

Guia Completo sobre PL/Python

Lucas Vinicius Ribeiro¹, Lucas Souza Santos¹

¹Departamento de Computação (DACOM) – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87.301-899 – Campo Mourão – PR – Brazil

{ribeiro1,lsantos.2016}@alunos.utfpr.edu.br

Resumo. *O PL/Python é uma linguagem procedural do PostgreSQL. Ao utilizar uma linguagem procedural, é possível desfrutar dos recursos de uma linguagem de programação que não seria possível utilizando apenas os recursos da linguagem SQL. Além disso, uma das vantagens de se utilizar o PL/Python é poder fazer uso da grande quantidade de bibliotecas que o Python possui. No decorrer deste tutorial são apresentados exemplos práticos de criação de funções e gatilhos utilizando o PL/Python.*

1. Introdução

Neste tutorial, daremos uma breve introdução sobre a linguagem de programação Python e também sobre o PostgreSQL. Em seguida, daremos foco nas explicações sobre como instalar e utilizar o PL/Python. Além disso, daremos alguns exemplos práticos de como funciona, e elencaremos algumas das vantagens e desvantagens de se utilizar o PL/Python.

2. Sobre o Python

Python é uma linguagem de programação de alto nível multi-paradigma e de tipagem dinâmica e forte. O conceito multi-paradigma é derivado do seu suporte à orientação a objeto, programação imperativa e programação funcional. Python é considerada uma linguagem de tipagem dinâmica e forte pois o próprio interpretador define os tipos de dados a serem recebidos por uma variável sem a necessidade de o programador deixar explícito o tipo dado a ser recebido.

Sua característica de alto nível, ou seja, seu alto nível de abstração é o que faz muitos optarem por essa linguagem justamente por cumprir com seu objetivo de projeto e sua filosofia: favorecer a sintaxe mantendo a produtividade e a legibilidade.

3. Sobre o PostgreSQL

Chamado por muitos de Postgres, o PostgreSQL é um sistema de gerenciamento de banco de dados objeto-relacional de código aberto. O que torna o PostgreSQL um SGBD objeto-relacional são suas características de um SGBD relacional combinadas com algumas características de orientação à objetos como herança e tipos personalizados.

Uma grande característica do PostgreSQL é a capacidade de executar funções escritas em outras linguagens além de SQL, essas linguagens são chamadas de Linguagens Procedurais (PLs). As PLs dão a possibilidade de escrita de Gatilhos (*Triggers*), Procedimentos Armazenados (*Stored Procedures*) e Funções.

De forma sucinta, gatilhos são instruções realizadas para eventos específicos, ou seja, uma função que é disparada mediante alguma ação.

Em muitos SGBDs (Sistemas Gerenciadores de Banco de Dados) temos o conceito de Procedimentos Armazenados, que são programas desenvolvidos em uma determinada linguagem de script e armazenados no servidor do banco de dados onde esse script será executado. Em outros SGBDs há diferenças entre Procedimentos Armazenados e Funções, no PostgreSQL os dois conceitos são tratados como Funções.

4. Sobre o PL/Python

PL/Python é implementação da linguagem de programação Python como uma das linguagens procedurais do PostgreSQL. Ao criar ou fazer a chamada de uma função escrita em PL/Python, por exemplo, um interpretador do Python é invocado dentro do banco de dados para interpretar o código em Python. Desta forma, é possível aproveitar os recursos que esta poderosa linguagem oferece para trabalhar dentro do banco de dados. Neste sentido, uma das vantagens é justamente poder utilizar a grande quantidade de bibliotecas que o Python disponibiliza. No entanto, uma das desvantagens é a não-confiabilidade.

Como é possível observar, no PostgreSQL a linguagem é denominada plpythonu, plpython2u ou plpython3u. Este "u" ao final significa "untrusted". Isto quer dizer que a linguagem não oferece algum tipo de restrição com relação a qual usuário pode utilizar algum recurso da linguagem na base de dados. O usuário pode utilizar bibliotecas com acesso a arquivos ou alterar configurações do sistema, por exemplo. Entretanto, somente superusuários podem criar ou executar funções em PL/Python dentro do Banco de Dados. Sendo assim, caso alguma ação indevida seja realizada, é de total responsabilidade do (super)usuário que a fez.



Figura 1. PostgreSQL + Python

4.1. Mas... Por que utilizar?

Como mencionado na Seção 3, as linguagens procedurais permitem a criação e execução de funções e gatilhos no banco de dados. Sem elas não seria possível, por exemplo, aplicar uma regra de negócio sob um campo que possui valor calculado. Ou então fazer o tratamento de exceção de um campo que não pode receber um valor negativo (como a idade de uma pessoa). Ou ainda fazer a verificação se um número de CPF é válido. Enfim, são muitos os exemplos.

A grande vantagem das linguagens procedurais é, portanto, poder utilizar, dentro do banco de dados, recursos e estruturas de programação como, por exemplo, estruturas de seleção (*if then else*), laços de repetição (*while*, *do while*, *for*), etc.

Mas, se tratando apenas de estruturas de programação, há muitas linguagens procedurais que fornecem recurso. A mais comum, PL/pgSQL, é a linguagem procedural do próprio PostgreSQL. Além disso, existem várias outras como PL/Java, PL/Perl, PL/Tcl e a própria *PL/Python*. Então, qual a vantagem do PL/Python em relação às outras linguagens?

Muitos conceitos devem ser levados em conta quando se precisa escolher uma linguagem de programação: Desempenho, suporte ao programador, quantidade de recursos dentro da linguagem, ou até mesmo sua facilidade de aprendizagem.

Quando se trata de linguagens procedurais é importante colocar em peso não só a sua familiaridade com a linguagem, como também quais recursos da linguagem se pode aproveitar. E como exposto no tópico anterior, quando uma função em PL/Python é chamada, um interpretador Python é invocado para tratar da linguagem, garantindo assim um alto nível de aproveitamento da linguagem Python, podendo então o programador usufruir não só de seu extenso arsenal de bibliotecas como também de suas características de tipagem forte e dinâmica, orientação a objetos, seu alto nível de abstração e sua sintaxe intuitiva e fácil.

5. Mãos à obra: Vamos utilizar o PL/Python

Para este tutorial utilizamos o Python 3.7 (plpython3u) e PostgreSQL 10.6 com pgAdmin 4. Não foram realizados testes e exemplos em outras versões.

Antes de iniciar, vale ressaltar que, para este tutorial, consideraremos que você, leitor, já possui conhecimento básico sobre a linguagem de programação Python e também sobre como utilizar o PostgreSQL, além de já o ter instalado em seu computador. Vamos lá!

5.1. Instalando o PL/Python

Assim como o PL/Perl e PL/Tcl, o PL/Python é uma linguagem procedural que já vem instalada junto ao servidor do PostgreSQL. Sendo assim, é preciso apenas criar a linguagem dentro do PostgreSQL para poder usá-la.

Para isso, com o pgAdmin aberto, crie uma nova base de dados. Com a nova base de dados aberta, vá em `Languages, Create, Language . . .` (Figura 2).

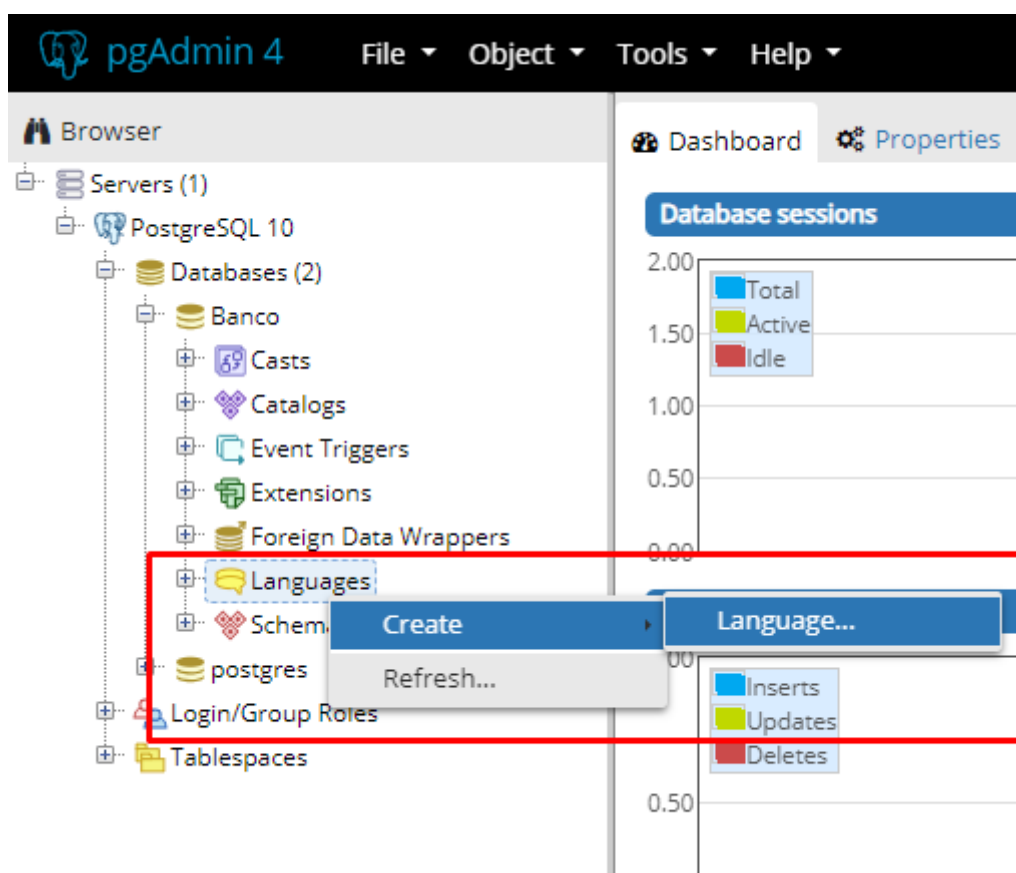


Figura 2. Criar linguagem PL/Python

Após isso, no campo "Name", selecione a opção `plpython3u` e salve. Com isso o PL/Python já foi criado na base de dados que está sendo utilizada.

Se preferir, ao invés de seguir o passo a passo acima, também é possível obter o mesmo resultado executando o seguinte script:

```
CREATE LANGUAGE plpython3u;
```

5.2. Criando a função "Hello, World!"

A sintaxe básica para criação de uma função em PL/Python é a seguinte:

```
CREATE FUNCTION nome_da_funcao(parametros)
RETURNS TIPO_DE_RETORNO AS $$
funcao_em_python
$$ LANGUAGE plpython3u;
```

Para ficar mais claro, o exemplo a seguir é um exemplo básico sobre como retornar uma mensagem de "Hello, World!" utilizando o PL/Python.

```
CREATE FUNCTION hello_world()  
RETURNS VARCHAR AS $$  
return "Hello, World!"  
$$ LANGUAGE plpython3u;
```

Se tudo der certo, ao executar o comando acima, dê um Refresh em sua base de dados e será possível ver a função criada ao acessar o menu Schemas, public, Functions. Nesta opção também é possível visualizar as propriedades e o conteúdo da função, bem como alterá-la. (Figura 3)

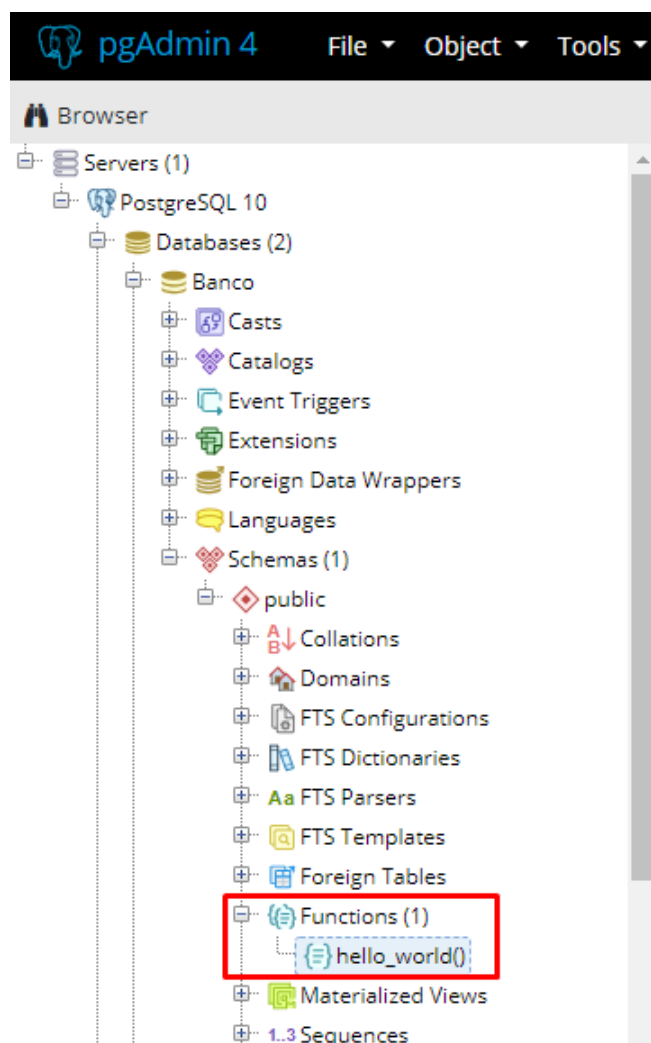


Figura 3. Função "hello_world" criada

Após isso, podemos chamar a execução da função criada utilizando o comando SELECT.

```
SELECT hello_world();
```

O resultado deverá ser como mostrado na Figura 4.

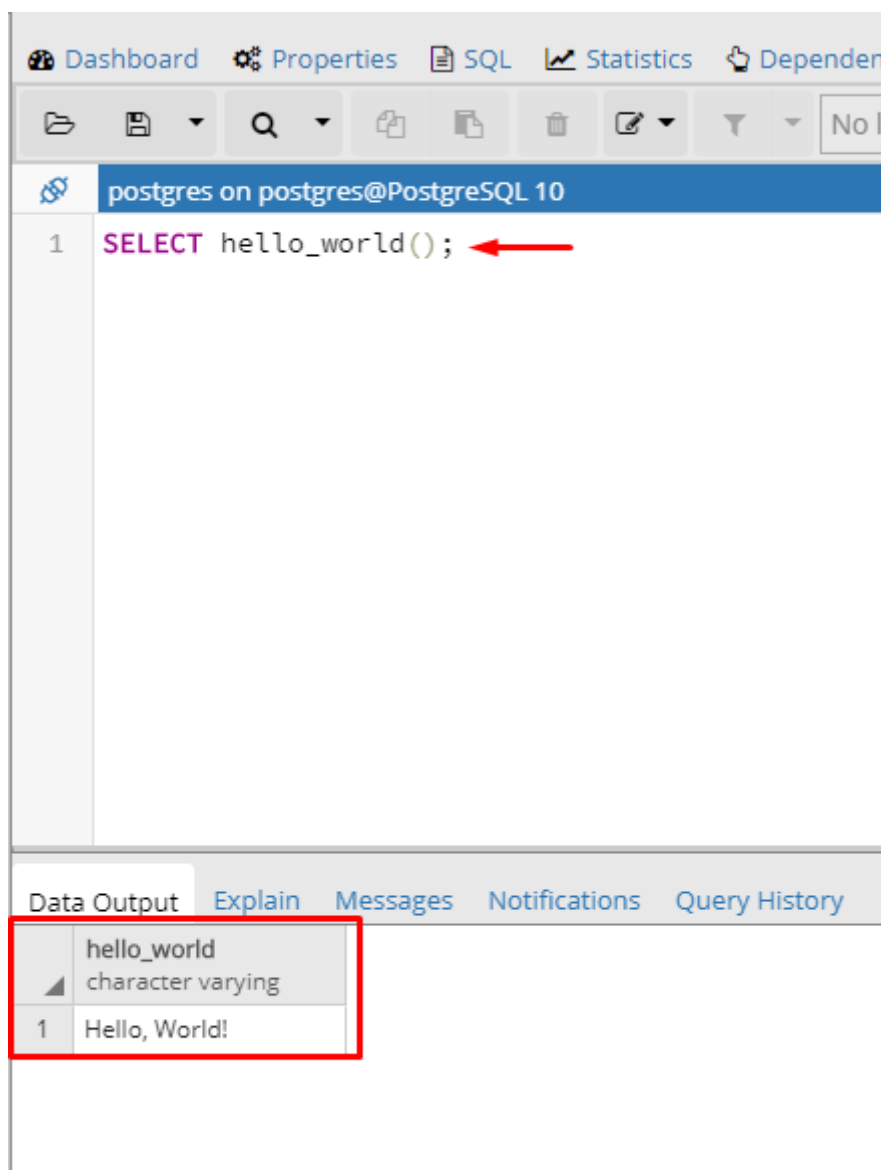


Figura 4. Chamada da função "hello_world"

5.3. OR REPLACE

O que acontece caso tenha executado o comando `CREATE FUNCTION` uma vez e queira alterar essa mesma função e executar o comando novamente? Da forma como está sendo feito, isto irá gerar um erro (Figura 5). Para corrigir este problema, usaremos o comando `OR REPLACE` após o `CREATE`. Ficando da seguinte forma:

```
CREATE OR REPLACE FUNCTION [...]
```

Vale ressaltar que, como o Python é uma linguagem orientada a objetos, é permitido que sobrescrevamos funções, utilizando tipos e quantidades de parâmetros diferentes. Por exemplo, seria possível criar a função a seguir sem algum problema. (Figura 6)

```

1 CREATE FUNCTION hello_world()
2 RETURNS VARCHAR AS $$
3
4     return "Olá, Mundo!" ← Linha alterada
5
6 $$ LANGUAGE plpython3u;

```

Data Output Explain Messages Notifications Query History

ERROR: function "hello_world" already exists with same argument types
SQL state: 42723

Figura 5. Erro ao alterar a função

```

1 CREATE OR REPLACE FUNCTION hello_world(texto VARCHAR)
2 RETURNS VARCHAR AS $$
3
4     return texto
5
6 $$ LANGUAGE plpython3u;
7
8 SELECT hello_world('Teste de Sobrescrita de Função!');

```

Data Output Explain Messages Notifications Query History

hello_world	character varying
1	Teste de Sobrescrita de Função!

Figura 6. Teste de Sobrescrita de Função

5.4. Exemplo: Criando uma função

Agora, sabendo como funciona o PL/Python e como utilizá-lo, daremos alguns exemplos práticos de sua aplicação. Para isso, utilizamos o modelo de banco de dados indicado na Figura 7. Além disso, o script para criar esta base de dados está disponível em: <https://github.com/lucasvribeiro/PLPython-Exemplos>. Neste momento, é imprescindível que você tenha executado os scripts de criação das tabelas e inserção das tuplas para poder executar o exemplo a seguir.

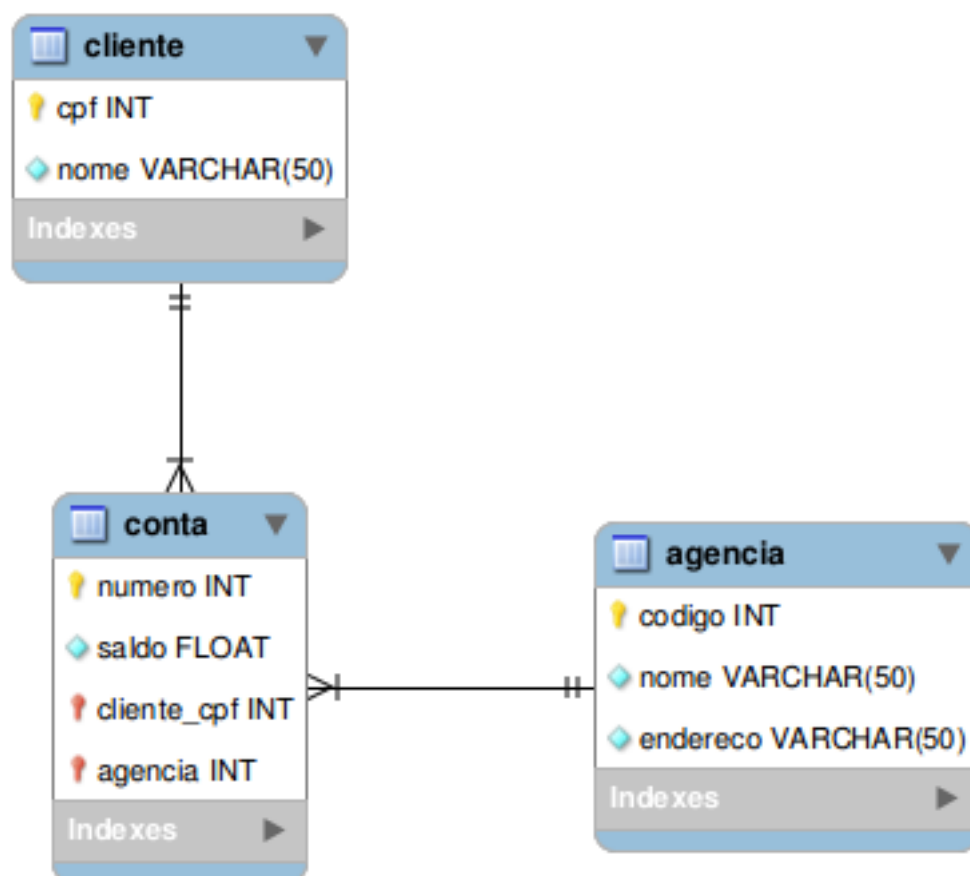


Figura 7. Modelo Entidade-Relacionamento da base de dados utilizada

No exemplo a seguir (Figura 8) criaremos uma função para aplicar rendimento sobre o montante disponível na conta do cliente. O script para criação da função também está disponível no link citado anteriormente.


```

1 CREATE OR REPLACE FUNCTION aplica_rendimento(num_conta INTEGER)
2 RETURNS VARCHAR AS
3 $$
4     saldo = plpy.execute('SELECT * FROM Conta C WHERE C.numero = ' + str(num_conta) + ';'')[0]['saldo']
5     novo_saldo = saldo
6     cliente = plpy.execute('SELECT CC.nome FROM Conta C, Cliente CC WHERE C.numero = ' + str(num_conta) + ' and CC.cpf = C.cliente;')[0]['nome']
7     ret = ''
8     if(saldo >= 100000):
9         novo_saldo += novo_saldo * (0.5/100)
10    elif(saldo >= 10000):
11        novo_saldo += novo_saldo * (1/100)
12    elif(saldo >= 1000):
13        novo_saldo += novo_saldo * (1.5/100)
14    else:
15        novo_saldo += novo_saldo * (2/100)
16    plpy.execute("UPDATE conta SET saldo = " + str(novo_saldo) + " WHERE numero = " + str(num_conta) + ";")
17    ret = "Cliente: " + str(cliente) + " || Novo Saldo: " + str(novo_saldo)
18    return ret
19 $$ LANGUAGE plpython3u;
20
21 SELECT aplica_rendimento(701);
22

```

Data Output	Explain	Messages	Notifications	Query History
<div> <div>aplica_rendimento</div> <div>character varying</div> </div>				
<div>1</div> <div>Cliente: Lucas Souza Novo Saldo: 3090.675</div>				

Figura 8. Script de criação da função aplica_rendimento()

Para facilitar o entendimento, explicaremos o detalhadamente qual a função de cada linha da função.

- Linhas 1 a 3: Comando de criação da função
- Linha 4: O comando `plpy.execute` possibilita a execução de um script dentro do própria função Python. Ele retorna uma lista de dicionários com os valores encontrados no resultado da consulta. Sendo assim, a posição `[0]` indica o primeiro elemento da lista (neste caso, teremos apenas um), e o valor `['saldo']` indica que queremos o valor referente à chave 'saldo' do dicionário.
- Linha 5: Novo saldo recebe o valor de saldo, para fins de cálculos
- Linha 6: De acordo com o número da conta recebido por parâmetro na função, executamos a consulta para armazenar o nome do cliente que possui esta conta. Esta linha serve para podermos imprimir, ao final da execução da função, o nome do cliente do qual a conta foi aplicado o rendimento.
- Linha 7: Criação da string de retorno.
- Linhas 8 a 15: Condições e atribuição do valor de rendimento. As condições estabelecidas são apenas para fins de exemplificação.
- Linha 16: Execução do comando de UPDATE para atualizar o saldo da conta, já com o rendimento aplicado.
- Linha 17: Formatação da string de retorno.
- Linha 18: Retorno da função.
- Linha 19: Indicação da linguagem procedural sendo utilizada.
- Linha 21: Chamada da função para a conta de número 701 (Saldo anterior do cliente era de R\$3.045. Ou seja, foi aplicado o rendimento de 1,5%).

5.5. Exemplo: Criando um gatilho

Sabendo agora como escrever uma função em PL/Python, apresentaremos um exemplo prático da criação de gatilhos e sua execução.

Para essa abordagem, atualizaremos nossa base de dados acrescentando uma nova tabela com o nome `operacoes`. É nela onde serão adicionadas todas as atualizações

de saldo da base de dados, o script da nova tabela está no repositório dado e a nova relação pode ser vista na Figura 9.

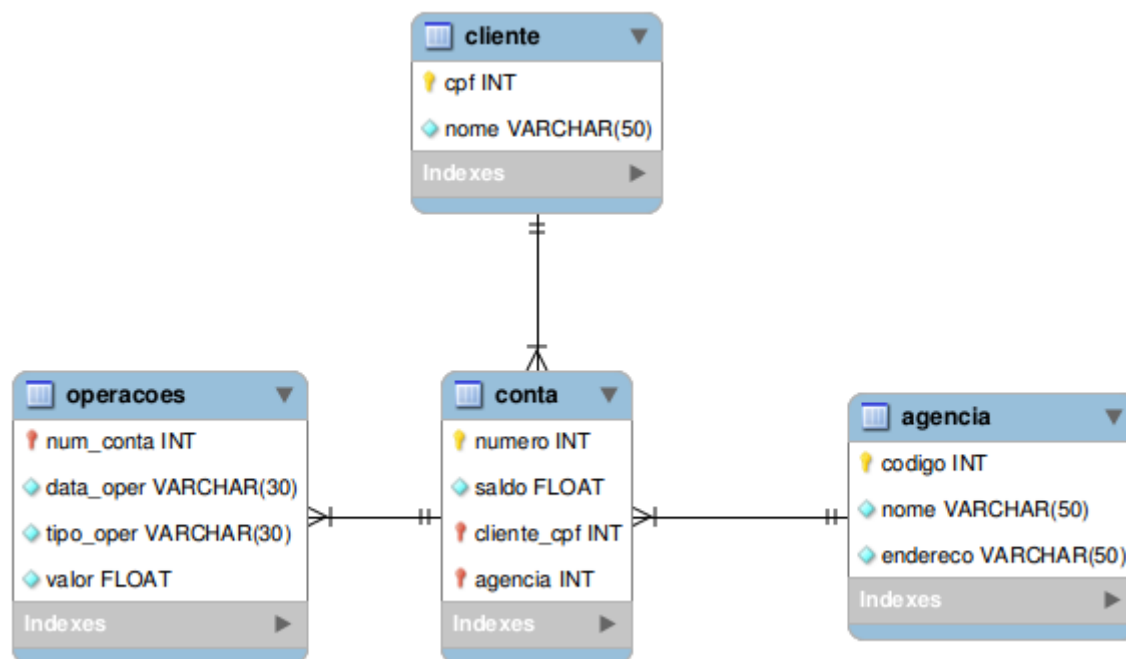


Figura 9. Modelo Entidade - Relacionamento com a nova tabela operacoes

Após a criação da nossa tabela, criaremos uma função de gatilho com o nome de `nova_operacao`, que tratará de inserir na tabela `operacoes` todas as atualizações de saldo dos nossos clientes, como apresentado na Figura 10.

```

1 CREATE OR REPLACE FUNCTION nova_operacao() RETURNS trigger AS '
2     from datetime import datetime
3
4     conta = TD["new"]["numero"]
5     antigo_saldo = TD["old"]["saldo"]
6     novo_saldo = TD["new"]["saldo"]
7
8     insert_operacao = plpy.prepare("INSERT INTO operacoes (num_conta, data_oper, tipo_oper, valor)
9         VALUES ($1, $2, $3, $4)", ["integer", "varchar(30)", "varchar(30)", "real"])
10
11     if(antigo_saldo < novo_saldo):
12         operacao = "Deposito"
13         valor = novo_saldo - antigo_saldo
14
15     elif(antigo_saldo > novo_saldo):
16         operacao = "Saque"
17         valor = antigo_saldo - novo_saldo
18
19     results = plpy.execute(insert_operacao, [conta, str(datetime.today()), operacao, valor])
20
21 ' LANGUAGE plpython3u;
  
```

Figura 10. Script de criação da função de gatilho `nova_operacao()`

De forma detalhada, será explicado a seguir o que faz cada linha da função de gatilho em questão.

- Linha 1: Uma nova função é criada com o nome `nova_operacao` e retornada para o gatilho que a disparar.
- Linha 2: Importando o módulo `datetime` que será utilizado para guardar a data e hora atual da operação.
- Linha 4: A variável `conta` recebe o numero da conta que disparou o gatilho.
- Linha 5: Utilizando o dicionário `TD`, com a tag `TD["old"]` ["saldo"] atribuímos a variável `antigo_saldo` o valor do atributo `saldo` antes do gatilho ser disparado.
- Linha 6: O mesmo conceito é utilizado, mas agora buscando o novo valor do atributo `acionador` com a tag `TD["new"]`.
- Linha 8: A instrução `plpy.prepare` está preparando o plano de execução para uma inserção. É chamado com uma string de inserção e uma lista de tipos de parâmetros. "integer" é o tipo da variável que você passará para o \$1.
- Linhas 11 a 13: Uma condição é dada: Se a variável `novo_saldo` apresenta-se maior que a variável `antigo_saldo`, ou seja, se o valor do saldo aumentou, a operação em questão trata-se de um depósito e deve ser registrado como "Depósito" na nossa tabela, assim como está sendo atribuído na variável `operacao`, e o valor atribuído na atualização é recebido por `valor`.
- Linhas 15 a 17: Diferente do depósito, se o o valor alterado se tornou menor em relação ao valor anterior, a operação deve ser tratada como um saque, e as atribuições são feitas como na condição anterior, mas com os atributos referenciando um saque.
- Linha 19: Após preparar a inserção na linha 8 e obter os valores a serem inseridos na nova linha da tabela, utilizamos uma variante da função `plpy.execute` para executar a inserção. Onde `str(datetime.today())` nos traz a data e hora atual para preenchermos as informações da operação.
- Linha 21: Indicação da linguagem procedural sendo utilizada.

Criada a função de gatilho, definiremos agora o gatilho ao qual chamaremos de `update_saldo`, associando, em seguida, à tabela de contas, como pode ser visto na Figura 11.

```

1 CREATE TRIGGER update_saldo
2 AFTER UPDATE ON conta
3 FOR EACH ROW
4 WHEN (OLD.saldo IS DISTINCT FROM NEW.saldo)
5 EXECUTE PROCEDURE nova_operacao();

```

Figura 11. Script de criação do gatilho `update_saldo`

Onde:

- Linha 1: um novo gatilho é criado com o nome `update_saldo`.
- Linha 2: Com a instrução `AFTER UPDATE` dizemos que o gatilho será disparado **após** uma instrução SQL `UPDATE`, logo após associamos o gatilho a tabela `conta`.
- Linha 3: Marcando o gatilho com `EACH ROW`, nos certificaremos de que o gatilho será chamado uma vez para cada linha que a operação modificar.

- Linhas 4 e 5: Com "OLD.saldo IS DISTINCT FROM NEW.saldo" enfatizamos que o gatilho será chamado apenas quando uma coluna em específico (no caso conta.saldo) for modificada no UPDATE. E então, se essa condição for aceita, o gatilho é disparado executando a função de gatilho nova_operacao().

Ao atualizarmos o valor de saldo de qualquer conta da nossa base de dados, podemos ver que uma nova linha será acrescentada a tabela operacoes com as informações da atualização do saldo. Para testarmos o gatilho, iremos adicionar uma nova conta com o saldo inicial de R\$900,00, como pode ser visto na Figura 12.



The screenshot shows a PostgreSQL query editor window titled "Banco on postgres@PostgreSQL 10". The query is an INSERT statement into the 'conta' table. The query is executed successfully, and the output shows "INSERT 0 1" and "Query returned successfully in 251 msec."

```

1 INSERT INTO conta (numero, saldo, agencia, cliente) VALUES
2 (702, 900, 101, 501);

```

INSERT 0 1

Query returned successfully in 251 msec.

Figura 12. Inserção da nova conta para teste de gatilho

Agora, faremos uma atualização no saldo dessa conta, acrescentando R\$150,00 obtendo assim, um saldo de R\$1.050,00 no total. Veja na Figura 13.



The screenshot shows a PostgreSQL query editor window titled "Banco on postgres@PostgreSQL 10". The query is an UPDATE statement that increases the 'saldo' of the account with 'numero' 702 by 150. The query is executed successfully, and the output shows the updated data in a table format.

```

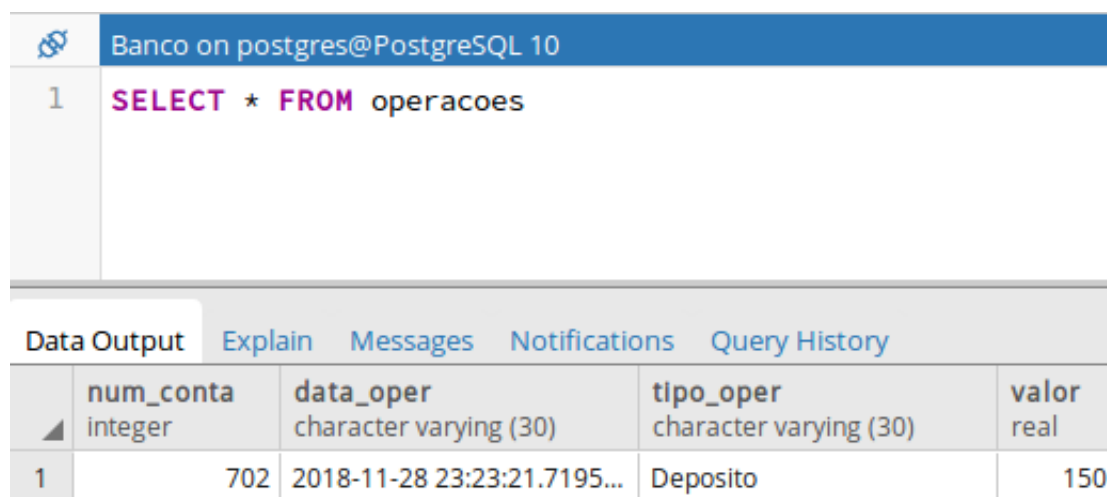
1 UPDATE conta
2 SET saldo = saldo + 150
3 WHERE numero = 702;
4
5 SELECT * FROM conta

```

	numero integer	saldo real	agencia integer	cliente integer
1	701	1300	101	501
2	702	1050	101	501

Figura 13. Atualização da conta

Por consequência, o gatilho será disparado e como pode ser visualizado na Figura 14 uma nova linha será inserida na tabela `operacoes` com o número da conta em operação, o tipo de operação como "Depósito", a data atual da operação e o valor adicionado (no caso R\$150,00).



The screenshot shows a PostgreSQL interface with a query window at the top containing the command: `SELECT * FROM operacoes`. Below the query window, there are tabs for 'Data Output', 'Explain', 'Messages', 'Notifications', and 'Query History'. The 'Data Output' tab is active, displaying a table with the following data:

	num_conta integer	data_oper character varying (30)	tipo_oper character varying (30)	valor real
1	702	2018-11-28 23:23:21.7195...	Deposito	150

Figura 14. Linha inserida na tabela operacoes

6. Considerações finais

Com este tutorial, esperamos que você, leitor, tenha aumentado seu conhecimento sobre Python, PostgreSQL e afins, mas principalmente, sobre o PL/Python, tema central deste documento. Esperamos ainda ter despertado a curiosidade necessária para que você se sinta empolgado a fazer seus próprios exemplos e também a utilizar o PL/Python no dia a dia.

Referências

- CHAPTER 38. PL/pgSQL - SQL Procedural Language. Disponível em: <https://www.postgresql.org/docs/8.3/plpgsql.html>. Acesso em: 27 nov. 2018.
- CHAPTER 45. PL/Python - Python Procedural Language. Disponível em: <https://www.postgresql.org/docs/10/plpython.html>. Acesso em: 22 nov. 2018.
- PL/PYTHON: Programando em Python no PostgreSQL. Disponível em: <https://speakerdeck.com/julianometalsp/python-programando-em-python-no-postgresql?slide=17>. Acesso em: 23 nov. 2018.
- PL/PYTHON – Pythonic Trigger Functions for Postgres. Disponível em: <https://anitagraser.com/2010/12/10/plpython-pythonic-trigger-functions-for-postgres/>. Acesso em: 23 nov. 2018.
- POSTGRESQL. Disponível em: <https://www.postgresql.org/>. Acesso em: 22 nov. 2018.
- PYTHON. Disponível em: <https://www.python.org/>. Acesso em: 22 nov. 2018.