



# Desenvolvimento de Sistemas

---

## Programação com SQL: *triggers*, *stored procedures* e *stored functions*

### Introdução

Assim como o Portugol, visto anteriormente no curso, também é possível escrever regras lógicas no banco de dados. Além das funcionalidades básicas que já se conhece para a manipulação e definição de dados (estruturando, consultando, inserindo etc.), é possível criar instruções complexas que buscam potencializar as capacidades do banco de dados, por meio do uso de regras condicionais, laços de repetição, variáveis e outros conceitos que já se conhece.

O MySQL, assim como outros SGBDs (Sistemas de Gerenciamento de Banco de Dados), contém uma linguagem de programação que permite a produção dessas instruções mais complexas, aumentando as potencialidades do seu banco de dados. Nesta seção, será utilizado o MySQL como base para seus estudos, mas os conceitos apresentados podem ser encontrados em diversos outros SGBDs, como o PostgreSQL, o Microsoft SQL Server, o Oracle Database, entre outros.

Também é importante salientar que existem pequenas diferenças na **sintaxe** da SQL (*structured query language*) quando comparada aos diferentes SGBDs. Isso quer dizer que as instruções podem não ser exatamente iguais, mudando algumas palavras ou a sua ordem. Entretanto, as instruções são **semanticamente** semelhantes. Ou seja, quando se olha para o Portugol, para a SQL no MySQL e para a SQL no Oracle Database, encontra-se um padrão, sendo necessário apenas visitar a documentação respectiva para conhecer a sintaxe exata, utilizando as palavras-chave corretas, na ordem certa.

Com este potencial da utilização de instruções complexas no banco de dados, será possível conhecer e abordar alguns outros conceitos. Por meio do uso da SQL, pode-se criar códigos compilados com diferentes instruções, que recebem parâmetros de entrada e definem, opcionalmente, um ou mais parâmetros de saída, podendo esse comando ser salvo e reutilizado diversas vezes no seu banco de dados (também conhecido como ***stored procedures*** – ou, em português, “procedimentos armazenados”). Também é possível criar funções que retornam um único valor e podem ser utilizadas dentro de outras declarações SQL (como o **SELECT** e o **INSERT**), chamadas de ***stored functions*** (ou “funções armazenadas”). Por último, podem ser criados gatilhos, adicionando, ao seu banco de dados, o comportamento por **reação**. Ou seja, define-se que, quando algo específico acontecer no banco de dados, ele mesmo, de maneira automatizada, realiza uma série de instruções, conhecida como ***trigger***. Esses três conceitos serão estudados neste material.

Diversos projetos optam por encapsular suas regras de negócio no banco de dados por meio do uso de *stored procedures*, *stored functions* e *triggers*. Dessa forma, diminui-se a complexidade do código-fonte do projeto, uma vez que todas as regras que acontecem no banco de dados são abstratas do ponto de vista do código-fonte, que se comunica com o banco de dados, podendo focar mais esforços na experiência de uso do usuário. Além disso, essa prática auxilia a centralizar as regras de negócio em um único lugar (evitando a duplicidade de código).

Para executar os exemplos desse conteúdo, execute o seguinte *script* SQL.

```
CREATE DATABASE senac_terrenos;

USE senac_terrenos;

CREATE TABLE vendedores (
  id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(25) NOT NULL
);

CREATE TABLE cidades (
  id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(25) NOT NULL,
  UF CHAR(2) NOT NULL,
  custo_metro_quadrado DECIMAL(6,2) NOT NULL
);

CREATE TABLE terrenos(
  id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  id_cidade INT(6) UNSIGNED,
  largura DECIMAL(8,2) NOT NULL,
  comprimento DECIMAL(8,2) NOT NULL,
  vendido BOOL NOT NULL DEFAULT FALSE,
  FOREIGN KEY(id_cidade) REFERENCES cidades(id)
);

CREATE TABLE vendas(
  id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  id_terreno INT(6) UNSIGNED,
  id_vendedor INT(6) UNSIGNED,
  vendido_em DATETIME NOT NULL,
  valor_total DECIMAL(24,2) NOT NULL,
  FOREIGN KEY(id_terreno) REFERENCES terrenos(id),
  FOREIGN KEY(id_vendedor) REFERENCES vendedores(id)
);

INSERT INTO vendedores(nome) VALUES ("Fulano");
INSERT INTO vendedores(nome) VALUES ("Ciclano");
INSERT INTO vendedores(nome) VALUES ("Beltrano");

INSERT INTO cidades (nome, UF, custo_metro_quadrado)
VALUES ("Porto Alegre", "RS", 6416.49);

INSERT INTO terrenos (id_cidade, largura, comprimento, vendido)
VALUES (1, 12, 4, false);
```

# Programação em SQL



Como você verá a seguir, a programação em SQL utiliza a sintaxe da SQL para a construção de instruções complexas que envolvem lógica de programação.

Normalmente, para pesquisar sobre essa sintaxe, deve-se pesquisar pelo SGBD específico que está sendo utilizado. Neste caso, será o MySQL. Por exemplo, se você quiser saber como montar um laço de repetição, busque por “laço de repetição no mysql” (ou, para aumentar a abrangência de sua busca, utilize as palavras-chave em inglês). Isso é necessário porque, assim como há diferença na sintaxe das diferentes linguagens de programação existentes, por mais que sejam semanticamente semelhantes, existe diferença na sintaxe da SQL quando comparada aos diferentes SGBDs (MySQL, Oracle, SQL Server, PostgreSQL etc.).

Você notará, inclusive, que alguns SGBDs têm nomenclaturas específicas para a programação em SQL, separando-a das instruções de manipulação e definição de dados (**SELECT**, **CREATE** etc.). São os casos do SQL Server, SGBD da Microsoft, que chama a sua sintaxe de T-SQL, e também do Oracle Database, da Oracle, que denomina a sua sintaxe como PL-SQL. De qualquer modo, orientando as suas buscas pelo SGBD utilizado, você será direcionado para a melhor documentação (sendo que a semântica das instruções é semelhante, o que diminui a curva de aprendizado sobre as instruções).

Antes da abordagem sobre os *stored procedures*, as *stored functions* e os *triggers*, comentados na seção anterior, entenda um pouco mais sobre a sintaxe da programação com SQL.

## Bloco de código

Para compreender como construir incríveis laços de repetição, condicionais e outras instruções lógicas, você precisa saber onde deverá fazer isso e também entender o que é um bloco de código quando se fala de programação em SQL.

Primeiramente, relembre um pouco das instruções de manipulação e definição com SQL (como o **SELECT**, o **UPDATE**, o **ALTER** etc.). Quando se fala de manipulação e definição com SQL, fala-se de **instruções simples**: há **um único comando, com um único objetivo**.

Embora se faça uma grande instrução – imagine uma instrução enorme de **SELECT**, na qual são selecionadas diversas colunas, de diversas tabelas diferentes, juntando-as com o comando **JOIN**, aplicando limitação de retorno, cláusula **WHERE** para filtrar os dados e mais uma série de cláusulas –, ainda haverá um **único comando com um único objetivo**: uma consulta por dados. Essa é a primeira grande diferença que você precisa entender a respeito de programação em SQL.

Na programação em SQL, utilizam-se **instruções compostas**. Por meio da definição de um **bloco de código** (isto é, de um escopo, um contexto...), com início e fim definidos, trabalha-se com **múltiplos comandos**, os quais, juntos, buscam um objetivo. Entre o início e o fim do bloco de código, cria-se um pequeno mundo, no qual existirá a execução das instruções que serão desenvolvidas. Nesse bloco de código, existirão as variáveis locais declaradas e serão executadas as instruções escritas, mas, **ao final do bloco de código, só existirá o resultado da operação** que foi montado.

Vejamos, a seguir, a sintaxe de um bloco de código escrito para o MySQL.

```
CREATE PROCEDURE minha_procedure()  
BEGIN  
    DECLARE id_porto_alegre INT UNSIGNED DEFAULT 12;  
    SELECT id_porto_alegre;  
END;
```

Concentre-se apenas nas instruções destacadas em azul nesse código. Os comandos **BEGIN** e **END** (que traduzidos do inglês significam “começar” e “terminar”, respectivamente) delimitam o início e o fim de um bloco de código. Entre

os comandos **BEGIN** e **END**, são declaradas as variáveis locais e escritas as instruções.

Como visto anteriormente, é comum haver instruções compostas na programação em SQL. Isso quer dizer que, normalmente, não se tem uma única instrução, mas sim várias, sendo executadas sequencialmente. Para delimitar quando termina uma instrução e começa a próxima, utiliza-se um **delimitador**. Por isso, conheça melhor o comando **DELIMITER**.

O comando **DELIMITER** define qual caractere é responsável por delimitar o fim de uma instrução, permitindo que se comece a próxima. Por padrão, o MySQL entende o ponto e vírgula [;] como delimitador, e, na maioria dos cenários, isso não será um problema. Entretanto, é comum que servidores de aplicações utilizem apenas a linha de comando para interação, e nesses casos pode-se enfrentar alguns problemas devido à utilização de ponto e vírgula como delimitador.

Analise o código a seguir:

```
CREATE PROCEDURE repetir(p1 INT)
BEGIN
  SET @x = 0;
  REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
END;
```

Novamente, atente-se apenas aos elementos destacados em azul no código. Como se pode ver, o ponto e vírgula está em quatro locais diferentes, representando dois **comportamentos** diferentes. As três primeiras utilizações de ponto e vírgula são relativas ao bloco de código que está sendo escrito. Note que elas se encontram entre os comandos **BEGIN** e **END**, apresentados na seção anterior. Entretanto, a quarta e última utilização de ponto e vírgula é relativa ao fim da instrução. Neste caso, está sendo utilizada a instrução **CREATE PROCEDURE** para criar um *stored procedure* (o que será mais estudo em breve).

Tanto os desenvolvedores quanto o ambiente de desenvolvimento integrado (IDE – *integrated development environment*) utilizado conseguem interpretar essa diferença no comportamento esperado de um ponto e vírgula, pois ambos enxergam o todo (e não comando por comando). Entretanto, a linha de comando não interpretará desta forma. Uma vez que o ponto e vírgula é o delimitador padrão do MySQL, o primeiro ponto e vírgula será interpretado como o fim da instrução e será executado de maneira incompleta, resultando em um erro.

Para resolver isso, pode-se utilizar o comando **DELIMITER** e determinar um delimitador diferente. Observe o código a seguir:

```
DELIMITER //
```

```
CREATE PROCEDURE repetir(p1 INT)
BEGIN
    SET @x = 0;
    REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
END
//
```

```
DELIMITER ;
```

```
CALL repetir(1000);
```

```
SELECT @x;
```

Nesse código existem cinco instruções diferentes.

A primeira instrução **DELIMITER** é responsável por determinar duas barras (**//**) como o novo delimitador. Em seguida, na segunda instrução, está sendo criado o *procedure* “repetir”, assim como foi criada no código do exemplo anterior, porém, dessa vez, ao final da instrução (logo após o comando **END**), utiliza-se o novo delimitador, as duas barras (ao invés do ponto e vírgula).

Isso acontece porque o comando **DELIMITER** determinou, para o MySQL, que o novo delimitador não é mais o ponto e vírgula, mas sim as duas barras. Dessa forma, consegue-se escrever o comando **CREATE PROCEDURE** utilizando ponto e vírgula dentro do bloco de código (entre os comandos **BEGIN** e **END**) normalmente e, após o fim da instrução (após o comando **END**), marca-se a sua conclusão utilizando o novo delimitador: as duas barras.

Continuando a leitura do código de exemplo, a terceira instrução é um novo **DELIMITER**, voltando o delimitador para o seu valor padrão (o ponto e vírgula)

A quarta instrução é o comando **CALL** executando o *stored procedure* “repetir” (você aprenderá mais sobre isso em breve). Por último, executa-se a quinta instrução, o comando **SELECT**. Note que, após a terceira instrução (o segundo **DELIMITER** dentro do exemplo, no qual foi determinado o ponto e vírgula como o delimitador), todas as próximas instruções voltam a utilizar o ponto e vírgula como delimitador. Novamente, é mais uma apresentação do objetivo do comando **DELIMITER**.

Como se pôde ver, a utilização do comando **DELIMITER** é bem simples. Basta escrever o comando **DELIMITER** e, em seguida, digitar o valor que será o novo delimitador. Confira o exemplo:

```
DELIMITER //
```

Normalmente, utilizam-se as duas barras como delimitador alternativo. É um valor amplamente utilizado pela comunidade por não impactar em nenhuma palavra-chave do SQL.

## Variáveis locais

Na programação, é comum a utilização de variáveis para auxiliar na simplificação do código e, para a programação em SQL, isso não é diferente. Por meio das variáveis locais, pode-se armazenar valores durante a execução de um



bloco de código, reutilizando-os entre as instruções.



Uma variável local pode ser manipulada e acessada **dentro de um bloco de código**. Como visto anteriormente, um bloco de código existe entre os comandos **BEGIN** e **END**. É importante entender esse conceito de blocos de código, pois não se consegue acessar variáveis locais entre diferentes contextos (ou seja, entre diferentes blocos de código).

Conheça, antes de colocar esses conceitos em prática, alguns dos tipos de dados existentes, os quais poderão ser (e serão) utilizados nas variáveis trabalhadas neste material.

Existem muitos tipos de dados, não só no MySQL, mas em outros SGBDs também. A sintaxe e a disponibilidade de alguns desses tipos de dados podem variar conforme o SGBD utilizado. Idealmente, deve-se sempre recorrer às documentações próprias do SGBD que estiver em uso. Nesta unidade, os estudos estão sendo orientados com o MySQL.

## Tipos de dados

### VARCHAR

O tipo de dados **VARCHAR** é um texto com um limite máximo de caracteres predefinido. A quantidade de caracteres de um dado **VARCHAR** pode estar entre 0 e 65.535 caracteres. É um dos tipos de dados mais utilizados pela sua flexibilidade para armazenar informações, trabalhando com letras, números, pontuações, palavras, frases etc. Quando se trabalha com **VARCHAR**, deve-se sempre estar atento ao *charset* do banco de dados, ou seja, a codificação de caracteres escolhida (uma vez que o *charset* limita as opções de caracteres especiais).

Em um futuro próximo, estão sendo criados *softwares* que se comunicam com os bancos de dados. Esteja atento ao *charset* definido tanto no banco de dados quanto na aplicação. As duas pontas dessa comunicação devem “falar a mesma língua” (no sentido de codificação de caracteres), sendo este um problema muito comum de se encontrar, principalmente nas primeiras experiências de um profissional.

Existem tipos de dados específicos para grandes quantidades de caracteres (como, por exemplo, o conteúdo de uma publicação em um *blog*). Nesses casos, evite o uso do **VARCHAR** e procure por outras opções, como *text*.

## INT

Abreviação de *integer*, que, em português, significa “inteiro”, é um tipo numeral e refere-se a números sem casas decimais. Também é um tipo de dado muito utilizado, podendo ser encontrado em casos como, por exemplo, laços de repetição ou identificadores únicos.

## DECIMAL

Tipo numérico que, diferentemente do INT, permite o uso de números com precisão, “com vírgula”. Para isso, é preciso informar o número total de dígitos e a quantidade de decimais. Por exemplo, poderia ser criada uma variável “salario” com o tipo de dado **DECIMAL(5, 2)**, que comporte até cinco dígitos, sendo dois desses decimais. Dessa forma, poderiam ser armazenados valores entre -999,99 e 999,99.

## DATETIME

Traduzido em “Data hora”, do inglês, é responsável por armazenar exatamente essas informações. O MySQL armazena a data e hora no seguinte formato: “ANO-MÊS-DIA HORA:MINUTO:SEGUNDO”. Por exemplo, a data/hora “14/04/2022 18:33:30” seria armazenada como “1998-04-14 18:33:33”. Esse formato é popularmente representado como “YYYY-MM-DD HH:MM:SS”.

## Declarando variáveis

Agora que já foram abordados alguns dos tipos de dados mais comuns e amplamente utilizados, aprenda a declarar novas variáveis dentro de um bloco de código. Observe a seguinte instrução:

```
DECLARE salario DECIMAL(5, 2);
```

O comando **DECLARE** é responsável por declarar novas variáveis. Nessa instrução, está sendo declarada uma nova variável chamada “salario”, do tipo **DECIMAL(5, 2)**. Como visto anteriormente, com essa declaração, a variável consegue armazenar valores entre -999,99 e 999,99. Além disso, como já estudado, deve-se adicionar um ponto e vírgula para delimitar o fim da instrução.

Anteriormente, verificou-se que o comando **DELIMITER** permite alterar o caractere responsável por delimitar o fim de uma instrução. Entretanto, o código de exemplo visto agora (que apresentou o comando **DECLARE**) existe apenas **dentro de um bloco de código** (entre os comandos **BEGIN** e **END**). Dessa forma, mesmo alterando o delimitador de instruções (por meio do comando **DELIMITER**), continua-se utilizando o ponto e vírgula, pois **o bloco de código é uma parte interna de uma única instrução** (como, por exemplo, uma instrução para a criação de um *stored procedure*).

No exemplo anterior, a variável "salario" foi declarada e seu valor é nulo (também conhecido como **NULL**). Isso acontece porque declara-se/inicializa-se a variável sem a definição de um valor padrão. Para inicializar a variável com um valor (ou seja, "não nula"), deve-se fazer como no seguinte exemplo:

```
DECLARE salario DECIMAL(5, 2) DEFAULT 150.99;
```

Observe que foi adicionado o comando **DEFAULT** à instrução. Logo após o comando **DEFAULT** (que pode ser traduzido para "padrão"), informou-se qual é o valor padrão para a variável declarada. No exemplo anterior, a variável "salario" está sendo declarada como o tipo **DECIMAL** e o valor padrão (ou, também, "inicial") igual a 150,99.

Lembre-se de que, no MySQL, como padrão, não se separam os decimais com uma vírgula, mas sim com um ponto.

Confira agora um código de exemplo completo, contendo a declaração de diferentes variáveis, com diferentes tipos de dados, utilizando o comando **DECLARE** junto a um bloco de código. Neste exemplo, o bloco de código será referente à criação de um *stored procedure*. Não se preocupe em entender a criação do *stored procedure* por enquanto, concentre-se apenas na declaração de variáveis em um exemplo completo.

```
DELIMITER $$
CREATE PROCEDURE buscar_resultado_prova ()
BEGIN
    DECLARE aluno VARCHAR(25) DEFAULT "Fulano";
    DECLARE id_prova INT UNSIGNED DEFAULT 12;
    DECLARE nota DECIMAL(3, 1) DEFAULT 9.5;
    DECLARE finalizada_em DATETIME DEFAULT "2022-01-14 10:32:35";
    SELECT aluno, id_prova, nota, finalizada_em;
END $$
DELIMITER ;
```

Como se observa nesse exemplo, alterou-se o delimitador (para evitar o conflito com o ponto e vírgula), iniciou-se a instrução para a criação de um novo *procedure* chamado "buscar\_resultado\_prova" e declarou-se o início do bloco de código (por meio do comando **BEGIN**) e as quatro variáveis diferentes: "aluno" do tipo **VARCHAR**, "id\_prova" tipo **INT**, "nota" um **DECIMAL** e "finalizada\_em" do tipo **DATETIME**. As quatro variáveis foram inicializadas com um valor padrão. Ainda dentro do bloco de código, adicionou-se uma instrução de **SELECT**, selecionando as quatro variáveis, e finalizou-se o bloco de código, por meio do comando **END**. Por último, o delimitador foi retornado ao seu valor padrão.

## Atribuindo novos valores a uma variável

Até agora, você aprendeu a declarar novas variáveis definindo valores padrões. Porém, o próprio nome já indica que são "variáveis". O objetivo, portanto, é conseguir um elemento cujo valor varie conforme a necessidade. Para isso, aprenda a atribuir novos valores a uma variável declarada. Confira o código a seguir:

```
DECLARE aluno VARCHAR(25) DEFAULT "Fulano";
SET aluno = "Ciclano";
```

Você já conhece a primeira instrução desse código. Está sendo declarada uma nova variável chamada "aluno", do tipo **VARCHAR**, com tamanho máximo de 25 caracteres, definindo "Fulano" como o valor padrão. A segunda instrução do exemplo está utilizando o comando **SET**.

**SET**, traduzido do inglês, significa “definir”, e é justamente essa a função do comando: ele **define** um novo valor para uma variável.

Até o momento, conhece-se apenas as variáveis locais. Porém, o comando **SET** também pode atribuir novos valores para parâmetros e variáveis de usuário, conceitos que você estudará em breve.

A sintaxe da instrução **SET** no MySQL é bem simples: informam-se o comando **SET**, o nome da variável cujo novo valor será definido, o sinal de igual e o novo valor que será atribuído. Também pode-se utilizar o sinal de atribuição (":=") em vez do sinal de igual.

É importante e necessário que você esteja atento ao tipo de dado da variável, pois não se pode atribuir um dado **VARCHAR**, como um nome, a uma variável que foi declarada como **INT**. O contrário também é verdade. Deve-se sempre respeitar o tipo de dado da variável.

Pode-se definir o valor de múltiplas variáveis em uma só instrução. Confira o exemplo:

```
DECLARE aluno VARCHAR(25) DEFAULT "Fulano";  
DECLARE nota DECIMAL(3,1) DEFAULT 10.0;  
SET aluno = "Ciclano", nota = 9.5;
```

No exemplo, por meio da inclusão de uma vírgula ao final da instrução, foi possível adicionar outra atribuição de valor a uma variável ao lado.

O comando **SET** também permite o trabalho com operações matemáticas, com valores literais ou a utilização de outras variáveis na instrução. Confira estes exemplos:

```
DECLARE contador INT UNSIGNED DEFAULT 0;  
SET contador = contador + 1;  
  
DECLARE ano_nascimento INT UNSIGNED DEFAULT 1987;  
DECLARE ano_atual INT UNSIGNED DEFAULT 2022;  
DECLARE idade INT UNSIGNED DEFAULT 0;  
SET idade = ano_atual - ano_nascimento;
```


Na primeira parte do código, declarou-se uma variável chamada "contador", com o tipo de dado **INT** e o valor padrão igual a 0. Então, utilizou-se o comando **SET** para definir que o novo valor da variável "contador" é o seu valor atual mais 1. Seguindo a execução do código, o valor atual da variável "contador" era 0, adiciona-se 1, o resultado final (sendo esse o valor final da variável neste exemplo) é igual a 1.

Na segunda parte do código, foram declaradas três variáveis. A primeira é chamada de "ano\_nascimento", com o tipo de dado **INT** e o valor padrão igual a 1987. A segunda é chamada de "ano\_atual", com o tipo de dado **INT** e o valor padrão igual a 2022. A terceira é chamada de "idade", com o tipo de dado **INT** e o valor padrão igual a 0. Então, utiliza-se o comando **SET** para atribuir um novo valor à variável "idade", realizando uma subtração entre as variáveis "ano\_atual" e "ano\_nascimento". Seguindo a execução do código, a variável "ano\_atual" tinha o valor 2022, menos o valor da variável "ano\_nascimento", cujo valor era 1987.

Completando o cálculo, a variável "idade" encerra a execução do código citado com o valor de 35.

Além de atribuir cálculos matemáticos a variáveis, também se pode atribuir resultados de uma consulta SQL. Observe o seguinte código.

```
DELIMITER //
```



```
CREATE PROCEDURE buscar_nome_vendedor ()  
BEGIN  
    DECLARE variavel_nome VARCHAR(25);  
    SELECT nome INTO variavel_nome FROM vendedores WHERE id = 1;  
    SELECT variavel_nome;  
END//  
  
DELIMITER ;
```

Conforme se pode ver na instrução destacada, utilizou-se o comando **INTO** junto ao comando **SELECT**. Com o comando **INTO**, é possível direcionar o valor de uma das colunas informadas no comando **SELECT** para uma variável local declarada. Nesse exemplo, declara-se a variável "variavel\_nome", do tipo **VARCHAR(25)**, direcionando o resultado da coluna "nome" para dentro da variável.

Em instruções semelhantes ao exemplo anterior, não nomeie as suas variáveis com o mesmo nome das colunas. Quando você tiver uma coluna e uma variável com o mesmo nome em uma instrução, o MySQL enxergará apenas a variável e você não conseguirá acessar o valor da coluna.

Além do comando **INTO**, também se admite a utilização do comando **SET**, apresentado anteriormente, para armazenar a coluna de uma consulta SQL em uma variável. Verifique o código a seguir:



```
DELIMITER //
```

```
CREATE PROCEDURE buscar_nome_vendedor ()
```

```
BEGIN
```

```
    DECLARE variavel_nome VARCHAR(25);
```

```
    SET variavel_nome := (SELECT nome FROM vendedores WHERE id = 1);
```

```
    SELECT variavel_nome;
```

```
END//
```

```
DELIMITER ;
```

Nesse exemplo, tem-se um código com o mesmo resultado que o exemplo anterior. Porém, em vez de utilizar o comando **INTO** para atribuir o valor da coluna "nome" à variável "variavel\_nome", optou-se por utilizar o comando **SET**, apresentado anteriormente. A sintaxe é muito semelhante à utilização básica do comando **SET**. Primeiramente, declara-se o comando e o nome da variável que receberá o valor, então coloca-se o sinal de igual (ou o sinal de atribuição, como neste exemplo) e, por último, a consulta SQL na qual **será retornada uma única coluna** (cujo valor será atribuído à variável). Como se trabalha neste momento com uma consulta SQL no lado direito da instrução, deve-se adicionar parênteses entre a instrução da consulta, criando um escopo exclusivo.

É importante perceber que, até o momento, estão sendo atribuídos apenas valores singulares, ou seja, uma coluna por vez. Tanto no exemplo do comando **INTO** quanto no exemplo do comando **SET**, foi utilizada a coluna "nome" em uma consulta que retornou apenas um resultado (isso porque a consulta foi realizada com a condição **WHERE** relacionada a uma chave primária – logo, um valor único).

Mas o que se deve fazer para armazenar diversos valores retornados em uma consulta SQL?

## Armazenando múltiplas colunas – Comando INTO

Observe o seguinte código:

```
DELIMITER //
CREATE PROCEDURE buscar_dados_vendedor ()
BEGIN
    DECLARE v_nome VARCHAR(25);
    DECLARE v_email VARCHAR(255);
    SELECT nome, email INTO v_nome, v_email
        FROM vendedores WHERE id = 1;
    SELECT v_nome, v_email;
END//
DELIMITER ;
```

Perceba que neste exemplo, trabalha-se com duas variáveis diferentes: "v\_nome" e "v\_email". As duas contêm o mesmo tipo que as colunas "nome" e "email" da tabela "vendedores". Na instrução **SELECT**, destacada, está sendo utilizado novamente o comando **INTO**, o qual, conforme visto anteriormente, permite direcionar o valor de uma coluna a uma variável: neste caso, não é diferente.

Está sendo utilizado o comando **INTO** para direcionar o valor das colunas "nome" e "email", selecionadas na instrução **SELECT** nas variáveis "v\_nome" e "v\_email". Perceba como é importante manter as variáveis na mesma ordem que as colunas.

A sintaxe do comando **INTO** permanece igual ao que foi visto antes. À esquerda, constam a lista de colunas selecionadas na instrução **SELECT** e, à direita, a lista de variáveis que receberão os valores das colunas selecionadas, na respectiva ordem. É importante ressaltar que as variáveis já devem ter sido declaradas antes de utilizadas.

## Variáveis de usuário

Além das variáveis locais aprendidas, existe um outro tipo de variável que se pode utilizar na programação em SQL: são chamadas de "variáveis de usuário".

Lembre-se de que as variáveis locais existem apenas dentro de um bloco de código (ou seja, entre os comandos **BEGIN** e **END**). Sendo assim, elas são declaradas dentro de um escopo e, após o fim deste bloco de código (ou seja, desse escopo), as variáveis deixam de existir.

Diferentemente das variáveis locais, as **variáveis de usuário** existem também fora de um bloco de código. Isso acontece porque **variáveis de usuário existem dentro de uma sessão de usuário**.

Sendo assim, pode-se declarar uma variável de usuário em uma instrução e utilizá-la em outra, pois variáveis de usuário não precisam de um bloco de código e podem ser utilizadas em diferentes instruções, compartilhando valores entre elas.

Além disso, as variáveis de usuário não têm uma tipagem forte, ou seja, elas não têm tipos de dados definidos na sua declaração. Isso acontece porque **variáveis de usuário não são declaradas, são apenas inicializadas**.

O que isso quer dizer?

Quer dizer que, ao criar uma variável de usuário, já se deve definir o valor dessa variável e, dessa forma, o MySQL conhecerá e definirá o seu tipo (baseado no valor definido), sem a necessidade de se especificar o tipo de dado da variável de usuário.

Mesmo com essa característica – de não ser preciso declarar explicitamente o tipo de uma variável de usuário –, as variáveis de usuário têm uma quantidade limitada de tipos de dados aceitos. Tentar atribuir valores de tipos de dados não aceitos a uma variável de usuário resultará em uma **conversão**, isto é, o próprio MySQL será responsável por converter o tipo de dado de origem para o tipo de dado aceito mais próximo. Por exemplo, se você tentar atribuir um valor de data a uma variável de usuário, o MySQL converterá o valor para um **VARCHAR** (uma *string*).

Que tal conhecer um pouco mais sobre os tipos de dados que são aceitos em uma variável de usuário?

## Tipos de dados



A seguir, veja a lista com os tipos de dados aceitos em variáveis de usuário:

- ◆ **INT**
- ◆ **DECIMAL**
- ◆ **VARCHAR**
- ◆ **FLOAT**: o tipo de dado **FLOAT** também trabalha com decimais, assim como o tipo de dado **DECIMAL**, entretanto tem precisão simples, limitando-se à norma técnica do Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE) 754 e variando conforme sistema operacional e *hardware*. Assim como no caso do tipo de dados **DECIMAL**, é preciso informar a precisão, que é o número total de dígitos, e a escala, que é quantidade de decimais. Utilizando o mesmo exemplo, pode-se criar uma variável “salario” com o tipo de dado **FLOAT(5, 2)**.

Os tipos de dados **INT**, **DECIMAL** e **VARCHAR** já foram apresentados na seção “Tipos de dados” do capítulo sobre as variáveis locais.

## Declarando variáveis de usuário

Conforme explicado, não é necessário um escopo (ou seja, de um bloco de código) para declarar variáveis de usuário. Dessa forma, a sintaxe para declarar variáveis de usuários é ligeiramente diferente do modo de declarar variáveis locais.

Considere se conectar ao seu banco de dados, inicializar uma variável de usuário, continuar utilizando o seu MySQL normalmente, realizando outras instruções e operações e, posteriormente (desde que dentro da mesma sessão de usuário, como explicado anteriormente), acessar a mesma variável de usuário declarada lá no início da sua conexão. O MySQL manterá salvo o valor dessa variável para você!

Observe este código:

```
SET @ola = "Olá mundo";
```

A primeira coisa que você deve notar é que, diferentemente das variáveis locais, utilizou-se um **@** antes do nome da variável de usuário. A partir do momento em que for executada essa instrução (do exemplo anterior), estará definido que a variável "**@ola**" possui o valor "Olá mundo". Além disso, por conta do valor atribuído, o MySQL entenderá que essa variável tem o tipo de dado **VARCHAR**.

Com isto, já se pode utilizar a variável de usuário em outras instruções (desde que a sessão/conexão não seja encerrada). Veja este código.

```
SELECT @ola;
```

No exemplo, está sendo realizada uma instrução básica de consulta, utilizando o comando **SELECT**. Perceba que se selecionou a variável "**@ola**" em uma outra instrução, diferentemente da instrução de declaração. Mesmo sendo uma instrução diferente, é viável utilizar a variável declarada, e o resultado dessa instrução será:

	@ola
1	Olá mundo

Agora, execute o código a seguir:

```
SELECT @variavelInexistente;
```

O resultado dessa operação será:



	@variavelInexistente
1	[NULL]

Isso acontece porque variáveis de usuário **não precisam ser declaradas**. Isso mesmo! A utilização de uma variável de usuário não declarada apresentará o valor nulo, valor padrão de uma variável de usuário.

## Atribuindo novos valores a variáveis de usuário

Para atribuir novos valores a variáveis de usuário já existentes, utiliza-se o mesmo comando **SET**, da mesma maneira que para declarar uma variável.

Veja o exemplo a seguir:

```
SET @ola = "Hello world";
```

Perceba que está sendo alterado o valor da variável “@ola”, criada na seção anterior. Agora, execute a seguinte instrução:

```
SELECT @ola;
```

O resultado será:



	@ola
1	Hello world

Conforme se pode notar, a variável “@ola” agora tem um novo valor. A mesma instrução para declarar novas variáveis de usuário é utilizada para alterar o seu valor.

Assim como foi feito com as variáveis locais, também se pode armazenar resultados de consultas em variáveis de usuário. Inclusive, você verá que as sintaxes são iguais! Observe:

```
SELECT id, nome INTO @v_id, @v_nome  
FROM vendedores  
WHERE id = 1;
```

Está sendo executada uma instrução de consulta nas colunas "id" e "nome" da tabela "vendedores". Por meio do comando **INTO**, armazenam-se os seus valores nas variáveis "@v\_id" e "@v\_nome", respectivamente. Perceba que a consulta contém uma cláusula **WHERE** em uma coluna de chave primária, ou seja, será retornado um único resultado.

Como você pôde ver, a sintaxe para armazenar resultados de consultas em variáveis de usuário por meio do comando **INTO** é a mesma de quando se utilizam variáveis locais. Agora, armazene o valor de uma consulta por meio do comando **SET**, como no exemplo a seguir:

```
SET @v_nome := (SELECT nome FROM vendedores WHERE id = 1);
```

Assim como no exemplo anterior, está sendo realizada uma instrução de consulta com o comando **SELECT**, selecionando a coluna "nome" da tabela "vendedores". Além disso, mantém-se a cláusula **WHERE** em uma coluna de chave primária, limitando o retorno a um único resultado.

Para armazenar o valor da sua consulta, utilizou-se o comando **SET**. Note que foi utilizada a mesma sintaxe da instrução quando com variáveis locais.

Além disso, é possível utilizar tanto variáveis de usuário que já foram definidas, alterando o seu valor atual, quanto variáveis de usuário novas, criando-as com o valor igual ao retorno da consulta.

Perceba que, assim como as variáveis locais, as variáveis de usuário são um conceito extremamente útil para a programação com SQL. Por meio das variáveis, é possível armazenar dados, permitindo o compartilhamento destes entre diferentes instruções. Além disso, o seu correto uso pode diminuir a complexidade do seu código, melhorando a sua legibilidade.

Diferentemente das variáveis locais, as variáveis de usuário não precisam de um bloco de código, existindo durante toda a sessão do usuário a partir do momento de sua declaração. Variáveis de usuário também podem ser utilizadas sem terem sido declaradas, retornando nulo.

Antes de estudar a próxima seção, ressalta-se que o melhor amigo do programador é a mensagem de erro. É importante aprender a ler as mensagens das ferramentas e dos compiladores quando se comete um erro. Por meio delas, consegue-se consultar em buscadores por soluções e entender onde está o erro. Portanto, não tenha medo das mensagens de erro!



# Condicional

Agora que você já conhece as variáveis, consegue flexibilizar as instruções, tornando-as muito mais dinâmicas. Para aumentar ainda mais as suas possibilidades com o SQL, observe alguns conceitos de lógica de programação aplicados ao SQL.

Os **comandos condicionais** serão o primeiro conceito. Eles são comumente utilizados no dia a dia, sendo um conceito para, por meio de uma ou mais condições, especificar comportamentos esperados para a sua instrução.

Existem duas formas principais de se utilizar uma condicional no SQL. A primeira que será estudada é conhecida como **IF statement**, na qual a condicional existe dentro de um bloco de código. Essa forma de condicional será muito utilizada dentro de *stored procedures*, *stored functions* e *triggers*. A segunda forma que será estudada é conhecida como **IF function**, na qual a condicional apenas declara dois possíveis retornos (de acordo com a condição provida). Confira mais sobre ambas as formas agora.

## IF statement

A **IF statement** tem a mesma semântica da condicional apresentada no Portugol. Confira:

```
DELIMITER //  
CREATE PROCEDURE minha_procedure ()  
BEGIN  
    DECLARE numero INT DEFAULT 1;  
    IF numero = 1 THEN  
        SET numero := 2;  
    END IF;  
    SELECT numero;  
END//  
DELIMITER ;
```

O código destacado começa com o comando **IF** (cuja tradução do inglês é "se") seguido da condição que deverá ser verificada. Nesse caso, verifica-se se a variável "numero" é igual ao valor "1". Após adicionar a condicional, utiliza-se o comando **THEN** (cuja tradução do inglês é "então"). A partir dessa linha, inicia-se um novo bloco de código. Para fechar o bloco de código, utiliza-se o comando **END IF** (em uma tradução literal do inglês, "fim se", marcando o fim do comando **IF**).

Todas as instruções escritas dentro desse bloco de código só serão executadas caso a condição definida seja verdadeira. Neste exemplo, a condição é a variável "numero" ser igual ao valor "1". Se isso for verdade, entra-se no bloco de código, executa-se o que estiver dentro dele e segue-se para a próxima instrução normalmente (logo após o comando **END IF**).

O que acontece se a condição não for verdade? No exemplo visto, o bloco de código simplesmente não será executado, pulando para a próxima instrução após a condicional. Tudo o que estiver dentro do bloco de código será ignorado.

Como proceder caso queira adicionar um comportamento **específico** para quando a sua instrução não for verdade? Veja o código a seguir:

```
DELIMITER $$
CREATE PROCEDURE minha_procedure ()
BEGIN
    DECLARE numero INT DEFAULT 2;
    IF numero = 1 THEN
        SET numero := 10;
    ELSE
        SET numero := 20;
    END IF;
    SELECT numero;
END$$
DELIMITER ;
```

Perceba que foi adicionado um novo comando no meio do bloco de código. O comando **ELSE** (traduzido do inglês significa "senão") permite adicionar **dois blocos de código** à condicional. O primeiro, entre os comandos **THEN** e **ELSE**, será executado caso a condição seja verdadeira. O segundo, entre os comandos **ELSE** e **END IF**, será executado caso a condição **não seja verdadeira**.

Comparando esse exemplo ao anterior, foi alterado o valor padrão da variável "numero". Agora, a variável contém o valor 2. Entretanto, a condição continua a mesma (se a variável "numero" for igual ao valor "1"). Como se pode perceber, a condição é **falsa**. Nesse caso, o segundo bloco de código será executado em vez do primeiro.

Além disso, assim como no exemplo anterior, o código continuará sua execução normalmente após a condicional, seguindo com as instruções após o comando **END IF**.

Confira um exemplo mais voltado ao dia a dia de um desenvolvedor. Observe o fluxograma a seguir:



Figura 1 – Fluxograma de exemplo

Nesse fluxograma, consta um exemplo básico de uma estrutura condicional. Por meio de uma condição (losango amarelo), tem-se dois caminhos possíveis (se a condição for verdadeira, flecha "sim"; se a condição for falsa, flecha "não"). Para cada caminho possível dessa condição há um bloco de código exclusivo (retângulos azul e vermelho). Isso é, o bloco de código azul só será executado caso a condição seja verdadeira no momento de sua execução. Já o bloco de código vermelho só será executado caso a condição seja falsa no momento de sua execução. Esse fluxograma representa fielmente uma condicional. Que tal transformá-lo em código SQL? Veja o código a seguir.

```
DELIMITER $$
CREATE PROCEDURE buscar_terrenos (id_vendedor INT)
BEGIN
    DECLARE id_encontrado INT;

    SELECT id INTO id_encontrado FROM vendedores
        WHERE id = id_vendedor;

    IF id_encontrado IS NULL THEN
        SELECT "Sem acesso";
    ELSE
        SELECT * FROM terrenos;
    END IF;
END $$
DELIMITER ;
```

Lendo o código dentro do bloco de código, primeiramente está sendo declarada uma variável sem valor padrão (conforme visto na seção anterior, isso significa que a variável contém valor nulo no momento de sua declaração).

Em seguida, há uma instrução de consulta buscando pela coluna "id" da tabela "vendedores". Nessa instrução de consulta consta uma cláusula **WHERE**, na qual estão filtrados todos os resultados em que a coluna "id" (da tabela "vendedores") for igual à variável "id\_vendedor".

A variável "id\_vendedor" está sendo recebida por parâmetro no *stored procedure*, mas esse assunto será estudado em breve.

Por último, no bloco de código, constrói-se o comando condicional. Nesse código, ocorre o mesmo comportamento desenhado na figura anterior, no fluxograma. Lendo o código, **se** a variável "**id\_encontrado**" for **igual** a **NULL** (nulo, no SQL), **então** será apresentada como resultado do *stored procedure* a informação "**Sem acesso**". **Senão**, será apresentada uma **consulta por todos os terrenos cadastrados**. Agora, leia apenas as palavras em negrito desse parágrafo. Parabéns! Você acaba de ler o comando condicional programado anteriormente.

Além disso, também pode haver múltiplos caminhos específicos em uma condicional. Veja o fluxograma a seguir:

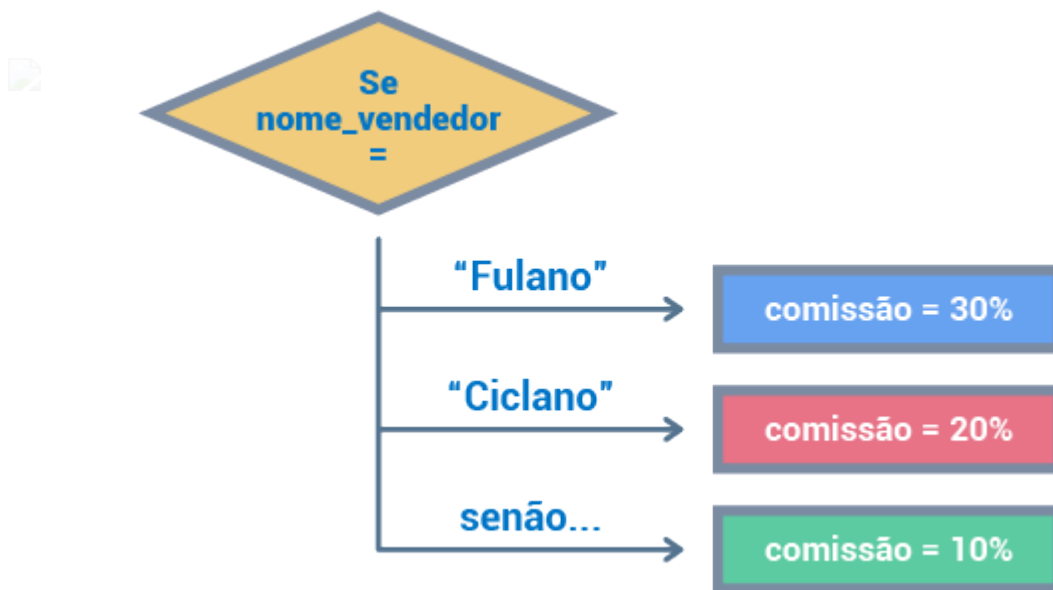


Figura 2 – Fluxograma de exemplo

Fluxograma apresentando um ponto de início condicional, questionando "Se variável nome vendedor for igual a". A partir do ponto de início condicional, existem três flechas diferentes conectadas a três blocos de ação diferentes. A primeira flecha, com o texto "Fulano", conecta-se a um bloco de ação escrito "comissão igual a trinta por cento". A segunda flecha, com o texto "Ciclano", conecta-se a um bloco de ação escrito "comissão igual a vinte por cento". A terceira e última flecha, com o texto "senão", conecta-se a um bloco de ação escrito "comissão igual a dez por cento".

Perceba que nesse fluxograma estão caminhos específicos. Anteriormente, foram vistos cenários nos quais a condição era verdadeira ou falsa, tendo, no máximo, dois blocos de código. Nesse fluxograma, estão duas verificações específicas (se a variável "nome\_vendedor" for igual a "Fulano" ou se a variável "nome\_vendedor" for igual a "Ciclano") e, além disso, um caminho "senão", para o caso em que nenhuma das outras condições seja verdadeira.

É possível ter quantos caminhos específicos forem necessários. Porém, é sempre bom considerar a qualidade do código, para que ele não fique muito complexo, facilitando as manutenções ao longo do tempo.

Agora, fluxograma anterior será transformado em código.

```
DELIMITER $$
CREATE PROCEDURE buscar_comissao (id_vendedor INT)
BEGIN
    DECLARE comissao DECIMAL(2,1) DEFAULT 1.0;
    DECLARE nome_vendedor VARCHAR(25);
    SELECT nome INTO nome_vendedor
        FROM vendedores
        WHERE id = id_vendedor;

    IF nome_vendedor = "Fulano" THEN
        SET comissao := 1.3;
    ELSEIF nome_vendedor = "Ciclano" THEN
        SET comissao := 1.2;
    ELSE
        SET comissao := 1.1;
    END IF;
    SELECT comissao;
END$$
DELIMITER ;
```

Observando o código destacado, nota-se o comando **ELSEIF** (junção das palavras *e/se* e *if*, traduzidas do inglês para "senão" e "se", respectivamente). Conforme a tradução, esse comando é uma forma de declarar caminhos com outras condições. No código de exemplo, utilizou-se o comando **ELSEIF** e foi definida uma condição diferente da condição declarada no comando **IF**. Neste caso, definiu-se como condição se a variável "nome\_vendedor" é igual ao valor "Ciclano". Após declarar a condição, utilizou-se o comando **THEN** para marcar o início do bloco de código referente a essa condição.

Pode-se fechar este bloco de código declarando um novo **ELSEIF**, criando uma cadeia de condições, declarando um **ELSE**, definindo um bloco de código para o caso de nenhuma das condições ser verdadeira, ou declarando um **END IF**, encerrando assim a condicional.

Existem outras instruções feitas para esse tipo de cenário (nas quais há diversos caminhos específicos). Utilizando o seu buscador, procure por "MySQL CASE" para conhecer o comando **CASE**, muito útil para cenários como este.

## IF function

Uma **IF function** (traduzida do inglês para "função se") tem o mesmo conceito da **IF statement**, estudada na seção anterior. A principal diferença da **IF function** é a sua utilização.

Enquanto a **IF statement** apenas existe dentro de blocos de código, uma **IF function** pode ser utilizada dentro de uma instrução simples. Confira o exemplo a seguir:

```
SELECT IF(10>1, "É maior", "É menor");
```

Perceba que foi escrita uma instrução de consulta simples, por meio do comando **SELECT**. Porém, onde seriam colocadas as colunas ou os valores a serem consultados, foi inserida uma **IF function**.

O comando **IF**, seguido de parênteses, representa a chamada da função **IF**, nativa do MySQL. A função **IF** recebe três parâmetros: o primeiro parâmetro é a condição; o segundo parâmetro é o retorno para o caso de a condição ser verdadeira; e o terceiro parâmetro é o retorno para o caso de a condição ser falsa.

Neste exemplo, declara-se a função **IF**, abrem-se parênteses e define-se, como a condição, "se 10 for maior que 1". Em seguida, adiciona-se uma vírgula e insere-se o segundo parâmetro, sendo este o retorno para o caso de a condição ser



verdadeira. Neste exemplo, o segundo parâmetro é o valor "É maior". Novamente, adiciona-se mais uma vírgula e define-se o terceiro parâmetro, sendo este o retorno para o caso de a condição ser falsa. Nesse parâmetro, passa-se o valor "É menor". Para fechar a função **IF**, fecha-se um parênteses.

O resultado do código de exemplo será o valor "É maior", uma vez que a condição é verdadeira. Caso a condição seja alterada para "1>10", então o retorno seria o valor "É menor".

Leia a documentação do MySQL sobre funções. Existem algumas variações da **IF function** que podem simplificar o seu código final.

## Laço de repetição

O laço de repetição é um conceito muito utilizado no dia a dia de um desenvolvedor, independentemente da linguagem de programação utilizada. Logo, isso não seria diferente para a programação em SQL.

O laço de repetição no MySQL tem uma semântica muito semelhante à do Portugol ou de linguagens de programação. O MySQL implementa as repetições pelos comandos **LOOP** e **WHILE**, sendo este último o mais comum e adequado.

## WHILE

O comando **WHILE** (traduzido do inglês para "enquanto") recebe uma condição e um bloco de código. Enquanto a condição provida for verdadeira, executa-se o bloco de código repetidas vezes. Lembre-se de que, por conta disso, é necessário garantir que alguma instrução dentro do bloco de código seja responsável por tornar a condição falsa. Por exemplo, veja o seguinte código:

```
DELIMITER $$
CREATE PROCEDURE inserindo_vendedores ()
BEGIN
    DECLARE v1 INT DEFAULT 5;
    WHILE v1 > 0 DO
        INSERT INTO vendedores (nome)
            VALUES (CONCAT("Vendedor ", v1));
        SET v1 = v1 - 1;
    END WHILE;
END$$
DELIMITER ;
```

Observe que, no código destacado, declarou-se o comando **WHILE** seguido da condição do laço de repetição. Nesse exemplo, estará sendo executado o laço de repetição enquanto a variável "v1" for maior que zero.

Após a condição do laço de repetição, declara-se o comando **DO** (traduzido do inglês para o verbo "fazer"), indicando o início do bloco de código. Para fechar o bloco de código, utiliza-se o comando **END WHILE**, que pode ser traduzido como "fim enquanto".

Perceba que foi destacado, também, o comando **SET**, que, como já se sabe, é responsável por atribuir um valor a uma variável. Isso foi feito porque, como explicado anteriormente, é preciso declarar, dentro do laço de repetição, a instrução que será responsável por tornar falsa a condição. Nesse exemplo, a condição é a variável "v1" ser maior do que zero. Então, deve-se, dentro do laço de repetição, diminuí-la de forma que, em algum momento, ela se torne igual a zero ou menor que zero (sendo assim falsa a condição).

## Stored procedures

Ao longo dos estudos, falou-se várias vezes sobre a criação de *stored procedures*. Na maioria dos exemplos, focou-se no bloco de código do *stored procedure*, entendendo condicionais, laços de repetição, variáveis locais etc. Agora, conheça melhor o que é um *stored procedure* e como utilizá-los.

*Stored procedures* (traduzido do inglês para "procedimentos armazenados") é um dos recursos disponíveis nos principais SGBDs, incluindo o MySQL. Por meio dos *stored procedures*, é possível declarar métodos responsáveis por executar uma série de instruções lógicas. É uma ferramenta muito útil para simplificar o seu código, separar responsabilidades e, em alguns casos, melhorar a *performance* do seu *software*.

O seu uso é simples. Cria-se um *stored procedure* informando o seu nome, os seus parâmetros e o seu bloco de código. Com isso, o banco de dados armazena o *stored procedure* (muito semelhante à criação de tabelas) e permite utilizá-lo o quanto for necessário. Um *stored procedure* pode tanto retornar valores quanto apenas executar uma série de instruções sem retorno. Além disso, também se pode alterar o modo com que o retorno é feito. Você estudará mais sobre isso em breve.

## Criando um *stored procedure*

Observe o código a seguir:

```
DELIMITER //  
CREATE PROCEDURE minha_procedure ()  
BEGIN  
    DECLARE variavel INT DEFAULT 15;  
    SELECT variavel;  
END//  
DELIMITER ;
```

De acordo com os comandos destacados, percebe-se que foi utilizado o comando **CREATE PROCEDURE** para iniciar a declaração de um novo *stored procedure*. Em seguida, declarou-se o nome do *procedure*. Após o nome, foram adicionados parênteses, entre os quais se colocam os parâmetros do *stored procedure* (o que será estudado em seguida). Após, foram utilizados os comandos

**BEGIN** e **END** para encapsular o bloco de código que será executado quando o *stored procedure* for chamado. Dentro do bloco de código, podem-se utilizar as instruções aprendidas até agora (condicionais, laços de repetição, consultas, atualizações etc.).

Para o nome de *stored procedures*, utiliza-se o mesmo conceito de nome de outros objetos, sendo permitido apenas caracteres ASCII (letras maiúsculas e minúsculas sem acentuação, números inteiros e *underline*).

Experimente criar o *stored procedure* anterior em um banco de dados já existente em sua máquina. Para isso, basta que o banco esteja selecionado e que você execute no Workbench o *script*. Note que, ao executar, na área **Navigator**, expandindo o item ***stored procedure*** do banco de dados, aparecerá o procedimento criado.

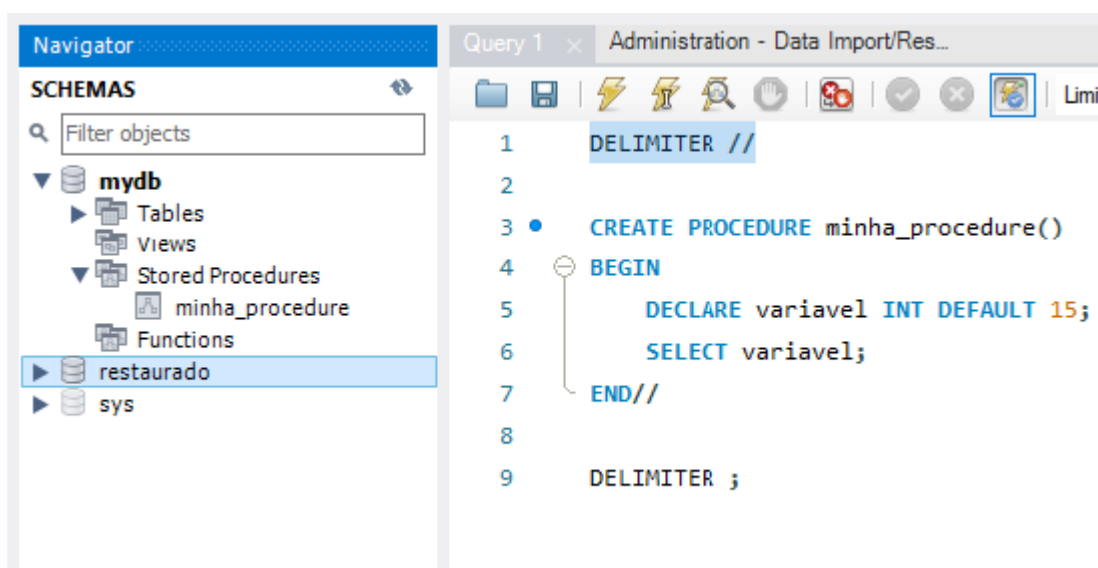


Figura 3 – Novo *procedure* para o banco “mydb” no MySQL Workbench

Tela principal do MySQL Workbench, na qual, à direita, está a área “Navigator” com o banco de dados “mydb”, e, abaixo dele, o item “Tables” retraído e o item “*stored procedures*” expandido, revelando, abaixo, “minha\_procedure”. À direita está o editor com o código mencionado no texto.

Outro fator importante é que os nomes dos *stored procedures* devem ser únicos. Dessa forma, não se pode executar a criação de um *stored procedure* duas vezes e, para atualizar um já existente, deve-se antes excluí-lo do banco de dados. Confira o seguinte comando:

```
DROP PROCEDURE minha_procedure;
```

Perceba o comando **DROP** entrou no lugar do comando **CREATE**. Como se está excluindo um *stored procedure*, não é necessário declarar os seus parâmetros e o seu bloco de código novamente, e sim apenas informar o nome do *stored procedure* que se deseja excluir.

## Passando parâmetros em um *stored procedure*

Quando se cria um *stored procedure*, definem-se quais serão os parâmetros passados ao chamar a *stored procedure*, podendo ser zero ou mais parâmetros. Cada parâmetro terá um **modo**, um **nome** e um **tipo de dado**. Observe estes importantes conceitos:

## Tipos de dados

Os tipos de dados disponíveis já são conhecidos, pois são os mesmos tipos de dados que podem ser utilizados nas variáveis e colunas (das tabelas que foram criadas).

## Nome

O nome do parâmetro deve respeitar a mesma regra para nomear o *stored procedure* (ou seja, utilizar apenas caracteres ASCII – letras maiúsculas e minúsculas sem acentuação, números inteiros e *underline*).

## Modo

O modo do parâmetro indicará se ele é um parâmetro de entrada, de saída ou ambos.

## Parâmetro de entrada

Representado pelo comando **IN**, esse parâmetro tem apenas o objetivo de informar um valor que será utilizado no bloco de código. Esse é o valor padrão para o caso de nenhum “modo” ser especificado no parâmetro. Você analisou alguns exemplos utilizando esse modo nos estudos anteriores.

## Parâmetro de saída

Representado pelo comando **OUT**, esse parâmetro tem o objetivo de receber uma variável, permitindo alterar o seu valor por meio da execução do *stored procedure*. É uma forma de retornar algo de dentro do *stored procedure* para fora, utilizando o seu valor fora do bloco de código.

## Parâmetro que seja tanto de entrada quanto de saída

É representado pelo comando **INOUT** e tem o objetivo de atuar tanto como um parâmetro de entrada quanto como um parâmetro de saída.

Analise o exemplo a seguir:

```
DELIMITER //  
CREATE PROCEDURE soma1 (INOUT numero INT)  
BEGIN  
    SET numero := numero + 1;  
END//  
DELIMITER ;
```

Nesse exemplo, declarou-se um parâmetro no modo **INOUT**, do tipo de dado **INT**. Dessa forma, é possível utilizá-lo para receber um valor no bloco de código, atualizando-o também fora do procedimento.

Execute as seguintes instruções:

```
SET @numero := 2;  
CALL soma1(@numero);  
SELECT @numero;
```

Note que, primeiramente, foi atribuído o valor "2" para uma variável de usuário "@numero". Em seguida, executou-se o *stored procedure* "soma1" por meio do comando **CALL** (você estudará mais sobre esse comando em seguida) e executou-se um comando de consulta, selecionando a variável de usuário "@numero". Veja que o resultado será:

	@numero
1	3

Como você pôde notar, a variável de usuário "@numero", que inicializou com o valor "2" atribuído, recebeu, após a execução do *stored procedure* "soma1", o valor "3", uma vez que o respectivo parâmetro foi marcado com o modo **INOUT**.

Agora, confira este código:

```
DELIMITER //
CREATE PROCEDURE buscar_nome_vendedor (
    id_vendedor INT,
    OUT nome_vendedor VARCHAR(25)
)
BEGIN
    SELECT nome INTO nome_vendedor
    FROM vendedores
    WHERE id = id_vendedor;
END//
DELIMITER ;
```

Perceba que, no exemplo, foram declarados dois parâmetros diferentes. O parâmetro "id\_vendedor", do tipo **INT**, e modo **IN** (não é necessário especificar esse modo, uma vez que ele é o padrão), e o parâmetro "nome\_vendedor", do tipo **VARCHAR(25)**, e modo **OUT**.

Esses dois parâmetros são utilizados no bloco de código da mesma forma que variáveis locais. O parâmetro com modo **OUT** pode ser utilizado no comando **INTO**, na instrução de consulta, recebendo o valor da coluna selecionada (como o que foi visto anteriormente com as variáveis). Da mesma forma, o parâmetro do modo **IN** foi utilizado na condição **WHERE** da instrução de consulta.

Que tal executar o recém-criado *stored procedure*?

## Executando um *stored procedure*



Para executar os *stored procedures* criados, utilize o comando **CALL**, como neste exemplo:

```
CALL buscar_nome_vendedor(1, @meu_vendedor);
```

Depois de passar o comando **CALL**, declare o nome do *stored procedure* que deseja executar e adicione parênteses. Caso o *stored procedure* tenha parâmetros, você deverá colocar os valores dentro dos parênteses, conforme apresentado no exemplo. O comando **CALL** citado executa o seu último *stored procedure* criado.

Perceba que o segundo parâmetro do *stored procedure* "buscar\_nome\_vendedor" é uma variável de usuário. Isso acontece porque, no *stored procedure*, o segundo parâmetro trata-se de um parâmetro de modo **OUT**, que retorna um valor atualizado. Dessa forma, provê-se uma variável de usuário, a fim de que se possa utilizar essa variável novamente, fora do bloco de código do *stored procedure*.

Além disso, você aprenderá em seguida que existe outra maneira de retornar um valor em um *stored procedure*, por meio de uma instrução de consulta. Nesse cenário, depois de utilizar o comando **CALL**, haverá uma tabela de resultados, exatamente como quando se executou uma instrução de consulta.

Caso você esteja utilizando uma linguagem de programação para interagir com o seu banco de dados, você conseguirá capturar e iterar esse resultado do seu *stored procedure*, semelhantemente ao que se pode fazer com o resultado de uma instrução de consulta.

## Retornando valores em um *stored procedure*

Você já conhece um dos meios que se tem para retornar valores em um *stored procedure*. Trata-se dos parâmetros com modo **OUT**. Passa-se uma variável para dentro do *stored procedure*, em sua chamada, conseguindo alterar o seu valor. Dessa forma, ao fim da execução do *stored procedure*, a variável terá sido alterada, conseguindo receber um valor de dentro do *stored procedure*.

Outro modo de retornar valores de um *stored procedure* é por meio da execução de uma instrução de consulta (**SELECT**), desde que não haja o comando **INTO**, uma vez que esse comando armazena o resultado da consulta em uma variável. Caso se tenha mais de uma instrução de consulta no bloco de código do *stored procedure*, apenas a primeira instrução será executada. Confira exemplo de código a seguir:

```
CREATE PROCEDURE exemplo ()  
BEGIN  
    SELECT 1;  
    SELECT 2;  
END;
```

Esse código retorna o segundo resultado:

	1
1	1

Altere agora a primeira instrução de consulta do exemplo, adicionando o comando **INTO**. Lembre-se de removê-la e recriá-la, uma vez que não pode haver dois *stored procedures* com mesmo nome.

```
CREATE PROCEDURE exemplo ()  
BEGIN  
    SELECT 1 INTO @primeiro_valor;  
    SELECT 2;  
END;
```

O resultado desse código será:

	2
1	2

### Que tal realizar alguns desafios?

Crie um *stored procedure* que liste os *top 10* dos vendedores que realizaram mais vendas em um dado período. A lista deverá apresentar o nome do vendedor e o total de vendas realizadas, além de estar ordenada pelo maior número de vendas.

Crie ao menos um dos *stored procedures* da seção “*IF statement*” e o *procedure* da seção “*While*”, do subtítulo “*Laço de repetição*”, deste conteúdo.

Agora que você conhece os *stored procedures* e aprendeu a criá-los e executá-los, conheça um pouco mais sobre as *stored functions*.

## Stored functions

As *stored functions* (traduzidas do inglês para “funções armazenadas”) são muito semelhantes aos *stored procedures*. Ambos armazenam instruções complexas, permitindo sua utilização posteriormente, e nesses dois casos haverá um nome, parâmetros de entrada e um bloco de código. A principal diferença está no retorno da *stored function* e na sua execução.

### Criando uma *stored function*

Observe o seguinte código:

```
DELIMITER //
CREATE FUNCTION buscar_preco (
    id_terreno INT
) RETURNS DECIMAL(24,2) DETERMINISTIC
BEGIN
    DECLARE v_custo_m2 DECIMAL(6,2);
    DECLARE v_largura INT;
    DECLARE v_comprimento INT;

    SELECT c.custo_metro_quadrado, t.largura, t.comprimento
    INTO v_custo_m2, v_largura, v_comprimento
    FROM terrenos t
        LEFT JOIN cidades c ON c.id = t.id_cidade
    WHERE t.id = id_terreno;

    RETURN (v_comprimento * v_largura) * v_custo_m2;
END//
DELIMITER ;
```

Assim como nos *stored procedures*, começa-se declarando o comando **CREATE**, desta vez acompanhado da palavra-chave **FUNCTION**, já que está sendo criada uma *stored function*.

Em seguida, define-se o nome da *stored function*, respeitando o mesmo padrão de nomenclatura do *stored procedures* e de outros objetos do MySQL.

Após definir o nome da *stored function*, é preciso adicionar parênteses e, caso haja parâmetros na *stored function*, eles deverão ser inseridos dentro dos parênteses.

Diferentemente dos *stored procedures*, nas *stored functions* não há os “modos” de um parâmetro (**IN**, **OUT** ou **INOUT**). Nas *stored functions*, todos os parâmetros têm o comportamento do modo **IN**, servindo apenas para alimentar o bloco de código da *stored function* com um valor de fora.

Todos os parâmetros da *stored function* devem ter um nome único, respeitando o mesmo padrão de nomenclatura da *stored function*, e um tipo de dado (sendo estas informações as mesmas já conhecidas das variáveis locais e dos parâmetros dos *stored procedures*).

Agora, porém, há uma outra grande diferença da sintaxe das *stored functions* em comparação com a sintaxe dos *stored procedures*. Nas *stored functions*, é preciso especificar o **retorno** da função. Para isso, escreve-se o comando **RETURNS** seguido do “tipo de dado” que será retornado na função. No exemplo anterior, está sendo retornado um **DECIMAL(24,2)**.

Após especificar o tipo de dado do retorno da função, deve-se especificar a característica do retorno, sendo **DETERMINISTIC** ou **NOT DETERMINISTIC**. Uma função determinística (**DETERMINISTIC**) **sempre retorna o mesmo valor quando são providos os mesmos parâmetros de entrada**. Já uma função não determinística (**NOT DETERMINISTIC**) é justamente o contrário, podendo variar o retorno, mesmo que sejam providos os mesmos parâmetros de entrada.

Após a correta especificação do retorno da função, utiliza-se os comandos **BEGIN** e **END** para empacotar o bloco de código da função. Outra diferença é a existência do comando **RETURN** dentro do bloco de código (como se pode ver no exemplo).

Uma vez especificado um retorno para a *stored function*, deve-se realizá-lo dentro do bloco de código. Para isso, utiliza-se o comando **RETURN** seguido do que se está retornando (garantindo sempre que está sendo retornado o mesmo tipo de dado especificado para a função em questão).

No exemplo anterior, a *stored function* está recebendo (por meio dos parâmetros) a ID (a chave primária) de um terreno (um registro da tabela "terrenos") e retornando um dado **DECIMAL**, sendo, de acordo com o nome da *stored function*, o preço do terreno informado. No bloco de código, declaram-se as variáveis necessárias para se descobrir o preço do terreno. Então, executa-se uma instrução

de consulta para buscar a largura, o comprimento e o preço do metro quadrado do respectivo terreno. Por último, declara-se o comando **RETURN** seguido da expressão matemática que retornará o preço do terreno (largura vezes comprimento vezes preço do metro quadrado).

No MySQL Workbench, a criação de uma *stored function* é análoga à de um *procedure*. O objeto criado pelo comando aparecerá no item **Functions** do banco de dados.

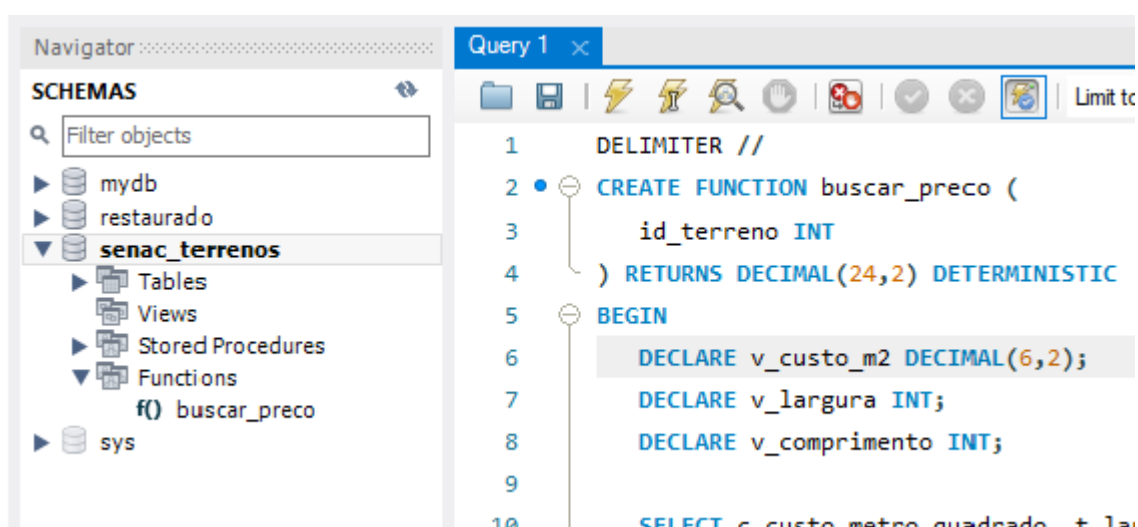


Figura 4 – Nova **Function** para o banco “mydb” no MySQL Workbench

A figura mostra a tela principal do MySQL Workbench. À direita está a área “Navigator”, com o banco de dados “senac\_terrenos” e, abaixo dele, os itens “Tables”, “Views” e “*stored procedures*” retraídos e o item “Functions” expandido, revelando, abaixo, “buscar\_preco”. À direita está o editor com o trecho do código mencionado no texto.

Que tal aprender agora a executar essa *stored function*?

## Executando uma *stored function*

Ao trabalhar com o *stored procedures*, era preciso executar uma instrução utilizando o comando **CALL**. Agora, com as *stored functions*, é possível executá-las **dentro de outras instruções**. Isso mesmo! Pode-se executar um **SELECT** de uma função ou executá-la em um **UPDATE**. O retorno da função a substituirá na instrução.

Desta forma, veja o código a seguir:

```
SELECT buscar_preco(1);
```

Com essa instrução, estaria sendo executada a função "buscar\_preco", criada no exemplo anterior, e sendo selecionado o seu retorno (de acordo com os registros do banco de dados, o seu retorno seria "307991.52").

Assim, analise o seguinte código:

```
INSERT INTO vendas (  
    id_terreno,  
    id_vendedor,  
    vendido_em,  
    valor_total  
) values (1, 1, "2022-02-07 10:00:00", buscar_preco(1));
```

Aqui está sendo executada a instrução de inserção por meio do comando **INSERT**, e, entre os valores informados, utilizou-se a função "buscar\_preco" para prover o valor da coluna "valor\_total". Portanto, como no exemplo anterior, com a instrução de consulta, a chamada da função estaria sendo substituída pelo valor do seu retorno: "307991.52".

**Que tal realizar alguns desafios?**

Crie uma função que retorne o valor total de uma venda específica, descontando 10%.

Adapte o código do *procedure* "buscar\_nome\_vendedor()" para que seja declarado como uma *stored function*.

## Triggers

Até o momento, você aprendeu a criar funções e procedimentos, o que é rotina com bloco de código e permite a criação de instruções complexas e a execução destas posteriormente. Tanto para funções quanto para procedimentos, é preciso chamar o momento da execução dessas rotinas.

Entretanto, os principais SGBDs têm um conceito chamado *trigger* (traduzido do inglês para "gatilho"). Um *trigger* é capaz de executar um bloco de código em um formato **reativo**. Isto é, por meio de um comportamento pré-especificado, o *trigger* será ativado e executado. Um exemplo é a criação de um *trigger* no qual, após a inserção de um registro em uma tabela específica, atualiza-se um outro registro em outra tabela. Perceba que, nesse formato, não é preciso ativamente executar a rotina. Por meio de um comportamento pré-especificado, a rotina será executada por conta própria.

## Criando um *trigger*

Observe o código a seguir:



```
DELIMITER //
```

```
CREATE TRIGGER marcar_venda AFTER INSERT
```

```
ON vendas
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    UPDATE terrenos
```

```
    SET vendido = true
```

```
    WHERE id = NEW.id_terreno;
```

```
END//
```

```
DELIMITER ;
```

Assim como nos procedimentos e nas funções, começa-se utilizando o comando **CREATE** para declarar a criação de um objeto, seguido da palavra-chave **TRIGGER**, uma vez que se está criando um *trigger* novo, e declara-se o **NOME** do *trigger* (mais uma vez respeitando o padrão de nomenclatura de objetos do MySQL).

Declara-se então o **momento** do gatilho, podendo ser **BEFORE** (cuja tradução do inglês é "antes") ou **AFTER** (que significa "depois"). Os *triggers* são disparados conforme comportamentos especificados. Nesse comando, informa-se se o *trigger* deve ser executado antes ou depois do respectivo comportamento.

Depois de declarar o momento do *trigger*, declara-se o **evento** que engatilhará o *trigger*. As opções são **INSERT**, para quando um registro for inserido, **UPDATE**, para atualizações, e **DELETE** para remoções.

O conjunto **momento** e **evento** será o **gatilho deste *trigger***, por exemplo, "antes de uma inserção" (**BEFORE INSERT**), ou "depois de uma remoção" (**AFTER DELETE**).

A próxima informação que deverá ser provida para a criação do *trigger* é qual tabela o gatilho estará observando. Para isso, utiliza-se o comando **ON** seguido do nome de uma das tabelas do banco de dados. Neste exemplo, está sendo observada a tabela de vendas, logo, define-se **ON vendas**.

Em seguida, é necessário declarar o comando **FOR EACH ROW** na instrução de criação de um *trigger*. Traduzido do inglês, *for each row* significa "para cada linha", e indica que o bloco de código do *trigger* será executado para cada linha que tenha ativado o gatilho. Por exemplo, caso se tenha feito uma instrução de **INSERT** na qual foram inseridas múltiplas linhas (de uma só vez), o *trigger* será executado **para cada uma das linhas inseridas**. Isso ocorrerá não apenas uma vez e também não para todas as linhas da tabela, mas sim para todas as linhas que ativaram o gatilho.

Por último, utiliza-se os comandos **BEGIN** e **END** para encapsular o bloco de código que será executado no *trigger*. Assim como nos procedimentos e nas funções, o bloco de código poderá conter condicionais, laços de repetição, variáveis de usuário, variáveis locais e múltiplas instruções.

Nota-se no bloco de código que há outra palavra-chave destacada: **NEW**. Dentro do bloco de código de um *trigger*, tem-se acesso a duas variáveis, de acordo com o evento que está sendo trabalhado. Tratam-se das variáveis **NEW** e **OLD**.

A variável **NEW** (que significa "novo") carrega o valor que está sendo "inserido" ou "atualizado" (ou seja, ela existe nos eventos **INSERT** e **UPDATE**). Note que, quando você executa uma instrução de inserção com o comando **INSERT**, você está **enviando um novo valor**. Dentro do bloco de código de um gatilho, tem-se acesso a estes novos valores que estão sendo inseridos (e isso será feito por meio da variável **NEW**). No código em questão, está buscado o valor da coluna "id\_terreno" do **novo registro inserido** (isso porque o gatilho foi configurado no evento **INSERT** – inserção). Da mesma forma, seria possível utilizar a variável **NEW** no evento **UPDATE**, acessando, assim, os **novos valores que estão sendo salvos na atualização**.

A variável **OLD** (que significa "velho"), por sua vez, carrega o valor que foi "atualizado" ou "removido" (dessa forma, essa variável existe nos eventos **UPDATE** e **DELETE**). Quando você executa uma atualização, estará inserindo novos valores

sobre um registro antigo. Por meio da variável **OLD**, tem-se acessos aos **velhos valores que serão sobrescritos**. Da mesma forma, em um evento de **DELETE**, é possível acessar os velhos valores do registro que será removido.

Com o *trigger* devidamente criado no evento de inserção da tabela de vendas, insira um registro nessa tabela para executar o bloco de código especificado no *trigger* (você pode reexecutar o comando **INSERT** do último exemplo das *stored functions*, no qual foi inserido um registro na tabela "vendas"). Com a execução desse comando de inserção, você verá que o respectivo terreno informado na inserção na tabela "vendas" terá a coluna "vendido" marcado como *true*/verdadeiro ("1") na tabela "terrenos", conforme especificado no bloco de código do *trigger*.

### Que tal realizar um desafio?

Crie um *trigger* que adicione uma comissão de 10% ao valor total de uma venda antes da inserção de um registro na tabela "vendas".

## Encerramento

A utilização de *procedures*, *functions* e *triggers* em banco de dados pode ser benéfica em vários cenários, sendo, entretanto, necessário cuidado em seu uso excessivo. Uma das vantagens da programação via linguagem é que as operações via banco de dados podem ficar muito mais rápidas. Já uma desvantagem é que a separação de regras de negócio *versus* dados fica prejudicada, o que pode complicar a manutenção de um sistema.

Para um administrador de banco de dados, é essencial o conhecimento dessas ferramentas e das possibilidades que elas proporcionam. Explore os conceitos, crie novos códigos e aprimore seus conhecimentos treinando a programação com SQL.