Trigger

Trigger é uma espécie de procedimento armazenado que é automaticamente executado quando ocorre um evento específico no banco de dados. Esse evento pode ser uma inserção, atualização ou exclusão de dados em uma tabela. Os triggers são úteis para impor regras de negócios, manter a integridade dos dados e automatizar tarefas.

Eles são definidos para monitorar certas ações em uma tabela e responder a essas ações com um conjunto de instruções pré-definidas. Por exemplo, um trigger pode ser configurado para atualizar automaticamente uma coluna em uma tabela sempre que um novo registro é inserido ou para impedir que certos tipos de alterações sejam feitas nas tabelas.

Os triggers podem ser classificados em diferentes tipos, como "Before Triggers" (triggers antes da ação), "After Triggers" (triggers após a ação) e "Instead of Triggers" (triggers em vez da ação), dependendo de quando são disparados em relação à ação que os acionou.

TRIGGERS (GATILHOS)

- Em caso de atualizações em colunas de tabelas, podemos nos referir aos valores novos e antigos da coluna como NEW.nome_coluna e OLD.nome_coluna, respectivamente

New - código novo para ser inserido ou atualizado
OLD - código antes de fazer atualizacao

TRIGGERS (GATILHOS)

- Em pgplsql, para criarmos triggers, devemos criar uma função que retorna a trigger e em seguida criar a trigger em si

```
CREATE TRIGGER name
        { BEFORE | AFTER | INSTEAD OF }
        { event [ OR ... ] } ON table
        [ FROM referenced_table_name ]
       [ FOR [ EACH ] { ROW | STATEMENT } ]
      EXECUTE PROCEDURE function_name ( arguments )
```

evento pode ser um dentre: `INSERT UPDATE [ OF column_name [, ... ] ] DELETE TRUNCATE`

- Os eventos podem ser executados por linha alterada (for each row) ou por comando executado (for each statement)
- A diferença é transacional
  - FOR EACH ROW executa cada linha como uma transação
  - FOR EACH STATEMENT executa cada comando como uma transação

Vamos criar uma trigger para cada atualização de salário ele insere a informação na tabela de atualizacao salarial

```
CREATE TABLE empregado (id_empregado serial primary key,
 nome VARCHAR(50) NOT NULL, sobrenome VARCHAR(50) NOT NULL,
email VARCHAR(100) NOT NULL UNIQUE, cargo VARCHAR(50), salario numeric NOT NULL);
```

```
INSERT INTO empregado (nome,sobrenome,email,cargo,salario) values ('José', 'Ciclano', 'ciclano@teste.com', 'Programador',5000.);
INSERT INTO empregado (nome,sobrenome,email,cargo,salario) values ('Antonio', 'Silva', 'silva@gmail.com', 'Analista de Sistemas',6000.);
INSERT INTO empregado (nome,sobrenome,email,cargo,salario) values ('Marina', 'Oliveira', 'm@gmail.com', 'DBA',9000.);
```

```
CREATE TABLE empregado_auditoria (
        id_empregado_auditoria SERIAL PRIMARY KEY,
        id_empregado int,
        nome VARCHAR(50) NOT NULL,
        modificadoem date DEFAULT NULL,
        salario numeric  NOT NULL);
```

Criando a função para guardar a atualização de salários

```
CREATE OR REPLACE FUNCTION empregado_atua_sal_func()
RETURNS trigger AS $$
BEGIN
INSERT INTO empregado_auditoria(id_empregado,nome,salario,modificadoem)
        VALUES(OLD.id_empregado,OLD.nome,OLD.salario,now());
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

Criando Trigger para chamar a função que atualiza salário

```
CREATE TRIGGER atualiza_salario
AFTER UPDATE
ON empregado
FOR EACH ROW
EXECUTE PROCEDURE empregado_atua_sal_func();
```

Fazendo o teste atualizando o salário de todos funcionários

```
UPDATE empregado SET salario = salario * 1.12;
```

Selecionando e visualizando o salário antigo que foi colocado no empregado_auditoria

Query Editor | Query History

```
1  select * from empregado_auditoria;
2
```

Data Output | Explain | Messages | Notifications

	id_empregado_auditoria [PK] integer	id_empregado integer	nome character varying (50)	modificadoem date	salario numeric
1	1	1	José	2023-04-22	5000
2	2	2	Antonio	2023-04-22	6000
3	3	3	Marina	2023-04-22	9000

CRIAR UMA TABELA E UMA FUNÇÃO PARA QUE TODOS FUNCIONARIOS QUE FOREM APAGADOS
SEJAM JOGADOS EM UMA TABELA

```
CREATE TABLE empregado_auditoria_delete (id_empregado_delete serial primary key,
 id_empregado int, nome VARCHAR(50) NOT NULL, sobrenome VARCHAR(50) NOT NULL,
email VARCHAR(100) NOT NULL UNIQUE, cargo VARCHAR(50), salario numeric NOT NULL, data date NOT
NULL);
```

```
CREATE OR REPLACE FUNCTION  empregado_del_func()
RETURNS trigger AS $$
BEGIN
INSERT INTO empregado_auditoria_delete(id_empregado,nome,sobrenome,email,cargo,salario,data)
      VALUES(OLD.id_empregado,OLD.nome,OLD.sobrenome,OLD.email,OLD.cargo,OLD.salario,now());
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

Criando Trigger e associando a função

```
CREATE TRIGGER apagar_empregado
AFTER DELETE
ON empregado
FOR EACH ROW
EXECUTE PROCEDURE empregado_del_func();
```

Testando a Trigger

```
1  select * from empregado;
2  /*delete from empregado where id_empregado =1;*/
3  select * from empregado;
4  select * from empregado_auditoria_delete;
5
```

ata Output | Explain | Messages | Notifications

id_empregado_delete [PK] integer	id_empregado integer	nome character varying (50)	sobrenome character varying (50)	email character varying (100)	cargo character varying (50)	salario numeric	data date
1	1	José	Ciclano	ciclano@teste.com	Programador	5600.00	2023-04-22

Novo exemplo utilizando Trigger
vamos criar duas tabelas conta e
conta_total
e inserir dados somente na conta total

```
CREATE TABLE conta(numero_conta int PRIMARY KEY,titular VARCHAR(60),saldo NUMERIC,numero_banco int,tipo
CHAR(1));
```

```
CREATE TABLE conta_total(tipo CHAR(1),quantidade int);
INSERT INTO conta_total(tipo,quantidade)VALUES('F',0);
INSERT INTO conta_total(tipo,quantidade)VALUES('J',0);
```

Criando função para incrementar conta na Trigger

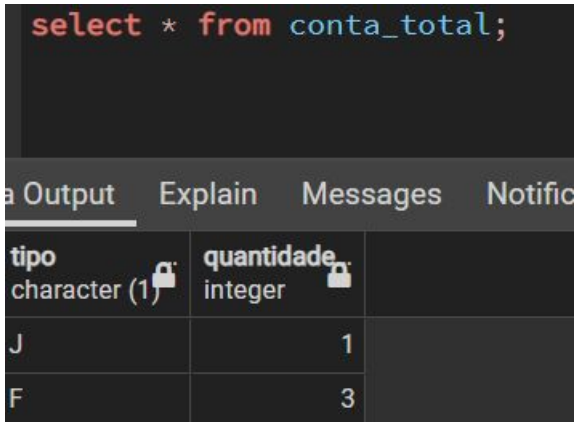
```
CREATE OR REPLACE FUNCTION incrementar() RETURNS TRIGGER AS $$
BEGIN
  IF NEW.tipo='F' THEN
    UPDATE conta_total SET quantidade=quantidade+1 WHERE tipo='F';
  ELSE
    UPDATE conta_total SET quantidade=quantidade+1 WHERE tipo='J';
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Criando trigger para chamar função incrementarContas

```
CREATE TRIGGER triggerIncrementar AFTER INSERT ON conta FOR EACH ROW EXECUTE
PROCEDURE incrementar();
```

Vamos inserir dados na conta para testar

```
INSERT into conta(numero_conta,titular,saldo,tipo)VALUES('234','Antônio',2000,'F');
INSERT into conta(numero_conta,titular,saldo,tipo)VALUES('235','Jorge',1000,'F');
INSERT into conta(numero_conta,titular,saldo,tipo)VALUES('237','Andréa',2400,'J');
INSERT into conta(numero_conta,titular,saldo,tipo)VALUES('238','Marcos',10700,'F');
```



PostgreSQl - VIEWS

View (Exibição / Visão) é uma tabela virtual (estrutura de dados) baseada no conjunto de resultados de uma consulta SQL, criada a partir de um conjunto de tabelas (ou outras views) presentes no banco, que servem com tabelas-base.

Mostra sempre resultados de dados atualizados, pois o motor do banco de dados recria os dados toda vez que um usuário consulta a visão.

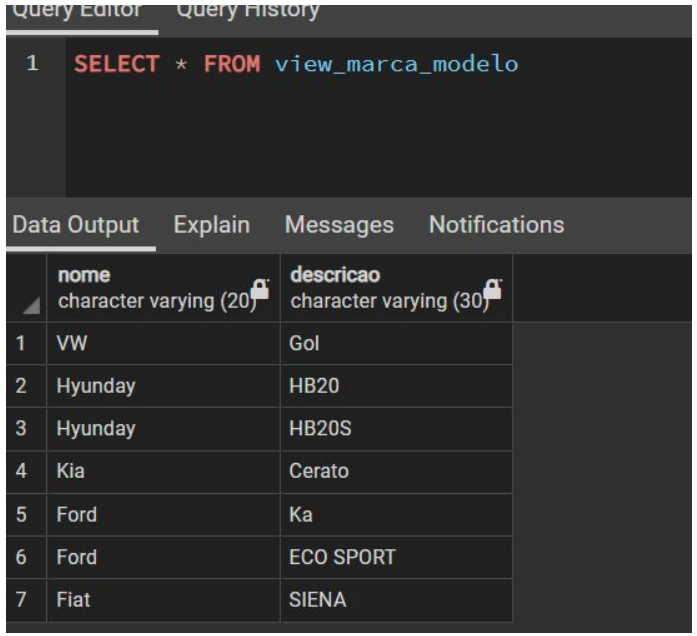
Aplicações das Views

- Simplificar o acesso a dados que estão armazenados em múltiplas tabelas relacionadas
- Implementar segurança nos dados de uma tabela, por exemplo criando uma visão que limite os dados que podem ser acessados, por meio de uma cláusula WHERE
- Prover isolamento de uma aplicação da estrutura específica de tabelas do banco acessado.

No exemplo abaixo temos uma consulta SQL que iremos transformar em uma VIEW.

```
create or replace view view_marca_modelo AS
select ma.nome, mo.descricao from marca ma
INNER JOIN modelo mo ON ma.codigo_marca = mo.codigo_marca
```

Fazendo um select na view



	nome character varying (20)	descricao character varying (30)
1	VW	Gol
2	Hyunday	HB20
3	Hyunday	HB20S
4	Kia	Cerato
5	Ford	Ka
6	Ford	ECO SPORT
7	Fiat	SIENA

Fazendo uma subconsulta para retornar o nome do veículo mais caro

Query Editor | Query History

```
1  SELECT descricao, preco FROM view_marca_modelo  WHERE preco =
2  (SELECT  max(preco) FROM view_marca_modelo);
```

Data Output | Explain | Messages | Notifications

	descricao character varying (30)	preco numeric
1	SIENA	45000