MODELAGEM E DESENVOLVIMENTO DE BANCO DE DADOS

Pedro Henrique Chagas Freitas

SAGAH SOLUÇÕES EDUCACIONAIS INTEGRADAS

Constraints

Objetivos de aprendizagem

- Identificar uma *constraint*.
- Exemplificar uma *constraint*.
- Implementar uma *constraint*.

Introdução

Neste capítulo, você vai estudar a utilização das *constraints* para delimitação de dados, isto é, a partir das *constraints*, restrições são colocadas na abstração lógica das tabelas, o que colabora para que as instruções SQL com *constraints* realizem restrições nas colunas das tabelas nas quais os dados serão inseridos.

Logo, as *constraints* são utilizadas na criação e na alteração das tabelas a fim de aplicar restrições frente à inserção dos dados nas colunas das tabelas.

Conceituando a utilização das *constraints*

A linguagem SQL (*Structured Query Language*) é uma linguagem para intercomunicação com bancos de dados que apresenta diversos recursos, desde a criação, exclusão e alteração de tabelas até a restrição do funcionamento das tabelas, delimitando, então, o tipo de dado que pode ser inserido em uma dada tabela.

A utilização das *constraints* pode ocorrer por diversos motivos: um dos principais é a própria modelagem de negócio presente no momento da modelagem e do desenvolvimento do banco de dados que vai atender a um dado modelo de negócio. Assim, a partir do modelo de negócio, é possível perceber restrições ou limitações que vão abstrair as necessidades do negócio e, posteriormente, aplicá-lo nas soluções que buscam atender determinada necessidade de negócio mapeada.

O surgimento das *constraints*, então, permeia a própria a linguagem SQL, tendo em vista que toda linguagem precisa conseguir abstrair a delimitação de cenários. Vamos ver um exemplo?

A *constraint* **NOT NULL** foi utilizada na coluna **Repasses_Monetários** da tabela **Bancos_Públicos** do banco de dados do Banco Central do Brasil, tendo em vista que, conforme legislação vigente (modelo de negócio), não poderíamos ter valores nulos na coluna **Repasses_Monetários**.

Esse é um exemplo puramente ilustrativo, mas que demonstra bem que a utilização das *constraints*, muitas vezes, é uma derivação de uma necessidade de negócio, já que as *constraints* se comprometem a restringir um determinado dado ou tipo de dado em uma tabela.

Obviamente, não é possível exaurir todos os tipos de *constraints* que podem ser utilizados na linguagem SQL, a linguagem mais difundida mundialmente. Assim, abordaremos as principais *constraints* e demonstraremos que o conceito de *constraints* sempre permeará algum tipo de restrição ou delimitação no tratamento dos dados em uma tabela.

Inicialmente, trabalharemos com as principais *constraints*, as quais são:

- PRIMARY KEY (chave primária);
- FOREIGN KEY (chave estrangeira);
- UNIQUE;
- NOT NULL;
- DEFAULT.

Temos aqui, então, os cinco principais tipos de *constraints* que são, sem dúvida, os mais utilizados em instruções SQL para intercomunicação com os bancos dados.

Quando o Sistema Gerenciador de Banco de Dados (SGBD) introduz uma instrução SQL para comunicação com o banco de dados, essa instrução pode, ou não, passar restrições, isto é, *constraints*. Por meio dessa comunicação entre o SGBD e o banco de dados utilizando a linguagem SQL, as *constraints* vão desempenhar um papel fundamental, que é manter a integridade do banco de dados durante a ocorrência de transações. Como as *constraints* são restrições, elas delimitam o funcionamento do banco de dados frente a instruções SQL.

A *constraint* PRIMARY KEY é uma restrição utilizada para criar uma identificação única em uma tabela, criando, então, um índice único. Logo, uma tabela pode conter somente uma única *constraint* desse tipo. Por exemplo:

```
CREATE TABLE Jogadores (

NumeroCamisa NUMBER( ) CONSTRAINT PK_Jogadores PRIMARY KEY );
```

Neste caso, estamos dizendo que o campo **NumeroCamisa** (número da camisa do jogador) será nossa PRIMARY KEY, e essa *constraint* terá o nome de **PK_Jogadores**.

É uma boa prática sempre dar nome para a *constraint* que criamos. Obviamente, a *constraint* criada funcionará sem o nome, mas, para melhor visualização e indentação de código SQL, é indicado dar nome para as *constraints*, até porque isso facilita se, posteriormente, deseje-se retirar uma dada *constraint*.

A *constraint* FOREIGN KEY também representa uma restrição, fazendo com que uma coluna ou um conjunto de colunas referencie outra tabela; logo, uma *constraint* FOREIGN KEY em uma tabela vai apontar para uma PRIMARY KEY em outra tabela. Temos, então, a partir da FOREIGN KEY, a criação do que conhecemos como relacionamento entre as tabelas, de forma que uma FOREIGN KEY apontará sempre para uma PRIMARY KEY em outra tabela. Por exemplo:

```
CONSTRAINT FK_ID_Cliente FOREIGN KEY (ID_Cliente)
REFERENCES Tabela_Clientes (ID_Cliente)
```

Neste exemplo, temos que a PRIMARY KEY está na tabela **Tabela_Clientes**, e uma FOREIGN KEY de nome **ID_Cliente** foi criada em outra tabela com o nome **FK_ID_Clientes**.

Logo, criamos um relacionamento entre duas tabelas, sendo que, na primeira, que poderíamos chamar de **Tabela_Região**, por exemplo, temos uma **FK_ID_Clientes**, que se relaciona com outra tabela — no caso, aqui, com a **Tabela_Clientes**.

Na *constraint* UNIQUE, temos uma semelhança com a *constraint* PRIMARY KEY; todavia, temos como características que a *constraint* UNIQUE é uma restrição que identifica, também de forma única, cada registro em uma tabela do banco de dados; logo, tanto a *constraint* UNIQUE como a *constraint* PRIMARY KEY garantem a unicidade de uma coluna ou de um conjunto de colunas. Isso ocorre porque a *constraint* PRIMARY KEY possui, automaticamente, uma restrição UNIQUE já definida, de modo que não é necessário especificar a *constraint* UNIQUE se utilizarmos a *constraint* PRIMARY KEY.

No entanto, a diferença é que podemos ter várias *constraints* UNIQUE em uma mesma tabela, mas podemos ter apenas uma *constraint* PRIMARY

KEY, isto é, podemos ter somente uma chave primária por tabela; logo, temos a mesma finalidade. Enquanto a *constraint* PRIMARY KEY pode ser utilizada uma única vez, a *constraint* UNIQUE pode identificar de forma única um dado registro e ser utilizada inúmeras vezes. Veja o exemplo a seguir.

Se quisermos vários valores únicos em uma tabela, faríamos:

```
CREATE TABLE EMPREGADO (

ID NUMBER ( ) CONSTRAINT PK_EMPREGADO PRIMARY KEY,
NOME VARCHAR ( ) NOT NULL,
E-MAIL VARCHAR ( ) CONSTRAINT UN_EMAIL_EMP UNIQUE KEY

);
```

Neste caso, estamos dizendo que nenhum empregado cadastrado poderá ter o mesmo e-mail, ou seja, o e-mail deverá ser único. Além disso, utilizamos também a *constraint* PRIMARY KEY para ID; logo, todo empregado tem um ID único e o ID é a PRIMARY KEY da tabela empregado.

Temos também a *constraint* **NOT NULL**, que, inclusive, já utilizamos nos exemplos anteriores que mostramos com outras *constraints*. Isso ocorre porque, sem dúvida, é natural utilizar a *constraint* **NOT NULL**, já que, por meio dela, conseguimos criar a obrigatoriedade do preenchimento de um determinado campo, ou seja, o campo não pode ser nulo. No exemplo anterior, fizemos isso:

```
CREATE TABLE EMPREGADO (

ID NUMBER ( ) CONSTRAINT PK_EMPREGADO PRIMARY KEY,
NOME VARCHAR ( ) NOT NULL,
E-MAIL VARCHAR ( ) CONSTRAINT UN_EMAIL_EMP UNIQUE KEY

);
```

Quando utilizamos NOT NULL no campo NOME, estamos dizendo que o preenchimento do nome é obrigatório, que não se pode ter um ID e um E-MAIL de um empregado que não tenha nome. Logo:

```
NOME VARCHAR ( ) NOT NULL
```

Por último, temos a *constraint* DEFAULT, que também é muito utilizada e é bem simples. A *constraint* DEFAULT é responsável por aplicar uma restrição para inserir um valor padrão especificado em uma coluna. Logo, a partir da *constraint* DEFAULT, o valor padrão será adicionado a todos os novos

registros caso nenhum outro valor seja especificado no momento da inserção do dado. Exemplo:

Vamos inserir um valor padrão, que será a data de hoje. Então, caso no momento da inserção não seja inserida uma data, a data de hoje será utilizada.

```
CREATE TABLE Cadastro (

Nome VARCHAR ( ) NOT NULL
Data_Cadastro DATATIME NOT NULL CONSTRAINT Data_Cadastro DEFAULT getDATA ( ) )
```

Temos, então, duas *constraints*: uma para designar que o valor do nome não pode ser nulo e outra para dizer que, caso não haja preenchimento do campo **Data_Cadastro** na tabela Cadastro, por padrão, será utilizada a data atual. Neste caso, a data atual será preestabelecida anteriormente, por exemplo: setDATA (03/04/2018).

Saiba mais

Neste caso, estamos usando a função **setDATA ( )** para atribuir uma data e a **getDATA ( )** para receber uma data. Em lógica de programação, principalmente no paradigma orientado a objeto, é comum a utilização desses dois recursos: SET e GET.

Saiba mais

A *constraint* PRIMARY KEY sempre terá as seguintes características:

- identifica de forma única cada registro em uma tabela do banco de dados;
- terá sempre valores únicos, por isso que, normalmente, a PRIMARY KEY em uma tabela é um ID, CPF, Código, etc.;
- por padrão, uma coluna de PRIMARY KEY não pode ter valores nulos (NULL);
- cada tabela deve ter somente uma PRIMARY KEY.

Exemplos da utilização das *constraints*

Vamos exemplificar novamente a utilização das *constraints* por meio da implementação de instruções SQL:

```
PRIMARY KEY

CREATE TABLE LojaCarros (

PLACA NUMBER( ) CONSTRAINT PK_Placa PRIMARY KEY );
```

Neste caso, estamos dizendo que nossa restrição primária (PRIMARY KEY) para a tabela **LojaCarros** é a placa do carro.

```
FOREIGN KEY

CREATE TABLE Aeroporto (

CONSTRAINT FK_Cartao_Embarque FOREIGN KEY (Cartao_Embarque) REFERENCES
Tabela_Passageiros (ID_Passageiro);
```

Estamos dizendo, agora, que a FOREIGN KEY **FK_Cartao_Embarque** está interligando-se ou relacionando-se com o ID_Passageiro, que é PRIMARY KEY na **Tabela_Passageiros** — o que, a propósito, é uma situação real, tendo em vista que o cartão de embarque muda, mas o ID do passageiro, como é PRIMARY KEY, faz com que seja possível identificá-lo em qualquer voo. Todavia, ao pensarmos no cartão de embarque como uma FOREIGN KEY, percebemos que todo passageiro tem um cartão de embarque; assim, o relacionamento ou referenciamento entre **Cartao_Embarque** e **ID_Passageiro** é válido.

```
UNIQUE

CREATE TABLE FUNCIONARIO (

CPF NUMBER ( ) CONSTRAINT PK_CPF_FUNCIONARIO PRIMARY KEY,
NOME VARCHAR ( ) NOT NULL,
RG VARCHAR ( ) CONSTRAINT RG_FUNC UNIQUE KEY

);
```

Neste caso, estamos dizendo que, na tabela FUNCIONARIO, temos como PRIMARY KEY o CPF; contudo, o campo RG também é único, ou seja,

nossa restrição é que nenhum funcionário poderá ter o RG igual ao de outro funcionário.

```
NOT NULL
CREATE TABLE CORREIOS
ENDERECO VARCHAR ( ) NOT NULL
```

A restrição NOT NULL é muito simples: neste caso, estamos dizendo que, na Tabela Correios, o campo ENDERECO não poderá ser nulo, isto é, deverá ser preenchido.

```
DEFAULT
ALTER TABLE Tabela_Autor (
MODIFY COLUMN Sobrenome_Autor Varchar ( ) DEFAULT 'Henrique';
```

Estamos, agora, dando um exemplo com a alteração de uma tabela (ALTER TABLE) a fim de demonstrar a utilização da *constraint* em uma alteração, tendo em vista que temos dois cenários para sua utilização na criação de tabelas (CREATE TABLE) e na alteração (ALTER TABLE).

Logo, temos que, na **Tabela_Autor**, a coluna **Sobrenome_Autor** será modificada ou alterada, e, como valor padrão, vamos passar 'Henrique' para ser o sobrenome de todos os autores.

Fique atento

Existem várias *constraints*, mas todas têm como fundamento restringir, ou seja, implementar uma restrição em uma coluna de uma tabela. As *constraints* são utilizadas na criação de uma tabela (CREATE TABLE) ou na alteração de uma tabela (ALTER TABLE), e é possível tanto adicionar *constraints* quanto remover as *constraints* adicionadas.

Implementação de *constraints* com a linguagem SQL

Nesta implementação, utilizaremos *constraints* em conjunto com instruções SQL para criar tabelas e implementar relacionamentos entre as tabelas.

Teremos, então, duas tabelas que se relacionam; logo, temos uma chave primária, isto é, a *constraint* PRIMARY KEY e uma chave estrangeira, ou seja, a *constraint* FOREIGN KEY. Assim, teremos duas tabelas já criadas (CREATE TABLE): tb_funcionario e tb_pessoas.

Agora, vamos apagar essas tabelas para recriá-las dentro da situação proposta de utilização das *constraints*. Teríamos, então:

```
DROP TABLE tb_funcionarios;
DROP TABLE tb_pessoas;
```

Vamos criá-las novamente, mas agora colocando as *constraints* e começando pela tabela: tb_pessoas:

```
CREATE TABLE tb_pessoas (

IDpessoa INT AUTO_INCREMENT NOT NULL,
Nome VARCHAR ( ) NOT NULL,
DtCadastro TIMESTAMP NOT NULL, DEFAULT CURRENT_TIMESTAMP ( ),
CONSTRAINT PK_pessoas PRIMARY KEY (IDpessoa)
);
```

Neste caso, criamos a tabela **tb_pessoa** com os campos **IDpessoa** e utilizamos a *constraint* **NOT NULL**, ou seja, o **IDpessoa** deve ser preenchido. Criamos o campo Nome, utilizando também a *constraint* **NOT NULL**, e o campo **DtCadastro**, utilizando a *constraint* DEFAULT para delimitar a data como sendo uma data padrão que será recebida. Por fim, definimos que **PK_pessoa** será o nome da nossa PRIMARY KEY, que será o **IDpessoa**.

Agora, vamos criar a tabela **tb_funcionarios**, observando que a tabela pessoa já tem alguns campos que servem para a tabela funcionários, tendo em vista que funcionário tem nome e data de cadastro; logo, podemos fazer uma relação em que um funcionário necessariamente tem que ser uma pessoa e, como já temos a tabela pessoa, vamos relacionar o funcionário com a pessoa. Para isso, vamos criar uma FOREIGN KEY:

```
CREATE TABLE tb_funcionarios (
Idfuncionario INT AUTO_INCREMENET NOT NULL,
IDpessoa INT NOT NULL,
Vlsalario DECIMA ( ),
Dtadmissao DATE NOT NULL,
CONSTRAINT PK_funcionarios PRIMARY KEY (idfuncionario),
CONSTRAINT FK_funcionarios_pessoas foreign key (idpessoa)
REFERENCES tb_pessoas (idpessoa)
```

É possível verificar, então, que o que for inserido na tabela **tb_funcionarios** será verificado, antes, se existe na **tb_pessoa**; logo, as duas colunas estão interligadas, e nós chamamos isso de integridade referencial.

Logo, se inserimos uma pessoa:

```
INSERT INTO tb_pessoas VALUES(NULL, 'Joao', NULL);
```

Depois, realizamos uma consulta para verificar o dado que inserimos, observando que, neste caso, colocamos o NULL no campo **IDpessoa** e **DTCadastro** porque estão sendo retornados automaticamente da tabela funcionários.

Fazemos a consulta:

```
SELECT * FROM tb_pessoas;
```

Agora que João é uma pessoa, vamos torná-lo um funcionário:

```
INSERT INTO tb_funcionarios VALUES (NULL, 1, 10000, CURRENT_DATE ( ) );
```

O ID 1 é referenciado como o primeiro a ser inserido; no caso, é João, por isso estamos utilizando o 1. Fazemos, então, uma consulta à tabela funcionário: **SELECT * FROM tb_funcionarios**; agora, teremos João como funcionário, tendo em vista que as tabelas pessoa e funcionários (Figuras 1 e 2) estão interligadas por meio das *constraints* PRIMARY KEY e FOREIGN KEY.

tb_pessoa

Idpessoa	Nome	Dtadmissao
1	João	25/05/2017

Figura 1. Tabela pessoa.

tb_funcionarios

Idfuncionario	Idpessoa	Vlsalario	Dtadmissao
1	1	10000	25/05/2017

Figura 2. Tabela funcionários.

Por fim, então, teremos um relacionamento criado por duas *constraints* (PRIMARY KEY e FOREIGN KEY), sendo que, ao longo da criação das tabelas, utilizamos, também, outras *constraints*. Dessa forma, quando consultamos a tabela funcionário, conseguimos retornar o **IDpessoa**, que está referenciado na tabela pessoa conforme mostrado, realizando, assim, o relacionamento entre as tabelas "pessoa" e "funcionário".

Leituras recomendadas

ELMASRI, R.; NAVATHE, S. B. *Sistemas de banco de dados*. 6. ed. São Paulo: Pearson, 2011.

HEUSER, C. A. *Projeto de banco de dados*. 6. ed. Porto Alegre: Bookman, 2010. (Série Livros Didáticos Informática UFRGS, v. 4).

KORTH, H. F.; SILBERSHATZ, A.; SUDARSHAN, S. *Sistema de banco de dados*. 6. ed. Rio de Janeiro: Campus, 2012.

RAMAKRISHNAN, R. *Sistemas de gerenciamento de banco de dados*. 3. ed. Porto Alegre: Penso, 2009.

SETZER, V. W. *Banco de dados:* conceitos, modelos, gerenciadores, projeto lógico, projeto físico. 3. ed. São Paulo: Blucher, 2002.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS