
alpha
<ed/tech>
BANCOS DE DADOS
</>

Banco de dados Relacional

Chaves em PostgreSQL

Olá pessoal! Hoje vamos explorar um conceito fundamental em bancos de dados relacionais: **Chaves e relacionamentos**.

Se você está começando agora, não se preocupe! Vamos explicar tudo de forma clara e simples.

Chave Primária

Imagine uma sala de aula cheia de alunos. Cada aluno tem um número de identificação único, certo?

Essa é a chave primária! É como se fosse a identidade de cada aluno na sala. Ela nos ajuda a distinguir um aluno do outro de maneira rápida e eficiente.

Em uma lista de compras, cada item pode ter um código de barras único. Esse código seria a chave primária, pois nos permite identificar cada item individualmente na lista.

Resumindo, uma chave primária é um campo ou conjunto de campos que identifica exclusivamente cada registro em uma tabela. São características importantes:

- Conter valores únicos.
- Não conter valores nulos.
- A partir de um único campo ou vários campos combinados (Composta).

Agora vou precisar de ainda mais imaginação... Estamos criando uma biblioteca virtual de uma escola e provavelmente precisaremos de uma tabela de livros, onde o número de identificação do livro é a chave primária.

```
CREATE TABLE livros(
      id SERIAL,
      nome VARCHAR(50),
      categoria VARCHAR(100),
      descrição TEXT,
      ebook BYTEA, /*pdf*/
      PRIMARY KEY (id)
);
```

Banco de dados Relacional

Chaves em PostgreSQL

Chave Estrangeira

Ainda está comigo? Se sim, então deve ter percebido que além da lista de livros que temos, também precisaremos de uma lista alunos.

Como podemos estabelecer uma relação sobre qual aluno está lendo qual livro?

Aqui entra a chave estrangeira! Ela é como um elo entre duas listas diferentes. Com ela, podemos conectar o aluno à lista de livros que ele está lendo.

Pois bem, agora vamos analisar a chave estrangeira, ela é um campo em uma tabela que se refere à chave primária em outra tabela.

- Estabelece uma relação entre duas tabelas.
- Garante a integridade referencial.

```
CREATE TABLE alunos (
      id SERIAL PRIMARY KEY,
      nome VARCHAR(50),
      curso id INTEGER ,
      FOREIGN KEY (curso_id)
      REFERENCES cursos(id)
);
```

Em uma tabela de alunos, o campo "curso" pode ser uma chave estrangeira que referencia a tabela de cursos.

```
CREATE TABLE cursos (
      id SERIAL,
      nome VARCHAR(50),
      requisitos JSONB,
      grade JSONB,
      PRIMARY KEY (id)
);
```

Agora, pare um instante para imaginar como você resolveria aquele nosso problema dos livros...

A chave estrangeira estaria no livro referente ao aluno? Ou será no aluno referente ao livro?

Aqui entramos no nosso próximo tópico: **Relacionamentos**.

Relacionamentos

Que bom que chegamos bem até aqui, o momento que vamos falar de um relacionamento muito especial... Relacionamento entre Tabelas!

E qualquer um deles pode ser agrupado em um destes grupos:

- One-One: Relacionamento entre um carro e seu número de chassi. Cada carro tem um número de chassi único e vice-versa.
- One-Many: Relacionamento entre um país e suas cidades. Um país pode ter várias cidades, mas cada cidade pertence a apenas um país.
- Many-Many: Relacionamento entre alunos e disciplinas. Cada aluno pode estar matriculado em várias disciplinas e cada disciplina pode ter vários alunos.

Banco de dados Relacional

Relacionamentos SQL

Agora que sabemos os tipos de relacionamentos, podemos voltar ao nosso problema anterior...

Qual tipo de relação você escolheria? Vamos explorar na prática pressupondo o uso de cada uma dela:

Um para um: One-One

Vamos pressupor que nessa biblioteca, apenas um aluno pode pegar um, um único, livro por vez. Mas não mantemos registro de por quais alunos cada livro já foi pego.

```
CREATE TABLE alunos (
      id SERIAL PRIMARY KEY,
      nome VARCHAR(50),
      livro_id INTEGER FOREIGN KEY REFERENCES livros(id),
);
```

Um para muitos: One-Many

Agora mudaremos nossa perspectiva e vamos pressupor que nessa biblioteca, apenas um aluno pode pegar mais de um livro por vez.

```
CREATE TABLE livros(
      id SERIAL PRIMARY KEY,
      nome VARCHAR(50),
      categoria VARCHAR(100),
      descrição TEXT,
      ebook BYTEA, /*pdf*/
      livro_id INTEGER FOREIGN KEY REFERENCES alunos(id),
);
```

Muitos para muitos: Many-Many

Mudaremos novamente nossa perspectiva e finalmente vamos manter registro de por quais alunos cada livro já foi pego. Mas antes disso, vamos precisar criar outra tabela, que chamamos de "Tabela Auxiliar", ela será o nó que amarra as tabelas de livros e alunos.

```
CREATE TABLE alunos (
      id SERIAL PRIMARY KEY,
      nome VARCHAR(50),
);
```

```
CREATE TABLE livros(
      id SERIAL PRIMARY KEY,
      ...
);
```

```
CREATE TABLE livro_aluno (
      livro id INTEGER FOREIGN KEY REFERENCES livros(id),
      aluno_id INTEGER FOREIGN KEY REFERENCES alunos(id),
);
```

Banco de dados Relacional

Índices

Você agora entende os fundamentos das chaves e relacionamentos no PostgreSQL. Então vamos para outro assunto muito importante, mas não muito distante, o conceito de indices.

E não muito distante, se pensarmos na nossa biblioteca, como podemos consultar um livro de forma otimizada e eficiente? Se você respondeu "usando a chave primária", está correto! Chave primárias são indexadas! Porém, também podemos criar um índice para colunas que não são chaves.

Os Índices são estruturas de dados utilizadas para otimizar consultas em bancos de dados. Eles funcionam como um guia rápido para encontrar registros em tabelas grandes, permitindo que o sistema de gerenciamento de banco de dados (SGBD) localize os dados desejados de forma mais eficiente.

Tipos de Índices

1. **Índice B-Tree**

   O índice B-Tree é o tipo mais comum de índice utilizado no PostgreSQL. Ele organiza os dados em uma estrutura de árvore balanceada, onde cada nó contém um intervalo de valores e ponteiros para subárvores. Isso permite uma busca rápida e eficiente, ideal para colunas que são frequentemente utilizadas em cláusulas WHERE e ORDER BY.

   Suponha que temos uma tabela "produtos" com uma coluna "preço". Para criar um índice B-Tree nesta coluna, podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_preco ON produtos(preco);
```

   Isso criará um índice B-Tree na coluna "preço" da tabela "produtos", melhorando o desempenho de consultas que envolvem esta coluna, como consultas de produtos ordenados por preço.

2. **Índice Hash**

   O índice Hash organiza os dados em uma estrutura de tabela hash, onde os valores das colunas são transformados em chaves de hash, permitindo uma busca extremamente rápida em tabelas grandes. No entanto, o índice Hash é adequado apenas para consultas de equivalência, como consultas utilizando a cláusula WHERE com "=".

   Suponha que temos uma tabela "clientes" com uma coluna "cpf". Para criar um índice Hash nesta coluna, podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_cpf ON clientes USING HASH (cpf);
```

   Isso criará um índice Hash na coluna "cpf" da tabela "clientes", melhorando o desempenho de consultas que buscam um cliente específico pelo seu CPF.

Banco de dados Relacional

Índices

Tipos de Índices

3. Índice GiST (Generalized Search Tree)

O índice GiST é um tipo de índice que suporta várias formas de consultas, incluindo consultas espaciais e de texto. Ele é especialmente útil para consultas que envolvem tipos de dados complexos, como geometrias ou documentos de texto.

Suponha que temos uma tabela "enderecos" com uma coluna "localizacao" do tipo geometria. Para criar um índice GiST nesta coluna, podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_localizacao ON enderecos USING GiST (localizacao);
```

Isso criará um índice GiST na coluna "localizacao" da tabela "enderecos", melhorando o desempenho de consultas espaciais, como encontrar todos os endereços próximos a uma determinada coordenada.

4. Índice GIN (Generalized Inverted Index)

O índice GIN é adequado para consultas que envolvem tipos de dados complexos, como arrays e texto. Ele cria uma estrutura de índice invertido que permite a busca eficiente de valores dentro destes tipos de dados.

Suponha que temos uma tabela "posts" com uma coluna "tags" contendo um array de tags. Para criar um índice GIN nesta coluna, podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_tags ON posts USING GIN (tags);
```

Isso criará um índice GIN na coluna "tags" da tabela "posts", permitindo consultas eficientes por posts que contenham determinadas tags.

5. Índice BRIN (Block Range Index)

O índice BRIN é uma opção eficiente para tabelas grandes e para consultas que envolvem colunas que possuem correlação com a ordem física dos dados. Ele divide a tabela em blocos e mantém informações sobre o intervalo de valores em cada bloco, permitindo uma busca rápida em tabelas de grande volume de dados.

Suponha que temos uma tabela "registros" com uma coluna "data" que representa a data de registro. Para criar um índice BRIN nesta coluna, podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_data ON registros USING BRIN (data);
```

Isso criará um índice BRIN na coluna "data" da tabela "registros", o que pode melhorar o desempenho de consultas que buscam registros em intervalos de datas.

Banco de dados Relacional

Índices

Clusters

No PostgreSQL, os índices B-Tree são geralmente não agrupados, o que significa que os dados na tabela física não são organizados na mesma ordem do índice. No entanto, é possível criar um índice agrupado (clustered index), onde os dados na tabela física são organizados na mesma ordem do índice, o que pode melhorar o desempenho de consultas que exploram a ordenação dos dados.

Suponha que queremos criar um índice agrupado na coluna "data" da tabela "registros". Podemos usar o seguinte comando SQL:

```
CREATE INDEX idx_data ON registros(data) CLUSTER;
```

Isso criará um índice agrupado na coluna "data" da tabela "registros", organizando os dados na mesma ordem do índice.

Monitoramento e Manutenção

É importante monitorar e manter os índices em um banco de dados PostgreSQL para garantir um bom desempenho das consultas. Isso pode envolver identificar índices não utilizados, reconstruir índices fragmentados e atualizar estatísticas de índice regularmente.

Podemos usar as visualizações do catálogo do PostgreSQL, como "pg_stat_user_indexes" e "pg_statio_user_indexes", para monitorar a utilização e o desempenho dos índices. Além disso, podemos usar o comando "REINDEX" para reconstruir um índice e o comando "VACUUM" para atualizar estatísticas de índice.

Estes conceitos são de fundamental importância para construirmos um banco de dados eficiente e performático, voltado a escalabilidade e que irá comportar inúmeros usuários!

Então é importante que você saiba que os índices desempenham um papel crucial na otimização de consultas em bancos de dados.

Ao entender os diferentes tipos de índices disponíveis e como utilizá-los de forma eficiente, é possível melhorar significativamente o desempenho das suas aplicações e consultas de banco de dados.

Banco de dados Relacional

Comandos, Cláusulas e Funções
Parte 1

Vamos continuar nossa jornada explorando os principais comandos e cláusulas utilizados no PostgreSQL!

INSERT

O comando INSERT é usado para adicionar novos registros a uma tabela.

Imagine uma tabela como uma planilha, onde cada linha representa um registro e cada coluna representa um campo de informação.

O comando INSERT nos permite inserir novas linhas nessa planilha.

Suponha que temos uma tabela chamada "clientes" com colunas como "id", "nome" e "email".

Para adicionar um novo cliente, podemos usar o comando INSERT da seguinte forma:

```
INSERT INTO clientes (nome, email)
VALUES ('João', 'joao@example.com');
```

Este comando adicionará um novo registro à tabela "clientes" com o nome "João" e o email "joao@example.com".

SELECT

A cláusula SELECT é usada para recuperar dados de uma ou mais tabelas.

É como se estivéssemos fazendo uma pergunta ao banco de dados, solicitando que nos forneça informações específicas com base em determinados critérios.

Suponha que desejamos recuperar todos os registros da tabela "clientes".

Podemos usar a cláusula SELECT da seguinte forma:

```
INSERT INTO clientes (nome, email)
VALUES ('João', 'joao@example.com');
```

Este comando retornará todos os registros da tabela "clientes", mostrando todas as colunas (representadas pelo asterisco "*").

Banco de dados Relacional

Comandos, Cláusulas e Funções
Parte 1

SELECT

Nós já aprendemos a indexar colunas das tabelas e através das consultas poderemos usufruir da performance e eficiencia deles.

Então veremos exemplos práticos de consultas que se beneficiam de índices B-Tree, Hash, GiST, GIN e BRIN, demonstrando como cada tipo de índice pode ser aplicado em situações específicas para otimizar o desempenho das consultas

1. **Índice B-Tree:**
Suponha que temos uma tabela "produtos" com uma coluna "nome" e um índice B-Tree nesta coluna. Podemos usar o índice para realizar uma consulta por igualdade da seguinte forma:

```
SELECT * FROM produtos WHERE nome = 'ProdutoX';
```

2. **Índice Hash**
Suponha que temos uma tabela "clientes" com uma coluna "cpf" e um índice Hash nesta coluna. Podemos usar o índice para realizar uma consulta por igualdade da seguinte forma:

```
SELECT * FROM clientes WHERE cpf = '123.456.789-10';
```

3. **Índice GiST**
Suponha que temos uma tabela "enderecos" com uma coluna "localizacao" do tipo geometria e um índice GiST nesta coluna. Podemos usar o índice para realizar uma consulta por proximidade da seguinte forma:

```
SELECT * FROM enderecos
WHERE localizacao <-> 'POINT(40.7128, -74.0060)' < 0.01;
```

4. **Índice GIN**
Suponha que temos uma tabela "posts" com uma coluna "tags" contendo um array de tags e um índice GIN nesta coluna. Podemos usar o índice para realizar uma consulta por uma tag específica da seguinte forma:

```
SELECT * FROM posts WHERE tags @> ARRAY['tecnologia'];
```

5. **Índice BRIN**
Suponha que temos uma tabela "registros" com uma coluna "data" que representa a data de registro e um índice BRIN nesta coluna. Podemos usar o índice para realizar uma consulta por intervalo de datas da seguinte forma:

```
SELECT * FROM registros
WHERE data BETWEEN '2024-01-01' AND '2024-01-31';
```

Banco de dados Relacional

Comandos, Cláusulas e Funções
Parte 1

WHERE

A cláusula WHERE é usada para filtrar os resultados de uma consulta com base em condições específicas.

É como se estivéssemos aplicando um filtro aos dados para obter apenas as informações que atendem a certos critérios.

Suponha que queiramos recuperar apenas os clientes com o nome "Maria".

Podemos usar a cláusula WHERE da seguinte forma:

```
SELECT * FROM clientes WHERE nome = 'Maria';
```

Este comando retornará apenas os registros da tabela "clientes" onde o nome é igual a "Maria".

JOIN

A cláusula JOIN é usada para combinar dados de duas ou mais tabelas com base em uma condição relacionada entre elas.

É como se estivéssemos juntando diferentes conjuntos de dados para obter uma visão mais completa.

Suponha que temos duas tabelas, "clientes" e "pedidos", e queremos obter os detalhes dos clientes juntamente com os detalhes de seus pedidos.

Podemos usar a cláusula JOIN da seguinte forma:

```
SELECT clientes.nome, pedidos.produto
FROM clientes
JOIN pedidos ON clientes.id = pedidos.cliente_id;
```

Este comando combinará os dados das tabelas "clientes" e "pedidos" com base na correspondência dos IDs dos clientes nos pedidos.

Banco de dados Relacional

Comandos, Cláusulas e Funções
Parte 1

ARRAY_AGG()

A função ARRAY_AGG() é usada para agregar valores em um array.

Ela é útil quando queremos agrupar dados em uma única linha, especialmente quando estamos lidando com valores relacionados.

Suponha que queremos listar todos os produtos comprados por cada cliente.

Podemos usar a função ARRAY_AGG() da seguinte forma:

```
SELECT clientes.nome, ARRAY_AGG(pedidos.produto) AS
produtos comprados
FROM clientes
JOIN pedidos ON clientes.id = pedidos.cliente_id
GROUP BY clientes.nome;
```

Este comando agregará os produtos comprados por cada cliente em um array, facilitando a visualização dos dados.

Hoje aprendemos muitos conceitos importantes para bancos de dados, é muito importante que você entendam e explorem estes conceitos.

Não tenham medo de praticar e buscar mais conhecimento, uma recomendação é a documentação oficial do Postgres e do Postgres Pro.

Obrigado e até a próxima aula!