cidade": "Pará de Minas",
    "ativo": true
   },
   {
    "_id": {
     "$oid": "6835ca43b90d341c91cbea9e"
    },
    "nome": "Saulo",
    "cidade": "Pará de Minas",
    "ativo": true
   }
  ]

  // desativar todos os usuários da cidade de Pará de Minas
  db.usuarios.updateMany(
   {cidade: {$eq: 'Pará de Minas'},},
   {$set: {ativo: false}}
  )
```

Relacionamento entre coleções (collections)

MongoDB é um banco de dados NoSQL orientado a documentos, e não usa joins nativamente como bancos relacionais. No entanto, é possível representar relações entre dados de duas formas:

- Referenciado (Normalized) – semelhante a foreign key.
- Incorporado (Embedded) – dados aninhados dentro de um documento.

1. Relacionamento Referenciado (Normalized)

Vantagens:

- Evita duplicação de dados.
- Melhor para dados que mudam com frequência.

Exemplo: clientes e pedidos em coleções separadas

```
// clientes
{
 "_id": 1,
 "nome": "Sílvio Santos"
}

// pedidos
{
 "_id": 100,
 "cliente_id": 1,
 "data": "2024-05-27”,
 "itens": [
  { "produto": "Teclado", "quantidade": 1 },
  { "produto": "Mouse", "quantidade": 2 }
```

```
 ]
}

// retornar um json com pedidos e clientes
db.pedidos.aggregate([
 {
  $lookup: {
   from: "clientes",                // coleção que será "referenciada"
   localField: "cliente_id",        // campo da coleção atual (pedidos)
   foreignField: "_id",             // campo da coleção clientes
   as: "cliente"                    // nome do campo de saída
  }
 }
])

// Json de retorno
{
  "_id": 1,
  "cliente_id": 1,
  "itens": [
   {
    "produto": "Teclado",
    "quantidade": 1
   },
   {
    "produto": "Mouse",
    "quantidade": 2
   }
  ],
  "cliente": [
   {
    "_id": 1,
    "nome": "Silvio Santos"
   }
  ]
 }
```

2. Relacionamento Incorporado (Embedded)

Vantagens:

- Leitura mais rápida
- Mais simples quando os dados são sempre usados juntos.

Exemplo: cliente com pedidos embutidos

```
{
 "_id": 1),
 "nome": "Maria Oliveira",
 "email": "maria@email.com",
 "pedidos": [
  {
```

```
    "data": "2024-05-27”,
    "itens": [
     { "produto": "Monitor", "quantidade": 1 },
     { "produto": "HD", "quantidade": 2 }
    ]
   }
 ]
}

// Retornar cliente e os pedidos
db.pedidos.find(
 {_id: {$eq: 1}}
)

// Json de retorno
{
  "_id": 1),
  "nome": "Maria Oliveira",
  "email": "maria@email.com",
  "pedidos": [
   {
    "data": "2024-05-27”,
    "itens": [
     { "produto": "Monitor", "quantidade": 1 },
     { "produto": "HD", "quantidade": 2 }
    ]
   }
 ]
}
```

Desenvolvendo um banco para um sistema de E-Commerce

Objetivo: Criar um banco de dados MongoDB que simule um e-commerce simples para um supermercado. Usar o tipo de relacionamento incorporado, já que os dados não mudarão com frequência.

- **Cadastro de clientes:** contendo e-mail e senha e todas as informações cadastrais.
- **Cadastro de produtos organizados por categoria**
- **Pedido:** deverá ter as informações do cliente, produto e status (aberto e fechado). Quando ele for fechado, deverá ter as informações de pagamento e o valor.

Todos os comandos utilizados deverão ser gravados em um arquivo e enviado para o e-mail saulinhomorais@gmail.com com o título: **Desenvolvendo um banco para um sistema de E-Commerce**

1. Criar o banco de dados

```
use ecommerce
```

2. Estrutura da coleção de clientes

```
{
  "nome": "João da Silva",
  "email": "joao@email.com",
  "senha": "senha123",
  "endereco": {
    "rua": "Rua A",
    "numero": 123,
    "cidade": "Nova Serrana",
    "estado": "MG",
    "cep": "35519-000"
  },
  "telefone": "37999999999"
}
```

3. Inserir 5 clientes com e-mails diferentes e cidades diferentes

4. Estrutura da coleção de produtos

```
{
  "nome": "Arroz",
  "descricao": "Arroz integral tipo 1",
  "preco": 39.00,
  "categoria": "Alimentício",
  "estoque": 10
}
```

5. Inserir 10 produtos com categorias diferentes e com estoque de 10 quantidades

6. Estrutura de coleção do pedido

```
{
  "cliente": {
    "nome": "João da Silva",
    "email": "joao@email.com",
    "endereco": {
      "rua": "Rua A",
      "numero": 123,
      "cidade": "Nova Serrana",
      "estado": "MG",
      "cep": "35519-000"
    },
    "telefone": "37999999999"
  },
  "itens": [
    {
      "nome": "Arroz",
      "descricao": "Arroz integral tipo 1",
      "preco": 39.00,
      "categoria": "Alimentício",
      "quantidade": 1
```

```
  },
  {
   "nome": "Feijão",
   "descricao": "Feijão carioquinha tipo 1",
   "preco": 19.00,
   "categoria": "Alimentício",
   "quantidade": 2
  }
 ],
 "data_criacao": "2025-05-27",
 "status": "aberto"
}
```

7. Inserir 3 pedidos em aberto, cada um com no mínimo 3 produtos.

8. Pegar o último pedido inserido acima e atualizar seu status para “fechado”. Ao ser alterado o status, deverá adicionar um novo objeto com o nome de “pagamento” com a forma de pagamento e o valor pago.

```
{
 "cliente": {...},
 "itens": [
  {...}
 ],
 "data_criacao": "2025-05-27",
 "status”: "fechado",
 "pagamento": {
    "forma_pagamento": "dinheiro",
    "valor_pago": 58.00
  }
}
```

MongoDB Aggregation Framework

É um sistema poderoso de consultas usado para processar e transformar dados dentro do MongoDB, especialmente quando você precisa ir além de simples buscas (como find()). Ele permite realizar operações como:

- Agrupamento ($group)
- Ordenação ($sort)
- Soma e média de valores ($sum, $avg)
- Junções ($lookup)
- E muito mais

$sum: Calcula e retorna a soma coletiva de valores numéricos. $sum ignora valores não numéricos.

Exemplo 1: Somar todos os pedidos de uma coleção

```
db.pedidos.aggregate([
 {
  $group: {
   _id: null,
   valor_total_pedidos: { $sum: "$valor_total" }
  }
 }
])
```

Exemplo 2: Somar todos os pedidos com status fechado

```
db.pedidos.aggregate([
 {
  $match: {"status": {$eq: "fechado"}}
 },
 {
  $group: {
   _id: null,
   valor_total_pedidos: { $sum: "$valor_total" }
  }
 }
])
```

Exemplo 3: Somar todos os pedidos agrupados por data

```
use('ecommerce');
db.pedidos.aggregate([
 {
  $group: {
   _id: { data: "$data_criacao" },
   valor_total_pedidos: { $sum: "$valor_total" }
  }
 }
])
```

$avg: Retorna o valor médio dos valores numéricos. $avg ignora valores não numéricos.

Exemplo 1: Fazer a média dos pedidos da coleção

```
db.pedidos.aggregate(
 {
  $group: {
   _id: null,
   valor_total_pedidos:  { $avg: "$valor_total" }
  }
 }
)
```

Exemplo 2: Fazer a média dos pedidos da coleção e arredondar para duas casas decimais

```
 {
```

```
   $group: {
    _id: null,
    media_valor: { $avg: "$valor_total" }
   }
  },
  {
   $project: {
    _id: 0,
    media_arredondada: { $round: ["$media_valor", 2] }
   }
  }
)
```

$count: Você pode usar o estágio $count para contar os documentos.

Exemplo 1: Contas todos os pedidos (documentos) da coleção.

```
db.pedidos.aggregate( [
  { $count: "total_pedidos" }
])
```

Exemplo2: Caso você precisar de retornar a cidade e quantidade de clientes daquela cidade, utilize a função $sum.

```
db.clientes.aggregate(
 {
   $group: {
    _id: "$endereco.cidade",
    quantidade: { $sum: 1 }
   }
 }
)
```

$sort: Ordena todos os documentos de entrada e os retorna ao pipeline em ordem.

Exemplo1: Retornar todos os pedidos em ordem crescente

```
db.pedidos.aggregate( [
  { $sort: {data_criacao : 1 } }
])
```

Exemplo2: Retornar todos os pedidos em ordem decrescente

```
db.pedidos.aggregate( [
  { $sort: {data_criacao : -1 } }
])
```

Exemplo 3: Somar todos os pedidos agrupados e ordenados por data em ordem crescente

```
db.pedidos.aggregate([
 {
```

```
  $group: {
    _id: { data: "$data_criacao" },
    valor_total_pedidos: { $sum: "$valor_total" }
  }
 },
 {
  $sort: { data: -1 }
 }
])
```

Para saber mais sobre as operações de agregação acesse o link https://www.mongodb.com/pt-br/docs/manual/reference/operator/aggregation/

Desenvolvendo um banco de dados em MongoDB para um sistema de gestão financeira.

- Deverá ser utilizado relacionamento Referenciado (Normalized) – semelhante a foreign key.

1. Criar as collections necessárias para nosso sistema.

- **empresa:** Informações cadastrais da empresa.
  - Deverá conter somente 1 documento.

- **usuarios:** Deverá armazenar e-mail, senha de acesso e um campo para informar se o usuário está ativo ou não. Utilizar um campo boolean.
  - Deverá conter 3 documentos com usuários diferentes.

- **clientes:** Informações cadastrais completas e um campo para informar a data do cadastro.
  - Deverá conter 5 documentos com clientes diferentes.

- **fornecedores:** Informações cadastrais completas e um campo para informar a data do cadastro.
  - Deverá conter 5 documentos com fornecedores diferentes.

- **contas_receber:** Deverá conter uma referência para o cliente (cliente_id) e informações da conta como: data de lançamento, data de vencimento, valor a receber e status (aberto ou pago). Após a conta ser paga, deverá ser alterado o status para pago e criado um objeto com informações do pagamento com os campos: data do pagamento, forma de pagamento e valor do pagamento.
  - Deverá ser gerado 5 lançamentos com datas e valores diferentes e com o status aberto.

- **contas_pagar:** Deverá conter uma referência para o fornecedor (fornecedor_id) e informações da conta como: data de lançamento, data de vencimento, valor a pagar e status (aberto ou pago). Após a conta ser paga, deverá ser alterado o status para pago e criado um objeto com informações do pagamento com os campos: data do pagamento, forma de pagamento e valor do pagamento.
  - Deverá ser gerado 7 lançamentos com datas e valores diferentes e com o status aberto.

2. Fazer uma consulta que retorne o e-mail de todos os usuários que estão ativos.

3. Fazer uma consulta que retorne todos os clientes por ordem crescente de nome.

4. Fazer uma consulta que retorne todos os fornecedores por ordem decrescente de data do cadastro.

5. Efetuar o pagamento de duas contas a receber alterando seu status para “pago” e adicionando um objeto de pagamento com as informações da data de pagamento, forma de pagamento (dinheiro, cartão de crédito, etc) e valor do pagamento.

Dica: Faça um find() para verificar qual o ID de cada documento e depois faça um updateOne para cada conta paga:

```
db.contas_receber.updateOne(
 {
  _id: ObjectId("683f32d468ed8ed47980a5b9")
 },
 {
  $set: { ... }
 }
)
```

6. Fazer uma consulta que retorne o nome do cliente, a data de vencimento e o valor de todas as contas a receber com status “aberto”.

7. Efetuar o pagamento de três contas a pagar alterando seu status para “pago” e adicionando um objeto de pagamento com as informações da data de pagamento, forma de pagamento (dinheiro, cartão de crédito, etc) e valor do pagamento.

    1. Dica: Faça um find() para verificar qual o ID de cada documento e depois faça um updateOne para cada conta paga:

```
db.contas_pagar.updateOne(
 {
  _id: ObjectId("683f32d468ed8ed47980a5b9")
 },
 {
  $set: { ... }
 }
)
```

8. Fazer uma consulta que retorne o nome do fornecedor, a data de vencimento e o valor de todas as contas a receber com status “pago”.

9. Fazer uma consulta que retorne a data e o valor de pagamento de todas as contas a pagar com status “aberto”, agrupado por data de vencimento.

10. Fazer uma consulta que retorne o nome da cidade e a quantidade de clientes daquela cidade.