



Introdução a 'Otimização de Consultas no Banco de Dados'

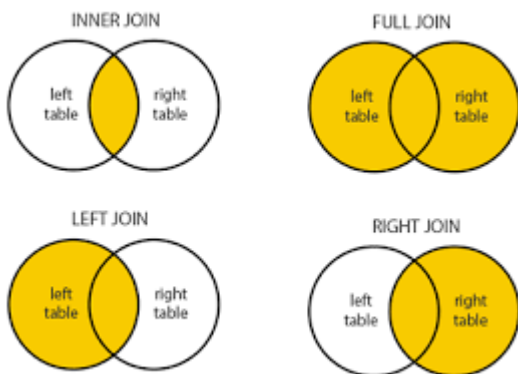
Até agora vimos as aplicações da sintaxe do SQL em consultas de dados em uma única tabela. Embora seja bastante útil, o cenário real é bem mais complexo. Na prática, uma empresa pode ter dezenas, centenas de tabelas em seu banco de dados.

Hoje em dia, com o paradigma de microsserviços em alta, uma empresa em geral possui dezenas de bancos de dados em sua arquitetura e não há qualquer obrigação desses bancos serem todos Postgres ou sequer relacionais. O cenário de múltiplos bancos heterogêneos é bastante comum.

Aqui vamos mostrar como fazemos para conectar dados que estão espalhados por mais de uma tabela num banco de dados relacional.

Vamos começar pelos mais famoso representante dessa categoria: **JOINS**!

Join



O diagrama acima ilustra, usando a linguagem da Teoria dos Conjuntos, a lógica por trás de cada um dos principais tipos de **JOINS**. Vamos ver cada um deles mais de perto.

INNER JOIN

Com o **INNER JOIN**, conectamos duas tabelas pelo que há de comum em ambas (*através das chaves usadas, claro; neste caso, as `customer_id`, colunas presentes em ambas as tabelas*). Abaixo, ligamos as tabelas **Orders** e **Customers** via este tipo de join.

```
SELECT Orders.order_id,  
       Customers.contact_name  
FROM Orders  
INNER JOIN Customers ON Orders.customer_id = Customers.customer_id;
```

Ou, o que é uma boa prática, usando alias para as tabelas:

```
SELECT o.order_id,  
       c.contact_name  
FROM Orders AS o  
INNER JOIN Customers AS c ON o.customer_id = c.customer_id;
```

LEFT JOIN

Com o **LEFT JOIN**, obtemos tudo que está na primeira tabela e, deste conjunto, o que der match na segunda tabela. A sintaxe é praticamente idêntica à usada no caso do join anterior:

```
SELECT o.order_id,  
       c.contact_name  
FROM Orders AS o  
LEFT JOIN Customers AS c ON o.customer_id = c.customer_id;
```

RIGHT JOIN

Neste caso, obtemos tudo que está na segunda tabela e, deste conjunto, o que der match na primeira tabela - o contrário do **LEFT JOIN** (*por isto, em geral, não há por que usar o RIGHT JOIN se você pode usar o LEFT JOIN - de fato é isto que ocorre na prática*).

```
SELECT o.order_id,  
       c.contact_name  
FROM Orders AS o  
RIGHT JOIN Customers AS c ON o.customer_id = c.customer_id;
```

FULL JOIN

Também descrito como **FULL OUTER JOIN**. Com ele consultamos tudo o que está em ambas as tabelas - ou, na linguagem da Teoria de Conjuntos, a união dos seus conteúdos.

```
SELECT o.order_id,  
       c.contact_name  
FROM Orders AS o  
FULL JOIN Customers AS c ON o.customer_id = c.customer_id;
```

Desses todos, os dois primeiros, os **INNER JOIN** e **LEFT JOIN**, são provavelmente os mais usados na prática.

Além disso, embora as operações acima estejam descritas para duas tabelas, elas foram concebidas para conectar um número arbitrário delas, mas sempre aos pares, repetindo a estrutura.

UNIONS

As operações de **UNION** no **SQL** servem para concatenar verticalmente os conteúdos de duas tabelas.

UNION

A operação de **UNION**, mantém apenas valores distintos das colunas exibidas em ambas as tabelas. Sua sintaxe é:

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers;
```

A query acima constrói uma tabela (*que, neste caso, é de apenas uma coluna, City*), na qual estão listadas todas as cidades que aparecem em ambas as colunas **City** das duas tabelas, **Customers** e **Suppliers**, sem repetir nomes de cidades.

UNION ALL

A **UNION ALL** é bastante semelhante à operação anterior, com a diferença de que não se preocupa em ter elementos únicos (*no exemplo abaixo, cidades únicas*). Ela simplesmente "empilha" os registros das duas tabelas.

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
```

Stored Procedures

Stored Procedure (*Procedimento Armazenado*) é um conjunto de comandos em **SQL** que podem ser executados de uma só vez, como em uma função. Ele armazena tarefas repetitivas e aceita parâmetros de entrada para que a tarefa seja efetuada de acordo com a necessidade individual.

Um **Stored Procedure** pode reduzir o tráfego na rede, melhorar a performance de um banco de dados, criar tarefas agendadas, diminuir riscos e criar rotinas de processamento.

Em muitos **SGDBs** temos o conceito de **Stored Procedures**, programas desenvolvidos em uma determinada linguagem de script e armazenados no servidor, onde serão processados. No **PostgreSQL**, as **Stored Procedures** são conhecidas com o nome de **Functions**.

Criando um Stored Procedure

Como exemplo, criaremos uma função que retorna o **NomeContato** que atende a condição **order_id = 10831**:

```
CREATE FUNCTION NomeContato() RETURNS SETOF VARCHAR AS
    'SELECT Customers.contact_name FROM Orders
    INNER JOIN Customers ON Orders.customer_id = Customers.customer_id
    WHERE Orders.order_id = 10831;'
LANGUAGE SQL;
```

Consultando a **stored procedure**:

```
SELECT NomeContato()
```

Views

Views são consideradas pseudo-tables, ou seja, elas são usadas junto a instrução SELECT para apresentar subconjuntos de dados presentes em tabelas reais.

Assim, podemos apresentar as colunas e linhas que foram selecionadas da tabela original ou associada. E como as Views possuem permissões separadas, podemos utilizá-las para restringir mais o acesso aos dados pelos usuários, para que veja apenas o que é necessário.

Criando uma View

Como exemplo, criaremos uma view da tabela **employees** restringindo a consulta, ou seja, apresentando apenas as colunas **employee_id**, **last_name**, **first_name**, **title** e **city**.

```
CREATE VIEW view_employees
AS SELECT employee_id, last_name, first_name, title, city
FROM employees;
```

Consultando a **view**:

```
select * from view_employees;
```

Index

Ao trabalharmos com bancos de dados, temos a necessidade de apresentar resultados com tamanha eficiência e rapidez, no entanto, chega um determinado momento em que o desempenho da base de dados cai, deixando de ser satisfatório dessa forma.

Eis que quando isto acontece, um recurso bastante utilizado para a resolução desse problema é a utilização da **indexação no banco de dados**.

Quando estamos lidando com **SGDBs**, como é o caso do **PostgreSQL**, temos que o **índice** é uma “cópia” do item que desejamos combinar com uma referência à localização real dos dados.

Quando realizamos buscas nas tabelas sem a utilização de **índices**, dependendo da quantidade de registros, podemos perceber que a busca é um pouco lenta, pois dessa forma, a pesquisa é realizada de forma sequencial.

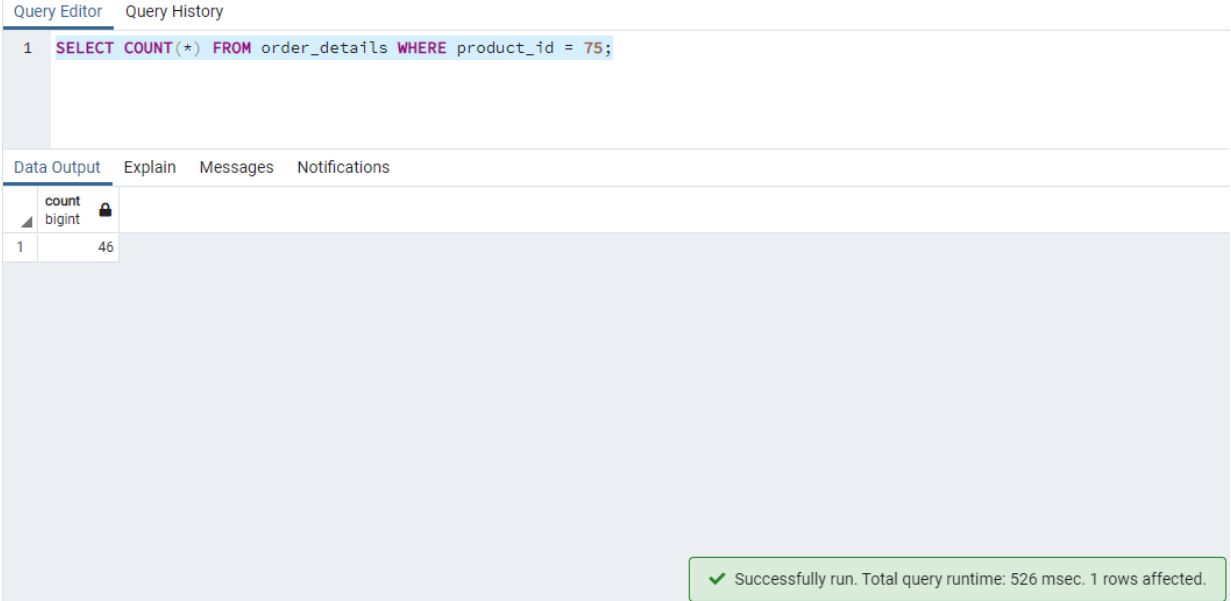
Quando dizemos que existe uma pesquisa sequencial, estamos nos referindo a uma busca linha a linha em toda a tabela (*ou conjunto de tabelas*) da base de dados com o intuito de obter a informação necessária.

Criando um Índice

Antes de criar o índice vamos realizar a seguinte consulta:

```
SELECT COUNT(*) FROM order_details WHERE product_id = 75;
```

Note que o retorno levou *526msec*.



Query Editor Query History

```
1 SELECT COUNT(*) FROM order_details WHERE product_id = 75;
```

Data Output Explain Messages Notifications

	count bigint
1	46

✓ Successfully run. Total query runtime: 526 msec. 1 rows affected.


Como exemplo, criaremos um índice que chamamos de **order_details_index**, presente na coluna **product_id** da tabela **order_details**.

```
CREATE INDEX order_details_index ON order_details (product_id);
```

Para consultar utilizamos o mesmo comando:

```
SELECT COUNT(*) FROM order_details WHERE product_id = 75;
```

Note que o retorno levou *364msec*.

northwind/letscode@letscode.c5kdoqkvyzml.us-east-2.rds.amazonaws.com

Query EditorQuery History

1 SELECT COUNT(*) FROM order_details WHERE product_id = 75;

Data OutputExplainMessagesNotifications

	count bigint
1	46

✓ Successfully run. Total query runtime: 364 msec. 1 rows affected.

Como podemos ver, o resultado da consulta com o índice levou alguns ms a menos, sendo assim um pouco mais rápida em relação a utilização das consultas sem índices. A pouca diferença ocorre devido ao fato de termos poucos registros na base de dados, mas considerem uma base em que tenhamos milhares de registros e perceberam que o ganho em tempo de consulta se torna

< Tópico anterior

Próximo Tópico >

Referências