1 Introdução

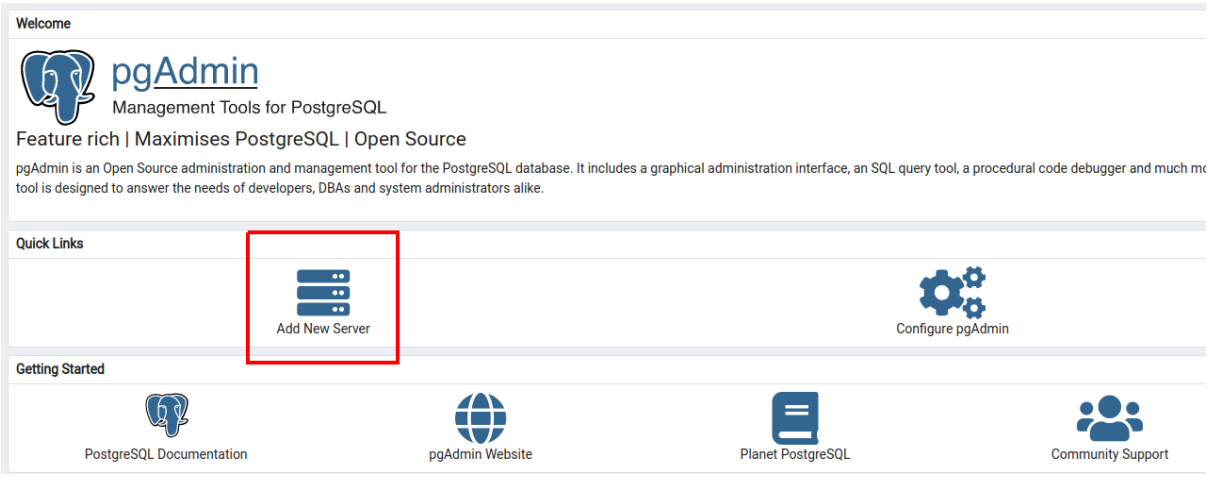
Neste material, estudaremos sobre blocos de código denominados **stored procedures**. Um stored procedure é um dos tipos de blocos de código armazenados pelo servidor. O link 1.1 dá acesso a uma página da documentação oficial sobre o assunto.

Link 1.1
https://www.postgresql.org/docs/current/sql-createprocedure.html

2 Passo a passo

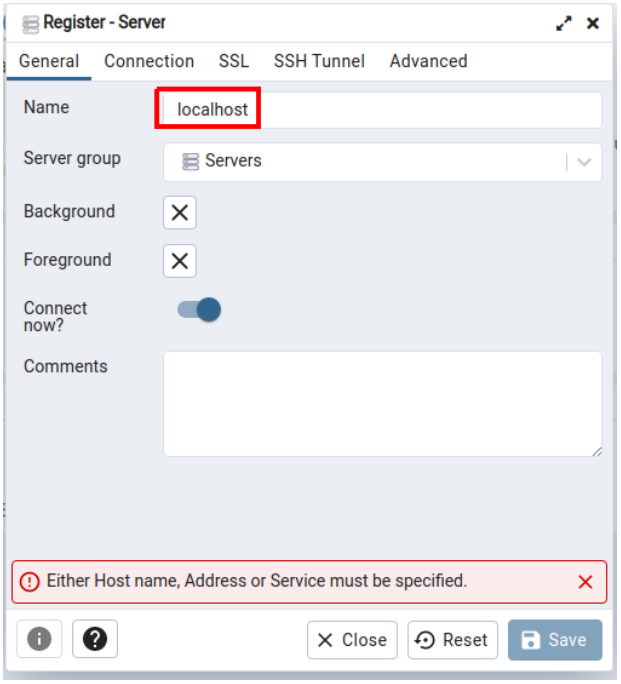
2.1 (Criando um servidor) Caso ainda não possua um servidor, abra o pgAdmin4 e clique em **Add New Server**, como mostra a Figura 2.1.1.

Figura 2.1.1



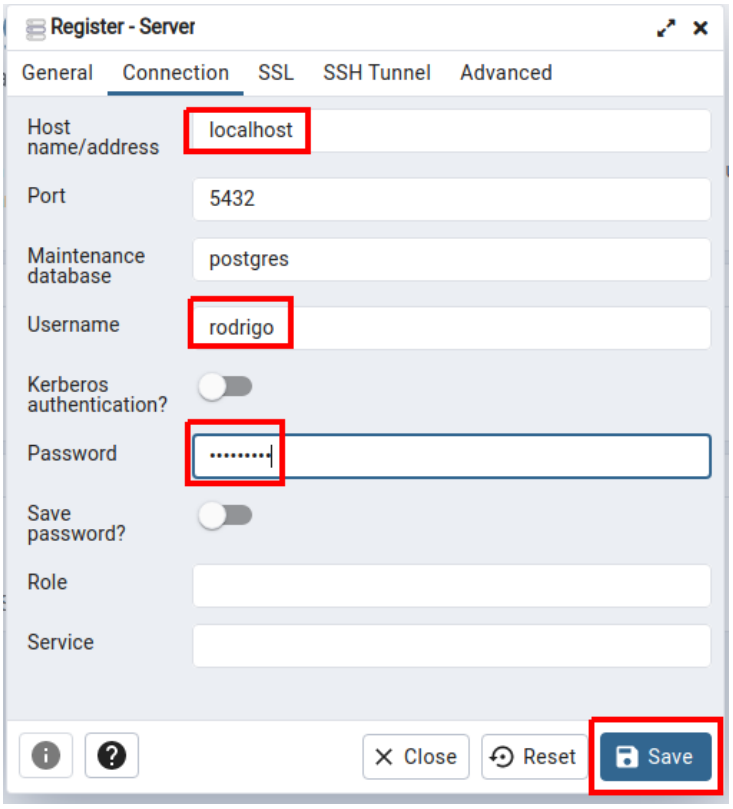
O nome do servidor pode ser algo que lhe ajude a lembrar a razão de ser dele. Como é um servidor que está executando localmente, podemos chamá-lo de algo como **localhost**, como na Figura 2.1.2. Depois de preencher o nome, clique na aba **Connection**.

Figura 2.1.2



Agora clique na aba **Connection**, como na Figura 2.1.3. Preencha os campos como destacado e clique em **Save**.

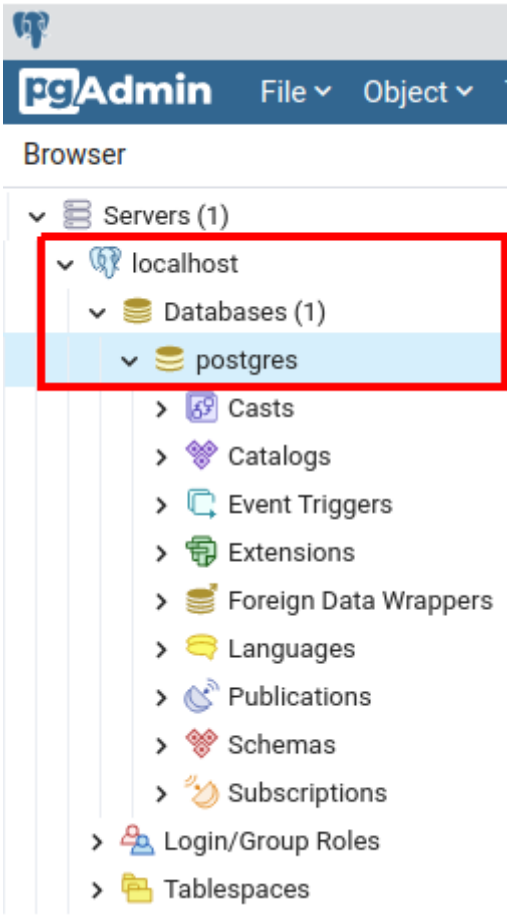
Figura 2.1.3



Nota. Usuário e senha dependerão de suas configurações. No Windows, é comum a existência de um usuário chamado postgres com a senha também igual a postgres.

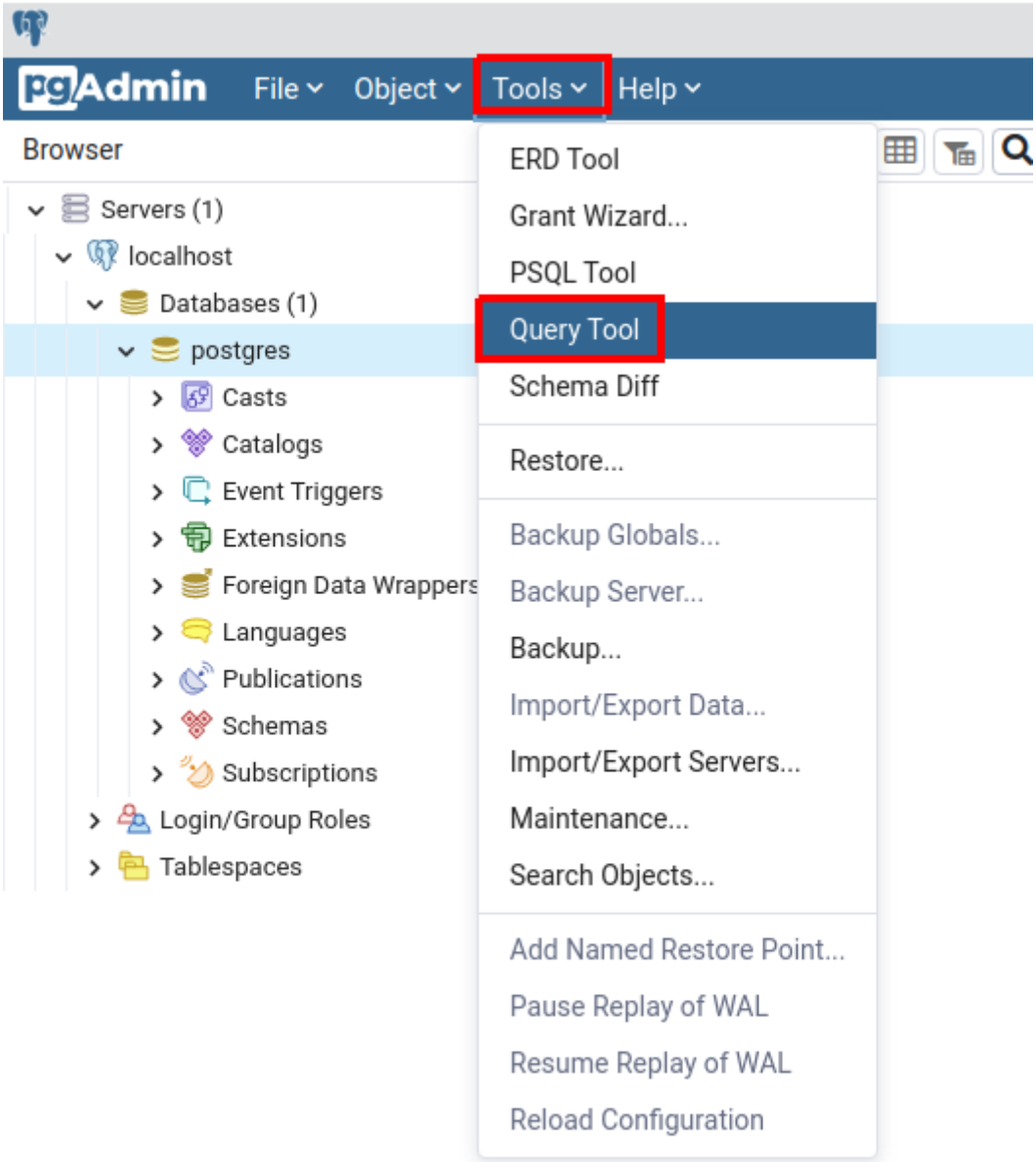
No canto superior esquerdo, encontre o seu servidor e clique sobre ele. Expanda **Databases** e encontre o database chamado **postgres**, cuja existência é muito comum. Veja a Figura 2.1.4.

Figura 2.1.4



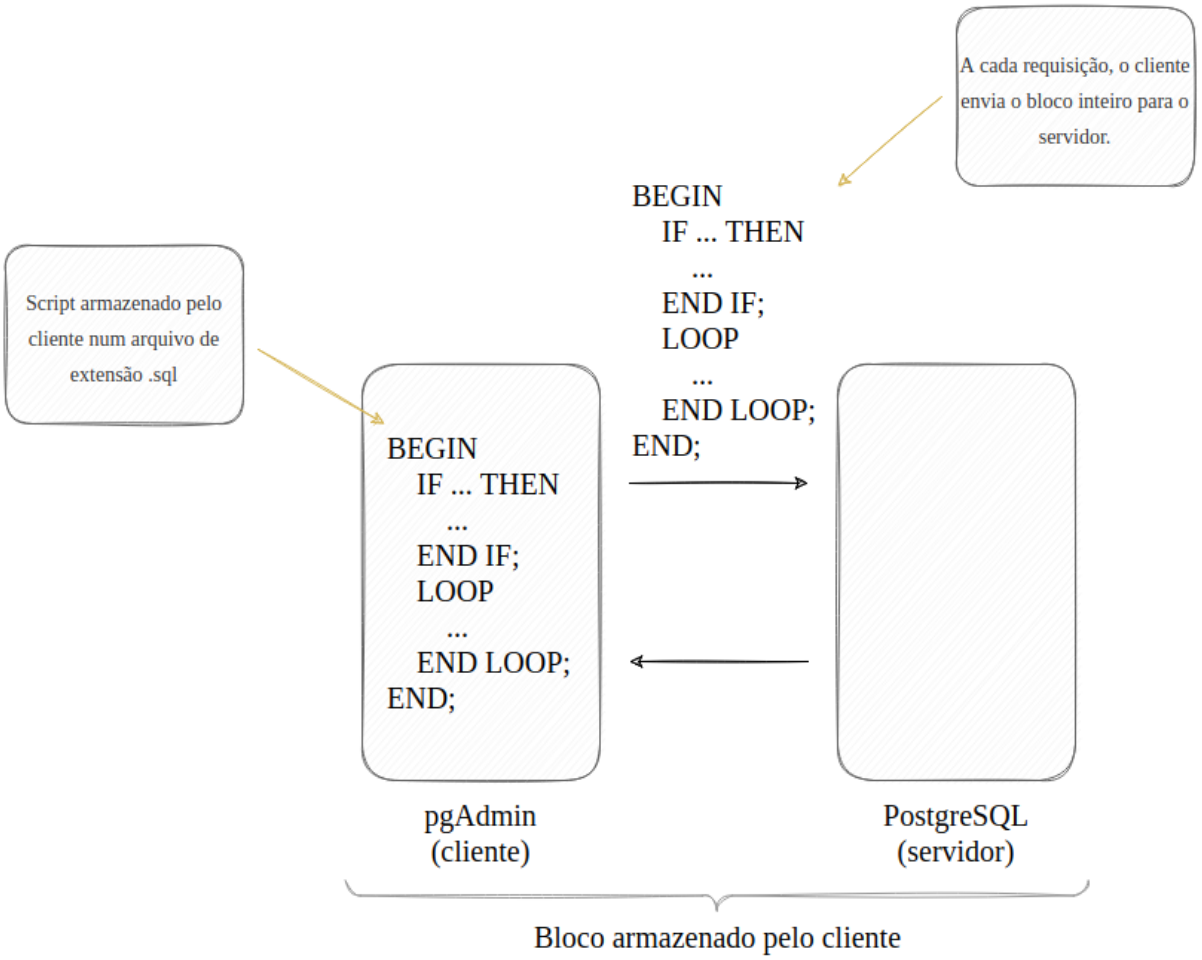
Para abrir um editor em que possa digitar seus comandos SQL, clique em **Tools >> Query Tool**, como mostra a Figura 2.1.5.

Figura 2.1.5


pgAdmin
File
Object
Tools
Help
Browser
Servers (1)
localhost
Databases (1)
postgres
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas
Subscriptions
Login/Group Roles
Tablespaces
ERD Tool
Grant Wizard...
PSQL Tool
Query Tool
Schema Diff
Restore...
Backup Globals...
Backup Server...
Backup...
Import/Export Data...
Import/Export Servers...
Maintenance...
Search Objects...
Add Named Restore Point...
Pause Replay of WAL
Resume Replay of WAL
Reload Configuration

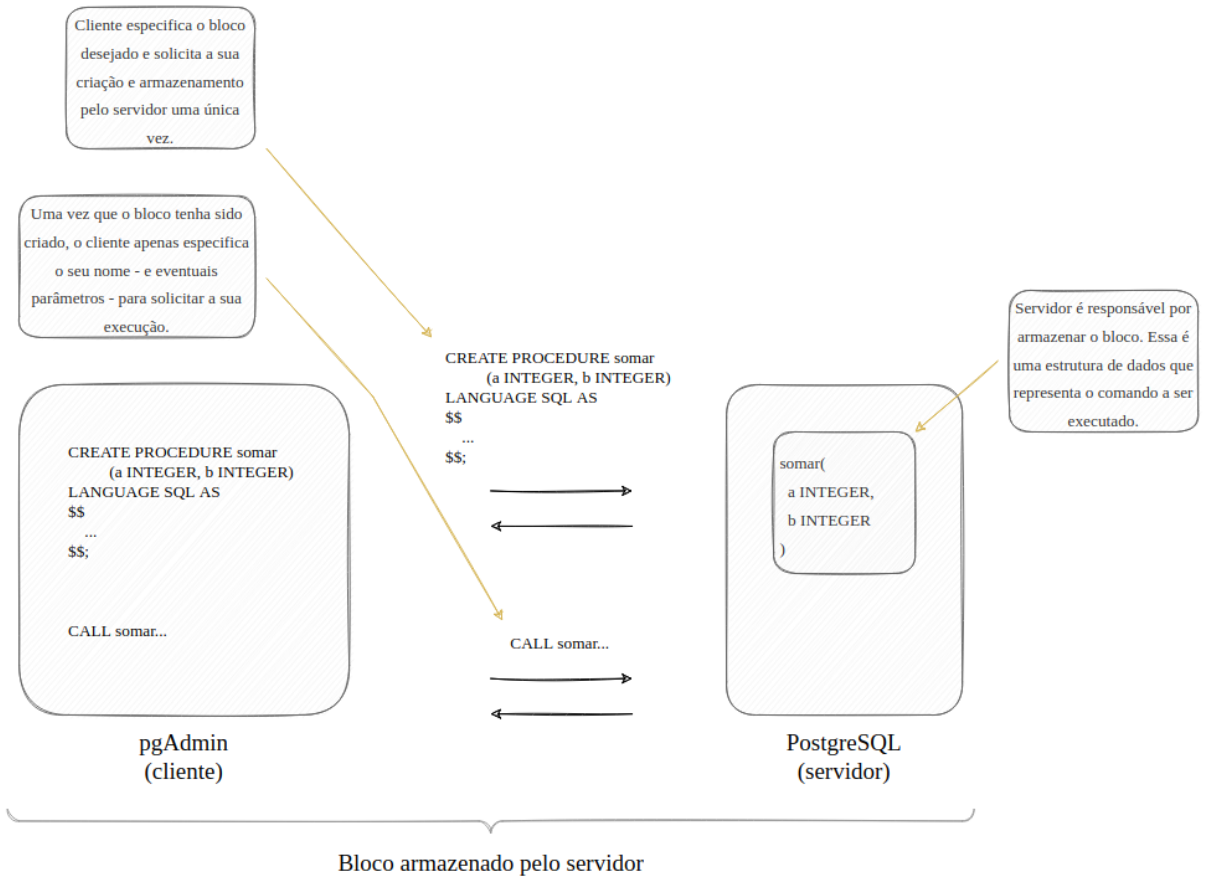
2.2 (Blocos de código armazenados no cliente e no servidor) Há blocos de código que são armazenados pelo próprio cliente, geralmente num arquivo de extensão .sql. Veja a Figura 2.2.1.

Figura 2.2.1



Há também diferentes tipos de blocos armazenados pelo servidor. Um desses tipos leva o nome de **stored procedure**. Veja a Figura 2.2.2.

Figura 2.2.2



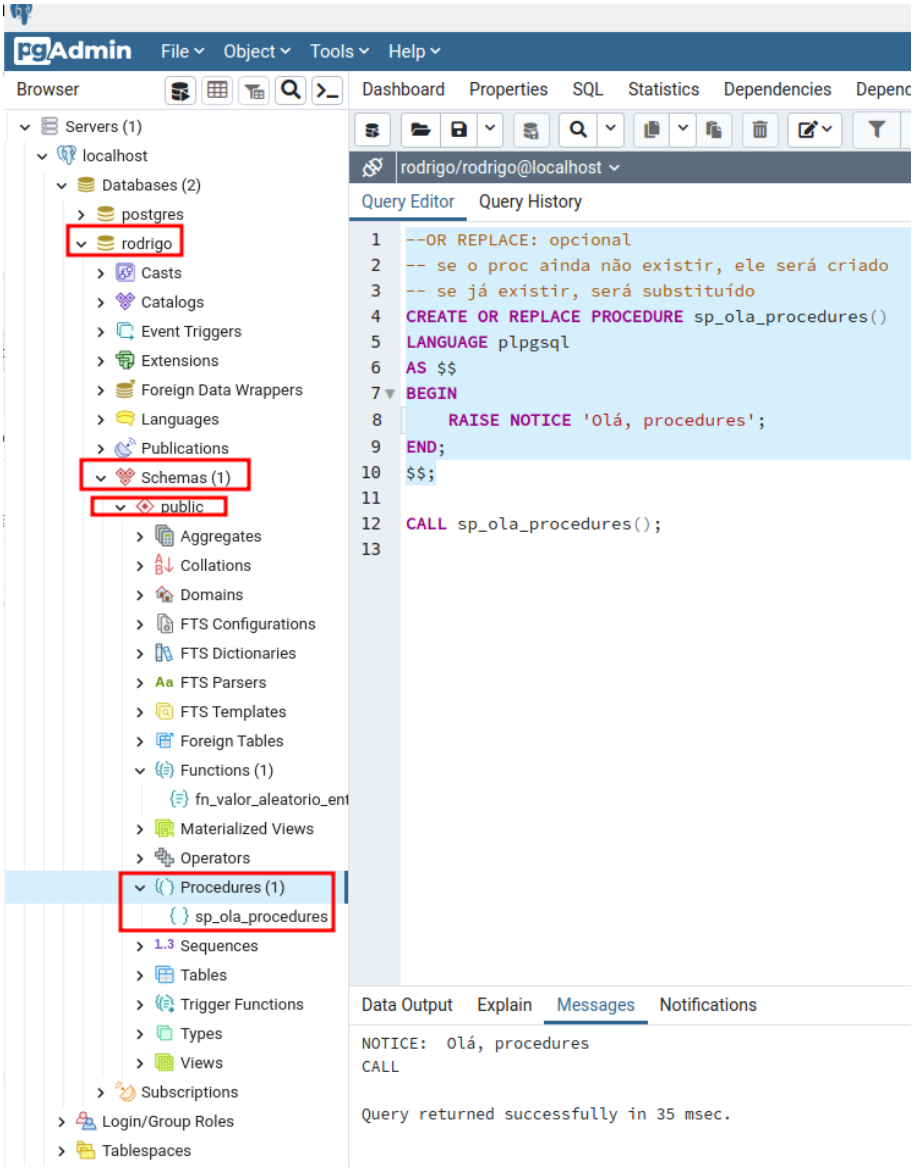
2.3 (Stored procedure: Olá, procedures!) O Bloco de Còdigo 2.3.1 mostra como fazer a criação de um stored procedure que exibe uma mensagem simples.

Bloco de Código 2.3.1

```
--OR REPLACE: opcional
-- se o proc ainda não existir, ele será criado
-- se já existir, será substituído
CREATE OR REPLACE PROCEDURE sp_ola_procedures()
LANGUAGE plpgsql
AS $$
BEGIN
        RAISE NOTICE 'Olá, procedures';
END;
$$;
```

O resultado esperado se parece com aquele exibido pela Figura 2.3.1.

Figura 2.3.1



O Bloco de Código 2.3.2, por sua vez, mostra como executar o procedimento criado.

Bloco de Código 2.3.2

```
CALL sp_ola_procedures( );
```

2.4 (Stored procedure: **usando um parâmetro**) Stored procedures podem receber valores como parâmetro. Veja um exemplo no Bloco de Código 2.4.1.

Bloco de Código 2.4.1

```
-- criando
CREATE OR REPLACE PROCEDURE sp_ola_usuario (nome VARCHAR(200))
LANGUAGE plpgsql
AS $$
BEGIN
        -- acessando parâmetro pelo nome
        RAISE NOTICE 'Olá, %', nome;
        -- assim também vale
        RAISE NOTICE 'Olá, %', $1;
END;
$$;

--colocando em execução
CALL sp_ola_usuario('Pedro');
```

2.5 (Parâmetros e seus diferentes modos) Os parâmetros de um stored procedure têm um modo associado. Veja.

- **IN**: parâmetros de “entrada”. Servem para que o cliente possa enviar dados a serem utilizados pelo procedimento. Não podem ser alterados. Quando um parâmetro não tem modo especificado explicitamente, **IN é o seu modo padrão**.
- **OUT**: parâmetros de saída. Podem ser utilizados para que o procedimento devolva valores ao cliente. Diferente dos parâmetros de modo IN, eles devem ter valor atribuído antes de o procedimento terminar.
- **INOUT**: É uma combinação dos modos IN e OUT. Quando o parâmetro tem modo igual a INOUT, o cliente é capaz de entregar um valor ao procedimento que o utiliza em seu processamento e, eventualmente, lhe atribui novo valor, o qual é devolvido ao cliente.

2.6 (Stored procedure: Parâmetros IN) O procedimento do Bloco de Código 2.6.1 promete calcular o maior valor entre dois parâmetros recebidos.

Bloco de Código 2.6.1

```
--criando
--ambos são IN, pois IN é o padrão
CREATE OR REPLACE PROCEDURE sp_acha_maior (IN valor1 INT, valor2 INT)
LANGUAGE plpgsql
AS $$
BEGIN
	IF valor1 > valor2 THEN
		RAISE NOTICE '% é o maior', $1;
	ELSE
		RAISE NOTICE '% é o maior', $2;
	END IF;
END;
$$

-- colocando em execução
CALL sp_acha_maior (2, 3);
```

2.7 (Stored procedure: Parâmetros OUT) Observe como os procedimentos não têm um tipo de retorno. De fato, eles não podem fazer uso da instrução RETURN. Entretanto, o uso de parâmetros em modo OUT permite que um procedimento entregue um resultado ao cliente. Veja o exemplo do Bloco de Código 2.7.1.

Bloco de Código 2.7.1

```
-- aqui estamos removendo o proc de nome sp_acha_maior para poder reutilizar o nome
DROP PROCEDURE IF EXISTS sp_acha_maior;
CREATE OR REPLACE PROCEDURE sp_acha_maior (OUT resultado INT, IN valor1 INT, IN valor2 INT)
LANGUAGE plpgsql
AS $$
BEGIN
	CASE
		WHEN valor1 > valor2 THEN
			$1 := valor1;
		ELSE
			resultado := valor2;
	END CASE;
END;
$$

--colocando em execução
DO $$
DECLARE
	resultado INT;
BEGIN
	CALL sp_acha_maior (resultado, 2, 3);
	RAISE NOTICE '% é o maior', resultado;
END;
$$
```

2.8 (Stored procedure: Parâmetros INOUT) Quando um parâmetro tem modo igual a INOUT, como o nome sugere, ele é tanto de entrada quanto de saída. Isso quer dizer que o código cliente pode utilizar um mesmo parâmetro para enviar um valor de interesse para que o procedimento operar e esperar que ele - o parâmetro - esteja alterado, armazenando o resultado esperado, quando o procedimento terminar. Veja o Bloco de Código 2.8.1.

Bloco de Código 2.8.1

```
DROP PROCEDURE IF EXISTS sp_acha_maior;
-- criando
CREATE OR REPLACE PROCEDURE sp_acha_maior (INOUT valor1 INT, IN valor2 INT)
LANGUAGE plpgsql
AS $$
BEGIN
	IF valor2 > valor1 THEN
		valor1 := valor2;
	END IF;
END;
$$

-- colocando em execução
DO
$$
DECLARE
	valor1 INT := 2;
	valor2 INT := 3;

BEGIN
	CALL sp_acha_maior(valor1, valor2);
	RAISE NOTICE '% é o maior', valor1;
END;
$$
```

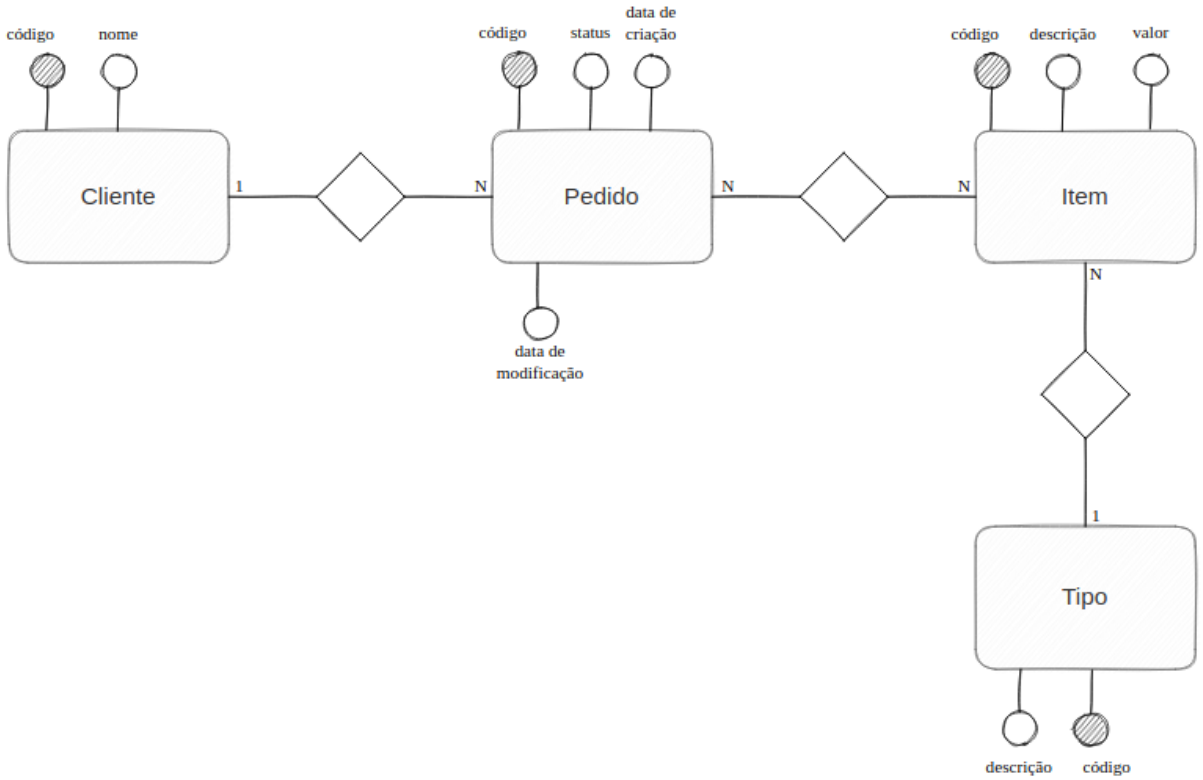
2.9 (Stored procedure: parâmetros VARIADIC) Um parâmetro VARIADIC permite que o cliente especifique uma coleção de tamanho maior ou igual a 1. Veja o exemplo do Bloco de Código 2.9.1.

Bloco de Código 2.9.1

```
CREATE OR REPLACE PROCEDURE sp_calcula_media ( VARIADIC  valores INT [])
LANGUAGE plpgsql
AS $$
DECLARE
	media NUMERIC(10, 2) := 0;
	valor INT;

BEGIN
	FOREACH valor IN ARRAY valores LOOP
		media := media + valor;
	END LOOP;
	--array_length calcula o número de elementos no array. O segundo parâmetro é o
número de dimensões dele
	RAISE NOTICE 'A média é %', media / array_length(valores, 1);
END;
$$

-- 1 parâmetro
CALL sp_calcula_media(1);

-- 2 parâmetros
CALL sp_calcula_media(1, 2);

-- 6 parâmetros
CALL sp_calcula_media(1, 2, 5, 6, 1, 8);

-- não funciona
CALL sp_calcula_media (ARRAY[1, 2]);
```

2.10 (Stored procedures: implementação de um restaurante) A seguir, implementaremos o funcionamento básico de um restaurante utilizando stored procedures. O modelo de dados utilizado aparece na Figura 2.10.1.

Figura 2.10.1



(Criação de tabelas) O Bloco de Código 2.10.1 mostra a criação das tabelas e a inserção de alguns itens.

Bloco de Código 2.10.1

```
DROP TABLE tb_cliente;
CREATE TABLE tb_cliente (
	cod_cliente SERIAL PRIMARY KEY,
	nome VARCHAR(200) NOT NULL
);


SELECT * FROM tb_pedido;
DROP TABLE tb_pedido;
CREATE TABLE IF NOT EXISTS tb_pedido(
	cod_pedido SERIAL PRIMARY KEY,
	data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
	data_modificacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
	status VARCHAR DEFAULT 'aberto',
	cod_cliente INT NOT NULL,
	CONSTRAINT fk_cliente FOREIGN KEY (cod_cliente) REFERENCES
tb_cliente(cod_cliente)
);
```

```
DROP TABLE tb_tipo_item;
CREATE TABLE tb_tipo_item(
	cod_tipo SERIAL PRIMARY KEY,
	descricao VARCHAR(200) NOT NULL
);
INSERT INTO tb_tipo_item (descricao) VALUES ('Bebida'), ('Comida');
SELECT * FROM tb_tipo_item;

DROP TABLE tb_item;
CREATE TABLE IF NOT EXISTS tb_item(
	cod_item SERIAL PRIMARY KEY,
	descricao VARCHAR(200) NOT NULL,
	valor NUMERIC (10, 2) NOT NULL,
	cod_tipo INT NOT NULL,
	CONSTRAINT fk_tipo_item FOREIGN KEY (cod_tipo) REFERENCES
tb_tipo_item(cod_tipo)
);

INSERT INTO tb_item (descricao, valor, cod_tipo) VALUES
('Refrigerante', 7, 1), ('Suco', 8, 1), ('Hamburguer', 12, 2), ('Batata frita', 9, 2);
SELECT * FROM tb_item;

DROP TABLE tb_item_pedido;
CREATE TABLE IF NOT EXISTS tb_item_pedido(
	--surrogate key, assim cod_item pode repetir
	cod_item_pedido SERIAL PRIMARY KEY,
	cod_item INT,
	cod_pedido INT,
	CONSTRAINT fk_item FOREIGN KEY (cod_item) REFERENCES tb_item (cod_item),
	CONSTRAINT fk_pedido FOREIGN KEY (cod_pedido) REFERENCES tb_pedido
(cod_pedido)
);
```

(Cadastro de novos clientes) O Bloco de Código 2.10.2 mostra um procedimento que faz o cadastro de clientes.

Bloco de Código 2.10.2

```
-- cadastro de cliente
-- se um parâmetro com valor DEFAULT é especificado, aqueles que aparecem depois dele também deve ter valor DEFAULT
CREATE OR REPLACE PROCEDURE sp_cadastrar_cliente (IN nome VARCHAR(200), IN codigo INT DEFAULT NULL)
LANGUAGE plpgsql
AS $$
BEGIN
	IF codigo IS NULL THEN
		INSERT INTO tb_cliente (nome) VALUES (nome);
	ELSE
		INSERT INTO tb_cliente (codigo, nome) VALUES (codigo, nome);
	END IF;
END;
$$
CALL sp_cadastrar_cliente ('João da Silva');
CALL sp_cadastrar_cliente ('Maria Santos');
SELECT * FROM tb_cliente;
```

(Inserção de novos pedidos, ainda sem itens) O procedimento do Bloco de Código 2.10.3 faz a criação de um pedido para um cliente. A ideia é simular a entrada do cliente no restaurante, momento em que ele pega a sua comanda.

Bloco de Código 2.10.3

```
-- criar um pedido, como se o cliente entrasse no restaurante e pegasse a comanda
CREATE OR REPLACE PROCEDURE sp_criar_pedido (OUT cod_pedido INT, cod_cliente INT)
LANGUAGE plpgsql
AS $$
BEGIN
	INSERT INTO tb_pedido (cod_cliente) VALUES (cod_cliente);
	-- obtém o último valor gerado por SERIAL
	SELECT LASTVAL() INTO cod_pedido;
END;
$$

DO
$$
DECLARE
	--para guardar o código de pedido gerado
	cod_pedido INT;
	-- o código do cliente que vai fazer o pedido
	cod_cliente INT;
BEGIN
	-- pega o código da pessoa cujo nome é "João da Silva"
	SELECT c.cod_cliente FROM tb_cliente c WHERE nome LIKE 'João da Silva' INTO cod_cliente;
	--cria o pedido
	CALL sp_criar_pedido (cod_pedido, cod_cliente);
	RAISE NOTICE 'Código do pedido recém criado: %', cod_pedido;

END;
$$
```

(Adição de item a um pedido) O procedimento do Bloco de Código 2.10.4 viabiliza a associação de itens a um determinado pedido. Ele deve ser chamado quando um cliente desejar um novo item.

Bloco de Código 2.10.4

```
-- adicionar um item a um pedido
CREATE OR REPLACE PROCEDURE sp_adicionar_item_a_pedido (IN cod_item INT, IN
cod_pedido INT)
LANGUAGE plpgsql
AS $$
BEGIN
	--insere novo item
	INSERT INTO tb_item_pedido (cod_item, cod_pedido) VALUES ($1, $2);
	--atualiza data de modificação do pedido
	UPDATE tb_pedido p SET data_modificacao = CURRENT_TIMESTAMP WHERE
p.cod_pedido = $2;
END;
$$

CALL sp_adicionar_item_a_pedido (1, 1);
SELECT * FROM tb_item_pedido;
SELECT * FROM tb_pedido;
```

(Cálculo do valor total de um pedido: somatório dos valores de seus itens) A qualquer momento, é natural que um cliente queira saber quanto já gastou, especialmente quando for pagar a conta. O procedimento do Bloco de Código 2.10.5 faz as contas para um determinado pedido.

Bloco de Código 2.10.5

```
--calcular valor total de um pedido
DROP PROCEDURE sp_calcular_valor_de_um_pedido;
CREATE OR REPLACE PROCEDURE sp_calcular_valor_de_um_pedido (IN p_cod_pedido
INT, OUT valor_total INT)
LANGUAGE plpgsql
AS $$
BEGIN
	SELECT SUM(valor) FROM
		tb_pedido p
		INNER JOIN tb_item_pedido ip ON
		p.cod_pedido = ip.cod_pedido
		INNER JOIN tb_item i ON
		i.cod_item = ip.cod_item
		WHERE p.cod_pedido = $1
		INTO $2;
END;
$$

DO $$
DECLARE
	valor_total INT;
BEGIN
	CALL sp_calcular_valor_de_um_pedido(1, valor_total);
	RAISE NOTICE 'Total do pedido %: R$%', 1, valor_total;

END;
$$
```

O procedimento do Bloco de Código 2.10.6 fecha um pedido, desde que o valor entregue pelo cliente seja suficiente para pagar a conta.

Bloco de Código 2.10.6

```
CREATE OR REPLACE PROCEDURE sp_fechar_pedido (IN valor_a_pagar INT, IN
cod_pedido INT)
LANGUAGE plpgsql
AS $$
DECLARE
	valor_total INT;
BEGIN
	--vamos verificar se o valor_a_pagar é suficiente
	CALL sp_calcular_valor_de_um_pedido (cod_pedido, valor_total);
	IF valor_a_pagar < valor_total THEN
		RAISE 'R$% insuficiente para pagar a conta de R$%', valor_a_pagar,
valor_total;
	ELSE
		UPDATE tb_pedido p SET
		data_modificacao = CURRENT_TIMESTAMP,
		status = 'fechado'
		WHERE p.cod_pedido = $2;
	END IF;
END;
$$

DO $$
BEGIN
	CALL sp_fechar_pedido(200, 1);
END;
$$
SELECT * FROM tb_pedido;
```

(Cálculo do troco) Embora simples, o cálculo do troco pode ser útil em diferentes contextos. Por isso, pode ser de interesse fazer a implementação usando um procedimento, como mostra o Bloco de Código 2.10.7.

Bloco de Código 2.10.7

```
CREATE OR REPLACE PROCEDURE sp_calcular_troco (OUT troco INT, IN valor_a_pagar
INT, IN valor_total INT)
LANGUAGE plpgsql
AS $$
BEGIN
	troco := valor_a_pagar - valor_total;
END;
$$

DO
$$
DECLARE
 troco INT;
 valor_total INT;
 valor_a_pagar INT := 100;
BEGIN
	CALL sp_calcular_valor_de_um_pedido(1, valor_total);
	CALL sp_calcular_troco (troco, valor_a_pagar, valor_total);
	RAISE NOTICE 'A conta foi de R$% e você pagou %, portanto, seu troco é de R$%.',
valor_total, valor_a_pagar, troco;
END;
$$
```

O Bloco de Código 2.10.8 mostra um procedimento que calcula as notas a serem utilizadas para compor um determinado valor de troco.

Bloco de Código 2.10.8

```
CREATE OR REPLACE PROCEDURE sp_obter_notas_para_compor_o_troco (OUT resultado
VARCHAR(500), IN troco INT)
LANGUAGE plpgsql
AS $$
DECLARE
	notas200 INT := 0;
	notas100 INT := 0;
	notas50 INT := 0;
	notas20 INT := 0;
	notas10 INT := 0;
	notas5 INT := 0;
	notas2 INT := 0;
	moedas1 INT := 0;
BEGIN
	notas200 := troco / 200;
	notas100 := troco % 200 / 100;
	notas50 := troco % 200 % 100 / 50;
	notas20 := troco % 200 % 100 % 50 / 20;
	notas10 := troco % 200 % 100 % 50 % 20 / 10;
	notas5 := troco % 200 % 100 % 50 % 20 % 10 / 5;
	notas2 := troco % 200 % 100 % 50 % 20 % 10 % 5 / 2;
	moedas1 := troco % 200 % 100 % 50 % 20 % 10 % 5 % 2;
```

```
	resultado := concat (
		-- E é de escape. Para que \n tenha sentido
		-- || é um operador de concatenação
		'Notas de 200: ',
		notas200 || E'\n',
		'Notas de 100: ',
		notas100 || E'\n',
		'Notas de 50: ',
		notas50 || E'\n',
		'Notas de 20: ',
		notas20 || E'\n',
		'Notas de 10: ',
		notas10 || E'\n',
		'Notas de 5: ',
		notas5 || E'\n',
		'Notas de 2: ',
		notas2 || E'\n',
		'Moedas de 1: ',
		moedas1 || E'\n'
	);
END;
$$

DO
$$
DECLARE
	resultado VARCHAR(500);
	troco INT := 43;
BEGIN
	CALL sp_obter_notas_para_compor_o_troco (resultado, troco);
	RAISE NOTICE '%', resultado;
END;
$$
```

Bibliografia

LOPES, A.; GARCIA, G..**Introdução à Programação - 500 Algoritmos Resolvidos**. 1a Ed., Elsevier, 2002.

PostgreSQL: Documentation: 14: PostgreSQL 14.2 Documentation. PostgreSQL, 2022. Disponível em <https://www.postgresql.org/docs/current/index.html>. Acesso em abril de 2022.