Banco de dados NoSQL

Os bancos de dados NoSQL surgiram como uma resposta às limitações dos bancos de dados relacionais (SQL) em lidar com os desafios modernos de escalabilidade, flexibilidade e performance em grandes volumes de dados.

1. Surgimento dos Bancos de Dados Relacionais (1970s)

- Antes dos bancos NoSQL, os **bancos relacionais** dominavam o cenário de armazenamento de dados, baseados no modelo proposto por **Edgar F. Codd** em 1970.
- Ferramentas como **Oracle, IBM DB2, MySQL** e **PostgreSQL** se tornaram os padrões por décadas.
- Apesar de eficazes, os bancos relacionais tinham limitações para lidar com:
    - **Escalabilidade horizontal:** Crescer adicionando mais servidores (em vez de aumentar a capacidade de um único servidor).
    - **Flexibilidade no esquema:** Alterar a estrutura de tabelas é difícil e pode impactar a performance.
    - **Altos volumes de dados desestruturados.**

2. O Crescimento da Web e o "Big Data" (2000s)

- Com o surgimento da **web 2.0**, empresas como Google, Amazon e Facebook começaram a lidar com volumes massivos de dados gerados por usuários em tempo real.
- Bancos relacionais começaram a mostrar suas limitações, especialmente em:
    - **Gerenciar grandes volumes de dados distribuídos.**
    - **Atender a milhões de transações simultâneas.**
    - **Trabalhar com dados semi-estruturados e desestruturados** (como JSON, XML, logs e dados multimídia).

Soluções Internas

- Empresas começaram a criar soluções próprias:
    - **Amazon Dynamo (2007):** Um banco de dados chave-valor usado para escalar sua plataforma de e-commerce.
    - **Google Bigtable (2006):** Banco baseado em colunas, projetado para lidar com dados massivos.
    - **Facebook Cassandra (2008):** Criado para gerenciar grandes quantidades de mensagens distribuídas.

Esses projetos abriram o caminho para o surgimento dos bancos NoSQL como tecnologias acessíveis.

3. Popularização do NoSQL (2009-2010)

- O termo **"NoSQL"** foi cunhado em 2009 por **Johan Oskarsson** em uma conferência, destacando alternativas aos bancos relacionais tradicionais.
- Primeiras ferramentas NoSQL a ganhar popularidade:
    - **MongoDB (2009):** Banco de dados baseado em documentos.
    - **CouchDB (2005):** Banco de dados distribuído para sincronização offline.
    - **Apache Cassandra:** Otimizado para grandes volumes de dados distribuídos.
    - **Redis (2009):** Banco chave-valor em memória, focado em alta performance.

Por que "NoSQL"?

- Originalmente, significava **"Not Only SQL"**, destacando a possibilidade de usar diferentes abordagens além do modelo relacional.

4. Evolução e Consolidação (2010-2020)

- **Adaptação às Necessidades Modernas:**
    - Aplicações de redes sociais, IoT, Big Data e inteligência artificial impulsionaram o uso de NoSQL.
    - Empresas como Netflix, Twitter e Uber adotaram NoSQL para atender milhões de usuários.
- **Principais Desafios:**
    - O NoSQL inicialmente tinha suporte limitado para **transações complexas** e consistência estrita, algo em que os bancos relacionais sempre se destacaram.
    - Soluções modernas começaram a integrar modelos híbridos para atender a esses desafios (ex.: **Cosmos DB** da Microsoft).

5. Tendências Atuais e Futuras

- **Modelos Multimodais:** Bancos como **ArangoDB** e **Couchbase** suportam múltiplos modelos (documentos, grafos, chave-valor) em uma única solução.
- **NoSQL na Nuvem:** Serviços como **Firebase (Google)**, **DynamoDB (AWS)** e **Cosmos DB (Azure)** popularizaram o uso de NoSQL como parte das plataformas de nuvem.
- **Integração com IA e Machine Learning:** O NoSQL é amplamente usado para armazenar e processar dados usados em modelos de aprendizado de máquina.

Comparação: Relacional vs. NoSQL

Aspecto	**Relacional (SQL)**	**NoSQL**

Estrutura de Dados	Estruturada (tabelas)	Flexível (documentos, grafos, etc.)
Escalabilidade	Vertical (mais hardware)	Horizontal (mais servidores)
Performance	Ideal para transações complexas	Ideal para leitura e escrita rápidas
Consistência	Consistência forte	Eventual ou ajustável

Por Que o NoSQL É Importante?

- É indispensável para sistemas distribuídos, grandes volumes de dados (Big Data) e dados dinâmicos.
- Sua flexibilidade e escalabilidade horizontal permitem que empresas modernas inovem em velocidade e escala.

Aqui estão exemplos de como modelar a estrutura de um **carrinho de compras** usando bancos de dados relacionais e NoSQL. Ambos têm abordagens diferentes por causa de suas características.No modelo relacional, os dados são organizados em tabelas com relacionamentos definidos entre elas.

Estrutura:

1. **Tabela `users`** – Contém os usuários do sistema.
2. **Tabela `products`** – Contém os produtos disponíveis para compra.
3. **Tabela `carts`** – Representa o carrinho de compras de cada usuário.
4. **Tabela `cart_items`** – Contém os itens dentro do carrinho, associando o produto ao carrinho.

Exemplo de Modelagem:

```
-- Tabela de usuários

CREATE TABLE users (

    user_id INT PRIMARY KEY,

    name VARCHAR(100),

    email VARCHAR(100) UNIQUE

);
```

```sql
-- Tabela de produtos
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100),
    price DECIMAL(10, 2),
    stock INT
);

-- Tabela de carrinhos
CREATE TABLE carts (
    cart_id INT PRIMARY KEY,
    user_id INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);

-- Tabela de itens do carrinho
CREATE TABLE cart_items (
    cart_item_id INT PRIMARY KEY,
    cart_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (cart_id) REFERENCES carts(cart_id),
```

```
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Como Funciona:

1. Cada usuário possui um **carrinho único**, identificado por `cart_id`.
2. Os itens dentro do carrinho estão na tabela `cart_items`, que armazena a quantidade de cada produto.
3. Os produtos estão relacionados à tabela `products`, que mantém os detalhes dos itens.

Adicionar um produto ao carrinho:

```
INSERT INTO cart_items (cart_id, product_id, quantity)

VALUES (1, 101, 2); -- Adiciona 2 unidades do produto 101 ao carrinho 1
```

Consulta para listar o carrinho de um usuário:

```
SELECT p.name, p.price, ci.quantity, (p.price * ci.quantity) AS total

FROM cart_items ci

JOIN products p ON ci.product_id = p.product_id

WHERE ci.cart_id = 1;
```

No modelo NoSQL, os dados são organizados de forma mais flexível, muitas vezes evitando normalização. Um documento pode conter informações aninhadas.

Estrutura:

- Cada **usuário** tem um documento representando seu carrinho de compras.
- Os itens do carrinho são armazenados diretamente como subdocumentos.

Exemplo de Documento:

```
{

  "user_id": 1,
```

```
    "name": "João",
    "email": "joao@email.com",
    "cart": {
      "created_at": "2024-11-19T10:00:00Z",
      "items": [
        {
          "product_id": 101,
          "name": "Camiseta",
          "price": 49.99,
          "quantity": 2,
          "total": 99.98
        },
        {
          "product_id": 102,
          "name": "Tênis",
          "price": 199.99,
          "quantity": 1,
          "total": 199.99
        }
      ],
      "cart_total": 299.97
    }
  }
```

Como Funciona:

1. O carrinho está embutido como um campo no documento do usuário (`cart`).
2. Os itens do carrinho (`items`) são armazenados como uma lista de subdocumentos, onde cada item contém os detalhes do produto.
3. O total do carrinho (`cart_total`) pode ser atualizado dinamicamente quando itens são adicionados ou removidos.

Operações:

Adicionar um item ao carrinho:

```
db.users.updateOne(
  { user_id: 1 },
  {
    $push: {
      "cart.items": {
        product_id: 103,
        name: "Mochila",
        price: 149.99,
        quantity: 1,
        total: 149.99
      }
    },
    $inc: { "cart.cart_total": 149.99 }
  }
);
```

Listar o carrinho de um usuário:

```
db.users.find(
  { user_id: 1 },
```

```
    { "cart.items": 1, "cart.cart_total": 1 }
  );
```

Comparação SQL x NoSQL no Carrinho de Compras

Aspecto	SQL	NoSQL
Flexibilidade	Estrutura fixa e bem definida	Estrutura flexível e aninhada
Desempenho	Requer JOINs para consultar tabelas	Dados relacionados já embutidos
Complexidade	Normalizado, mas pode ser mais complexo	Simples para alterações rápidas
Atualizações	Mais difícil com várias tabelas	Mais fácil devido à estrutura embutida
Escalabilidade	Vertical (servidores maiores)	Horizontal (distribuição entre servidores)