



Digital
College

FORMAÇÃO EM

DATA ANALYTICS

UNIDADE 1:

BANCO DE DADOS

MÓDULO 3:

> SQL PARA ANÁLISE DE DADOS >



Sumário

1. Estrutura do banco de dados de exemplo	03
2. Subconsultas	06
2.1. O que é uma Subconsulta SQL (Subquery)?	06
2.2. Regras gerais para aplicação de subconsultas	06
2.3. Sintaxe Básica de uma Subconsulta SQL	06
2.3.1. Subconsulta com operação SELECT	07
2.3.2. Subconsulta como uma nova coluna da consulta	07
2.3.3. Subquery como uma nova coluna da consulta	10
2.3.4. Subquery como filtro de uma nova consulta	12
2.3.5. Subquery como fonte de dados de uma consulta principal	15
2.3.6. Subconsulta com operação INSERT INTO	17
2.3.7. Subconsulta com operação UPDATE	19
2.3.8. Subconsulta com operação DELETE	19
3. Operadores UNION EXCEPT e INTERSECT	20
3.1. Operador UNION	21
3.2. Operador UNION ALL	22
3.3. Operador INTERSECT	22
3.4. Operador EXCEPT	23
4. Views	24
4.1. Trabalhando com as Views materializadas	29
5. Triggers	33
5.1. Trigger function	34
5.2. 2 passos para escrever uma trigger no PostgreSQL	34
5.3. Visibilidade dos dados nas Triggers	36
Referências bibliográficas	40

1. Estrutura do banco de dados de exemplo

Para ilustração dos exemplos a seguir, considere a estrutura de tabelas apresentada na Figura 1, onde temos a tabela projetos (Tabela 1), a tabela comentários (Tabela 2), a tabela usuários (Tabela 3), a tabela likes_por_projeto (Tabela 4) e a tabela likes_por_comentario (Tabela 5).

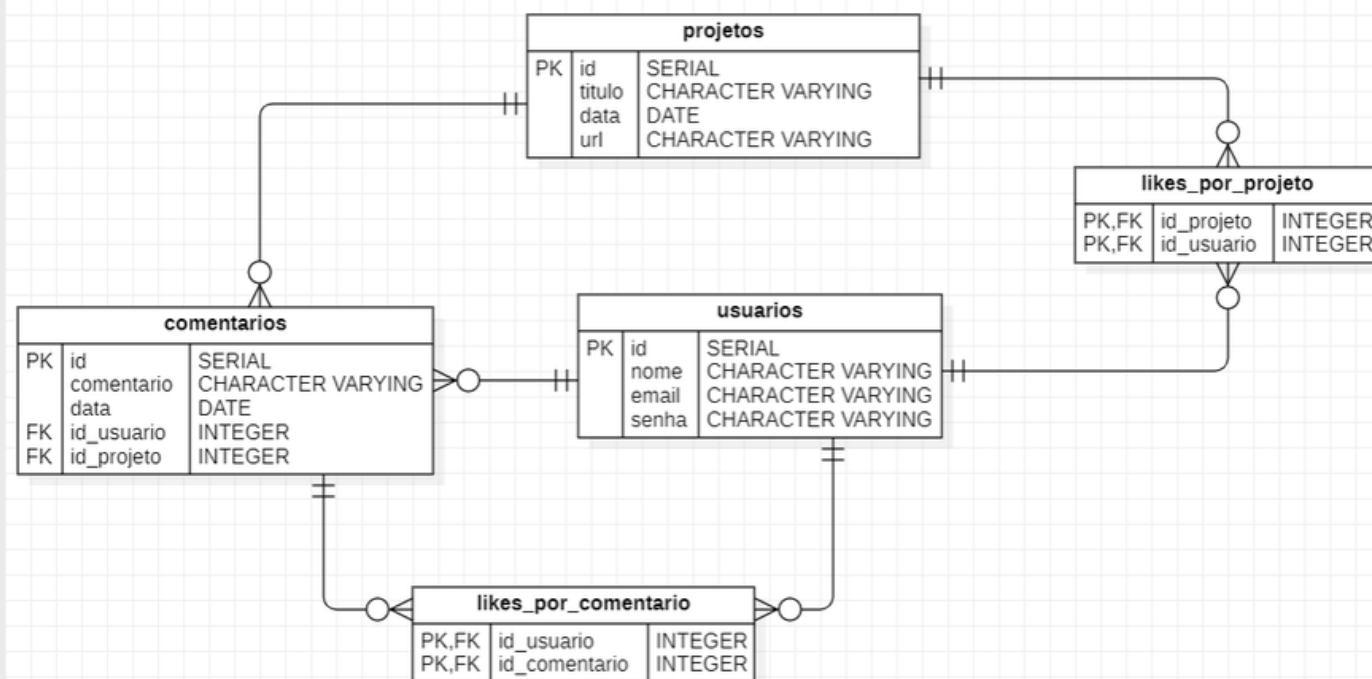


Figura 1. Modelagem do banco

id	título	data
1	Aplicação C#	2018-04-01
2	Aplicação Ionic	2018-05-07
3	Aplicação Python	2018-08-05

Tabela 1. Estrutura da tabela projetos



id	comentário	id_projeto	id_usuario
1	A Microsoft acertou com essa linguagem!	1	1
2	Parabéns pelo projeto! Bem legal!	1	3
3	Superinteressante! Fácil e rápido!	2	4
4	Cara, que simples fazer um app assim!	2	1
5	Linguagem muito diferente.	3	3
6	Adorei aprender Python! Parabéns!	3	2
7	Muito maneiro esse framework!	2	2

Tabela 2. Estrutura da tabela comentários

id	comentário	id_projeto	id_usuario
1	Bruna Luiza	bruninha@gmail.com	abc123.
2	Thiago Braga	thiagobraga_1@hotmail.com	pena093
3	Osvaldo Justino	osvaltino@yahoo.com.br	osvaldit1_s
4	Gabriel Fernando	gabriel_fnd@gmail.com	gabss34

Tabela 3. Estrutura da tabela Usuários



id_projeto	id_usuario
1	1
1	3
2	1
2	2
2	3
2	4
3	2

Tabela 4. Estrutura da tabela likes_por_projeto

id_projeto	id_usuario
7	1
7	2
7	4

Tabela 5. Estrutura da tabela likes_por_comentarios

2. Subconsultas

>> 2.1. O que é uma Subconsulta SQL (Subquery)?

Uma subconsulta é uma consulta embutida dentro de outra consulta, de forma aninhada, passando os resultados da consulta mais interna para a consulta mais externa por meio de uma cláusula WHERE ou de uma cláusula HAVING.

Desta forma, é possível restringir mais ainda os dados retornados por uma consulta, permitindo a criação de filtros bastante aprimorados. A subconsulta retorna os dados que serão empregados pela consulta “principal”, alimentando-a com informações a serem utilizadas como condições de filtragem.

Podemos utilizar subconsultas não apenas em consultas com a cláusula SELECT, mas também em operações INSERT, UPDATE e DELETE.

Quando usamos uma subconsulta em uma query, a subconsulta é resolvida primeiro, e, então, a consulta externa (principal) é resolvida de acordo com o resultado retornado pela subconsulta (subquery).

>> 2.2. Regras gerais para aplicação de subconsultas

Algumas regras gerais precisam ser seguidas para que seja possível empregar subconsultas. Entre elas, temos:

- A subconsulta pode ter apenas uma única coluna em sua cláusula SELECT, exceto quando a consulta principal tiver múltiplas colunas para comparação com as colunas selecionadas;
- Se a subconsulta retornar mais de uma linha de dados, é necessário usar operadores de valores múltiplos, como o operador lógico IN;
- As subconsultas são escritas dentro de parênteses;
- Não é possível usar a cláusula ORDER BY em uma subconsulta – mas na consulta principal pode;
- É possível usar uma cláusula GROUP BY em uma subconsulta;
- Não podemos usar o operador BETWEEN com uma subconsulta (na consulta principal), mas podemos usar esse operador dentro da subconsulta.

>> 2.3. Sintaxe Básica de uma Subconsulta SQL

A sintaxe básica para a criação de uma subconsulta SQL é a seguinte:

```
SELECT coluna(s)
FROM tabela(s)
WHERE coluna operador (SELECT coluna
                        FROM tabela WHERE condições);
```

Onde “operador” pode ser qualquer operador lógico ou relacional, como >, <, >=, <=, =, <>, IN, NOT, AND, OR etc

>> 2.3.1. Subconsulta com operação SELECT

Vamos efetuar uma consulta à tabela de “comentários” de nosso banco de dados db_digital, retornando os comentários publicados pelo projeto Aplicação Python, porém usando uma subconsulta em vez de usar INNER JOIN:

```
SELECT comentario
FROM comentarios
WHERE id_projeto =
      (SELECT id
       FROM projetos
       WHERE titulo = 'Aplicação Python');
```

Listagem 1. Consulta SQL

>> 2.3.2. Subconsulta como uma nova coluna da consulta

Uma das formas possíveis de *realizar uma subquery* é fazendo com que o resultado de outra consulta seja uma coluna dentro da sua consulta principal.

Como exemplo, buscaremos o título de todos os projetos cadastrados e adicionaremos uma coluna com a quantidade de comentários existentes em cada projeto, realizando, assim, uma consulta principal na tabela projetos e uma subconsulta na tabela comentários, que gerará uma nova coluna.

Veja o resultado esperado na Tabela 6.

título	Quantidade_Comentarios
Aplicação C#	2
Aplicação Ionic	3
Aplicação Python	2

Tabela 6. Resultado da query

No resultado acima, existem duas colunas: título e Quantidade_Comentarios. Essa tabela foi obtida por meio da consulta SQL da Listagem 2.

```
SELECT
    P.titulo,
    (SELECT
        COUNT(C.id_projeto)
    FROM
        comentarios C
    WHERE
        C.id_projeto = P.id ) AS Quantidade_Comentarios
FROM
    projetos P
GROUP BY
    P.id
```

Listagem 2. Consulta SQL

Observe na query acima que a consulta principal é feita na tabela projetos, porém, na seleção das colunas que virão no resultado, existe uma outra consulta – essa na tabela comentários – responsável por trazer o total de ocorrências do ID do projeto específico na tabela comentários. Essa coluna foi nomeada de Quantidade_Comentarios.



Podemos utilizar o trecho de código da Listagem 3 para adicionar mais uma coluna a esta consulta. Adicionaremos, por exemplo, o valor total de likes recebidos por projeto.

```
SELECT
    P.titulo,
    (SELECT
        COUNT(C.id_projeto)
    FROM
        comentarios C
    WHERE
        C.id_projeto = P.id ) AS Quantidade_Comentarios,
    (SELECT
        COUNT(LP.id_projeto)
    FROM
        likes_por_projeto LP
    WHERE
        LP.id_projeto = P.id ) AS Quantidade_Likes
FROM
    projetos P
GROUP BY
    P.id
```

Listagem 3. Consulta do valor total de likes recebidos por projeto

Veja o resultado da query acima na Tabela 6.

título	Quantidade_Comentarios	Quantidade_likes
Aplicação C#	2	2
Aplicação Ionic	3	4
Aplicação Python	2	1

Tabela 6. Resultado da query

Observe que no resultado acima existem três colunas, sendo que apenas a coluna título faz parte da tabela projetos. As colunas Quantidade_Comentarios e Quantidade_likes são providas pelas duas subqueries utilizadas em tabelas diferentes: comentarios e likes_por_projeto, respectivamente. Sempre após a criação de uma subquery como nova coluna, será necessário definir um nome para esta coluna, por meio da palavra reservada AS. Uma subquery como nova coluna deve retornar apenas uma única coluna com um único valor.

>> 2.3.3. Subquery como uma nova coluna da consulta

Uma das formas possíveis de *realizar uma subquery* é fazendo com que o resultado de outra consulta seja uma coluna dentro da sua consulta principal.

Como exemplo, buscaremos o título de todos os projetos cadastrados e adicionaremos uma coluna com a quantidade de comentários existentes em cada projeto, realizando, assim, uma consulta principal na tabela projetos e uma subconsulta na tabela comentários, que gerará uma nova coluna.

Veja o resultado esperado na Tabela 7.

título	Quantidade_Comentarios
Aplicação C#	2
Aplicação Ionic	3
Aplicação Python	2

Tabela 7. Resultado da query

No resultado acima existem duas colunas: título e Quantidade_Comentarios. Essa tabela foi obtida por meio da consulta SQL da Listagem 2.

```
SELECT
    P.titulo,
    (SELECT
        COUNT(C.id_projeto)
    FROM
        comentarios C
    WHERE
        C.id_projeto = P.id ) AS Quantidade_Comentarios,
FROM
    projetos P
GROUP BY
    P.id
```

Listagem 2. Consulta SQL

Observe na query acima que a consulta principal é feita na tabela projetos, porém, na seleção das colunas que virão no resultado, existe uma outra consulta – essa na tabela comentários – responsável por trazer o total de ocorrências do ID do projeto específico na tabela comentários. Essa coluna foi nomeada de Quantidade_Comentarios.

Podemos utilizar o trecho de código da Listagem 3 para adicionar mais uma coluna a esta consulta. Adicionaremos, por exemplo, o valor total de likes recebidos por projeto.

```
SELECT
    P.titulo,
    (SELECT
        COUNT(C.id_projeto)
    FROM
        comentarios C
    WHERE
        C.id_projeto = P.id ) AS Quantidade_Comentarios,
    (SELECT
        COUNT(LP.id_projeto)
    FROM
        likes_por_projeto LP
    WHERE
        LP.id_projeto = P.id ) AS Quantidade_Likes
FROM
    projetos P
GROUP BY
    P.id
```

Listagem 3. Consulta do valor total de likes recebidos por projeto

Veja o resultado da query acima na Tabela 8.

título	Quantidade_Comentarios	Quantidade_likes
Aplicação C#	2	2
Aplicação Ionic	3	4
Aplicação Python	2	1

Tabela 8. Resultado da query

Observe que no resultado acima existem três colunas, sendo que apenas a coluna título faz parte da tabela projetos. As colunas Quantidade_Comentarios e Quantidade_likes são providas pelas duas subqueries utilizadas em tabelas diferentes: comentarios e likes_por_projeto, respectivamente.

Sempre após a criação de uma subquery como nova coluna, será necessário definir um nome para esta coluna, por meio da palavra reservada AS. Uma subquery como nova coluna deve retornar apenas uma única coluna com um único valor.

>> 2.3.4. Subquery como filtro de uma nova consulta

Outro exemplo da utilização de subqueries é fazendo filtros no resultado de outras consultas. Para esse modelo podemos utilizar as cláusulas IN, EXISTS ou operadores de comparação, como =, >=, <=, dentre outros.

Para exemplificar, buscaremos todos os projetos que possuam algum comentário, ou seja, uma consulta principal na tabela projetos, e filtraremos o resultado com base no resultado da subconsulta na tabela comentarios.

Veja ao resultado esperado na Tabela 9.

id	título	data
1	Aplicação C#	2018-04-01
2	Aplicação Ionic	2018-05-07
3	Aplicação Python	2018-08-05

Tabela 9. Resultado da query

Observe no resultado acima que vieram apenas informações da tabela projetos, mas que foram filtradas com base em uma pesquisa pelo ID do projeto na tabela comentarios. Cada projeto listado neste resultado é um projeto que possui algum comentário. Essa informação foi obtida através da query da Listagem 4.



```
SELECT
    P.id,
    P.titulo,
    P.data
FROM
    projetos P
WHERE
    P.id IN
    (
        SELECT
            C.id_projeto
        FROM
            comentarios C
        WHERE
            P.id = C.id_projeto
    );
```

Listagem 4. Subquery que gerou a Tabela 8

Observe que, na query acima, a consulta principal é feita na tabela projetos, mas que em seguida utilizamos a cláusula WHERE para realizar um filtro na seleção, definindo através da cláusula IN que o valor da coluna ID deve estar incluso no resultado da subconsulta realizada, que, no caso, é feita na tabela comentários.

Uma subquery utilizada como filtro de uma consulta principal pode retornar N valores, porém, apenas uma única coluna.

Podemos utilizar uma subquery como filtro com a cláusula EXISTS. Para exemplificar, podemos realizar a mesma consulta anterior e buscar todos os projetos que possuam algum comentário, como vemos na Listagem 5.



```
SELECT
    P.id,
    P.titulo,
    P.data
FROM
    projetos P
WHERE
    EXISTS
    (
        SELECT
            C.id_projeto
        FROM
            comentarios C
        WHERE
            P.id = C.id_projeto
    );
```

Listagem 5. Subquery com o uso de EXISTS

Na query acima, continuamos com o mesmo resultado que a query anterior, mudando apenas a forma como é feita a subquery. No caso da cláusula EXISTS, a query principal realiza uma verificação se o resultado da segunda query conseguiu encontrar algum valor, e em caso positivo, retorna esse item, ou seja, ao encontrar um comentário para algum projeto, os dados solicitados na consulta principal a respeito desse projeto serão exibidos.

Outro exemplo do mesmo modelo de subquery seria utilizando operadores lógicos. Logo, buscaremos agora o título e a data do último projeto que recebeu likes, ou seja, uma consulta principal na tabela projetos com um filtro na tabela likes, como vemos na Listagem 6.

```
SELECT
    P.titulo,
    P.data
FROM
    projetos P
WHERE
    P.id = (SELECT
        MAX(LP.id_projeto)
    FROM
        likes_por_projeto LP);
```

Listagem 6. Subquery com o uso de operadores lógicos



Observe que, na query acima, a consulta principal é feita na tabela projetos e o filtro realizado por meio da cláusula WHERE é que o ID do projeto seja IGUAL ao valor retornado pela subquery, que busca pelo MAIOR valor de ID do projeto na tabela likes_por_projeto. Veja o resultado obtido na Tabela 9.

título	data
Aplicação Python	2018-08-05

Tabela 10. Resultado da query

> > 2.3.5. Subquery como fonte de dados de uma consulta principal

Este outro formato faz com que o resultado de uma subquery seja utilizado como tabela fonte de dados de uma consulta principal.

Para exemplificar esse modelo, realizaremos primeiro a query que servirá como fonte de dados para a consulta principal, como vemos na Listagem 7.

```
SELECT
    P.id,
    P.titulo,
    (SELECT
        COUNT(C.id_projeto)
    FROM
        comentarios C
    WHERE
        C.id_projeto = P.id ) AS Quantidade_Comentarios
FROM
    projetos P
```

Listagem 7. Fonte de dados

Veja na Tabela 11 o resultado da query acima.

título	Quantidade_Comentarios
Aplicação C#	2
Aplicação Ionic	3
Aplicação Python	2

Tabela 11. Resultado da query

Com base no resultado acima, selecionaremos com o código da Listagem 8 apenas o projeto que teve a quantidade de comentários maior de 2, dessa forma, utilizaremos a query acima como fonte de dados.

```
SELECT
    F.titulo,
    F.Quantidade_Comentarios
FROM
    (SELECT
        P.id,
        P.titulo,
        (SELECT
            COUNT(C.id_projeto)
        FROM
            comentarios C
        WHERE
            C.id_projeto = P.id ) AS Quantidade_Comentarios
    FROM
        projetos P
    ) as F
WHERE
    F.Quantidade_Comentarios > 2
```

Listagem 8. Utilizando a fonte de dados

Observe que, na query acima, a consulta principal solicita por meio do FROM as colunas título e Quantidade_Comentarios da fonte de dados baseada em uma outra consulta, e, por fim, realiza um filtro no resultado por intermédio da cláusula WHERE para buscar somente aqueles projetos com a quantidade de comentários maior que 2.

Sempre após a criação de uma subquery como fonte de dados de uma consulta principal, será necessário definir um nome para esta fonte de dados, utilizando a palavra reservada AS.

Apesar de a maioria das consultas feitas por quem está começando em SQL ser com SELECT externos, nesse conteúdo será apresentado um recurso bastante útil que vai ajudar você a melhorar a legibilidade da sua query assim como, em alguns casos, otimizar o tempo do retorno das suas informações para o usuário, as chamadas subconsultas.

Uma subconsulta nada mais é do que uma instrução SELECT dentro de outro SELECT que retorna algumas colunas específicas usadas em algumas funções como INSERT e UPDATE por exemplo.

>> 2.3.6. Subconsulta com operação INSERT INTO

Podemos usar subconsultas também com a instrução de inserção de registros INSERT INTO, e outras instruções de linguagem de manipulação de dados (DML). Com a instrução INSERT INTO, os dados retornados por uma subconsulta são usados para realizar a inserção de um registro em outra tabela.

Sintaxe:

```
INSERT INTO tabela (colunas)
    (SELECT coluna(s)
     FROM tabela(s)
     WHERE coluna operador valor);
```

Exemplo:

Vamos criar uma tabela de exemplo chamada de “projetos_usuarios”, contendo os campos id, título e nome:

```
CREATE TABLE IF NOT EXISTS public.projeto_usuario(
    id serial NOT NULL,
    titulo character varying,
    nome character varying,
    PRIMARY KEY (id)
);
```

Usando uma subconsulta, vamos popular essa nova tabela com os nomes e títulos:

```
INSERT INTO public.projeto_usuario(titulo, nome)
SELECT titulo, nome
FROM projetos p
INNER JOIN likes_por_projeto lpp on lpp.id_projeto = p.id
INNER JOIN usuarios u on u.id = lpp.id_usuario;
```

Listagem 9. Insert usando Select

Após executar o comando, efetuamos uma consulta na nova tabela para verificar se os dados foram inseridos com sucesso:

```
SELECT * FROM projeto_usuarios
```

Resultado:

id [PK] integer	titulo character varying	nome character varying
1	Aplicação C#	Bruna Luiza
2	Aplicação C#	Osvaldo Justino
3	Aplicação Ionic	Bruna Luiza
4	Aplicação Ionic	Thiago Braga
5	Aplicação Ionic	Osvaldo Justino
6	Aplicação Ionic	Gabriel Fernando
7	Aplicação Python	Thiago Braga

Figura 2. Resultado da consulta em projeto_usuario



>> 2.3.7. Subconsulta com operação UPDATE

Também podemos usar uma subconsulta com uma instrução UPDATE, permitindo atualizar os valores de registros de uma tabela com base no resultado retornado pela subconsulta.

Sintaxe:

```
UPDATE tabela
SET coluna(s) = valor
WHERE coluna operador (SELECT coluna(s)
                        FROM tabela
                        WHERE coluna operador valor);
```

Exemplo:

Vamos criar uma coluna a mais na tabela “comentário” chamada de vip. Esse campo será Boolean e terá como opção verdadeira quando o usuário for o Thiago Braga:

```
ALTER TABLE public.comentarios
ADD COLUMN vip boolean;

UPDATE comentarios
SET vip = true
WHERE id_usuario = (SELECT id FROM usuarios WHERE nome = 'Thiago Braga')
```

Listagem 10. Update com select

A subconsulta retorna o ID do usuário de acordo com o seu nome, e este ID é então empregado na consulta principal para aplicar a atualização nos valores dos registros desejados.

>> 2.3.8. Subconsulta com operação DELETE

Finalmente, podemos usar subconsultas em instruções DELETE, selecionando o valor a ser excluído de uma tabela de acordo com o retorno da subconsulta. Estamos excluindo todos os comentários de um determinado usuário.

```
DELETE FROM comentarios
WHERE id_usuario = (SELECT id FROM usuarios WHERE nome = 'Thiago Braga')
```

Listagem 11. Delete de múltiplos registros com select

3. Operadores UNION EXCEPT e INTERSECT

Os operadores de conjunto trabalham em linhas completas das consultas, portanto, os resultados das consultas devem ter o mesmo nome de coluna, mesma ordem de coluna e os tipos de colunas devem ser compatíveis.

- **UNION**: combine dois ou mais conjuntos de resultados em um único conjunto, sem duplicatas;
- **UNION ALL**: combine dois ou mais conjuntos de resultados em um único conjunto, incluindo todas as duplicatas;
- **INTERSECT**: pegue os dados de ambos os conjuntos de resultados que são comuns;
- **EXCEPT**: pegue os dados do primeiro conjunto de resultados, mas não do segundo conjunto de resultados (ou seja, sem correspondência entre si).

Regras sobre Operações de Conjunto:

- Os conjuntos de resultados de todas as consultas devem ter o mesmo número de colunas;
- Em cada conjunto de resultados, o tipo de dados de cada coluna deve ser compatível (bem combinado) com o tipo de dados de sua coluna correspondente em outros conjuntos de resultados;
- Para classificar o resultado, uma cláusula ORDER BY deve fazer parte da última instrução select. Os nomes das colunas ou aliases devem ser descobertos pela primeira instrução select;
- Entenda as diferenças entre esses operadores com exemplos.

Use o script SQL abaixo para criar e preencher as duas tabelas que usaremos em nossos exemplos.

```
CREATE TABLE public.tabela_a(  
    id serial NOT NULL,  
    nome character varying,  
    genero character varying,  
    departamento character varying,  
    PRIMARY KEY (id));
```

```
INSERT INTO public.tabela_a(nome, genero, departamento) VALUES ('João', 'M', 'TI');  
INSERT INTO public.tabela_a(nome, genero, departamento) VALUES ('Maria', 'F', 'TI');  
INSERT INTO public.tabela_a(nome, genero, departamento) VALUES ('José', 'M', 'RH');  
INSERT INTO public.tabela_a(nome, genero, departamento) VALUES ('Lara', 'F', 'RH');
```

Listagem 12. Criação da tabela A

```
CREATE TABLE public.tabela_b(  
  id serial NOT NULL,  
  nome character varying,  
  genero character varying,  
  departamento character varying,  
  PRIMARY KEY (id));  
  
INSERT INTO public.tabela_b(nome, genero, departamento) VALUES ('João', 'M', 'TI');  
INSERT INTO public.tabela_b(nome, genero, departamento) VALUES ('Maria', 'F', 'TI');  
INSERT INTO public.tabela_b(nome, genero, departamento) VALUES ('José', 'M', 'RH');
```

Listagem 13. Criação da Tabela B

>> 3.1. Operador UNION

O operador Union retornará todas as linhas exclusivas de ambas as consultas. Observe que as duplicatas são removidas do conjunto de resultados.

```
SELECT nome, gênero, departamento FROM tabela_a  
UNION  
SELECT nome, gênero, departamento FROM tabela_b
```

nome character varying	genero character varying	departamento character varying
João	M	TI
Maria	F	TI
José	M	RH
Lara	F	RH
João	M	TI
Maria	F	TI
José	M	RH

Figura 3: Union

>> 3.2. Operador UNION ALL

O operador UNION ALL retorna todas as linhas de ambas as consultas, incluindo as duplicatas.

```
select nome, gênero, departamento from tabela_a  
UNION ALL  
select nome, gênero, departamento from tabela_b
```

nome character varying	genero character varying	departamento character varying
João	M	TI
Maria	F	TI
José	M	RH
Lara	F	RH
João	M	TI
Maria	F	TI
José	M	RH

Figura 4: Union All

>> 3.3. Operador INTERSECT

O operador INTERSECT recupera as linhas exclusivas comuns das consultas esquerda e direita. Observe que as duplicatas foram removidas.

```
select nome, gênero, departamento from tabela_a  
INTERSECT  
select nome, gênero, departamento from tabela_b
```

nome character varying	genero character varying	departamento character varying
João	M	TI
José	M	RH
Maria	F	TI

Figura 5: INTERSECT

>> 3.4. Operador EXCEPT

O operador EXCEPT retornará linhas exclusivas da consulta à esquerda que não estão presentes nos resultados da consulta à direita.

```
SELECT ID, Nome, Gênero, Departamento FROM TableA  
EXCEPT  
SELECT ID, Name, Gender, Department FROM TableB
```

Resultado: se você quiser as linhas que estão presentes na Tabela B, mas não na Tabela A, inverta as consultas.

Para que todos esses 4 operadores funcionem, as 2 condições a seguir devem ser atendidas

- O número e a ordem das colunas devem ser os mesmos em ambas as consultas;
- Os tipos de dados devem ser iguais ou pelo menos compatíveis.

Por exemplo, se o número de colunas for diferente, você receberá o seguinte erro:
Msg 205, Level 16, State 1, Line 1

Todas as consultas combinadas usando um operador UNION, INTERSECT ou EXCEPT devem ter um número igual de expressões em suas listas de destino.

nome	genero	departamento
character varying	character varying	character varying
Lara	F	RH

Figura 6: EXCEPT

4. Views

Primeiramente, precisamos entender o que são as Views, que são consideradas pseudo-tables, ou seja, elas são usadas junto à 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 da tabela associada. Como as Views possuem permissões separadas, podemos utilizá-las para restringir mais o acesso aos dados pelos usuários, para que este veja apenas o que é necessário.

Vamos começar apresentando a sintaxe básica de uma View, conforme a Listagem 14.

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name  
[ ( column_name [, ...] ) ]  
[ WITH ( view_option_name [= view_option_value] [, ...] ) ]  
AS query  
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Listagem 14. Sintaxe de criação de uma View.

O código começa com a instrução de criação CREATE VIEW. Como ela é virtual, sempre que uma consulta é executada, a View é referenciada na consulta.

A instrução CREATE [ou REPLACE] VIEW é semelhante à anterior, mas verifica se já existe uma View com o mesmo nome e, caso exista, faz a substituição desta pela mais recente. O nome dado à nova View não pode ser o mesmo já atribuído também a qualquer outro item presente no banco.

Caso o nome do Schema tenha sido fornecido, a View será criada com base nele e não no Schema global. Quando temos um Schema sendo especificado, este não pode ser temporário, assim como as Views.

O parâmetro [TEMPORARY | TEMP] é especificado quando queremos que a View seja temporária, as quais são eliminadas de forma automática quando a sessão atual é finalizada. Caso tenhamos referenciado tabelas temporárias para a View, esta será criada como temporária, mesmo que não tenhamos especificado este parâmetro.

Já o RECURSIVE cria Views recursivas, ou seja, aquelas que trabalham dentro delas próprias, como na sintaxe a seguir:

```
CREATE VIEW name AS WITH RECURSIVE name (columns) AS (SELECT ...) SELECT  
columns FROM name;
```

Listagem 15. Sintaxe de criação de uma View Recursiva.

Podemos também utilizar a seguinte sintaxe, que é uma versão reduzida:

```
CREATE RECURSIVE VIEW name (columns) AS SELECT ...;
```

Listagem 16. Sintaxe de criação de uma View Recursiva reduzida.

Um exemplo real da aplicação desse conceito seria ter uma tabela de usuários, onde procuraríamos dentre estes quais seriam administradores:

```
CREATE VIEW administradores AS WITH RECURSIVE usuarios (columns) AS  
(SELECT ...) SELECT isAdministrador FROM usuarios;
```

Listagem 17. Sintaxe de criação de uma View.

O Name é o nome da View, podendo este ser opcional. Já o column_name, é uma lista opcional que contém os nomes a serem utilizados para as colunas da View.

O check_option (string) pode ser local ou em cascata e é equivalente a quando especificamos o WITH [CASCADED | LOCAL] CHECK OPTION. Esta opção pode ser alterada em Views existentes quando utilizamos a instrução ALTER VIEW. O security_barrier (string) é utilizado quando a View é criada com o intuito de fornecer segurança em nível de linha.

O WITH [CASCADE | LOCAL] CHECK OPTION é uma opção utilizada para controle do comportamento da View com base nas atualizações automáticas. No caso desta opção ser especificada, os comandos de Insert e Update são verificados para garantir que as novas linhas satisfaçam as condições que definem a View. Caso não estejam em conformidade, a atualização não ocorrerá. Caso o CHECK OPTION não seja especificado, os comandos INSERT e UPDATE presentes na View serão autorizados a criar linhas não visíveis através da própria View.

As opções de verificação suportadas, são as seguintes:

- **LOCAL** – onde novas linhas são verificadas apenas contra as condições definidas diretamente na própria View;
- **CASCADED** – aqui, novas linhas são verificadas em relação às condições da View e todas as Views subjacentes.

Se a cláusula CHECK OPTION for especificada e não tivermos a especificação da LOCAL nem da CASCADED, então o CASCADED é assumido. Precisamos ter cuidado apenas com as Views recursivas, pois o CHECK OPTION não pode ser utilizado nelas. Ou seja, esse parâmetro é suportado apenas em Views que são atualizadas automaticamente e não possuem triggers do tipo Instead Of ou mesmo que contenham regras do tipo Instead. Se uma View que é atualizada automaticamente for definida com base em uma View que contenha uma trigger Instead of, então a LOCAL CHECK OPTION pode ser utilizada para verificar as condições sobre esta View, mas as condições referentes à View que contém a trigger INSTEAD OF não será verificada.

Vamos criar uma tabela funcionários que será responsável por manter os registros reais, como mostra a Listagem 18.

```
CREATE TABLE funcionarios
(
    codigo integer NOT NULL,
    nome_func character varying(100) NOT NULL,
    data_entrada date,
    profissao character varying(100) NOT NULL,
    salario real,
    CONSTRAINT funcionarios_pkey PRIMARY KEY (codigo)
)
WITH (
    OIDS=FALSE
);
ALTER TABLE funcionarios
    OWNER TO postgres;
```

Listagem 18. Criando a tabela de funcionários.

Vamos criar uma View para essa tabela usando a sintaxe a seguir:

```
CREATE VIEW view_funcionarios AS SELECT * FROM Funcionarios;
```

Listagem 19. Sintaxe de criação de uma View Funcionários.

A view_funcionarios contém uma instrução SELECT para receber os dados da tabela. Para que possamos ver as Views que criamos, podemos utilizar a seguinte declaração:

```
SELECT codigo, nome_func, profissao FROM view_funcionarios;
```

Listagem 20. Consultando a View.

Uma View é um objeto que permite a visualização de dados da tabela à qual esteja associada. Como ela não existe por conta própria, é criada com base em consultas nas tabelas para selecionar colunas, dando, assim, acessos restritos ou com privilégios. Além disso, possuem a capacidade de juntar informações contidas em diversas tabelas em um único lugar como, por exemplo, na geração de relatórios.

Devido às suas especificações de restrição, os dados retornados são apenas aqueles que o usuário pode ver. Por serem tabelas virtuais, não podemos realizar por elas as operações DML de inserção, atualização e exclusão, mas podemos fazê-las apenas usando o SELECT.

Para exemplificarmos esse ponto, utilizaremos a tabela funcionários em conjunto com uma nova tabela registro_ponto, que conterá o registro de entrada e saída de cada um das empresas, conforme o código da Listagem 3. Veja que obteremos apenas o nome do funcionário, profissão, data e hora de entrada.

```
CREATE TABLE registro_ponto
(
    registro_ponto_id integer NOT NULL,
    hora_entrada time without time zone,
    "codFunc" integer NOT NULL,
    entrada date,
    CONSTRAINT registro_ponto_pkey PRIMARY KEY (registro_ponto_id),
    CONSTRAINT "codFuncFK" FOREIGN KEY ("codFunc")
        REFERENCES funcionarios (codigo) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
    OIDS=FALSE
);
ALTER TABLE registro_ponto
    OWNER TO postgres;

-- Index: "fki_codFuncFK"

-- DROP INDEX "fki_codFuncFK";

CREATE INDEX "fki_codFuncFK"
    ON registro_ponto
    USING btree
    ("codFunc");
```

Listagem 21. Criação da tabela de registro_ponto.

Agora criaremos uma segunda View View_Ponto_funcionario que utilizará o código do funcionário na cláusula where, como vemos a seguir:

```
CREATE VIEW View_Ponto_funcionario AS SELECT nome_func, profissao,
entrada, hora_entrada FROM funcionarios, registro_ponto WHERE
funcionarios.codigo = registro_ponto."codFunc";
```

Listagem 22. Sintaxe de criação de uma View Ponto.

Feito isso, podemos executar a instrução a seguir para ver a View criada:

```
SELECT * FROM View_Ponto_funcionario;
```

Listagem 23. Consultando a View.

Para vermos o resultado, iremos inserir alguns registros em ambas as tabelas, como mostra a Listagem 24.

```
INSERT INTO funcionarios(codigo, nome_func, data_entrada, profissao,
salario) VALUES (1, 'Edson Dionisio', '2015-09-01', 'Desenvolvedor Web',
2000.00);
INSERT INTO funcionarios(codigo, nome_func, data_entrada, profissao)
VALUES (2, 'Marília Késsia', '2013-02-01', 'Coordenadora', 5000.00);
INSERT INTO funcionarios(codigo, nome_func, data_entrada, profissao)
VALUES (3, 'Caroline França', '2015-06-15', 'Esteticista', 2500.00);
INSERT INTO registro_ponto(registro_ponto_id, entrada, hora_entrada,
"codFunc") VALUES (1, '2015-10-03', '13:00:00', 1);
INSERT INTO registro_ponto(registro_ponto_id, entrada, hora_entrada,
"codFunc") VALUES (2, '2015-11-04', '08:00:00', 2);
INSERT INTO registro_ponto(registro_ponto_id, entrada, hora_entrada,
"codFunc") VALUES (3, '2015-10-05', '08:00:00', 1);
Feita a inserção dos dados, podemos ver as mudanças na nossa segunda
View utilizando a seguinte instrução:
SELECT * FROM View_Ponto_funcionario;
```

Listagem 24. Inserção de dados nas tabelas funcionários e registro_ponto.

Como mostra a Figura 7, as informações apresentadas na View contêm os dados solicitados das duas tabelas.

Data Output	Explain	Messages	History	
	nome_func character varying(100)	profissao character varying(100)	entrada date	hora_entrada time without time zone
1	Edson Dionisio	Desenvolvedor Web	2015-11-09	08:00:00
2	Edson Dionisio	Desenvolvedor Web	2015-10-01	13:00:00
3	Marília Kássia	Coordenadora	2015-10-06	08:00:00

Figura 7. **Resultado da consulta à View View_Ponto_funcionario.**

Para excluir uma View, usamos o seguinte comando:

```
DROP VIEW view_funcionarios;
```

Listagem 25. Excluindo a View.

>> 4.1. Trabalhando com as Views materializadas

Como as Views são apenas para leitura e representação lógica dos dados que estão armazenados nas tabelas do banco de dados, podemos “materializá-las”, ou seja, armazená-las fisicamente no disco.

A criação dessas Views, em vez de criar novas tabelas, melhora o desempenho em operações de leitura. Os dados das Views materializadas serão, então, armazenados em uma tabela, que pode ser indexada rapidamente quando esta for associada e em momentos em que se precise atualizar as Views materializadas. Isso ocorre com frequência em Data warehouse e aplicações de Business Intelligence, por exemplo.

Ao contrário da View tradicional, que nos apresentam dados atualizados automaticamente, as Views materializadas precisam de um mecanismo de atualização. A sintaxe é apresentada na **Listagem 26**.

```
CREATE MATERIALIZED VIEW table_name  
[ (column_name [, ...] ) ]  
[ WITH ( storage_parameter [= value] [, ...] ) ]  
[ TABLESPACE tablespace_name ]  
AS query  
[ WITH [ NO ] DATA ]
```

Listagem 26. Sintaxe básica de uma View materializada.

A instrução `CREATE MATERIALIZED VIEW` cria View e a consulta é executada para preenchê-la. Para atualizar os dados, usamos o comando `REFRESH MATERIALIZED VIEW`.

Uma View materializada tem muitas das mesmas propriedades de uma tabela, mas não há suporte para materialização temporária ou geração automática de OIDs.

O parâmetro `TABLESPACE tablespace_name` é o nome do espaço de tabelas no qual a nova View materializada deve ser criada. Se este não for especificado, a consulta será realizada no `default_tablespace`.

Com base no exemplo que desenvolvemos neste conteúdo, a instrução a seguir cria uma View materializada para a tabela funcionários:

```
CREATE MATERIALIZED VIEW view_materializada_funcionario AS SELECT * FROM  
funcionarios WITH NO DATA;
```

Listagem 27. Sintaxe básica de uma View materializada.

Para inserir dados nessa view, usaremos o código da Listagem 28.

```
INSERT INTO funcionarios (codigo, nome_func, data_entrada, profissao)  
VALUES (5, 'Gustavo França', '2014-10-11', 'Estagiário');  
INSERT INTO funcionarios (codigo, nome_func, data_entrada, profissao)  
VALUES (6, 'Mayara Silva', '2015-06-10', 'Analista de testes');  
INSERT INTO funcionarios (codigo, nome_func, data_entrada, profissao)  
VALUES (7, 'João dos testes', '2011-01-01', 'Gerente de negócios');  
INSERT INTO funcionarios (codigo, nome_func, data_entrada, profissao)  
VALUES (8, 'Marina França', '2012-03-07', 'Analista de negócios');  
INSERT INTO funcionarios (codigo, nome_func, data_entrada, profissao)  
VALUES (9, 'Paulo Dionisio', '2013-07-07', 'DBA Sênior');
```

Listagem 28. Inserindo dados na View materializada

Agora, utilizaremos o comando `SELECT` para verificarmos se os registros foram realmente inseridos com sucesso:

```
SELECT * FROM view_materializada_funcionario;
```

Listagem 29. Consultando a View materializada

Podemos ver que obtivemos um erro ao tentarmos consultar a nossa View, como mostra a **Figura 8**.



Figura 8. Visualização de erro ao consultar View materializada.

Este erro ocorreu porque a view não se atualizou automaticamente, então precisamos do comando Refresh Materialized View:

```
REFRESH MATERIALIZED VIEW view_materializada_funcionario;
```

Listagem 30. Refresh na view materializada

Feito isso, em seguida, ao tentar consultar novamente os dados, obtemos êxito.

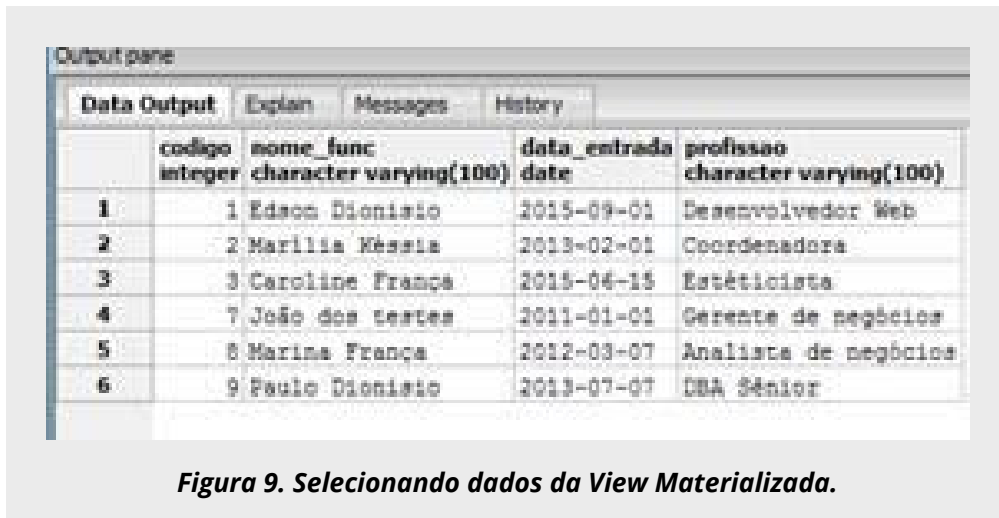
Com o PostgreSQL 9.4 podemos consultar Views materializadas enquanto são atualizadas, porém, nas versões anteriores isso não é possível. Para esse procedimento, utilizamos a palavra-chave CONCURRENTLY, como mostra o comando a seguir:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY view_materializada_funcionario;
```

Listagem 31. Refresh na View materializada com CONCURRENTLY

Para que isso ocorra, um índice exclusivo passa a ser necessário para existir na View materializada. Sendo assim, ao executarmos o comando select novamente, teremos todos os dados atualizados, como podemos ver na **Figura 9**.

Podemos ver que obtivemos um erro ao tentarmos consultar a nossa View, como mostra a **Figura 8**.



Output pane

	Data Output	Explan	Messages	History
	codigo integer	nome_func character varying(100)	data_entrada date	profissao character varying(100)
1	1	Edson Dionisio	2015-09-01	Desenvolvedor Web
2	2	Marilia Messia	2013-02-01	Coordenadora
3	3	Caroline Franca	2015-04-15	Esteticista
4	7	João dos testes	2011-01-01	Gerente de negócios
5	8	Marina Franca	2012-03-07	Analista de negócios
6	9	Paulo Dionisio	2013-07-07	DBA Sênior

Figura 9. Selecionando dados da View Materializada.

Para excluir esse tipo de View, basta utilizarmos o seguinte comando:

```
DROP MATERIALIZED VIEW view_materializada_funcionario;
```

Listagem 32. Excluindo a View materializada

Todos os recursos de otimização são importantes, pois lembre-se que as Views são apenas visões dos dados e são carregadas apenas quando necessárias. Mas elas criam uma camada extra para administrar, bem como podem limitar exageradamente, impedindo certas tarefas.

Além disso, não confunda uma View materializada com uma trigger, pois apesar de funcionarem de forma semelhante, a trigger tem muito mais poder sobre a tabela.

5. Triggers

Os gatilhos (trigger) do banco de dados são uma maneira de executar um pedaço de código quando uma operação pré-definida ocorre no banco de dados. Você pode se relacionar com gatilhos como manipuladores de eventos em javascript. Em javascript, você pode configurar um manipulador de eventos para um clique de botão. Os gatilhos do banco de dados são assim.

O que é um gatilho (trigger)

Um gatilho é uma especificação de que o banco de dados deve executar automaticamente uma determinada função sempre que um determinado tipo de operação for realizado. Podemos usar gatilhos em tabelas (particionadas ou não), visualizações e tabelas estrangeiras.

Os gatilhos podem ser definidos para executar antes ou depois de qualquer operação INSERT, UPDATE ou DELETE, uma vez por linha modificada ou uma vez por instrução SQL.

Uma trigger deve ter uma “função de gatilho”, que é executada sempre que uma operação suportada ocorre. Criar uma trigger inclui fornecer a função do gatilho. Não precisa de argumentos e devolve o gatilho do tipo.

A função de gatilho recebe seus argumentos de entrada por meio de uma variável específica incorporada ao PostgreSQL. Não precisamos passar nenhum argumento de função na definição.

Tipos de gatilhos

Há muitas variações em como um gatilho é invocado. Vamos olhar para eles:

- Gatilhos de nível de linha;
- Gatilhos de nível de declaração.

Os gatilhos de nível de linha são executados para cada linha na tabela, enquanto os gatilhos de nível de declaração são executados uma vez por transação. Por exemplo, se uma única transação atualizar mil linhas, o gatilho de nível de declaração será executado apenas uma vez, mas o gatilho de nível de linha será executado mil vezes.

Os gatilhos também são classificados de acordo com a ação de disparar antes, depois ou em lugar da operação. Estes são referidos como gatilhos BEFORE, gatilhos AFTER e em vez de gatilhos, respectivamente.

Para gatilhos por linha:

- O valor de retorno é ignorado para gatilhos de nível de linha disparados após uma operação para que possam retornar NULL;
- O gatilho de nível BEFORE pode retornar NULL para pular a operação para a linha atual. Isso instrui o executor a não executar a operação de nível de linha que invocou o gatilho (inserção, modificação ou exclusão de uma determinada linha de tabela).
- Para os gatilhos de nível de linha BEFORE DE INSERIR/ATUALIZAR, a linha retornada torna-se a linha que será inserida ou atualizada. Permite que a função do gatilho modifique a linha que está sendo inserida ou atualizada;
- As funções de gatilho invocadas por gatilhos por declaração devem sempre retornar NULL.

>> 5.1. Trigger function

Uma trigger é criada com o comando CREATE FUNCTION, declarando-o como uma função sem argumentos e um tipo de gatilho de retorno para gatilhos de alteração de dados. Variáveis locais especiais nomeadas TG_something são automaticamente definidas para descrever a condição que acionou a chamada.

Funções de trigger podem ser escritas na maioria das linguagens processuais, incluindo PL/pgsql, Perl, Python. Com a função PL/pgSQL como um gatilho, existem algumas variáveis criadas automaticamente no bloco de nível superior.

Observe que a função deve ser declarada sem argumentos, mesmo que espere receber alguns argumentos especificados no CREATE TRIGGER. Tais argumentos são passados via TG_ARGV.

Vamos dar uma olhada em alguns dos comuns:

NEW

Mantém a nova linha de banco de dados para operações INSERT/UPDATE em gatilhos de nível de linha. Esta variável é nula em gatilhos de nível de declaração e para operações DELETE.

OLD

Mantém a linha de banco de dados antiga para operações UPDATE/DELETE em gatilhos de nível de linha. Esta variável é nula em gatilhos de nível de declaração e para operações INSERT.

TG_WHEN

Uma sequência de BEFORE, AFTER, ou INSTEAD OF, dependendo da definição do gatilho.

TG_LEVEL

Indica se o gatilho é nível ROW ou STATEMENT.

TG_OP

Mostra a operação que acionou o gatilho. Podendo ser INSERT, UPDATE, DELETE, ou TRUNCATE.

TG_ARGV[]

Os argumentos da instrução CREATE TRIGGER. O índice conta a partir de 0. Índices inválidos (menores de 0, ou maiores, ou iguais, ou iguais à tg_nargs) resultam em um valor nulo.

>> 5.2. 2 passos para escrever uma trigger no PostgreSQL

Um gatilho em pleno funcionamento no PostgreSQL pode ser criado em apenas duas etapas:

1. Defina uma função de gatilho;
2. Crie o gatilho usando essa função de gatilho.



Primeiramente, criaremos uma função de gatilho que pode ser usada por um gatilho:

```
CREATE OR REPLACE FUNCTION function_name()  
  RETURNS trigger  
  LANGUAGE plpgsql  
AS $$  
DECLARE  
  -- declare variables if needed  
BEGIN  
  -- function body goes here  
END;  
$$;
```

Agora que temos a função de gatilho, podemos criar o gatilho:

```
CREATE OR REPLACE TRIGGER name  
  { BEFORE | AFTER } { event }  
  ON table_name  
  [ FOR [ EACH ] { ROW | STATEMENT } ]  
  [ WHEN ( condition ) ]  
  EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

CREATE OR REPLACE TRIGGER criará um novo gatilho ou substituirá um gatilho existente. Se você não quiser isso, use CREATE TRIGGER em vez disso. Ele não criará o gatilho se ele já existir.

O gatilho estará associado à tabela especificada e executará a função de gatilho especificada. Quando determinadas operações forem executadas naquela tabela BEFORE/AFTER, a operação aconteceu.

A mesma função de gatilho pode ser usada para criar vários gatilhos se tudo estiver escrito corretamente.

Você também pode colocar condições sobre quando o gatilho deve disparar, basta usar a cláusula WHEN. Em gatilhos de nível de linha, a condição WHEN pode examinar os valores antigos e/ou novos das colunas da linha. Não é útil no caso de gatilhos de nível de declaração porque eles não podem se referir a nenhum valor na tabela.

Se vários gatilhos do mesmo tipo forem definidos para o mesmo evento, eles serão disparados em ordem alfabética pelo nome.

Alguns casos de uso para diferentes gatilhos:

Os gatilhos BEFORE de nível de linha são usados para verificar ou modificar os dados que serão inseridos ou atualizados. Por exemplo, um gatilho BEFORE pode ser usado para inserir o tempo atual em uma coluna de tempo ou para verificar se dois elementos da linha são consistentes. Ele também pode atuar como uma fase de validação em que você rejeita os dados indesejados (verificações nulas, por exemplo).

Os gatilhos AFTER de nível de linha são frequentemente usados para propagar as atualizações para outras tabelas ou fazer verificações de consistência contra outras tabelas. Razão? After trigger sempre vê o valor final da linha enquanto um BEFORE não.

Se possível, usar o gatilho BEFORE é melhor do ponto de vista de desempenho. Não devemos salvar as informações sobre a operação até o final de uma declaração, se não for necessário.

Se uma função de gatilho executar comandos SQL, esses comandos podem disparar gatilhos novamente. Isso é conhecido como **gatilhos em cascata**. Não há limitação direta no número de níveis de cascata. As Cascatas podem causar uma invocação recursiva do mesmo gatilho.

Por exemplo, um gatilho INSERT pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o gatilho INSERT seja acionado novamente.

> > 5.3. Visibilidade dos dados nas Triggers

Existem muitos casos de uso para gatilhos, e acessar os valores novos ou antigos de uma linha em uma tabela é um deles. A capacidade de olhar para os dados nos permite realizar várias ações que não seriam possíveis de outra forma. Registros de auditoria, restrições de verificação e histórico de versão são alguns dos casos de usos comuns e simples.

Tendo isso em mente, é importante saber quando podemos acessar os dados antigos e novos de uma linha na tabela.

1. Em um gatilho de nível de declaração, nenhuma das alterações é visível para os gatilhos BEFORE, enquanto todas as alterações são visíveis para os gatilhos AFTER;
2. Quaisquer modificações nos dados que acontecem durante INSERT, UPDATE, DELETE não são visíveis aos gatilhos de nível BEFORE porque a alteração ainda não aconteceu;
3. BEFORE do gatilho pode ver os efeitos das corridas *anteriores* dos gatilhos BEFORE;
4. Todas as alterações são visíveis para AFTER gatilhos porque a mudança já aconteceu.

Exemplos:

Vamos dar uma olhada em alguns exemplos mostrando diferentes casos de uso de gatilhos. Estes são exemplos muito simples, mas não são os únicos casos de uso.

- Checking constraints

Podemos definir uma restrição de verificação em uma tabela para garantir que certos valores de coluna não sejam nulos ou tenham determinados valores. Isso pode ser alcançado com a ajuda de um gatilho BEFORE INSERT.

```
CREATE OR REPLACE FUNCTION name_null_checker() RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.name IS NULL THEN
        RAISE EXCEPTION 'Name cannot be null';
    END IF;
    RETURN NEW;
END;
$$;
```

Nós escrevemos uma função de gatilho básico que verifica se o nome do livro é nulo e lança uma exceção se o for. Com esta etapa, criamos uma função de gatilho reutilizável que pode ser usada por vários gatilhos para verificar a restrição do nome. Devidamente escrito, esta função de gatilho seria independente da tabela específica em que está acionando.

Agora, vamos criar um gatilho que disparará antes de uma operação insert na mesa de livros.

```
CREATE TRIGGER name_null_checker_trigger BEFORE INSERT ON books
FOR EACH ROW EXECUTE FUNCTION name_null_checker();
```

E agora temos um gatilho básico para realizar a verificação de restrição na tabela de livros.

- Audit logging

Audit logs são uma maneira de acompanhar as mudanças em uma tabela. Vamos configurar um gatilho para registrar todos os INSERT, UPDATE e DELETE na tabela de livros.

Precisamos criar uma tabela de auditoria de livros para armazenar os registros de auditoria. Vamos primeiro criá-la:

```
CREATE TABLE IF NOT EXISTS books_audit_store (  
  name VARCHAR(128) UNIQUE NOT NULL,  
  price float(2) NOT NULL,  
  rating INT NOT NULL,  
  operation VARCHAR(64) NOT NULL,  
  modified_at TIMESTAMP NOT NULL  
);
```

Agora que temos nossa books_audit_store tabela, podemos criar a função de gatilho que executará o comando INSERT SQL:

```
CREATE OR REPLACE FUNCTION audit_logger()  
  RETURNS trigger  
  LANGUAGE plpgsql  
AS $$  
BEGIN  
  INSERT INTO books_audit_store (name, price, rating, operation,  
modified_at)  
    VALUES (NEW.name, NEW.price, NEW.rating, TG_OP, now());  
  RETURN NEW;  
END;  
$$;
```

Finalmente, criando o gatilho na tabela books_audit_store:

```
CREATE TRIGGER audit_logger_trigger  
AFTER UPDATE ON books  
FOR EACH ROW  
WHEN (OLD.* IS DISTINCT FROM NEW.*)  
EXECUTE FUNCTION audit_logger();
```

Aqui está a tabela de livros antes da atualização:

```
postgres=# select * from books;
 id |      name      | price | rating |      updated_at
-----+-----+-----+-----+-----
  1 | The midnight library | 5.88 | 4 | 2022-01-29 19:10:25
  2 | Scrum          | 15.5 | 3 | 2022-01-30 02:13:25
  3 | The midnight library 2 | 5.88 | 4 | 2022-01-29 19:10:25
(3 rows)
```

Atualizamos o título do livro "Scrum" para ser "Mestre scrum":

```
postgres=# select * from books;
 id |      name      | price | rating |      updated_at
-----+-----+-----+-----+-----
  1 | The midnight library | 5.88 | 4 | 2022-01-29 19:10:25
  3 | The midnight library 2 | 5.88 | 4 | 2022-01-29 19:10:25
  2 | Scrum master    | 15.5 | 3 | 2022-01-30 02:13:25
(3 rows)
```

Como resultado da atualização na tabela de livros com um valor diferente, temos nossa entrada de login de auditoria na tabela books_audit_store:

```
postgres=# select * from books_audit_store;
 name      | price | rating | operation |      modified_at
-----+-----+-----+-----+-----
 Scrum master | 15.5 | 3 | UPDATE | 2022-01-29 16:38:26.155591
(1 row)
```

- Drop trigger

Um gatilho pode ser descartado emitindo uma instrução DROP TRIGGER.

```
DROP TRIGGER triggerName ON tableName;
```

Para excluir o gatilho, o usuário deve ser o proprietário da tabela em que o gatilho é definido. Podemos adicionar IF EXISTS à instrução para soltar o gatilho somente se ele existir:

```
DROP TRIGGER IF EXISTS triggerName ON tableName;
```

No padrão SQL, os nomes de gatilho não são locais para tabelas, então você não precisa especificar o nome da tabela na instrução DROP TRIGGER.



Referências bibliográficas

ALVES, William Pereira. **Banco de Dados [BV:MB]**. 1ª ed. São Paulo: Érica, 2014.

BALIEIRO, R. **Banco de Dados [BV:RE]**. 1ª ed. Rio de Janeiro: SESES, 2015.

ELMASRI, R.; NAVATHE, S. **Sistemas de Banco de Dados [BV:PE]**. 7. ed. São Paulo: Pearson, 2018.

FONSECA, Cleber Costa da. **Implementação de banco de dados. Banco de Dados [BV:RE]**. 1ª ed. Rio de Janeiro: SESES, 2016.

HEUSER, C. **Projeto de Banco de Dados. [BV:MB]**. 6. ed. Porto Alegre: ARTMED, 2009.

MACHADO, Felipe N. R. **Banco de Dados - Projeto e Implementação [BV:MB]**. 3. ed. São Paulo: Érica, 2014.

NETO, Geraldo H. **MODELAGEM DE DADOS. [BV:RE]**. 1ª ed. Rio de Janeiro: SESES, 2015.

PUGA, Sandra; FRANÇA, Edson; GOYA, Milton. **Banco de Dados: implementação em SQL, PL/SQL e Oracle 11g [BV:PE]**. 1ª ed. São Paulo: Pearson, 2013.

RAMAKRISHNAN, R. **Sistemas de gerenciamento de banco de dados.[BE:MB]**. 3. ed. Porto Alegre: McGraw-Hill, 2008.

Anexo I

```
CREATE TABLE IF NOT EXISTS public.projetos  
(  
    id serial NOT NULL,  
    titulo character varying,  
    data date,  
    url character varying,  
    PRIMARY KEY (id)  
);
```

```
ALTER TABLE public.projetos  
    OWNER to postgres;
```

```
CREATE TABLE IF NOT EXISTS public.usuarios  
(  
    id serial NOT NULL,  
    nome character varying,  
    email character varying,  
    senha character varying,  
    PRIMARY KEY (id)  
);
```

```
ALTER TABLE public.usuarios  
    OWNER to postgres;
```

```
CREATE TABLE IF NOT EXISTS public.comentarios  
(
```

```
id serial NOT NULL,  
comentario character varying,  
data date,  
id_usuario integer,  
id_projeto integer,  
PRIMARY KEY (id),  
FOREIGN KEY (id_usuario)  
REFERENCES public.usuarios (id) MATCH SIMPLE  
ON UPDATE NO ACTION  
ON DELETE NO ACTION  
NOT VALID,  
FOREIGN KEY (id_projeto)  
REFERENCES public.projetos (id) MATCH SIMPLE  
ON UPDATE NO ACTION  
ON DELETE NO ACTION  
NOT VALID  
);
```

```
ALTER TABLE public.comentarios  
OWNER to postgres;
```

```
CREATE TABLE IF NOT EXISTS public.likes_por_projeto  
(  
id_projeto integer NOT NULL,  
id_usuario integer NOT NULL,  
PRIMARY KEY (id_projeto, id_usuario),  
FOREIGN KEY (id_projeto)  
REFERENCES public.projetos (id) MATCH SIMPLE  
ON UPDATE NO ACTION  
ON DELETE NO ACTION  
NOT VALID,  
FOREIGN KEY (id_usuario)  
REFERENCES public.usuarios (id) MATCH SIMPLE  
ON UPDATE NO ACTION  
ON DELETE NO ACTION  
NOT VALID  
);
```

```
ALTER TABLE public.likes_por_projeto  
OWNER to postgres;
```

```
CREATE TABLE IF NOT EXISTS public.likes_por_comentario
(
    id_usuario integer NOT NULL,
    id_comentario integer NOT NULL,
    PRIMARY KEY (id_usuario, id_comentario),
    FOREIGN KEY (id_usuario)
        REFERENCES public.usuarios (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    FOREIGN KEY (id_comentario)
        REFERENCES public.comentarios (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
);

ALTER TABLE public.likes_por_comentario
    OWNER to postgres;
```

```
INSERT INTO projetos (titulo, data) values ('Aplicação C#','2022-04-01');
INSERT INTO projetos (titulo, data) values ('Aplicação Ionic','2022-05-07');
INSERT INTO projetos (titulo, data) values ('Aplicação Python','2022-08-05');
```

```
INSERT INTO usuarios (nome, email, senha) VALUES ('Bruna Luiza','bruninha@gmail.com','abc123');
INSERT INTO usuarios (nome, email, senha) VALUES ('Thiago
Braga','thiagobraga_1@hotmail.com','pena093');
INSERT INTO usuarios (nome, email, senha) VALUES ('Osvaldo
Justino','osvaltino@yahoo.com.br','osvaldit1_s');
INSERT INTO usuarios (nome, email, senha) VALUES ('Gabriel
Fernando','gabriel_fnd@gmail.com','gabss34');
```

```
INSERT INTO likes_por_projeto VALUES (1, 1);
INSERT INTO likes_por_projeto VALUES (1, 3);
INSERT INTO likes_por_projeto VALUES (2, 1);
INSERT INTO likes_por_projeto VALUES (2, 2);
INSERT INTO likes_por_projeto VALUES (2, 3);
INSERT INTO likes_por_projeto VALUES (2, 4);
INSERT INTO likes_por_projeto VALUES (3, 2);
```

```
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('A Microsoft acertou com essa linguagem!', 1, 1);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Parabéns pelo projeto! bem legal!', 1, 3);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Super interessante! Fácil e rápido!', 2, 4);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Cara, que simples fazer um app assim!', 2, 1);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Linguagem muito diferente.', 3, 3);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Adorei aprender Python! Parabéns!', 3, 2);
INSERT INTO comentarios (comentario, id_projeto, id_usuario) values ('Muito maneiro esse framework!', 2, 2);

INSERT INTO likes_por_comentario values (1,7);
INSERT INTO likes_por_comentario values (2,7);
INSERT INTO likes_por_comentario values (4,7);
```



Digital College

ENSINO DE HABILIDADES DIGITAIS

digitalcollege.com.br