

Victória Amaral

NOSQL



CAPTURANDO DADOS COM
A MAGIA DOS POKÉMONS!

01

INTRODUÇÃO

INTRODUÇÃO

Olá, alunos de BD II-NoSQL!

Gostaria de, como primeiro contato, expor um pouco de minha experiência e de meus colegas. Nesta matéria, vocês irão programar utilizando tanto o MongoDB (banco guiado a documentos e muito utilizado em empresas de bancos financeiros) quanto o Neo4J (banco guiado a grafos e utilizado em redes sociais como Facebook e LinkedIn). Minha turma especialmente preferiu o MongoDB por ser levemente familiar com o banco relacional. Aconselho a se dedicarem em uma boa nota de P1 se não quiserem se preocupar com a possibilidade de não se adaptarem tanto com o Neo4J na P2. Eu, particularmente, prefiro programar em Neo4J, mas exceção não é regra.

Vamos iniciar nossa jornada explorando a história dos bancos de dados não relacionais, também conhecidos como NoSQL.

02

HISTÓRIA DOS BANCOS NÃO RELACIONAIS

HISTÓRIA DOS BANCOS NÃO RELACIONAIS

Uma resposta às limitações dos bancos tradicionais.

Enquanto os bancos relacionais organizam dados em tabelas com linhas e colunas, os bancos NoSQL são mais flexíveis e podem armazenar dados em diversos formatos, como documentos, grafos, colunas e chave-valor.

A evolução dos bancos de dados NoSQL foi impulsionada pela necessidade de gerenciar grandes volumes de dados, a alta disponibilidade e a escalabilidade que os sistemas modernos requerem. Empresas como Amazon, Google e Facebook foram pioneiras na adoção de tecnologias NoSQL para lidar com seus imensos conjuntos de dados e a necessidade de respostas rápidas.

03

O MUNDO DE NOSQL: DOCUMENTOS E GRAFOS

Vamos imaginar que os dados são como Pokémons. Em vez de organizá-los em caixas e linhas rígidas, podemos armazená-los de maneiras mais flexíveis que nos permitem acessar e manipular informações de maneira mais eficiente.

O MUNDO DE NOSQL: DOCUMENTOS E GRAFOS

MongoDB: O Banco de Documentos

MongoDB é um banco de dados orientado a documentos, onde os dados são armazenados em documentos JSON semelhantes a objetos. Pense nos documentos como cartas Pokédex detalhando cada Pokémon: sua espécie, tipo, ataques e estatísticas. Cada carta é independente e pode ter informações diferentes, permitindo uma grande flexibilidade.

Por exemplo, um documento JSON para um Pokémon pode ser assim:

Exemplo

Criando Documentos Para Treinadores e Pokémons;

○ ○ ○

```
// Treinador
{
  "_id": ObjectId("treinador_ash"),
  "nome": "Ash Ketchum",
  "idade": 10,
  "pokemons": [
    { "_id": ObjectId("pokemon_pikachu"), "nome": "Pikachu", "tipo": "Elétrico" },
    { "_id": ObjectId("pokemon_charizard"), "nome": "Charizard", "tipo": "Fogo/Voador" },
    { "_id": ObjectId("pokemon_bulbasaur"), "nome": "Bulbasaur", "tipo": "Grama/Veneno" }
  ]
}
```

snappify.com

Inserindo Documentos no MongoDB;

○ ○ ○

```
db.treinadores.insertOne({
  "_id": ObjectId("treinador_ash"),
  "nome": "Ash Ketchum",
  "idade": 10,
  "pokemons": [
    { "_id": ObjectId("pokemon_pikachu"), "nome": "Pikachu", "tipo": "Elétrico" },
    { "_id": ObjectId("pokemon_charizard"), "nome": "Charizard", "tipo": "Fogo/Voador" },
    { "_id": ObjectId("pokemon_bulbasaur"), "nome": "Bulbasaur", "tipo": "Grama/Veneno" }
  ]
});
```

snappify.com

Consulta no MongoDB;

○ ○ ○

```
db.treinadores.find(
  { "nome": "Ash Ketchum" },
  { "pokemons.nome": 1, "pokemons.tipo": 1, "_id": 0 }
).pretty();
```


Neo4J: O Banco de Grafos

Neo4J é um banco de dados orientado a grafos, onde os dados são armazenados como nós e relações. Imagine um mapa de conexões entre treinadores e seus Pokémons, onde cada treinador (nó) está conectado aos seus Pokémons (outros nós) através de relações. Isso é ideal para modelar redes sociais, onde as conexões entre as entidades são tão importantes quanto as entidades em si.

Exemplo

Criando Nodos Para Treinadores e Pokémons;

```
CREATE (ash:Treinador {nome: 'Ash Ketchum', idade: 10})
CREATE (pikachu:Pokemon {nome: 'Pikachu', tipo: 'Elétrico'})
CREATE (charizard:Pokemon {nome: 'Charizard', tipo: 'Fogo/Voador'})
CREATE (bulbasaur:Pokemon {nome: 'Bulbasaur', tipo: 'Grama/Veneno'})
```

Criando Nós Para Treinadores e Pokémons;

```
MATCH (ash:Treinador {nome: 'Ash Ketchum'})
MATCH (pikachu:Pokemon {nome: 'Pikachu'})
MATCH (charizard:Pokemon {nome: 'Charizard'})
MATCH (bulbasaur:Pokemon {nome: 'Bulbasaur'})
CREATE (ash)-[:TREINA]->(pikachu)
CREATE (ash)-[:TREINA]->(charizard)
CREATE (ash)-[:TREINA]->(bulbasaur)
```

Consulta no Neo4J;

```
MATCH (ash:Treinador {nome: 'Ash Ketchum'})-[:TREINA]->(pokemon:Pokemon)
RETURN ash.nome, pokemon.nome, pokemon.tipo
```

04

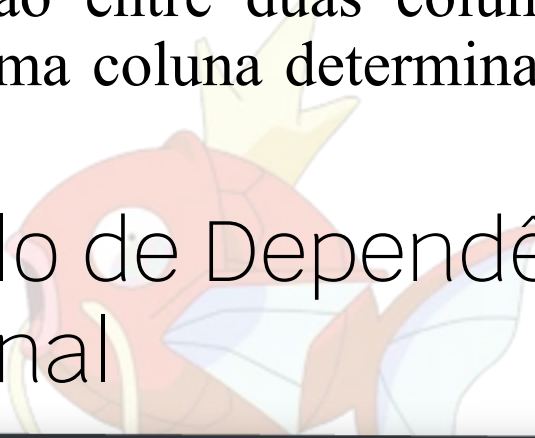
DEPENDENCIAS FUNCIONAIS

DEPENDENCIAS FUNCIONAIS

Teoria das Dependências

Suponha que temos uma tabela de Pokémons com atributos como nome, tipo e treinador. Uma dependência funcional é uma relação entre duas colunas, onde o valor de uma coluna determina o valor de outra.

Exemplo de Dependência Funcional



```
id_pokemon | nome      | tipo      | treinador
1           | Pikachu  | Elétrico  | Ash
2           | Bulbasaur | Planta    | Ash
3           | Charmander | Fogo     | Brock
```

Aqui, o nome do Pokémon depende unicamente do seu `id_pokemon`. Então, `id_pokemon` \rightarrow `nome`.

Etapas do Processo Para Encontrar Dependências Funcionais

1. Separação; Identificar as dependências funcionais básicas onde um atributo ou um conjunto de atributos determina outro(s).

Exemplo:

Dependência Funcional: TreinadorID → NomeTreinador, TimeTreinador

TreinadorID: Identificador único do treinador.

NomeTreinador: Nome do treinador.

TimeTreinador: Nome do time do treinador.

```
TreinadorID → NomeTreinador, TimeTreinador
```

2. Acumulação; Identificar dependências funcionais onde a combinação de atributos determina outro(s).

Exemplo:

Dependência Funcional: TreinadorID → TimeTreinador, portanto TreinadorID, TipoBatalha → TimeTreinador

TreinadorID: Identificador único do treinador.

TimeTreinador: Nome do time do treinador.

TipoBatalha: Tipo da batalha (por exemplo, ginásio, competição).

○ ○ ○

```
TreinadorID → TimeTreinador, portanto TreinadorID, TipoBatalha → TimeTreinador
```

3. Transitividade; Identificar dependências funcionais transitivas onde um atributo determina outro, que por sua vez determina um terceiro.

snapptify.com

Exemplo:

Dependência Funcional: PokémonID →
TipoPokémon & TipoPokémon →
VantagemBatalha, portanto PokémonID →
VantagemBatalha

PokémonID: Identificador único do Pokémon.

TipoPokémon: Tipo do Pokémon (por exemplo, Fogo, Água).

VantagemBatalha: Vantagem em batalhas contra tipos específicos.

○ ○ ○

PokémonID → TipoPokémon & TipoPokémon → VantagemBatalha, portanto PokémonID → VantagemBatalha

snappily.com

4. Pseudo-Transitividade; Identificar dependências funcionais onde um atributo determina outro, e a combinação deste segundo atributo com um terceiro atributo determina um quarto.

Exemplo:

Exemplo:

Dependência Funcional: PokémonID → TreinadorID & TreinadorID, GinásioID → VencedorBatalha, portanto PokémonID, GinásioID → VencedorBatalha

PokémonID: Identificador único do Pokémon.

TreinadorID: Identificador único do treinador.

GinásioID: Identificador único do ginásio.

VencedorBatalha: Determina se o treinador venceu a batalha no ginásio.

○ ○ ○

PokémonID → TreinadorID & TreinadorID, GinásioID → VencedorBatalha, portanto PokémonID, GinásioID → VencedorBatalha

snapify.com

05

TRANSAÇÕES EM BANCOS DE DADOS

TRANSAÇÕES EM BANCOS DE DADOS

Sistema de Gerenciamento de Bancos de Dados (SGBD)

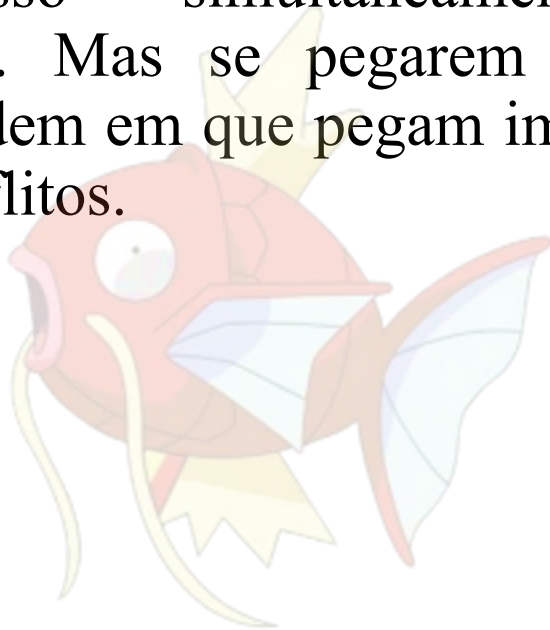
Os SGBDs processam operações simultâneas realizadas por múltiplos usuários, garantindo alta disponibilidade e tempos de resposta reduzidos. Sempre monitoram as transações quanto ao seu estado: ativa; em processo de efetivação, efetivada, em falha e em efetivação (concluída).

Execução Concorrente

A execução concorrente refere-se à capacidade de realizar múltiplas operações simultaneamente, aumentando a eficiência e a utilização dos recursos. Vamos usar uma analogia Pokémon para explicar a serialização por conflito:

- Itens diferentes: A ordem não importa.
- Mesmo item: A ordem importa.

Imagine duas transações de compras de itens no Pokémon Center. Se você e seu amigo pegarem itens diferentes, podem fazer isso simultaneamente sem problemas. Mas se pegarem o mesmo item, a ordem em que pegam importa para evitar conflitos.



06

SISTEMA CAP

O teorema CAP descreve as limitações dos sistemas distribuídos, forçando uma escolha entre Consistência, Disponibilidade e Tolerância a Partições.

Sistema CAP

Vamos usar Pokémons para ilustrar:

CA (Consistência e Disponibilidade): Sempre consistente, como um Pokémon com movimentos precisos.

AP (Disponibilidade e Tolerância a Partições): Alta disponibilidade, como um Pokémon que pode lutar em qualquer ambiente.

CP (Consistência e Tolerância a Partições): Consistência forte, garantindo que o Pokémon sempre execute o comando corretamente mesmo em um ambiente desafiador.

07

QUÓRUM DE LEITURA

Conceito fundamental em sistemas distribuídos, especialmente em bancos de dados NoSQL e sistemas de armazenamento distribuído. Ele é usado para garantir a consistência dos dados durante operações de leitura em um ambiente onde os dados são replicados em múltiplos nós.

QUÓRUM DE LEITURA

O que é?

Em sistemas distribuídos, os dados são frequentemente replicados em vários nós para melhorar a disponibilidade e a tolerância a falhas. No entanto, isso pode levar a problemas de consistência, onde diferentes réplicas podem ter versões diferentes dos dados. O quórum de leitura é uma técnica para mitigar esses problemas de consistência.

Como Funciona?

O conceito de quórum de leitura envolve definir um número mínimo de nós (réplicas) que devem concordar sobre o valor de um dado antes que ele seja retornado ao cliente. Isso ajuda a garantir que o valor lido seja consistente com o valor mais recente gravado.

Parâmetros Importantes

N (Número de Réplicas): O número total de réplicas dos dados.

R (Quórum de Leitura): O número mínimo de réplicas que devem responder a uma operação de leitura para que ela seja considerada bem-sucedida.

W (Quórum de Escrita): O número mínimo de réplicas que devem confirmar uma operação de escrita antes que ela seja considerada bem-sucedida.

Para garantir a consistência forte, a soma de R e W deve ser maior que N ($R + W > N$). Isso significa que há uma sobreposição entre as réplicas que participam da leitura e da escrita, garantindo que qualquer leitura incluirá pelo menos uma réplica que recebeu a última escrita.

Exemplo Prático

Imagine um sistema com três nós ($N = 3$). Podemos configurar o sistema com $W=2$ e $R=2$. Isso significa que:

Para uma operação de escrita ser bem-sucedida, pelo menos 2 dos 3 nós devem confirmar a escrita.

Para uma operação de leitura ser bem-sucedida, pelo menos 2 dos 3 nós devem responder.

Cenários de Consistência

Consistência Forte

Quando $R + W > N$, obtemos consistência forte. Por exemplo, com $N = 3$, $W = 2$ e

$R = 2$:

Escrita: Dois nós devem confirmar a escrita.

Leitura: Dois nós devem concordar sobre o valor lido.

Pelo menos um nó que participou da escrita mais recente estará entre os nós lidos.

Consistência Eventual

Quando $R + W \leq N$, obtemos consistência eventual. Por exemplo, com $N = 3$, $W = 1$ e $R = 1$:

Escrita: Apenas um nó precisa confirmar a escrita.

Leitura: Apenas um nó precisa responder.

Não há garantia de que o valor lido será o mais recente, mas eventualmente, todos os nós se tornarão consistentes.

Vantagens

Consistência: Garante que os dados lidos são consistentes e atualizados.

Resiliência a Falhas: Permite operações de leitura mesmo quando alguns nós estão indisponíveis, desde que o quórum seja alcançado.

Desvantagens

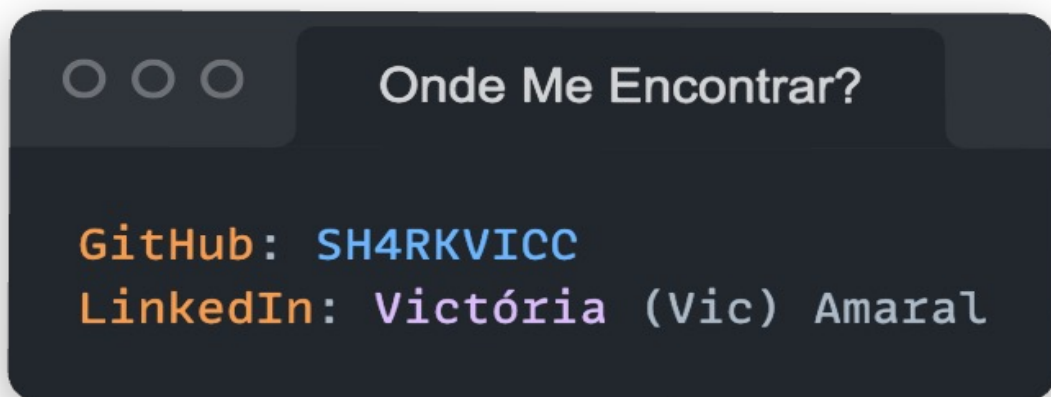
Latência: Pode aumentar a latência de leitura, já que mais nós precisam responder.

Complexidade: Requer um gerenciamento cuidadoso da configuração dos quóruns para equilibrar consistência, disponibilidade e desempenho.

AGRADECIMENTOS

AGRADEÇO A LEITURA!

Esse Ebook foi gerado por IA com base em meus resumos sobre realizados, e diagramados por humano. Este documento foi fundamental para meu aprendizado no Bootcamp DIO em que participei.



snappify.com

