rede e-Tec Brasil

Prática de Banco de Dados

Thiago Alves Elias da Silva
Nádia Mendes dos Santos
Wilson de Oliveira Júnior

Curso Técnico em Informática




INSITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA PIAUÍ

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PDE | PRONATEC
PROGRAMA NACIONAL DE ACESSO AO ENSINO TÉCNICO E EMPREGO

Ministério da Educação

Prática de Banco de Dados

Thiago Alves Elias da Silva
Nádia Mendes dos Santos
Wilson de Oliveira Júnior

INSITUTO FEDERAL DE EDUCAÇÃO
CIÊNCIA E TECNOLOGIA
PIAUÍ

Teresina-PI
2013

Presidência da República Federativa do Brasil

Ministério da Educação

Secretaria de Educação Profissional e Tecnológica

© Instituto Federal de Educação, Ciência e Tecnologia do Piauí

Este Caderno foi elaborado em parceria entre o Instituto Federal de Educação, Ciência e Tecnologia do Piauí e a Universidade Federal de Santa Catarina para a Rede e-Tec Brasil.

Equipe de Elaboração
Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI

Coordenação Insitucional
Francieric Alves de Araujo/IFPI

Coordenação do Curso
Thiago Alves Elias da Silva/IFPI

Professores-autores
Thiago Alves Elias da Silva/IFPI
Nádia Mendes dos Santos/IFPI
Wilson de Oliveira Júnior/IFPI

Comissão de Acompanhamento e Validação
Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional
Araci Hack Catapan/UFSC

Coordenação do Projeto
Silvia Modesto Nassar/UFSC

Coordenação de Design Instrucional
Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico
Juliana Tonietto/UFSC

Design Instrucional
Juliana Leonardi/UFSC

Web Master
Rafaela Lunardi Comarella/UFSC

Web Design
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação
Liana Domeneghini Chiaradia/UFSC
Marília Ceriolli Hermoso/UFSC
Roberto Gava Colombo/UFSC

Projeto Gráfico
e-Tec/MEC

Catalogação na fonte pela Biblioteca Universitária
Universidade Federal de Santa Catarina

S586p Silva, Thiago Alves Elias da
Prática de banco de dados / Thiago Alves Elias da Silva, Nádia Mendes dos Santos, Wilson de Oliveira Júnior. – Teresina : Instituto Federal de Educação, Ciência e Tecnologia do Piauí, 2013.
88 p. : il., tabs.

Inclui bibliografia
ISBN: 978-85-67082-07-3

1. Banco de dados. 2. Linguagem de programação computadores). 3. SQL (linguagem de programação de computador). I. Santos, Nádia Mendes dos. II. Oliveira Júnior, Wilson de. III. Título.

CDU: 681.31.06SQL

Apresentação e-Tec Brasil

Bem-vindo a Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira propiciando caminho de o acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (SETEC) e as instâncias promotoras de ensino técnico como os Institutos Federais, as Secretarias de Educação dos Estados, as Universidades, as Escolas e Colégios Tecnológicos e o Sistema S.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Abril de 2013

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

Atenção: indica pontos de maior relevância no texto.

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

Palavra dos professores-autores 9

Apresentação da disciplina 11

Projeto instrucional 13

Aula 1 – Introdução e projeto de banco de dados 15

1.1 Introdução 15

1.2 Dados e objetos do banco de dados 16

1.3 Projeto de banco de dados 17

1.4 Modelagem conceitual 18

1.5 Projeto lógico 20

Aula 2 – A linguagem SQL – principais comandos 23

2.1 Introdução a SQL 23

2.2 Tipos de dados básicos 23

2.3 O comando *create database* 24

2.4 O comando *create table* 25

2.5 O comando *alter table* 28

2.6 O comando *drop table* 30

2.7 Consultas simples 31

2.8 Manipulando dados de uma tabela 33

(*insert, update, delete*) 33

2.9 Funções agregadas 37

2.10 A cláusula *group by* 39

2.11 Junções (*join*) 40

Aula 3 – Procedimento e função 47

3.1 Introdução à programação em SQL 47

3.2 Procedimentos (*store procedure*) 59

3.3 Funções (*functions*) 64

Aula 4 – Gatilho e controle de acesso **71**
4.1 Gatilho (*trigger*) 71
4.2 Controle de acesso 76
4.3 Privilégios 79

Referências **86**

Currículo dos professores-autores **87**

Palavra dos professores-autores

Caro estudante!

Inicialmente, gostaríamos de dar as boas vindas a você na qualidade de estudante da disciplina de Prática de Banco de Dados e desejar boa sorte em mais esta etapa do nosso curso.

Esta disciplina, assim como as demais do curso, possui uma grande importância para a sua formação de técnico em Informática. Juntos, desbravaremos caminhos e conheceremos novos horizontes e desafios que a Prática de Banco de Dados traz para você.

Acreditamos em seu potencial e por isso esperamos que você se mantenha assíduo e se organize de tal modo a participar das atividades, a corresponder aos desafios impostos e nos oferecer sua contribuição enquanto um estudante autodidata que primará pela descoberta de novos conhecimentos.

Nós os professores-autores desejamos a você um ótimo rendimento e declaramos que acreditamos em sua capacidade de estudo, criatividade e participação.

Um grande abraço!

Prof. Thiago A. Elias da Silva
Profa. Nádia Mendes dos Santos
Prof. Wilson de Oliveira Júnior

Apresentação da disciplina

Bem vindo (a) à disciplina Prática de Banco de Dados.

Esta é o nosso caderno, material elaborado com o objetivo de contribuir para o desenvolvimento de seus estudos e para a ampliação de seus conhecimentos acerca desta disciplina.

Este texto é destinado aos estudantes aprendizes que participam do programa Escola Técnica Aberta do Brasil (e-Tec Brasil), vinculado à Escola Técnica Aberta do Piauí (ETAPI) do Instituto Federal de Educação, Ciências e Tecnologia do Piauí (IFPI).

O texto é composto de quatro (04) Aulas assim distribuídas:

Na **Aula 1 – Banco de Dados**, conceitos, definição de SGBD, projeto de banco de dados passando pelos modelos conceitual e lógico.

Na **Aula 2 – A linguagem SQL**, tipos de dados, instruções DDL e instruções DML.

Na **Aula 3 – Procedimentos e funções**, apresentamos uma visão geral acerca dos comandos em SQL para se programar no MySQL. Uma introdução aos procedimentos (*store procedure*) e às funções (*functions*) e como implementá-las no MySQL.

Na **Aula 4 – Gatilho e controle de acesso**, apresentamos uma visão sobre gatilhos (triggers) e controle de acesso no MySQL. E como implementar os gatilhos, criar usuários e atribuir privilégios a esses usuários.

Projeto instrucional

Disciplina: Prática de Banco de Dados (carga horária: 60h

Ementa: Linguagem SQL. Prática de banco de dados. Instruções DDL e DML. Estudo de caso usando o MySQL.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Introdução e projeto de banco de dados	Conhecer os conceitos de Banco de Dados e SGBDs. Modelar um Banco de Dados.	Vídeo-aulas; Webconferências; *Sites*; Ambiente Virtual de Ensino-Aprendizagem (AVEA).	15
2. A linguagem SQL – principais comandos	Construir e alterar tabelas em um banco de dados utilizando instruções SQL. Realizar consultas simples e complexas em um banco de dados utilizando instruções SQL.	Vídeo-aulas; Webconferências; SGBD MySQL; *Sites*; Ambiente Virtual de Ensino-Aprendizagem (AVEA).	15
3. Procedimentos e funções	Definir procedimento armazenado (*store procedure*). Conhecer a sintaxe de construção de procedimento. Implementar e executar procedimentos. Definir função (*function*). Conhecer a sintaxe de construção de função. Implementar e executar função.	Vídeo-aulas; Webconferências; SGBD MySQL; *Sites*; Ambiente Virtual de Ensino-Aprendizagem (AVEA).	15
4. Gatilho e controle de acesso	Definir gatilho (*trigger*). Conhecer a sintaxe de um gatilho (*trigger*). Implementar gatilhos (trigger). Definir usuário de banco de dados. Criar, renomear, deletar, alterar senha de usuário no MySQL. Definir privilégios de usuário de BD. Atribuir e remover privilégios de usuários.	Vídeo-aulas; Webconferências; SGBD MySQL; *Sites*; Ambiente Virtual de Ensino-Aprendizagem (AVEA).	15

Aula 1 – Introdução e projeto de banco de dados

Objetivos

Conhecer os conceitos de Banco de Dados e SGBDs.

Modelar um Banco de Dados.

1.1 Introdução

Nesta disciplina de Prática de Banco de Dados, como o próprio nome sugere, aplicaremos, na prática, alguns conceitos de banco de dados.

Daremos ênfase à implementação do banco de dados, estudando desde a modelagem, passando pela criação de uma base de dados até a sua manipulação através de consultas simples e complexas. Vamos criar procedimentos armazenados, dentre outros objetos de banco de dados.

Tentaremos, ao máximo, utilizar uma linguagem simples e de baixa complexidade. Vale lembrar que o objetivo da disciplina não é aprender a utilizar um SGBD em específico, focaremos no aprendizado da linguagem SQL. Porém, como não poderemos seguir na disciplina sem utilizarmos um determinado SGBD, indicamos o MYSQL como referência. Lembro-lhe também que existem outros SGBDs, alguns gratuitos, outros pagos. Listamos alguns exemplos a seguir (Figura 1.1): Postgresql, SQL Server, Oracle, Access, Firebird, dentre outros.

Figura 1.1: Exemplos de SGBDs
Fonte: Elaborada pelos autores

Para aqueles que estão esquecidos, SGBD significa Sistema Gerenciador de Banco de Dados, e nada mais é do que um conjunto de *software* que tem por objetivo administrar uma base dados, gerenciando o acesso, a manipulação e a organização dos dados. O SGBD provê uma interface amigável para o usuário de tal forma que ele possa incluir, alterar ou consultar os dados. A Figura 1.2 representa o esquema de um SGBD.

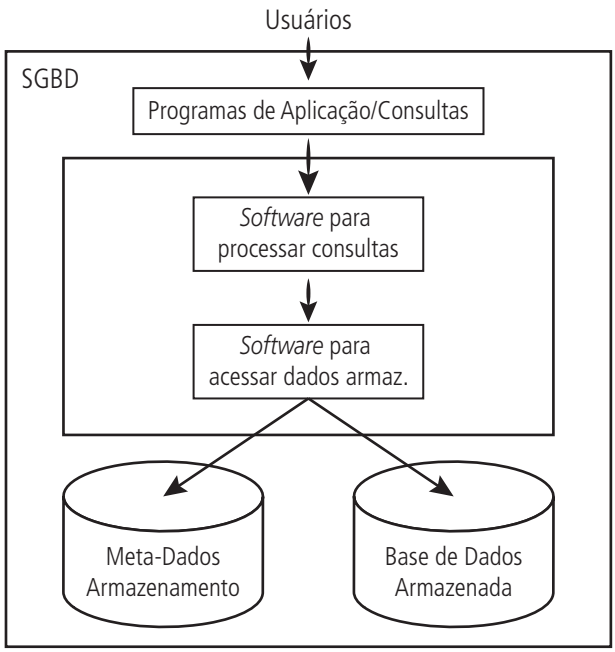

Figura 1.2: Organização de um SGBD
Fonte: Costa (2006)

Antes de colocarmos a mão na massa e começarmos a codificar a nossa base de dados e as nossas tabelas, precisamos conhecer alguns conceitos. Vamos lá?

1.2 Dados e objetos do banco de dados

Segundo COSTA (2006), os SGBDs foram projetados para armazenar dados. Nos SGBDs relacionais e objeto-relacionais, os dados são armazenados em tabelas. As tabelas são um dos tipos de objetos do banco de dados. Existem outros tipos de objetos, tais como visões, gatilhos, procedimentos, dentre outros. Note que, embora os SGBDs objeto-relacional e orientado a objetos possam armazenar objetos como os definidos na orientação a objetos, os objetos do banco de dados a que nos referimos não são os objetos da orientação a objetos e, sim, estruturas internas de Sistemas Gerenciadores.

Em outras palavras, a partir de agora, quando falarmos de objetos de banco de dados em SGBDs relacionais, não estaremos nos referindo a objetos estudados na disciplina de Introdução a Programação Orientada a Objetos e, sim, a estruturas internas do Sistema Gerenciador, tais como tabelas, procedimentos armazenados, gatilhos e tantas outras estruturas que ainda iremos estudar.

1.3 Projeto de banco de dados

Ao construirmos uma aplicação para controlar, por exemplo, o mercadinho do Sr. Raimundo, devemos criar um banco de dados para armazenar suas informações. Porém algumas perguntas nos vêm à cabeça: como os dados serão armazenados? Se pensarmos em armazenar em tabelas, quantas precisaremos criar, quais as informações serão armazenadas em cada uma delas e quais os tipos de dados de cada informação? Para respondermos a estas perguntas, devemos projetar o banco de dados a partir de modelos que se divide em três etapas: modelagem conceitual, modelagem lógica e modelagem física.

1. Na **modelagem conceitual**, há a representação dos conceitos e características observadas no ambiente para o qual se deseja projetar o banco de dados, ou seja, a empresa para o qual o sistema está sendo desenvolvido. Nesta etapa, devemos ignorar as particularidades de implementação.

2. Na **modelagem lógica**, buscamos transformar o esquema conceitual para informações técnicas que possam ser compreendidas e utilizadas por desenvolvedores e administradores de banco de dados.

3. Já na **modelagem física**, são consideradas características específicas do SGBD onde está sendo implementada a base de dados.

Como forma de exemplificação, utilizaremos um estudo de caso de um sistema que controlará uma empresa de desenvolvimento de projetos. Este mesmo banco de dados será utilizado no decorrer de todo o material. Logo abaixo, iniciaremos a explicação do estudo de caso e, em seguida, projetaremos o nosso banco de dados.

Uma empresa de desenvolvimento de projetos deseja informatizar-se. Após uma conversa com os proprietários e funcionários da empresa levantamos as seguintes informações:

Cada funcionário está lotado em um departamento e em um determinado departamento poderá existir vários funcionários trabalhando nele. Os departamentos são cadastrados a partir de seu código, descrição e sua localização no prédio. O sistema precisa manter também os dados de todos os funcionários, tais como: nome e data de nascimento. No cadastro dos projetos, deve existir seu código, que é o seu identificador único, o nome do projeto, o orçamento, a data de início e a data prevista para a sua conclusão.

Um projeto poderá ser desenvolvido por uma equipe ou por apenas um funcionário. Da mesma forma, um funcionário poderá desenvolver mais de um projeto, ao mesmo tempo ou não.

No ato da vinculação do funcionário ao projeto, o sistema deve controlar a data da vinculação, a data de desvinculação e a função desempenhada pelo funcionário naquele projeto. A escolha da função NÃO está relacionada ao departamento que o funcionário está lotado. Para cada função, existe um valor de salário mensal que poderá ser pago de forma integral ou proporcional, caso o funcionário não trabalhe um mês inteiro.

Quando um funcionário se vincula a um projeto e, após a conclusão da sua participação, este se desvincula, ele não poderá mais participar novamente do mesmo projeto, mesmo que desempenhando outra função, ou seja, cada funcionário só poderá associar-se apenas uma vez em cada projeto.

1.4 Modelagem conceitual

Para a modelagem conceitual, utilizaremos o modelo Entidade-Relacionamento (ER) que já foi estudado na disciplina de Introdução a Banco de Dados.

Segundo COSTA (2006), uma entidade representa um conjunto de objetos da realidade que está sendo modelada. No mundo real, os objetos representados pelas entidades podem interagir entre si de diferentes formas. Para modelar tais interações, utilizamos os relacionamentos. As entidades que possuem vínculos entre si estarão ligadas por relacionamentos. Além disso, no mundo real, cada entidade poderá ter uma série de propriedades, as quais são modeladas através de atributos. Algumas vezes, os relacionamentos também possuem atributos próprios. Entidades, relacionamentos e atributos possuem um nome. Para enriquecer o modelo, utilizamos as cardinalidades. Estas representam o número de interações que uma dada instância de uma entidade poderá ter com instâncias de outra entidade a ela relacionada.

Existem diferentes representações para modelarmos um banco de dados a partir do modelo entidade-relacionamento. Na Figura 1.3 seguem as representações que utilizamos neste material.

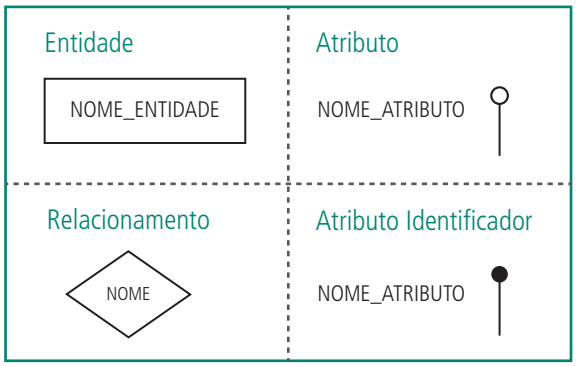

Figura 1.3: Notação para diagramas entidade-relacionamento
Fonte: Elaborada pelos autores

Seguindo com a análise do nosso estudo de caso, descobrimos a existência de quatro entidades, apresentadas na Figura 1.4: Funcionário, Projeto, Departamento e Função.

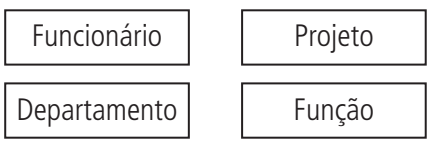

Figura 1.4: Entidades
Fonte: Elaborada pelos autores

Observamos também que a entidade Funcionário interage com as entidades Projeto e Departamento. Quanto às cardinalidades, a partir da análise do texto que descreve o estudo de caso, um funcionário poderá se relacionar a apenas um departamento, mas um departamento poderá ter um ou vários funcionários associados a ele. Por sua vez, um funcionário poderá desenvolver mais de um projeto e um projeto poderá ser desenvolvido por um ou por vários funcionários. Já a entidade Função se relaciona ao Funcionário e ao Projeto, já que a função do funcionário é determinada na hora em que este se associa a um determinado projeto.

Assim, temos o diagrama da Figura 1.5, representando o modelo conceitual do banco de dados do sistema que gerencia a nossa empresa de desenvolvimento de projetos:

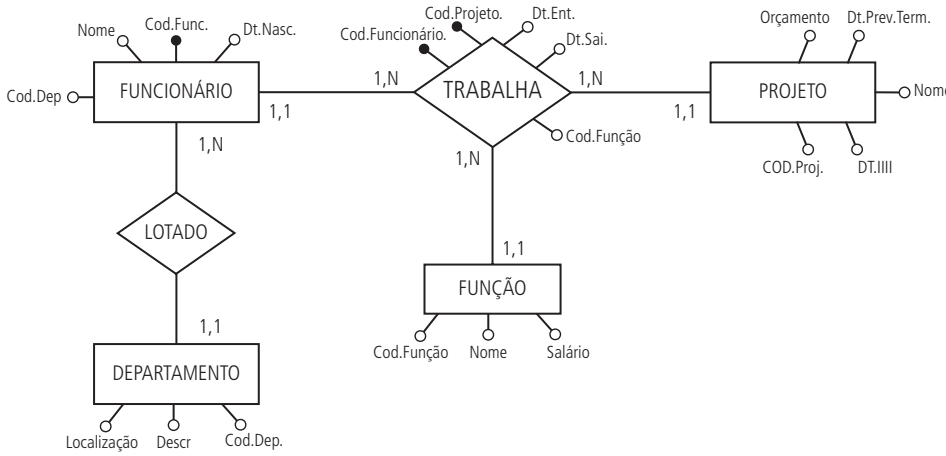

Figura 1.5: Diagrama Entidade-relacionamento
Fonte: elaborada pelos autores

1.5 Projeto lógico

Segundo Costa (2006), muitas vezes, uma entidade ou relacionamento do diagrama entidade-relacionamento se tornará uma tabela no projeto lógico. Existem algumas situações onde isso não é verdade. Essas situações são especificadas, principalmente, a partir das cardinalidades existentes.

A partir da aplicação das regras de cardinalidade no modelo conceitual apresentado na Figura 1.5, especificamos as suas respectivas tabelas no destaque abaixo. Os nomes em negrito representam as tabelas. Os itens delimitados por parênteses são as colunas de cada tabela. As colunas que têm o seu nome sublinhado formam as chaves primárias das tabelas. Já aquelas com o nome em itálico representam chaves estrangeiras.

Departamento (cod_dep, descr, localiz)

Funcionário (cod_func, nome, dt_nasc, *cod_dep*)

Projeto (cod_proj, nome, orcamento, dt_ini, dt_prev_term)

Função (cod_funcao, nome, sal)

Trabalha (*cod_func, cod_proj, cod_funcao*, dt_ent, dt_sai)

A partir da especificação das tabelas, poderemos definir no projeto lógico as restrições adicionais para cada coluna, indicando, por exemplo, se seu preenchimento é obrigatório ou não. Poderemos também especificar o tipo de informação que será armazenada em uma coluna, definindo o domínio utilizado para os dados. Veja o quadro 1.1:

Vamos fazer uma atividade diferente? Assista a vídeo-aula disponível em http://cefetpi.nucleoead.net/etapi/ referente aos conteúdos até agora estudados e, em poucas palavras, escreva um texto narrando o que você entendeu. Poste o texto em nosso AVEA.

Quadro 1.1: Descrição do banco de dados

TABELA	COLUNA	DESCRIÇÃO	DOMÍNIO	REQUERIDO
Departamento	cod_dep	Código do departamento	Numérico inteiro	Sim
	descr	Descrição do departamento	Cadeia de caracteres.	Sim
	localiz	Localização do departamento	Cadeia de caracteres	Sim
Funcionário	cod_func	Código do funcionário	Numérico inteiro	Sim
	nome	Nome do funcionário	Cadeia de caracteres	Sim
	dt_nasc	Data de nascimento do funcionário	Data	Não
	cod_dep	Código do departamento que o funcionário trabalha	Numérico inteiro	Não
Projeto	cod_proj	Código do projeto	Numérico inteiro	Sim
	nome	Nome do projeto	Cadeia de caracteres	Sim
	orcamento	Orçamento do projeto	Número Real	Sim
	dt_ini	Data de início do projeto	Data	Sim
	dt_prev_term	Data prevista para o término do projeto	Data	Sim
Função	cod_funcao	Código da função	Numérico inteiro	Sim
	nome	Nome da função	Cadeia de caracteres	Sim
	sal	Salário pago para quem desempenhar a função	Número Real	Sim
Trabalha	cod_func	Código do funcionário	Numérico inteiro	Sim
	cod_proj	Código do projeto	Numérico inteiro	Sim
	cod_funcao	Código da função desempenhada pelo funcionário no projeto	Numérico inteiro	Sim
	dt_ent	Data de entrada do funcionário no projeto	Data	Sim
	dt_sai	Data de saída do funcionário do projeto	Data	Não

Fonte: Elaborado pelos autores

Resumo

Nesta primeira aula foi feita uma revisão dos conceitos de banco de dados e sistema gerenciador de banco de dados. Conceituou-se a modelagem conceitual, lógica e física. Paralelo a isso, foi proposto um exemplo de modelagem utilizando o modelo entidade-relacionamento.

Atividades de aprendizagem

1. Defina SGBD, explique qual a sua principal função e cite, pelo menos, 3 exemplos.

2. Quando falamos em objetos de banco de dados, estamos nos referindo aos mesmos objetos definidos na orientação a objetos? Explique.

3. Defina a modelagem conceitual, modelagem lógica e modelagem física.

4. Modele, utilizando o modelo entidade-relacionamento, o banco de dados de um sistema que controlará uma clínica médica. Nesta clínica, deverão ser cadastrados todos os médicos, além de suas especialidades. Mesmo que o médico possua mais de uma especialidade, ele deverá escolher apenas uma delas para trabalhar na clínica. Todos os pacientes também deverão ser cadastrados com todos os seus dados. Os dados das consultas deverão ser armazenados também no banco de dados: a data em que a consulta aconteceu, o médico que atendeu o paciente, bem como o diagnóstico feito pelo médico.

Poste suas respostas no AVEA.

Aula 2 – A linguagem SQL – principais comandos

Objetivos

Construir e alterar tabelas em um banco de dados utilizando instruções SQL.

Realizar consultas simples e complexas em um banco de dados, utilizando instruções SQL.

2.1 Introdução a SQL

Segundo Neto (2003), *Structured Query Language* (SQL), ou Linguagem de Consulta Estruturada, é a linguagem estruturada padrão para acessar dados num sistema gerenciador de banco de dados. Ela foi criada originalmente no início dos anos 80, pela IBM. Em meados de 1980, a ANSI iniciou seus trabalhos de desenvolvimento para padronização de uma linguagem para banco de dados relacionais.

A ANSI e a ISO juntaram esforços e trabalharam num conjunto maior de padronização para o SQL, criando o chamado SQL ANSI-92, ou ainda SQL-92 ou simplesmente SQL2. Um grupo de trabalho estendeu o SQL2 para um novo padrão que incorpora características orientadas a objetos (como, por exemplo, a herança), o SQL3.

A-Z

ANSI
Significa *American Nacional Standards Instituite* – Instituto Americano de Padrões.

ISO
Significa *International Standards Organization* – Organização Internacional de Padrões.

2.2 Tipos de dados básicos

Em banco de dados relacionais, cada tabela pode conter diversas colunas, as quais armazenarão os dados. Para cada coluna, existirá um tipo de dado associado. Os tipos de dados são definidos durante a criação da tabela. No Quadro 2.1, apresentamos os principais tipos de dados simples definidos pela SQL:2003.

Eu sei que você vai querer saber um pouco mais sobre a linguagem SQL. Para isso, acesse o endereço http://pt.wikipedia.org/wiki/SQL

As instruções que estudaremos durante toda a aula 2 foram implementadas no MySQL. Porém, elas poderiam ser executadas em qualquer outro SGBD.

Quadro 2.1: Principais tipos de dados simples definidos pela SQL:2003	
Tipos de Dados	**Descrição**
CHARACTER	Caractere de tamanho fixo – usualmente conhecido como CHAR
CHARACTER VARYING	Caractere de tamanho variante – usualmente conhecido como VARCHAR
CHARACTER LARGE OBJECT	Caractere longo – usualmente conhecido como CLOB
BINARY LARGE OBJECT	String binária para objetos longos – usualmente conhecido como BLOB
NUMERIC	Numérico exato
DECIMAL	Numérico exato
SMALLINT	Numérico exato
INTERGER	Numérico exato
BIGINT	Numérico exato
FLOAT	Numérico aproximado
REAL	Numérico aproximado
DOUBLE PRECISION	Numérico aproximado
BOOLEAN	Booleano
DATE	Data com informações de dia, mês e ano
TIME	Hora com informações de hora, minuto e segundo
TIMESTAMP	Determina um momento, com informações de ano, mês, dia, hora, minuto e segundo

Fonte: Costa (2006)

Os vários fornecedores de Sistemas Gerenciadores de Banco de Dados (SGBDs) utilizam variações próprias dos tipos de dados definidos na SQL:2003. No Oracle, o tipo de dados mais utilizado para tratamento de informações numéricas é o tipo NUMBER. Já no SQL SERVER 2005 e no DB2 versão 9, são utilizados vários tipos de dados para armazenamento de informações numéricas, com denominações bem próximas do padrão SQL. No que se refere a tipos de dados referentes a cadeias de caracteres, os principais gerenciadores de banco de dados comerciais se aproximam bastante do padrão da linguagem (COSTA, 2006).

2.3 O comando *create database*

A instrução *create database*, como o próprio nome sugere, serve para criarmos a base de dados na qual as tabelas serão criadas.

Sua sintaxe é bastante simples. Vejamos, através da Figura 2.1, a criação de uma base de dados chamada *praticaBD*.

```
CREATE DATABASE praticaBD;
```

Figura 2.1: Criação da base dados chamada *praticaBD*
Fonte: Elaborada pelos autores

2.4 O comando *create table*

Após criarmos a nossa base de dados, criaremos as nossas tabelas. Para isso, faremos uso do comando *create table*.

O comando *create table* permite criarmos e definirmos a estrutura de uma tabela, definindo suas colunas (campos), suas respectivas restrições, além de suas chaves primárias e estrangeiras. Sua sintaxe é apresentada na Figura 2.2.

```
CREATE TABLE nome-tabela
(nome-coluna tipo-do-dado [NOT NULL]
                          [NOT NULL WITH DEFAULT]
CONSTRAINT nome PRIMARY KEY (nome-coluna-chave)
CONSTRAINT nome FOREIGN KEY (nome-coluna-chave-estrangeira)
     REFERENCES nome-tabela-pai(nome-coluna-pai))
ON DELETE [RESTRICT] [CASCADE]  [SET NULL]
```

Figura 2.2: Sintaxe do comando *create table*
Fonte: Costa (2006)

Os comandos entre colchetes [ ] são opcionais.

Onde:

nome-tabela representa o nome da tabela que será criada.
nome-coluna representa o nome da coluna que será criada. A definição das colunas de uma tabela é feita relacionando-as uma após a outra.
tipo-do-dado define o tipo e tamanho dos campos definidos para a tabela.
NOT NULL exige o preenchimento do campo, ou seja, no momento da inclusão é obrigatório que possua um conteúdo.
NOT NULL WITH DEFAULT preenche o campo com valores pré-definidos, de acordo com o tipo do campo, caso não seja especificado o seu conteúdo no momento da inclusão do registro.
CONSTRAINT nome PRIMARY KEY (**nome-coluna-chave**) define para o banco de dados a coluna que será a chave primária da tabela. Caso ela tenha mais de uma coluna como chave, elas deverão ser relacionadas entre os parênteses e separadas por vírgulas.
CONSTRAINT nome FOREIGN KEY (**nome-coluna-chave-estrangeira**) **REFERENCES nome-tabela-pai** (**nome-campo-pai**) define para o banco de dados as colunas que são chaves estrangeiras, ou seja, os campos que são chaves primárias de outras tabelas. Na opção **REFERENCES** deve ser especificado a tabela na qual a coluna é a chave primária.

ON DELETE especifica os procedimentos que devem ser feitos pelo SGBD quando houver uma exclusão de um registro na tabela pai quando existe um registro correspondente nas tabelas filhas. As opções disponíveis são:
RESTRICT - opção *default*. Esta opção não permite a exclusão na tabela pai de um registro cuja chave primária exista em alguma tabela filha.
CASCADE - esta opção realiza a exclusão em todas as tabelas filhas que possua o valor da chave que será excluída na tabela pai.
SET NULL - esta opção atribui o valor NULO nas colunas das tabelas filhas que contenha o valor da chave que será excluída na tabela pai.

Antes de iniciarmos a criação das tabelas do nosso estudo de caso, vale ressaltar que a ordem de criação dessas tabelas é de suma importância. Isso se deve ao fato das tabelas estarem conectadas através de suas chaves primárias e estrangeiras. Vamos explicar de uma maneira diferente. Sabemos, por exemplo, que a tabela **funcionário** "recebe", como chave estrangeira, a chave primária da tabela **departamento**. Assim, caso tentássemos criar primeiro a tabela **funcionário**, durante a sua declaração diríamos que ela possui um atributo chave estrangeira e que este se conecta com a chave primária da tabela **departamento**. Como a tabela **departamento** ainda não existiria na base de dados, o SGBD acusaria uma mensagem de erro informando que não conhece a tabela **departamento**. Dito isso, iniciaremos a criação das tabelas.

Na Figura 2.3, apresentamos o código SQL que cria a tabela **departamento**. Conforme observamos, a tabela **departamento** possui 3 atributos, sendo o código do departamento (cod_dep) do tipo inteiro e chave primária da tabela.

```
CREATE TABLE departamento
(cod_dep INTEGER NOT NULL,
descr CHAR(30) NOT NULL DEFAULT 'Não Informado',
localiz CHAR(30) NOT NULL,
CONSTRAINT pk_dep PRIMARY KEY(cod_dep));
```

Figura 2.3: Criação da tabela "departamento"
Fonte: Elaborada pelos autores

Observamos também que foi inserido um valor *default* para o atributo descrição (descr). Caso não seja informado um valor para o atributo descrição, o próprio SGBD incluirá o valor "Não informado".

Como não especificamos a cláusula *on delete*, o SGBD não permitirá a exclusão na tabela pai de um registro cuja chave primária exista em alguma tabela filha. A próxima tabela que criaremos será a tabela *funcionario*, mostrada na Figura 2.4.

```
CREATE TABLE funcionario
(cod_func INTEGER NOT NULL,
nome CHAR(50) NOT NULL,
dt_nasc DATE,
cod_dep INT,
CONSTRAINT pk_func PRIMARY KEY(cod_func),
CONSTRAINT fk_func FOREIGN KEY(cod_dep) REFERENCES departamento(cod_dep));
```

Figura 2.4: Criação da tabela "funcionario"
Fonte: Elaborada pelos autores

Observamos que a tabela **funcionario** possui duas restrições (*constraint*). A primeira determina o código do funcionário (cod_func) como a chave primária da tabela e a segunda restrição determina o atributo **cod_dep** como chave estrangeira que veio da tabela **departamento**.

Perceba que, de acordo com a sintaxe de criação das tabelas, não é obrigatório que as chaves primárias e estrangeiras tenham o mesmo nome. Usando como exemplo as tabelas **funcionario** e **departamento**, observe que o atributo **cod_dep** da tabela **funcionario** não precisaria ter o mesmo nome do atributo **cod_dep** da tabela **departamento**. Isso só é possível por que, durante a declaração da chave estrangeira, dizemos explicitamente com qual atributo devemos conectar o atributo **cod_dep** da tabela **funcionario**.

Nas Figuras 2.5 e 2.6 a seguir, seguem as criações das tabelas **funcao** e **projeto**, respectivamente.

```
CREATE TABLE funcao
(cod_funcao INTEGER NOT NULL,
nome CHAR(50) NOT NULL,
sal REAL NOT NULL,
CONSTRAINT pk_funcao PRIMARY KEY(cod_funcao));
```

Figura 2.5: Criação da tabela "funcao"
Fonte: Elaborada pelos autores

```
CREATE TABLE projeto
(cod_proj INTEGER NOT NULL,
nome CHAR(50) NOT NULL,
orcamento REAL NOT NULL,
dt_ini DATE NOT NULL,
dt_prev_term DATE NOT NULL,
CONSTRAINT pk_projeto PRIMARY KEY(cod_proj),
CONSTRAINT verifica_datas CHECK (dt_ini<dt_prev_term));
```

Figura 2.6: Criação da tabela "projeto"
Fonte: Elaborada pelos autores

A cláusula *check* serve para implementarmos restrições de domínio. Durante a criação da tabela **projeto**, inserimos uma restrição que garante que a data de início do projeto (dt_ini) seja menor que a data prevista de término (dt_prev_term). A cláusula *check* também poderia ser usada para comparar um atributo com um valor absoluto e não apenas para comparar um atributo com outro atributo, conforme exemplo anterior.

Por fim, na Figura 2.7, apresentamos a criação da tabela **trabalha**. Esta tabela, obrigatoriamente, deveria ser a última tabela a ser criada no nosso banco de dados. Isso se deve ao fato desta tabela receber, como chaves estrangeiras, atributos oriundos das tabelas **funcionario**, **projeto** e **funcao**.

```
CREATE TABLE trabalha
(cod_func INTEGER NOT NULL,
cod_proj INTEGER NOT NULL,
cod_funcao INTEGER NOT NULL,
dt_ent DATE NOT NULL,
dt_sai DATE,
CONSTRAINT pk_trabalha PRIMARY KEY(cod_func,cod_proj),
CONSTRAINT fk_trabalha1 FOREIGN KEY(cod_func) REFERENCES funcionario(cod_func),
CONSTRAINT fk_trabalha2 FOREIGN KEY(cod_proj) REFERENCES projeto(cod_proj),
CONSTRAINT fk_trabalha3 FOREIGN KEY(cod_funcao) REFERENCES funcao(cod_funcao),
CONSTRAINT checa_datas CHECK (dt_ent<dt_sai));
```

Figura 2.7: Criação da tabela "trabalha"
Fonte: Elaborada pelos autores

Na tabela **trabalha**, inserimos uma restrição chamada ch**eca_datas** para garantir que a data de entrada do funcionário no projeto (dt_ent) seja sempre menor que a sua data de saída (dt_sai).

2.5 O comando *alter table*

Segundo Pazin (2003), o comando *alter table* permite alterar a estrutura de uma tabela acrescentando, alterando, retirando e alterando nomes, formatos das colunas e a integridade referencial definidas em uma determinada tabela. A sintaxe para esse comando é apresentada na Figura 2.8.

```
ALTER TABLE nome-tabela
      DROP nome-coluna
      ADD   nome-coluna tipo-do-dado [NOT NULL]
                                     [NOT NULL WITH DEFAULT]

      MODIFY nome-coluna tipo-do-dado [NULL][NOT NULL]
                                      [NOT NULL WITH DEFAULT]
```

Figura 2.8: Sintaxe do comando *alter table*
Fonte: Elaborada pelos autores

Onde:

nome-tabela representa o nome da tabela que será atualizada.
nome-coluna representa o nome da coluna que será criada.
tipo-do-dado a cláusula que define o tipo e tamanho dos campos definidos para a tabela.
DROP nome-coluna realiza a retirada da coluna especificada na estrutura da tabela.
ADD nome-coluna tipo-do-dado realiza a inclusão da coluna especificada na estrutura da tabela. Na coluna correspondente a este campo nos registros já existentes será preenchido o valor NULL (Nulo). As definições NOT NULL e NOT NULL WITH DEFAULT são semelhantes à do comando CREATE TABLE.
MODIFY nome-coluna tipo-do-dado permite a alteração na característica da coluna especificada.

Apresentaremos exemplos utilizando as cláusulas anteriormente citadas.

2.5.1 Apagando uma coluna de uma tabela

Imagine que você deseja, por alguma razão, apagar a coluna que armazena a data de saída (dt_sai) da tabela **trabalha**. Como faríamos isso? A Figura 2.9 apresenta a solução.

```
ALTER TABLE trabalha
DROP dt_sai;
```

Figura 2.9: Remoção do atributo dt_sai da tabela trabalha
Fonte: Elaborada pelos autores

2.5.2 Adicionando uma coluna em uma tabela

Imagine que, após criarmos a tabela **funcionario** e já termos cadastrados alguns registros, percebemos que nos esquecemos de criar a coluna **telefone** na tabela. Como resolveríamos este problema? A Figura 2.10 apresenta a solução.

```
ALTER TABLE funcionario
ADD telefone CHAR(13) NOT NULL DEFAULT 'Nao Informado';
```

Figura 2.10: Adição do atributo telefone na tabela funcionario
Fonte: Elaborada pelos autores

Perceba que criamos a coluna telefone com um valor *default* 'Não Informado'. O que tentamos fazer utilizando este artifício? Você teria alguma explicação?

Bem, caso a inclusão desta coluna ocorra após alguns funcionários já terem sido cadastrados e caso tivéssemos criado a nova coluna **telefone** aceitando valores nulos (NULL), não teríamos nenhum problema, pois seria atribuído valor nulo aos telefones de todos os funcionários que já estivessem cadastrados na tabela. Porém, como queremos criar a coluna **telefone** não aceitando valores nulos (NOT NULL), fomos obrigados a criar este valor *default* 'Não Informado' para ser inserido na coluna **telefone** de todos os funcionários que já se encontravam cadastrados na tabela. Fomos claros na explicação?

Vamos praticar um pouco do que acabamos de estudar? Tenho certeza de que você é capaz! Remova, da tabela **projeto**, a coluna (atributo) referente à data prevista de término do projeto. Feito isso adicione novamente a mesma coluna. Poste no ambiente virtual de ensino-aprendizagem os códigos referentes aos dois procedimentos.

2.5.3 Modificando uma coluna de uma tabela

E se precisássemos mudar as características de uma coluna da tabela após a sua criação? Como exemplo, imagine que desejamos aceitar valores nulos no atributo salário (sal) da tabela **funcao**. Além disso, desejamos também alterar o domínio do atributo, passado de real para "*integer*". Para isso, observe o código da Figura 2.11 a seguir.

```
ALTER TABLE funcao
MODIFY sal INTEGER NULL;
```

Figura 2.11: Alteração do atributo sal da tabela funcao
Fonte: Elaborada pelos autores

2.6 O comando *drop table*

O comando *drop table* serve para destruirmos uma tabela. Se, por exemplo, precisássemos destruir a tabela **trabalha**, usaríamos o comando da Figura 2.12.

```
DROP TABLE trabalha;
```

Figura 2.12: Remoção da tabela trabalha
Fonte: Elaborada pelos autores

Perceba que a sintaxe do comando é bastante simples. Basta escrevermos, após *drop table*, o nome da tabela que desejamos destruir. Lembre-se que algumas tabelas podem ser dependentes da tabela que desejamos destruir. Por exemplo, caso fôssemos destruir a tabela **departamento**, teríamos que lembrar que a tabela **funcionario** é dependente de departamento, pois ela recebe o atributo **cod_dep** como chave estrangeira. Para resolvermos este problema, teríamos que destruir a referência de chave estrangeira da tabela **funcionario**, ou mesmo, chegarmos ao ponto de destruirmos primeiro, a tabela **funcionario**, para só depois eliminarmos a tabela **departamento**. Caso optássemos pela segunda solução, teríamos que lembrar que a tabela **trabalha** também é dependente de funcionário e o mesmo procedimento deveria ser tomado.

2.7 Consultas simples

Consultar dados em um banco de dados, normalmente, é a operação mais utilizada pelos usuários. Para isso, precisamos fazer uso da instrução *select*. Ela é considerada por muitos, como a instrução mais poderosa da linguagem SQL. Nesta seção, apresentaremos a sua estrutura básica. Nas páginas seguintes, apresentaremos formas avançadas de utilização dessa instrução. A sintaxe básica da instrução *select* é apresentada na Figura 2.13.

```
SELECT lista_atributos FROM nome-tabela [AS APELIDO] [,nome-tabela]
WHERE condição
```

Figura 2.13: Sintaxe da instrução *select*
Fonte: Costa (2006)

Onde:

lista_atributos representa, como o nome sugere, a lista dos atributos que se deseja apresentar no resultado da consulta.
nome-tabela representa o nome da(s) tabela(s) que contem as colunas que serão selecionadas ou que serão utilizadas para a execução da consulta.
APELIDO representa os nomes que serão usados como nomes de tabelas em vez dos nomes originais. A vantagem desse recurso é que, em casos de consultas muito grandes, com a utilização de apelidos, digitamos menos.
condição representa a condição para a seleção dos registros. Esta seleção poderá resultar em um ou vários registros.

Para melhor entendermos esta instrução, apresentaremos alguns exemplos:

I. Obter todas as informações de todos os funcionários. A solução é apresentada na Figura 2.14.

```
SELECT * FROM funcionario
```

Figura 2.14: Seleciona os dados de todos os funcionários
Fonte: Elaborada pelos autores

Neste exemplo, percebemos que não fizemos uso da cláusula *where*. Isso se deve ao fato da questão não colocar uma condição de busca. Assim, concluímos que o *where* só é necessário em consultas que possuem uma condição para a seleção.

II. Obter o nome e a data de nascimento do funcionário de código 2. A solução é apresentada na Figura 2.15.

```
SELECT nome, dt_nasc FROM funcionario WHERE cod_func=2;
```

Figura 2.15: Código SQL
Fonte: Elaborada pelos autores

Nesta consulta, como a questão apresentava uma condição para a seleção (código do funcionário igual a 2), utilizamos a cláusula *where*.

2.7.1 Subconsultas

Realizar subconsultas é uma forma de combinar mais de uma consulta (*select*) obtendo apenas um resultado. A seguir vamos apresentar exemplos como forma de explicar o assunto.

Imagine que precisamos obter o nome de todos os funcionários que estão lotados no departamento de contabilidade. Perceba que o nome do departamento está na tabela **departamento**, enquanto que o nome do funcionário está na tabela **funcionario**. Assim, precisaríamos utilizar as duas tabelas para obtermos o nosso resultado. A instrução que atende à nossa necessidade encontra-se na Figura 2.16.

```
Select nome from funcionario where cod_dep = (select cod_dep from departamento
where descr='contabilidade');
```

Figura 2.16: Código SQL
Fonte: Elaborada pelos autores

Observe que utilizamos o código do departamento como "ponte" para "pularmos" da tabela **funcionario** para a tabela **departamento**. Isso aconteceu, pois a chave primária de departamento (cod_dep) é a chave estrangeira da tabela **funcionario**.

2.8 Manipulando dados de uma tabela (*insert, update, delete*)

Como dissemos anteriormente, na linguagem SQL existem instruções para definição de dados (DDL), e instruções para manipulação de dados (DML). Conhecemos, até agora, alguns comandos DDL e, nas próximas páginas, conheceremos instruções de manipulação. São elas; *insert into, update* e *delete*.

2.8.1 Inserindo dados em uma tabela

Para incluirmos dados em uma tabela, utilizamos a instrução *insert into*. Este comando permite inserir um ou vários registros em uma tabela do banco de dados. A sintaxe é apresentada na Figura 2.17.

```
INSERT INTO nome-tabela [(nome-coluna, ...)]
      VALUES (relação dos valores a serem incluídos)
```

Figura 2.17: Sintaxe da instrução *insert into*
Fonte: Costa (2006)

Onde:

nome-tabela representa o nome da tabela onde será incluída o registro.
nome-coluna representa o nome da(s) coluna(s) que terão conteúdo no momento da operação de inclusão. Obs.: esta relação é opcional.
relação dos valores representa os valores a serem incluídos na tabela.

Existem três observações importantes para fazermos sobre este comando. Vamos comentá-las a partir de exemplos.

Vejamos o primeiro exemplo na Figura 2.18.

```
INSERT INTO departamento VALUES (1, 'Análise', 'Sala B2-30');
```

Figura 2.18: Inserção de um registro na tabela departamento
Fonte: Elaborada pelos autores

No exemplo anterior, cadastramos o departamento 1, chamado Análise e que se localiza na sala B2-30. Perceba que, após o nome da tabela **departamento**, não colocamos a lista com o nome das colunas que seriam povoadas. Isso é possível, porém temos que, obrigatoriamente, inserirmos as informações das colunas da tabela na mesma ordem em que elas foram criadas. No nosso caso, primeiro o código do departamento, depois a descrição e, por fim, a sua localização. Vejamos um segundo exemplo na Figura 2.19.

```
INSERT INTO funcionario(cod_func, cod_dep, nome) VALUES(1, 1, 'Maria');
```

Figura 2.19: Inserção de um registro na tabela funcionario
Fonte: Elaborada pelos autores

Neste segundo exemplo, cadastramos a funcionária de código 1 chamada Maria e que trabalha no departamento de código 1. Perceba que, após o nome da tabela **funcionario**, colocamos a lista com os nomes das colunas que deveriam ser preenchidas na tabela. Perceba também que a ordem não é a mesma utilizada na criação da tabela. E mais, não colocamos, na lista, o atributo referente à data de nascimento da funcionária. Então você poderia estar se perguntando: O que acontecerá com o atributo data de nascimento quando executarmos esta instrução? A explicação para sua pergunta é que o próprio SGBD atribuirá valor nulo para a data de nascimento da Maria. Isso só é possível porque, quando criamos a tabela **funcionario**, dissemos que o atributo data de nascimento aceitaria valores nulos. Caso você ainda tenha alguma dúvida, volte algumas páginas e veja o código de criação da referida tabela.

Agora, antes de comentarmos o nosso terceiro exemplo, imagine que possuímos, no banco de dados, a tabela **pessoa** com a seguinte estrutura, apresentada no Quadro 2.1 a seguir.

Quadro 2.1: Tabela "pessoa"

Codigo	Apelido	Data_nasc	Cod_setor	Nome_mae
100	Joãozinho	01/01/1980	1	Francisca
200	Maricota	02/02/1979	1	Raimunda
300	Franzé	03/03/1978	1	Joanete

Fonte: Elaborado pelos autores

Imagine que precisamos cadastrar, na tabela **funcionario**, todos os registros da tabela **pessoa**. Como faríamos isso de maneira rápida? Como forma de agilizarmos o nosso trabalho, poderíamos executar o seguinte comando apresentado na Figura 2.20.

```
INSERT INTO funcionario(cod_func, nome, dt_nasc, cod_dep)
SELECT codigo, apelido, data_nasc, cod_setor FROM pessoa;
```

Figura 2.20: Inserção de um bloco de registros na tabela funcionario
Fonte: Elaborada pelos autores

Perceba que conseguimos, através de uma única instrução, inserirmos vários registros na tabela **funcionario**. Isso só foi possível por que a instrução *insert into* permite que cadastremos o resultado de um *select*, desde que este *select* produza uma tabela compatível com a tabela na qual estamos inserindo.

2.8.2 Alterando dados de uma tabela

Para alterarmos uma informação contida numa tabela do banco de dados, utilizamos o comando *update*. Ele atualiza dados de um registro ou de um conjunto de registro.

A sua sintaxe é apresentada na Figura 2.21.

```
UPDATE nome-tabela
     SET <nome-coluna = <novo conteúdo para o campo>
        [nome-coluna = <novo conteúdo para o campo>]
     WHERE condição
```

Figura 2.21: Sintaxe da instrução *update*
Fonte: Costa (2006)

Onde:

nome-tabela representa o nome da tabela cujo conteúdo será alterado.
nome-coluna representa o nome da(s) coluna(s) terão seus conteúdos alterados com o novo valor especificado.
condição representa a condição para a seleção dos registros que serão atualizados. Esta seleção poderá resultar em um ou vários registros. Neste caso a alteração irá ocorrer em todos os registros selecionados.

Vejamos os exemplos apresentados nas Figuras 2.22 e 2.23 a seguir.

```
UPDATE projeto
SET orcamento=1000
WHERE cod_proj=1 OR cod_proj=5;
```

Figura 2.22: Alteração do orçamento dos projetos de código 1 ou 5
Fonte: Elaborada pelos autores

No exemplo acima, estamos alterando para 1000 os orçamentos dos projetos que possuem código igual a 1 ou igual a 5.

```
UPDATE projeto
SET orcamento=2000;
```

Figura 2.23: Alteração de todos os orçamentos de projetos
Fonte: Elaborada pelos autores

Já neste último exemplo, alteramos para 2000 os orçamentos de TODOS os projetos. Isso aconteceu por que não utilizamos a cláusula *where* para delimitar as linhas que seriam selecionadas para serem alteradas.

2.8.3 Excluindo dados de uma tabela

O comando *delete* é utilizado para excluir linhas de uma tabela. Na Figura 2.24, apresentamos a sua sintaxe.

```
DELETE FROM nome-tabela WHERE condição
```

Figura 2.24: Sintaxe do comando *delete*
Fonte: Costa (2006)

Caso desejássemos deletar os projetos que custam mais de 2000, usaríamos o comando da Figura 2.25.

```
DELETE FROM projeto
WHERE orcamento>2000;
```

Figura 2.25: Remoção dos projetos que custam mais de 2000
Fonte: Elaborada pelos autores

Quando vamos deletar qualquer registro, devemos nos lembrar da **Integridade Referencial**. Este conceito determina que um registro não pode fazer referência a um outro registro do banco de dados que não existe. Por exemplo, nós não poderíamos simplesmente deletar projetos, caso estes ainda estivessem sendo referenciados pela tabela **trabalha**.

2.9 Funções agregadas

Muitas vezes, precisamos de informações que resultado de alguma operação aritmética ou de conjunto sobre os dados contidos nas tabelas de um banco de dados. Para isso, utilizamos as funções agregadas. A seguir apresentaremos algumas delas.

2.9.1 Função *count( )*

A função *count*, como o próprio nome sugere, conta a quantidade de linhas de uma tabela que satisfazem uma determinada condição. A seguir veremos alguns exemplos.

Caso precisássemos saber quantas projetos existem cadastrados na tabela Projeto? Observe a Figura 2.26.

```
SELECT COUNT(cod_proj) FROM projeto;
```

Figura 2.26: Contagem da quantidade de códigos de projetos
Fonte: Elaborada pelos autores

Perceba que dentro dos parênteses da função *count* colocamos o atributo que será utilizado para a contagem. E se precisássemos contar a quantidade de projetos que custam mais de 2000? Observe a Figura 2.27.

```
SELECT COUNT(cod_proj) FROM projeto
WHERE orcamento>2000;
```

Figura 2.27: Contagem de código de projetos que custam mais de 2000
Fonte: Elaborada pelos autores

Perceba que, neste último exemplo, inserimos a cláusula WHERE. Isso aconteceu porque precisávamos contar apenas as linhas da tabela que atendiam à condição especificada.

2.9.2 Função *avg( )*

A função *avg* é responsável por extrair a média aritmética dos valores de uma coluna.

Por exemplo, se precisássemos calcular a média dos orçamentos de todos os projetos, executaríamos o seguinte comando apresentado na Figura 2.28.

```
SELECT AVG(orcamento) FROM projeto;
```

Figura 2.28: Média dos orçamentos dos projetos
Fonte: Elaborada pelos autores

2.9.3 Função *sum( )*

A função *sum* é responsável por realizar a soma dos valores de uma coluna.

Observe o exemplo da Figura 2.29:

```
SELECT SUM(orcamento) FROM projeto WHERE cod_proj<10;
```

Figura 2.29: Instrução SQL - soma
Fonte: Elaborada pelos autores

No exemplo acima, o SGBD realizará a soma dos orçamentos dos projetos cujo código seja menor que 10.

2.9.4 Função *min( )*

A função *min* obtém o valor mínimo dentre os elementos de uma coluna.

O exemplo da Figura 2.30 obtém o menor código de funcionário dentre aqueles que nasceram no ano de 1980.

```
SELECT MIN(cod_func) FROM funcionario
WHERE dt_nasc>='1980-01-01' AND dt_nasc<='1980-12-31';
```

Figura 2.30: Instrução SQL - mínimo
Fonte: Elaborada pelos autores

2.9.5 Função *max( )*

A função *max* obtém o maior valor dentre os elementos de uma coluna. O exemplo da Figura 2.31 obtém o maior código de funcionário dentre aqueles que nasceram no ano de 1980.

```
SELECT MAX(cod_func) FROM funcionario
WHERE dt_nasc>='1980-01-01' AND dt_nasc<='1980-12-31';
```

Figura 2.31: Instrução SQL - máximo
Fonte: Elaborada pelos autores

2.10 A cláusula *group by*

Os dados resultantes de uma seleção podem ser agrupados de acordo com um critério específico. Este procedimento é realizado usando a cláusula *group by*.

Para melhor entendermos como funciona o *group by*, analisaremos o exemplo a seguir.

Desejamos obter, para cada código de projeto, a quantidade de funcionários que trabalharam nele. Lembre-se que, para sabermos qual funcionário trabalha em qual projeto, teremos que observar a tabela **trabalha**, pois é nela que acontecem as associações entre funcionários e projetos. Para respondermos a pergunta anterior, vamos considerar as seguintes informações na tabela **trabalha** apresentadas no Quadro 2.2.

Quadro 2.2: Tabela "trabalha"

Cod_func	Cod_proj	Cod_funcao	Dt_ent	Dt_sai
1	1	1	2010-02-02	2010-03-03
2	1	2	2010-02-02	2010-03-03
1	2	1	2010-04-04	2010-05-05
4	2	2	2010-04-04	2010-05-05
3	1	3	2010-02-02	2010-03-03

Fonte: Elaborado pelos autores

Perceba que o funcionário 1 trabalhou no projeto 1 e no projeto 2. Perceba também que 3 funcionários trabalharam no projeto 1 e apenas 2 funcionários trabalharam no projeto 2. No projeto 1, trabalharam os funcionários de código 1, 2 e 3. Já no projeto 2, trabalharam os funcionários de código 1 e 4.

Bem, agora voltando para a questão inicial, como escreveríamos um comando SQL que mostra, para cada código de projeto, a quantidade de funcionários que trabalharam nele? Na realidade, o que estamos buscando está representado na tabela do Quadro 2.3.

Quadro 2.3: Número de funcionários por projeto

Cod_proj	Quantidade_funcionários
1	3
2	2

Fonte: Elaborado pelos autores

A solução para o nosso problema é a apresentada na Figura 2.32 a seguir.

```
SELECT cod_proj, COUNT(cod_proj) 'Quantidade_funicionários'
FROM trabalha GROUP BY cod_proj;
```

Figura 2.32: Instrução SQL - contagem
Fonte: elaborada pelos autores

Observe que agrupamos os códigos dos projetos iguais, ou seja, foram criados dois grupos: um grupo para os projetos de código 1 e outro grupo para os projetos de código 2. Se existissem, na tabela **trabalha**, outros códigos de projetos diferentes, outros grupos também seriam criados. Além de criar os grupos, através da função agregada *count*( ), foi feita a contagem de elementos de cada grupo. Aqui, vale chamar a atenção no seguinte: toda vez que utilizamos uma função agregada junto com o GROUP BY, esta função será operada sobre cada um dos grupos gerados pela cláusula GROUP BY.

Outra observação que destacamos é o novo nome que demos para a coluna *count*(). Logo após a função agregada, entre aspas simples, escrevemos o novo nome que desejamos que possua a coluna *count*( ).

2.11 Junções (*join*)

Quando precisamos realizar consultas que envolvam mais de uma tabela, uma das soluções seria a utilização de junções. As junções permitem que acessemos mais de uma tabela utilizando apenas um *select*.

Na utilização de junções, normalmente, deve existir a chave primaria de uma tabela fazendo relação com a chave estrangeira da outra tabela que compõe a junção. Esta será a condição de ligação entre as tabelas.

Existem vários tipos de junções, cada uma delas variando a forma que cada tabela se relaciona com as demais.

Antes de iniciarmos o estudo dos diferentes tipos de junções, consideremos as tabelas **funcionario** e **departamento**. Elas servirão de base para os tópicos seguintes. A seguir, no Quadro 2.4, segue a tabela **funcionario**.

Quadro 2.4: Tabela "funcionario"

Cod_func	Nome	Dt_nasc	Cod_dep
1	João	1980-01-02	1
2	José	1981-02-03	2
3	Maria	1982-05-04	1
4	Antônio	1983-07-06	3

Fonte: Elaborado pelos autores

A seguir, no Quadro 2.5, segue a tabela **departamento**.

Quadro 2.5: Tabela "departamento"

Cod_dep	Descr	Localiz
1	Desenvolvimento	Sala C3-10
2	Análise	Sala B2-30
3	Testes	Sala C1-10
4	Contabilidade	Sala A1-20

Fonte: Elaborado pelos autores

2.11.1 Junção interna (*inner Join*)

A junção interna entre tabelas é a modalidade de junção que faz com que somente participem da relação resultante as linhas das tabelas de origem que atenderem à cláusula de junção. Por exemplo, caso quiséssemos saber o nome de todos os funcionários com seus respectivos nomes de departamentos, teríamos a instrução da Figura 2.33.

```
SELECT nome, descr FROM funcionario f INNER JOIN departamento d
ON f.cod_dep=d.cod_dep;
```

Figura 2.33: Instrução SQL – *inner join*
Fonte: Elaborada pelos autores

O comando anterior apresentaria como resultado, a partir das tabelas apresentadas nos Quadros 2.4 e 2.5, a relação resultante do Quadro 2.6.

Quadro 2.6: Relação resultante

Nome	Descr
João	Desenvolvimento
José	Análise
Maria	Testes
Antônio	Contabilidade

Fonte: Elaborado pelos autores

Observe, no comando apresentado, que selecionamos o nome do funcionário a partir da tabela **funcionario** e a descrição do departamento a partir da tabela **departamento**. Isso foi possível, pois realizamos uma junção interna, onde se juntou todas as colunas da tabela **funcionario** com todas as colunas da tabela **departamento**. Os registros que foram selecionados desta junção foram somente aqueles que satisfizeram a condição expressa após a cláusula ON.

Perceba também que, após o nome de cada tabela inserimos um apelido para elas. Demos, para a tabela **funcionario**, o apelido **f**, já para a tabela **departamento**, demos o apelido **d**, ou seja, poderemos substituir a palavra **funcionário** por **f** e a palavra departamento por **d**.

Logo após a cláusula ON, inserimos a condição para a junção. Observe que a condição, normalmente, acontece entre as chaves primárias e chaves estrangeiras das tabelas que, no caso específico, é o código do departamento. Antes do nome de cada atributo, colocamos o nome da tabela da qual ele se origina, ou seja, **f.cod_dep** quer dizer que é o atributo código do departamento que existe na tabela **funcionario**, já **d.cod_dep** quer dizer que é o código do departamento da tabela **departamento**.

2.11.2 Junções externas (*outer join*)

Na junção externa, os registros que participam do resultado da junção não obedecem obrigatoriamente à condição de junção, ou seja, a não inexistência de valores correspondentes não limita a participação de linhas no resultado de uma consulta.

Existem tipos diferentes de junção externa. Apresentaremos alguns deles a seguir.

2.11.3 Junção externa à esquerda (*left outer join*)

Suponha que desejemos uma listagem com os nomes de todos os departamentos cadastrados no nosso banco de dados e, para aqueles que possuam funcionários lotados nele, apresente os respectivos nomes. Para isso, teremos que utilizar a junção externa à esquerda. A instrução para resolver esta questão é apresentada na Figura 2.34.

```
SELECT descr, nome FROM departamento d LEFT OUTER JOIN funcionario f
ON f.cod_dep=d.cod_dep;
```

Figura 2.34: Instrução SQL – *left outer join*
Fonte: Elaborada pelos autores

A instrução anterior produzirá o resultado do Quadro 2.7, a partir das tabelas **funcionario** e **departamento**.

Quadro 2.6: Relação resultante

Descr	Nome
Desenvolvimento	João
Desenvolvimento	Maria
Análise	José
Testes	Antônio
Contabilidade	

Fonte: Elaborado pelos autores

Vamos repetir a mesma atividade que fizemos ao final da primeira aula deste caderno? Assista a vídeo-aula disponível em http://cefetpi.nucleoead.net/etapi/ referente aos conteúdos até agora estudados e, em poucas palavras, escreva um texto narrando o que você entendeu. Não se esqueça de utilizar exemplos que apresentem a utilização das instruções SQL até então estudadas. Poste o texto na nossa plataforma AVEA.

Perceba que, como a tabela **departamento** foi colocada à esquerda da junção, foi apresentada a listagem completa de todas as descrições de departamento e, quando havia alguma associação de uma descrição com um funcionário, este era apresentado. Observe que ninguém está lotado no departamento de contabilidade, logo ninguém aparece associado a este departamento na tabela anterior.

2.11.4 Junção externa à direita (*right outer join*)

A junção externa à direita é muito parecida com a junção externa à esquerda. A única diferença está no fato de que a tabela da qual todas as linhas constarão no resultado está posicionada à direita do termo *right outer join* no comando.

Assim, para realizarmos a mesma consulta do item anterior, porém, utilizando a junção externa à direita, teríamos que executar a instrução da Figura 2.35.

```
SELECT descr, nome FROM  funcionario f RIGHT OUTER JOIN departamento d
ON f.cod_dep=d.cod_dep;
```

Figura 2.35: Instrução SQL – *right outer join*
Fonte: Elaborada pelos autores

Resumo

Nesta segunda aula, foi definida a linguagem SQL e seus tipos básicos. Também foram apresentados os principais comandos da linguagem SQL, tais como: *cre ate database, create table, alter table, drop table*, e instruções para manipulação de dados em tabelas e consultas. Todos os conteúdos foram apresentados utilizando um estudo de caso elaborado na aula 1.

Atividades de aprendizagem

1. A partir da linguagem SQL, construa uma base de dados chamada Clínica e, dentro dela, crie as tabelas da quarta questão da aula anterior, com suas respectivas chaves primárias e chaves estrangeiras.

2. A partir do banco de dados da questão anterior e utilizando a linguagem SQL, responda as questões:

   a) Altere a tabela médico, adicionando a coluna "nome_cônjuge".

   b) Insira, pelo menos, dois registros em cada uma das tabelas.

   c) Delete um registro da tabela especialidade. Obs.: mantenha a integridade referencial.

   d) Obtenha o nome do paciente mais velho.

   e) Para cada CRM de médico, obtenha a quantidade de consultas relacionadas a ele.

   f) Obter o nome do(s) médico(s) que atendeu o paciente de nome 'João'.

   g) Para cada nome de médico, obtenha a quantidade de consultas relacionadas a ele.

Poste os códigos SQLs das duas atividades no AVEA.

Aula 3 – Procedimento e função

Objetivos

Definir procedimento armazenado (*store procedure*).

Conhecer a sintaxe de construção de procedimento.

Implementar e executar procedimentos.

Definir função (*function*).

Conhecer a sintaxe de construção de função.

Implementar e executar função.

3.1 Introdução à programação em SQL

A linguagem SQL foi estruturada como uma linguagem de programação comum, assumindo estrutura de controle, decisão, repetição, de forma que possa executar funções (*functions*), procedimentos (*store procedures*) e gatilhos (*triggers*) de maneira eficiente e efetiva. Este tipo de programação com SQL diretamente no SGBD trás as seguintes vantagens:

- Reduz a diferença entre o SQL e a linguagem de programação;
- Por ser armazenada no SGBD permite que possa ser invocado por diferentes aplicações evitando assim a duplicação de código;
- Reduz custos de comunicação, por ser executada no SGBD;
- Pode ser utilizado diretamente na elaboração de *functions, store procedures* e *triggers*;

O MySQL utiliza o SQL/PSM (SQL/*Persistent Stored Modules*) que é uma extensão ao SQL. Ele define para a escrita de procedimentos e funções em SQL que juntamente com a utilização de estruturas de controle aumentam consideravelmente o poder expressivo do SQL.

Para conhecer melhor o SQL/PSM que é bastante utilizado pelo MySQL iremos acessar o *link*: http://paginas.ulusofona.pt/p1662/BasedeDados/Cap_13_SQL-PSM.pdf

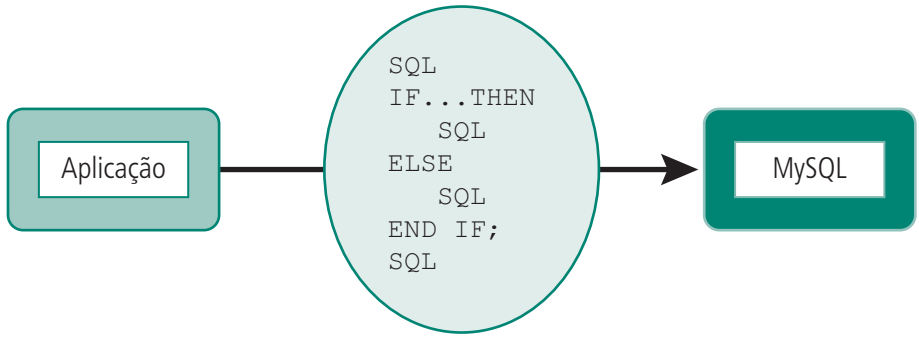

Figura 3.1: Bloco de comandos
Fonte: Manual de Referência do MySQL, 1997

Para se programar em SQL/PSM, ou seja, criar programas em SQL, se faz necessário familiaridade com as declarações de variáveis, cursores, atribuições de valores, operadores lógicos, condições, *loops*, rotinas de controle, comentários. Acumulado esse conhecimento, pode-se criar blocos de comandos para preparar funções (*function*), procedimentos (*store procedure*) e gatilhos (*triggers*), que serão compilados, executados e armazenados diretamente no SGBD. Fazendo com que dessa forma as regras de negócio sejam disponibilizadas para todas as aplicações que acessam o SGBD.

Utilizaremos o MySQL como nosso sistema de gerenciamento de banco de dados (SGBD). É atualmente um dos bancos de dados mais populares, com mais de 10 milhões de instalações pelo mundo.

Iniciaremos nosso estudo em SQL/PSM no MySQL, desde os blocos de comando até a elaboração das funções (*function*), procedimentos (*store procedure*), gatilhos (*triggers*) e ainda controle de acesso de usuário.

Vamos aprender a sintaxe, dos comandos para que possamos programar no MySQL.

3.1.1 Comando: *begin...end*

As palavras-reservas *begin* e *end* atuam como delimitadoras de um bloco de comandos, na criação de programas SQL. Os blocos de comandos são programas compostos por uma ou mais instruções escritas em SQL. Essas mesmas instruções *begin... end* também podem aparecer aninhadas. Temos a estrutura de um bloco de comandos em SQL, a seguir:

```
BEGIN
 [DECLARAÇÃO DE VARIÁVEIS];

 [DECLARAÇÃO DE CURSORES];

 COMANDOS SQL;

 COMANDOS SQL;

END
```

3.1.2 Comando: *declare*

Para que se possa usar uma variável em um programa no MySQL, é necessário fazer a declaração de variável antes. A declaração de variáveis simplesmente informa ao MySQL quais são os nomes dados as variáveis e quais são os tipos usados. No MySQL o nome da variável consiste, basicamente, de caracteres alfanuméricos, podendo ainda ser utilizados os caracteres '_', '**$**' e o '**.**'.

O comando *declare*, serve para fazer a declaração de variáveis locais, condições, cursores e *handlers*.

Introduzindo o conceito de cursor até então desconhecidos nas linguagens comuns de programação, o MySQL utiliza esse termo para armazenar resultados de algum tipo de processamento realizado no SGBD. No exemplo a seguir temos a declaração de dois cursores, o cur1 e o cur2, observemos que os dois acessam colunas e tabelas diferentes, portanto geram resultados diferentes.

Além de podermos trabalhar no MySQL com funções, procedimentos e gatilhos temos ainda possibilidade de tratarmos das *views* (visões) como está descrito no *link*: http://dev.mysql.com/doc/query-browser/pt/mysql-query-browser-creating-views.html

Quando necessário o cursor está presente em funções (*function*), procedimentos (*store procedure*) e gatilhos (*triggers*).

Temos também o termo novo *handler*. Um *handler* funciona como um artifício que existem em torno de funções de acesso ao banco de dados MySQL. Ele procura estabelecer conexões persistentes ou não persistentes com o MySQL, executar consultas SQL, recuperar o número de linhas entre um conjunto de resultados e além de obter o número de linhas afetadas por um *insert, update* ou *delete* numa consulta no MySQL. E ainda recupera mensagens de erros associadas à última consulta no MySQL que não chegou a ser finalizada devido a uma falha interna qualquer.

O comando *declare* segue a precedência, assumida pelo MySQL, que determinar que os cursores devem ser declarados antes dos handlers, e as variáveis e condições, antes dos cursores.

Sintaxe do comando *declare* para declarar variável:

DECLARE <nome da variável> <tipo de dados>;

Sintaxe do comando *declare* para declarar condição:

DECLARE <nome da condição> **CONDITION FOR** <valor da condição>;

Sintaxe do comando declare para declarar cursor:

DECLARE <nome do cursor> **CURSOR FOR** <comando **SELECT**>;

Sintaxe do comando *declare* para declarar *handler*:

DECLARE <tipo de handler> **HANDLER FOR** <valor da condição>;

<tipo de handler> pode ser: **CONTINUE**, **EXIT** ou **UNDO**.

Exemplificação do comando declare, a seguir:

```
BEGIN
 DECLARE a CHAR(16);
 DECLARE b, c INT;
 DECLARE processa CONDITION FOR FOUND SET done=1;
 DECLARE cur1 CURSOR FOR SELECT id, data FROM test.t1;
 DECLARE cur2 CURSOR FOR SELECT i from test.t2;
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;

END;
```

Figura 3.2: Comando *declare*
Fonte: Manual de Referência do MySQL, 1997

3.1.3 Comando: *set*

Uma vez que já se tenha declarado uma variável no MySQL, deve-se atribuir a mesma algum valor. A operação de atribuição é uma operação muito simples, consiste de atribuir um valor de uma expressão a uma variável, utilizando para isso o comando SET.

O símbolo de atribuição é o **=** (igual) que no MySQL, também pode ser usado como o operador que representa a igualdade de valores.

As variáveis não precisam ser inicializadas. Elas contém o valor NULL por padrão e podem armazenar valores numéricos (inteiro ou real), *string* (sequência de caracteres) ou data e hora.

Sintaxe do comando SET:

```
SET <nome da variável> = <valor a ser atribuído>;
```

```
BEGIN
  DECLARE a CHAR(16);
  DECLARE b, c INT;

  SET a = 'EAD-POLÓ';
  SET b=10;
  SET c =76;
  .....
END
```

Figura 3.3: Comando *set*
Fonte: Manual de Referência do MySQL, 1997

3.1.4 Comando: *open, fetch, close*

Já vimos como declarar um cursor, portanto, vamos agora aprender a utilizá-lo.

O curso age como um mecanismo para manipulação de linhas de uma tabela do MySQL, muitas vezes discriminadas linha por linha. E atuam ainda como ponteiros, uma vez que apontam para a(s) linha(s) do resultado dado pela consulta solicitada no MySQL. Após a declaração do cursor, ele deve ser inicializado através do comando open.

Sintaxe do comando *open*:

```
OPEN <nome do cursor>;
```

Posteriormente, a execução do comando *open*, o MySQL está pronto para manipular o resultado dos comandos SQL. Então, o comando *fetch* é executado para que o ponteiro seja posicionado numa linha e as informações atribuídas apontadas pra um conjunto de variáveis (o número de coluna do resultado apontado pelo cursor deve ser igual ao número de variáveis). Portanto, terminado a execução do *fetch*, já se pode manipular as variáveis que receberam o valor do cursor.

O comando *fetch* é usualmente encontrado nos comandos de iteração, como o *repeat* e o *while*, que serão vistos em seções posteriores.

Sintaxe do comando *fetch*:

```
FETCH <nome do cursor> INTO <nome(s) da(s) variável(s);
```

E para finalizar o uso do cursor, deve-se fechar o mesmo, através do comando *close* A ação de não fechar o cursor pode causar problemas graves no MySQL.

Sintaxe do comando *close*:

```
CLOSE <nome do cursor>;
```

Os comandos *open, fetch* e *close* demonstrados:

```
BEGIN
   DECLARE x1 CHAR(16);
   DECLARE y1, z1 INT;
   DECLARE CURSOR cursor1 CURSOR FOR SELECT rg, data1, FROM tabela1.dt1;
   DECLARE CURSOR cursor2 CURSOR FOR SELECT rg, data1, FROM tabela2.dt2;

   OPEN cursor1;
   OPEN cursor2;

   REPEAT

     FETCH cursor1 INTO x1, y1;
     FETCH cursor2 INTO z1;
     ..............................
   END REPEAT;
   .....
   CLOSE cursor1;
   CLOSE cursor2;

  END
```

Figura 3.4: Comandos *open, fetch* e *close*
Fonte: Manual de Referência do MySQL, 1997

3.1.5 Comando: *select...into*

Esse comando é usado para armazenar, no MySQL, o resultado de uma consulta em uma variável. O resultado da consulta deve ter sempre como retorno somente uma linha, caso o resultado tenha mais de uma linha, deve ter o mesmo número de variáveis para receber esses valores.

Se a consulta tiver mais de uma linha como resultado, e não existir variável suficiente receber os valores da consulta, ocorrerá o erro 1172, e aparecerá no MySQL, a seguinte mensagem "*Result consisted of more than one row*", caso a consulta retorne nenhum resultado, ocorrerá o erro 1329, e aparecerá no MySQL a mensagem "*No data*".

Sintaxe do comando *select...into*:

```
SELECT <nome da coluna1, coluna2,..N coluna> INTO <nome da variável1, variavél2,...N variável> FROM <nome da tabela>;
```

O comando *select...into* está demonstrado:

```
SELECT rg, datal INTO xl, yl FROM tabelal.tdl;
```

Figura 3.5: Comando *select...into*
Fonte: Manual de Referência do MySQL, 1997

3.1.6 Comando: *if*

A estrutura de decisão permite executar um entre dois ou mais blocos de instruções. No MySQL, temos a estrutura de decisão *if*, ele testa se uma condição é verdadeira ou falsa, se for verdadeira executa um conjunto de comandos.

Sintaxe do comando *if*:

```
IF <condição> THEN

<comandos SQL>;

<comandos SQL>;

[ELSE IF <condição> THEN <comandos SQL>;

<comandos SQL>;]...

[ELSE <comandos SQL>;<comandos SQL>;

END IF;
```

```
BEGIN
   DECLARE x1 CHAR(16);
   DECLARE y1, z1 INT;
   DECLARE CURSOR cursor1 CURSOR FOR SELECT rg, data1, FROM tabela1.dt1;
   DECLARE CURSOR cursor2 CURSOR FOR SELECT rg, data1, FROM tabela2.dt2;

   OPEN cursor1;
   OPEN cursor2;

   REPEAT

     FETCH cursor1 INTO x1, y1;
     FETCH cursor2 INTO z1;
     ..............................
   END REPEAT;
   .....
   CLOSE cursor1;
   CLOSE cursor2;

  END
```

Figura 3.6: Comando *if*
Fonte: Manual de Referência do MySQL, 1997

3.1.7 Comando: *case...when*

A estrutura *case...when* é uma estrutura de decisão que permite a execução de um conjunto de instruções SQL, conforme a pesquisa e posterior encontro de um determinado valor.

Sintaxe do comando *case...when*:

```
CASE <valor procurado>
 WHEN <valor da pesquisa1> THEN <comandos SQL>;
 WHEN <valor da pesquisa2> THEN <comandos SQL>;]...
 [ELSE <comandos SQL>;]
END CASE;
```

A Sintaxe do comando *case...when*, também pode ser a seguinte:

```
CASE
 WHEN <valor da pesquisa1> THEN <comandos SQL>;
 WHEN <valor da pesquisa2> THEN <comandos SQL>;]...
 [ELSE <comandos SQL>;
END CASE;
```

```
BEGIN
 DECLARE VALOR INT DEFAULT 1;

 CASE VALOR
  WHEN 4 THEN SELECT VALOR;
  WHEN 6 THEN SELECT 0;
  ELSE
   BEGIN
   END;
 END CASE;
END
```

Figura 3.7: Comando *case...when*
Fonte: Manual de Referência do MySQL, 1997

3.1.8 Comando: *loop* e *iterate*

O comando LOOP não tem uma condição a ser testada. Para que a repetição termine, o MySQL determina que o comando *leave* finalize o laço. O comando *iterate* é usado dentro da construção *loop... end loop*, serve para reiniciar a repetição, ou seja, o *loop*. Sintaxe do comando *loop*:

```
<nome do label> : LOOP

<comandos SQL>;

<comandos SQL>;

 ITERATE <nome do label>;

 <comandos SQL>

 <comandos SQL>;

END LOOP <nome do label>;
```

Os comandos *loop...end loop* e *iterate* está demonstrado na Figura 3.8:

```
BEGIN
 label2 : LOOP

  SET num1 = num2 + 20;
  IF num1 < 200 THEN
     ITERATE label2;
  END IF;
  LEAVE label2;

  END LOOP label2;
END
```

Figura 3.8: Comandos *loop...end* e *iterate*
Fonte: Manual de Referência do MySQL, 1997

3.1.9 Comando: *repeat*

Esse comando permite a repetição na execução de um conjunto de comandos. Os comandos serão executados ao menos uma vez, independente da condição testada. A execução do comando *repeat* será mantida enquanto a condição testada for falsa.

Sintaxe do comando REPEAT:

```
REPEAT

<comandos SQL>;

<comandos SQL>;

 UNTIL <condição testada>

END REPEAT;
```

Para melhor exemplificação, temos um exemplo comando *repeat*:

```
BEGIN
 label2 : LOOP

  SET num1 = num2 + 20;
  IF num1 < 200 THEN
     ITERATE label2;
  END IF;
  LEAVE label2;

  END LOOP label2;
END
```

Figura 3.9: Comando *repeat*
Fonte: Manual de Referência do MySQL, 1997

3.1.10 Comando: *while...do*

Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição é verdadeira os comandos SQL são executados uma vez e a condição é avaliada novamente. Caso a condição seja falsa a repetição é terminada sem a execução dos comandos SQL.

Sintaxe do comando *while...do*:

```
WHILE <condição testada> DO

<comandos SQL>;

<comandos SQL>;

<comandos SQL>;

END WHILE;
```

O comando *while...do* está demonstrado na Figura 3.10:

```
BEGIN
  ......................
  WHILE num > xx DO
    .....
    SET xx = xx - 1;

  END WHILE;
END
```

Figura 3.10: Comando *while..do*
Fonte: Manual de Referência do MySQL, 1997

3.1.11 Comando: *leave*

Esse comando é utilizando para sair de uma estrutura de controle, seja de repetição (*repeat, while, loop, iterate*) ou decisão (*if, case*).

Sintaxe do comando *leave*:

LEAVE <nome do label>;

OBSERVAÇÃO: o *label* pode ser o nome de uma função, procedimento ou gatilho, ou simplesmente o nome de um rótulo presente nas estrutura de controle. O comando *leave* está demonstrado na Figura 3.11:

```
BEGIN
  ......................
  label1 : xxx
    .....
    .....
    LEAVE label1;
  END xxxx label1;
    .....
END
```

Figura 3.11: Comando *leave*
Fonte: Manual de Referência do MySQL, 1997

3.1.12 Comando: *call*

Esse comando é utilizando para chamar um procedimento (*store procedure*) no MySQL. Posteriormente, veremos como criar um procedimento (*store procedure*).

Sintaxe do comando call:

CALL <nome-procedimento> (parâmetros do procedimento);

ou

CALL <nome-procedimento> ( );

O comando *call* está demonstrado:

```
CALL procedimento ();
CALL procedimento1;
```

Figura 3.12: Comando *call*
Fonte: Manual de Referência do MySQL, 1997

3.1.13 Comandos: *return* e *returns*

Esse comando é utilizando para retornar um valor de uma variável armazenada no MySQL. O comando *return* não é utilizado em procedimentos (*store procedure*). Sintaxe do comando *return*:

```
RETURN <valor de variável>;
```

O comando *return* está demonstrado na Figura 3.13:

```
CREATE FUNCTION Aumenta_Sal(aumento INT) RETURNS double
 BEGIN
     DECLARE maior_sal REAL;
     UPDATE funcao SET sal=(sal*(aumento/100))+sal;

    SELECT MAX(sal) INTO maior_sal FROM funcao;

    RETURN maior_sal;

 END
```

Figura 3.13: Comando *return*
Fonte: Manual de Referência do MySQL, 1997

O comando *return* é diferente do comando *returns*. Os dois são usados numa função, o 1º (*returns*) serve para definir qual tipo de dados irá retornar na função, e o 2º (*return*) diz o valor de qual variável será definida como retorno. No exemplo, acima, temos a função **Aumenta_Sal** (...) que irá retornar um valor do tipo *double*, que está armazenado na variável **maior_sal**.

Sintaxe do comando *returns*:

```
RETURNS <tipo da variável>;
```

Praticar todos os comandos vistos até agora. Poste no ambiente da disciplina o arquivo de sua atividade.

3.2 Procedimentos (*store procedure*)

Agora que já aprendemos a sintaxe, para que possamos construir as primeiras rotinas que serão executadas no MySQL. Temos que entender, que nossas rotinas poderão ser simples, com poucas linhas de código ou bastante complexas e de código extenso, vai depender muito do que se programa, como numa linguagem de programação comum. A diferença básica e essencial, é que esses nossos programas estarão armazenados no servidor e poderão ser chamados a partir da própria linguagem SQL.

Assim, teremos que utilizar os procedimentos (*store procedure*) e as funções (*function*), cujos conceitos estão definidos no SQL e ficam armazenados no servidor, podendo ser invocados, de acordo com a necessidade do SGBD.

O MySQL reconhece *store procedure* (procedimentos armazenados). Nesse livro vamos nos referir a eles somente como procedimentos.

Vamos a um assunto muito interessante referente à banco de dados, mais precisamente procedimentos armazenados (*store procedure*).

Um procedimento é um código procedural, semelhante ao utilizado em linguagens estruturadas, só que ao invés de ter que escrever seus próprios comandos SQL na aplicação, você cria o procedimento (*store procedure*) no banco de dados e pode ser executado por qualquer aplicação cliente, o que melhor e muito sua performance de execução.

Já que entendemos o que é um procedimento (*store procedure*). Saberemos, a partir de agora, os motivos para se utilizar um procedimento (*store procedure*):

- **Modularização**: os procedimentos (*store procedures*) utilizam a programação modular. Eles encapsulam conjuntos de operações sobre os dados, ou seja, qualquer possível alteração no SGBD fica "escondida" da aplicação que fazem o acesso o banco por meio de procedimento (*store procedure*). E ainda permite que aplicações possam acessar o SGBD de maneira uniforme;

- **Performance**: quando um procedimento (*store procedure*) é executado no banco de dados, um "modelo" daquele procedimento continua na memória para se aproveitada posteriormente, o que melhorar a velocidade de execução;

- **Segurança**: utilizando procedimento (*store procedure*), o acesso não é feito diretamente nas tabelas, portanto a probabilidade de um desenvolvedor da aplicação fazer algo de errado que possa comprometer a base do banco de dados diminui, o que aumenta a segurança da aplicação desenvolvida utilizando procedimento. Ou podem ter atributos de segurança, portanto os usuários podem ter permissões de executar procedimento (*store procedure*) , sem ter permissões sobre os objetos referenciados dentro do procedimento;

- **Tráfego de rede**: pode reduzir o tráfego na rede gerado pela aplicação, porque quando código SQL fica na aplicação, é necessário que o mesmo seja enviado ao banco (compilado e otimizado) a cada nova consulta, se tivermos muitas linhas de SQL isso irá gerar um tráfego maior, portanto é mais vantajoso ter uma linha de código para executar um procedimento (*store procedure*);

Os procedimentos (*store procedure*), podem ser de três tipos:

- **retornam registros do banco de dados**: um simples *select* em uma tabela;

- **retornam um simples valor**: pode ser o total de registros de uma tabela;

- **não retorna valor (realiza uma ação)**: pode ser a inserção.

Já que entendemos, o que é um procedimento, sua sintaxe é:

```
CREATE PROCEDURE nome-do-procedimento ([parâmetros[,...]])
BEGIN
[características ...] corpo da função
parâmetros:
 [ IN | OUT| INOUT ] nome do tipo do parâmetro
tipo:
 Qualquer tipo de dado válido no MySQL
 características:
 Linguagem SQL
 | [NOT] DETERMINISTIC
 | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES
 SQL DATA }
```

```
| SQL SECURITY {DEFINER | INVOKER}
| COMMENT string
corpo do procedimento:
comandos válidos no SQL
END
```

Onde:

nome-do-procedimento representa o nome do procedimento que será criado.

Após elaborarmos um *store procedure* (procedimento) no *MySQL* podemos fazer uma análise desse procedimento através do comando *select...from...where... procedure analyse* (max elementos [max memória]) conforme consta no *link*: http://www.hospedia.com.br/mysql_manual/html/extending-mysql.html#adding-procedures

Os três tipos de parâmetros que pode ser utilizados nos procedimentos são:

- **IN**: é um parâmetro de entrada, ou seja, um parâmetro cujo seu valor seu valor será utilizado no interior do procedimento para produzir algum;
- **OUT**: é um parâmetro de saída, retorna algo de dentro do procedimento, colocando os valores manipulados disponíveis na memória ou no conjunto de resultados;
- **INOUT**: faz o processamento dos IN ou OUT simultaneamente.

A característica *deterministic* diz que o procedimento sempre retorna o mesmo resultado para os mesmos parâmetros de entrada, e a característica *not deterministic* determina o contrário da característica *deterministic*. Atualmente, essas características são aceitas, pela MySQL, mas ainda não é usada.

Temos a característica que informa o modo de tratamentos dos dados durante a execução do procedimento. Para *contains* SQL é o *default*, determina que os dados não podem ser lidos ou escritos, pois ele já devem ser determinados, no bloco de comandos. O *no* SQL diz que o procedimento contém dados a serem lidos ou escritos. O *reads* SQL *data* indica que teremos somente leitura de dados, por meio do *select*. E o *modifies* SQL *data* determina que tem-se escrita ou remoção de dados, utilizando o *insert* ou *delete*.

A característica SQL security pode ser usada para especificar se o procedimento pode ser executado para usar as permissões do usuário que criou os procedimentos, ou do usuário que o invocou. O definer é o valor padrão, foi um recurso novo introduzido no SQL:2003.

A cláusula *comment* é uma extensão do MySQL, e pode ser usada para descrever o procedimento (*stored procedure*).

Depois que, aprendemos a sintaxe dos procedimentos, vamos ver alguns exemplos implementadas no MySQL, do nosso banco de dados da "empresa de desenvolvimento de projetos" utilizado desde o início em nosso caderno, a seguir.

Antes de elaborar os procedimentos, temos que ter como objetivos, criá-los de forma que eles possam trazer algum tipo utilização, realmente prática, para empresa ou organização, na qual forem implementados.

Poderia ser interessante, para cada departamento, saber relacionar os projetos e seus orçamentos, o usuário pode determinar solicitar um relatório de todos os projetos cujo orçamento for maior do R$ 10.000,00 e a data de início desse projeto. Na Figura 3.14 um exemplo bem simples de procedimento:

```
CREATE PROCEDURE Relatorio_Projeto()
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
   select nome, dt_ini from Projeto p
   where p.orcamento > 10000;

   END
```

Figura 3.14: Comando *procedure*
Fonte: Manual de Referência do MySQL, 1997

Criar um procedimento para relacionar os projetos e seus orçamentos, o usuário poderia solicitar um relatório de todos os projetos com seus orçamentos cuja data de início desse projeto seja janeiro de 2012. Poste no ambiente da disciplina o arquivo de sua atividade.

A gerência de pessoal, pode solicitar os nomes dos funcionários que estão lotados em mais de um projeto, ou solicitar em qual ou quais projeto(s) seu funcionário está lotado, a partir do código desse funcionário. Observem que são dois procedimentos diferentes um sem e outro com parâmetros de entrada. Para resolver estes problemas, vamos à utilização dos procedimentos (Figura 3.15).

```
CREATE PROCEDURE Relatorio_Func_Projeto()
    LANGUAGE SQL
    DETERMINISTIC
    READS SQL DATA

    BEGIN
     SELECT nome
     FROM funcionario f
     WHERE cod_func IN (SELECT cod_func FROM trabalha);

   END
```

Figura 3.15: Comando *procedure* Relatorio_Func_Projeto
Fonte: Manual de referência do MySQL, 1997

```
CREATE PROCEDURE Relatorio_Cod_Func(IN cod_func INT)
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
    SELECT nome
    FROM projeto
    WHERE  cod_proj IN(SELECT cod_proj FROM trabalha WHERE cod_func);

   END
```

Figura 3.16: Comando *procedure* Relatorio_Cod_Func
Fonte: Manual de referência do MySQL, 1997

A chamada para a execução de um procedimento é diferente da chamada para função. No procedimento, utilizamos o comando (*call*). Daqui a pouco, aprenderemos como utilizar função.

Vamos invocar os procedimentos criados, e verificar se eles realmente funcionam, no MySQL.

Agora é a hora de usar o comando *call*, toda vez que quiser executar um procedimento precisamos utilizar esse comando

```
CALL Relatorio_Projeto;

CALL Relatorio_Func_Projeto;

CALL Relatorio_Cod_Func(2);
```

Figura 3.17: Comando *call*
Fonte: Manual de Referência do MySQL, 1997

Algumas características de um procedimento podem ser alteradas, para isso, vamos ver a sintaxe da alteração de um procedimento, a seguir:

ALTER PROCEDURE nome-do-procedimento[características ...]

características:
NAME novo-nome

{CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES

SQL DATA }

SQL SECURITY {DEFINER | INVOKER}

COMMENT string

Onde:

nome-do-procedimento representa o nome do procedimento que terá sua(s) característica(s) alterada(s).

O comando *alter procedure* não é muito utilizado no MySQL, uma vez que não se pode alterar nem os parâmetros nem o corpo do procedimento. Caso, se queira fazer muitas modificações no procedimento, recomenda-se apagar o procedimento e criar um novo. A seguir, vamos aprender como remover uma função:

DROP PROCEDURE nome-do-procedimento [IF EXISTS] nome_warning

A cláusula ***if exists***, do comando ***drop procedure*** ou ***function*** é uma extensão do MySQL. Ela previne que um erro ocorra se a função ou o procedimento não exista mais no SGBD. Um aviso é produzido e pode ser visualizado.

Onde:

nome-do-procedimento representa o nome do procedimento que terá removido do servidor de banco de dados.

Veremos, a seguir, como remover os três procedimentos que foram criados no MySQL:

```
DROP PROCEDURE Relatorio_Cod_Func;
DROP PROCEDURE Relatorio_Func_Projeto;
DROP PROCEDURE Relatorio_Projeto;
```

Figura 3.18: Comando *drop procedure*
Fonte: Manual de Referência do MySQL, 1997

3.3 Funções (*functions*)

Vamos agora, aprender sobre funções (*functions*).

Funções (também chamadas de rotinas, ou subprogramas) são segmentos de programa que executam uma determinada tarefa específica. É possível ao administrador do SGBD, escrever suas próprias rotinas, no MySQL.

Para saber todas as informações sobre as funções do SGBD, podemos usar o comando ***show function status*** e ***show procedure status***, para os procedimentos.

Criando nossas próprias funções, temos a vantagem de adequá-las a nosso ambiente de trabalho, de acordo com as nossas necessidades diárias. Isso tudo, sem ter que modificar a aplicação, uma vez a função está implementada direto na aplicação.

Temos como exemplo, um Banco Comercial, onde um de seus maiores patrimônios são os dados dos correntistas, nesse tipo de organização a mudança de SGBD é remota, procura-se dessa forma obter o máximo possível do servidor de banco de dados, utilizando os recursos que lhes são oferecidos, como a utilização de funções. Outras organizações por utilizar tudo centralizado no SGBD, centralizam também às regras de negócios, tornando-as iguais para qualquer aplicação que venha a acessar o servidor do banco de dados. Dessa maneira, uma função que venha a ser padronizada no banco de dados, por ser executada por aplicações diferente, seja desenvolvida em Delphi, Java, C.

Então, função são programas armazenados no SGBD, pré-compilados, invocados de forma explícita para executar alguma lógica de manipulação de dados, e que sempre retorna algum valor.

A diferença básica entre o procedimento e uma função é que a função sempre retorna algum valor. Já que entendemos, o que é uma função, sua sintaxe é:

CREATE FUNCTION nome-da-função ([parâmetros[,...]])

[RETURNS tipo]

BEGIN
[características ...] corpo da função

parâmetros:nome do tipo do parâmetro

tipo:
Qualquer tipo de dado válido no MySQL

características:
Linguagem SQL
| [NOT] DETERMINISTIC

| {CONTAINS SQL | NO SQL | READS SQL DATA |

MODIFIES SQL DATA }

| SQL SECURITY {DEFINER | INVOKER}

```
| COMMENT string
corpo da função:
comandos válido no SQL
RETURN <valor>
END
parâmetros vazia de ( ) deve ser usada.
```

A função também apresenta as características *deterministic* e *not deterministic*. As características *contains* SQL, *no* SQL, *reads* SQL *data* e *modifies* SQL *data*, possuem as mesmas funcionalidades utilizadas nos procedimentos.

E na questão de segurança, a característica SQL *security* pode ser usada para especificar se a função possa ser executada usando as permissões do usuário que criou as funções, ou do usuário que a invocou. O *definer* é o valor padrão, foi um recurso novo introduzido no SQL:2003.

A cláusula *comment* é uma extensão do MySQL, e pode ser usada para descrever a função (*function*).

A cláusula *returns* pode ser especificada apenas por uma *function*. É usada para indicar o tipo de retorno da função, e o corpo da função deve conter uma instrução *return* e o valor de retorno. Depois que, aprendemos a sintaxe da função, vamos ver alguns exemplos de funções, implementadas no MySQL, do nosso banco de dados utilizado desde o início deste caderno.

Antes de elaborar as funções, temos que ter como objetivos criá-las de forma que elas possam trazer algum tipo utilização, realmente prática, para empresa ou organização, na qual forem implementadas.

Caso cada departamento, queira saber a classificação dos projetos de acordo com o seu orçamento, a função está exemplificada na Figura 3.19 a seguir.

```
CREATE FUNCTION Classifica_Orcament(orcamento INT)
       RETURNS VARCHAR(15)
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
      IF orcamento >=100000 THEN RETURN 'Muito Alto';
      ELSEIF orcamento >= 50000 THEN RETURN 'Alto' ;
      ELSEIF orcamento >= 30000 THEN RETURN 'Medio' ;
      ELSEIF orcamento >= 10000 THEN RETURN 'Baixo' ;
      ELSE  RETURN 'Muito Baixo' ;
      END IF;

   END
```

Figura 3.19: Comando *function* Classifica_Orcament
Fonte: Manual de Referência do MySQL, 1997

A gerência de pessoal poderia solicitar ao SGBD, um aumento salarial a todos os funcionários e ter como retorno o valor do maior salário, após o aumento. Vamos a elaboração dessa função no MySQL:

```
CREATE FUNCTION Aumenta_Sal(aumento INT)
       RETURNS REAL
   LANGUAGE SQL
   DETERMINISTIC
   MODIFIES SQL DATA

   BEGIN
     DECLARE maior_sal REAL;
     UPDATE funcao SET sal=(sal*(aumento/100))+sal;

     SELECT MAX(sal) INTO maior_sal FROM funcao;

    RETURN maior_sal;

   END
```

Figura 3.20: Comando *function* Aumenta_Sal
Fonte: Manual de Referência do MySQL, 1997

A gerência de pessoal poderia solicitar ao SGBD, um aumento salarial a todos os funcionários e ter como retorno o valor do maior salário, após o aumento, criar uma função para isso. Poste no ambiente da disciplina o arquivo de sua atividade.

Para verificarmos, se as funções acima funcionarão e se retornaram o que se esperava dele, podemos usar os *selects*. Vamos invocar as funções criadas para saber a classificação do orçamento, e verificar se elas realmente funcionam.

```
SELECT Classifica_Orcament(155000);
SELECT Classifica_Orcament(55000);
SELECT Classifica_Orcament(35000);
SELECT Classifica_Orcament(15000);
SELECT Classifica_Orcament(1500);
```

Figura 3.21: Comando *select* Classifica_Orcament
Fonte: Manual de Referência do MySQL, 1997

Agora, vamos invocar a Aumenta_Sal(aumento INT) e vê sua funcionalidade, para um aumento de 10%.

```
SELECT Aumenta_Sal(10);
```

Figura 3.22: Comando *select* Aumenta_Sal
Fonte: Manual de Referência do MySQL, 1997

Lembrando que uma função sempre retorna algum valor, por isso podemos utilizar o valor do retorno da função. E para utilizar esse valor, é preciso que seja criada uma variável do mesmo tipo do retorno da função, e o nome da função deve aparecer ao lado direito de um comando de atribuição de valores (*set*). As funções armazenadas no MySQL podem ser utilizadas em comandos SQL da mesma forma que as outras funções já criadas na própria linguagem.

Algumas características de uma função podem ser alteradas, para isso usamos a sintaxe, a seguir:

```
ALTER FUNCTION nome-da-função [características ...]

características:
 | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES

 SQL DATA }

 | SQL SECURITY {DEFINER | INVOKER}
```

Onde:

nome-da-função representa o nome da função que terá sua(s) característica(s) alterada(s).

Esse comando não é muito utilizando, uma vez que não se pode alterar os parâmetros ou o corpo da função. Caso se queira alterar, o código de uma função o ideal é removê-la e criar uma nova função. A seguir, vamos aprender como remover uma função.

Portanto, para finalizar o nosso estudo sobre função, podemos deletar essa rotina do SGBD, para isso usamos a sintaxe:

DROP FUNCTION nome-da-função [IF EXISTS] nome_warning

Onde:

nome-da-função representa o nome da função que terá removida do servidor de banco de dados.

Teremos, a seguir, a remoção das funções criadas no MySQL.

```
DROP FUNCTION Classifica_Orcament;
DROP FUNCTION Aumenta_Sal;
```

Figura 3.23: Comando *drop function*
Fonte: Manual de Referência do MySQL, 1997

Para listarmos todos os procedimentos (*store procedure*) e as funções (*functions*) armazenados no MySQL, temos que utilizar a sintaxe a seguir:

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES;
```

Figura 3.24: Comando *select*
Fonte: Manual de Referência do MySQL, 1997

Esse último comando lista somente os procedimentos e funções, gravadas na tabela *routines* do banco de dados *information_schema*.

Assista à vídeo-aula da disciplina de Prática de Banco de disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre **Procedimento** e **Função**.

Resumo

Começamos uma introdução na linguagem SQL/PSM (SQL/*Persistent Stored Modules*), voltada para utilização no MySQL. Depois, utilizamos o estudo de caso de uma empresa de desenvolvimento de projeto, para verificar como modelar o banco de dados do sistema dessa empresa, utilizando procedimentos e funções.

Portanto, a Aula 3, procurar fazer uma iniciação no estudo de procedimentos (*store procedure*) e funções (*functions*) no MySQL, buscando ensinar o manejo destes recursos que é de grande proveito, principalmente no sistemas que necessitam de modularidade e otimização quanto à performance.

Atividades de aprendizagem

1. Qual a diferença entre o comando *return* e *returns*?

2. Qual a diferença entre procedimento e função?

3. São segmentos do programa que executam uma determinada tarefa específica. Podemos chamar de rotinas definidas pelo usuário **UDFS** (*User Definition Functions*)?

A-Z

UDFS (*User Defined Functions*)
São as chamadas de funções ou de rotinas definidas pelo usuário

4. Utilizando a linguagem SQL para se programar, qual das alternativas abaixo possui o comando para "armazenar o resultado de uma consulta em uma variável".

   a) OPEN

   b) SET

   c) SELECT...INTO

   d) SELECT

   e) CALL

5. No que se refere a utilização de procedimento no MySQL, escolha a alternativa que contém o comando para que se possa mudar as características do procedimento:

   a) CREATE PROCEDURE

   b) DROP PROCEDURE

   c) ALTER PROCEDURE

   d) CREATE PROCEDURE FOR EACH ROW

   e) Nenhuma das alternativas anteriores

6. Utilizando o banco de dados **praticabd** da "empresa desenvolvedora de projetos", elabore um procedimento e uma função. Explique a funcionalidade de cada um deles para a empresa. O código do procedimento e da função deverá ser compilado e executado no MySQL, de acordo com o que for proposto pelo aluno.

Aula 4 – Gatilho e controle de acesso

Objetivos

Definir gatilho (*trigger*).

Conhecer a sintaxe de um gatilho (*trigger*).

Implementar gatilhos (*trigger*).

Definir usuário de Banco de dados.

Criar, renomear, deletar, alterar senha de usuário no MySQL.

Definir privilégios de usuário de BD.

Atribuir e Remover privilégios de usuários.

4.1 Gatilho (*trigger*)

A linguagem SQL além de tratar os procedimentos e funções, também permite a criação de gatilhos (*triggers*).

É considerado uma lógica de processamento procedural, armazenada no SGBD e disparada automaticamente pelo servidor sob condições específicas. Gatilhos (*triggers*) representam regras do mundo (negócio) que definem a integridade ou consistência do BD. Passaram a ser padrão SQL em 1999. Seu principal objetivo é monitorar o SGBD e realizar alguma ação quando uma condição ocorre.

Os gatilhos (*triggers*) devem ser armazenados na base de dados como objetos independentes e não podem ser locais a um bloco ou pacote. São na verdade, procedimentos disparados automaticamente pelo SGBD em resposta a um evento específico do banco de dados. Portanto, são bastante semelhantes aos procedimentos (*store procedure*) só que tem sua execução disparada pelo SGBD quando ocorre um acontecimento/evento desencadeador de "*trigerring*" suceder e não aceita argumentos. O evento desencadeador pode ser uma operação DML (*insert, update*, ou *delete*) em uma tabela da base de dados.

O ato de executar um gatilho (*trigger*) é conhecido como disparar o gatilho (*trigger*).

Os gatilhos (*triggers*) podem ser usados para:

- IN: é um parâmetro de entrada, ou seja, um parâmetro cujo seu valor seu valor será utilizado no interior do procedimento para produzir algum;
- Segurança sobre a base de dados, e ação de usuários, verificando quando uma operação é realizada sobre uma entidade, o gatilho (*trigger*) é disparado para verificar as permissões do usuário;
- Melhorar a segurança no acesso aos dados;
- Assegurar as restrições de integridade;
- Fazer a auditoria das informações de uma tabela, registrando as alterações efetuadas e quem as efetuou;
- Sinalizar automaticamente a outros programas que é necessário efetuar uma ação, quando são efetuadas alterações numa tabela.

A seguir são enumeradas algumas vantagens no uso de gatilhos (*triggers*):

- Um gatilho (*trigger*) sempre é disparado quando o evento ocorre, evitando assim esquecimentos ou falta de conhecimento sobre o banco de dados;
- São administrados de forma centralizada, ou seja, o DBA (Administrador de Banco de Dados) define suas situações, eventos e ações;
- A ativação central combina com o modelo cliente/servidor, portanto a execução da *trigger* é realizada no servidor, independente do aplicativo executado pelo cliente.

Com o uso de gatilho (*trigger*) procura-se eliminar o esforço manual e repetitivo de identificar e corrigir comandos e regras mal implementados, com baixo desempenho. Assim, o desenvolvedor da aplicação sairá ganhando em produtividade, uma vez que não irá mais perder seu tempo em corrigir e buscar regras já prontas e controladas no SGBD.

Por se tratar de mecanismos ativos, os gatilhos (*triggers*), utilizam como requisitos para sua elaboração, o paradigma **ECA**. Onde o "Evento", indica o momento do disparo da regra, a "Condição" precisa ser satisfeita para que a execução do gatilho (*trigger*) prossiga e a "Ação" determina o que deve ser feito, caso a condição seja realmente válida.

ECA
Significa **E**vento-**C**ondição-**A**ção.

Há três tipos principais de gatilhos (*triggers*): DML, *instead-of* e gatilhos (*triggers*) de sistema (DDL).

1. **Um gatilho (*trigger*) DML** é acionado em uma operação INSERT, UPDATE ou DELETE de uma tabela de dados. Ele pode ser acionado antes ou depois que a instrução é executada e pode ser acionado uma vez por linha problemática ou uma vez por instrução;

2. **Os gatilhos (*triggers*) *instead-of*** podem ser definidos apenas em visões (tanto de objeto como relacional);

3. **Um *trigger* de sistema** é acionado quando um evento de sistema como uma inicialização ou desativação de banco de dados ocorrer, em vez de em uma operação DML em uma tabela. Um *trigger* de sistema também pode ser acionado em operações de DDL como a criação de uma tabela.

Devemos ter cuidado quando utilizar gatilhos (*triggers*), pois erros na execução de um gatilho (*trigger*) pode causar falhas nas funções de incluir/deletar/atualizar da função que o disparou.

Não podemos criar gatilho (*trigger*) para disparar outro gatilho (*trigger*), ou seja, na pior das hipóteses gerar uma cadeia infinita de gatilhos (triggers) e também não se deve usar gatilho (*trigger*) na replicação da base de dados, porque quando alguma modificação é feita na base de dados principal, um gatilho (*trigger*) aplica a mesma modificação na cópia.

Cada SGBD utiliza sua própria linguagem e sintaxe para gatilhos (*triggers*). Iremos aprender agora, como elaborar gatilhos (*trigger*) no MySQL. Importa ressaltar que iremos nos ater aos gatilhos (*triggers*) DML. Já que aprendemos o que é um gatilho (*trigger*), sua sintaxe a seguir:

CREATE TRIGGER nome-do-gatilho momento-da-exec
evento-disparador
ON nome-da-tabela FOR EACH ROW comandos válidos no SQL OU
BEGIN
corpo do gatilho
comandos válido no SQL

Onde:

nome-do-gatilho representa o nome do gatilho (*trigger*) que será criado;
momento-da-execução diz em que tempo a ação ocorrerá, antes (BEFORE) ou depois (AFTER) do evento;
evento-disparador representa o evento que dispara o gatilho (*trigger*), são os comandos INSERT, UPDATE e DELETE do SQL;
nome-da-tabela diz o nome da tabela que será utilizado pelo gatilho (*trigger*);

Para um melhor entendimento do que seja o momento da execução e um evento-disparador durante a execução de um gatilho (*trigger*), observemos no Quadro 4.1, a seguir.

Quadro 4.1: Tipos de gatilhos

Tipo de Gatilho	Descrição
BEFORE INSERT	O gatilho é disparado antes de uma ação de inserção.
BEFORE UPDATE	O gatilho é disparado antes de uma ação de alteração.
BEFORE DELETE	O gatilho é disparado antes de uma ação de remoção.
AFTER INSERT	O gatilho é disparado depois de uma ação de inserção.
AFTER UPDATE	O gatilho é disparado depois de uma ação de alteração.
AFTER DELETE	O gatilho é disparado depois de uma ação de remoção.
BEFORE INSERT	O gatilho é disparado antes de uma ação de inserção.

Fonte: Manual de Referência do MySQL, 1997

É possível combinar alguns dos modos, desde que tenham a operação de AFTER ou BEFORE em comum, conforme mostra o Quadro 4.2 a seguir.

Quadro 4.1: Tipos de gatilhos

Tipo de Gatilho	Descrição
BEFORE INSERT ou UPDATE ou DELETE	O gatilho é disparado antes de uma ação de inserção ou de alteração ou de remoção.

Fonte: Manual de Referência do MySQL, 1997

Depois que, aprendemos a sintaxe do gatilho (*trigger*), vamos implementar no nosso banco de dados praticabd utilizando o MySQL, alguns gatilhos (*triggers*) realmente úteis para as regras de negócios da nossa empresa e seus projetos.

A empresa pode querer ter um controle de todas as datas de previsão de finalização prevista de seus projetos. Para isso podemos implementar um gatilho (*trigger*) para colher as datas de previsão de término do cadastro de todos os projetos e estas serão inseridas numa outra tabela, por exemplo, uma tabela com a seguinte estrutura.

```
CREATE TABLE DataProj
(novo_codproj INT auto_increment,
dt_prev_term DATE NOT NULL,
CONSTRAINT pk_DataProj PRIMARY KEY(novo_codproj));
```

Figura 4.1: Comando *create table*
Fonte: Manual de Referência do MySQL, 1997

Toda vez que um novo projeto for inserido no cadastro de projeto, tanto será acrescido na tabela do projeto, quanto na nova tabela criada (**dataproj**), isso automaticamente através do gatilho (*trigger*) da Figura 4.2 a seguir.

```
CREATE TRIGGER Data_Final_Projeto AFTER INSERT on projeto
FOR EACH ROW BEGIN

  IF (NEW.dt_prev_term IS NOT NULL) THEN
    INSERT INTO dataproj SET dtprevterm = NEW.dt_prev_term;
  END IF;

END
```

Figura 4.2: Comando *create trigger*
Fonte: Manual de Referência do MySQL, 1997

Aproveitando a criação da tabela **dataproj**, faça um gatilho para toda vez que um projeto for deletado no cadastro de projeto, que seja excluído também da tabela projeto. Poste no ambiente da disciplina o arquivo de sua atividade.

Para testar o gatilho (*trigger*), fazemos uma inserção na tabela **projeto**, no campo referente à data de previsão do término do projeto, e depois fazemos um *select* na tabela **dataproj**, pra verificarmos se a data 2015-01-12 também foi inserida automaticamente nela, sintaxe é:

```
INSERT INTO projeto SET dt_prev_term='2015-01-12';

SELECT * from dataproj;
```

Figura 4.3: Comando *insert into*
Fonte: Manual de Referência do MySQL, 1997

Para finalizar o nosso estudo sobre gatilho (*trigger*), iremos verificar como deletar um gatilho (*trigger*), sintaxe é:

DROP TRIGGER nome-do-gatilho [IF EXISTS] nome_warning

Onde:

nome-do-gatilho representa o nome do gatilho que terá removido do servidor de banco de dados.

Teremos, a seguir, a remoção dos gatilhos que foram criados no MySQL.

```
DROP TRIGGER Data_Final_Projeto;
```

Figura 4.4: Comando *drop trigger*
Fonte: Manual de Referência do MySQL, 1997

Para listarmos todos os gatilhos (*triggers*) armazenados no MySQL, temos que utilizar a sintaxe a seguir:

```
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;
```

Figura 4.5: Comando *select*
Fonte: Manual de Referência do MySQL, 1997

Esse último comando lista somente os gatilhos, gravados na tabela TRIGGERS do banco de dados INFORMATION_SCHEMA.

4.2 Controle de acesso

A integridade do sistema de Banco de Dados depende muito do tipo de segurança implementado ao SGBD. Para isso existe um DBA (Administrador de Banco de Dados) responsável pela segurança e controle de acesso a esse banco de Dados.

Iremos estudar como implementar o controle de acesso e melhorar a segurança no MySQL.

Quando iniciamos os trabalhos em um servidor MySQL, normalmente precisamos de uma identidade e uma senha. Essa identidade é o que vai fazer com que o MySQL reconheça o usuário. Cabe portanto ao DBA (Administrador de Banco de Dados) criar os usuários que poderão usar o MySQL, podendo também renomeá-los e removê-los permanentemente do SGBD.

A sintaxe para criar um usuário é:

CREATE USER usuario [IDENTIFIED BY[PASSWORD]] 'senha';

Onde:

usuario representa o nome do usuário que será criado;
'**senha**' representa a senha de acesso ao MySQL pelo usuário criado.

Teremos, a seguir, a criação de um usuário no MySQL:

```
CREATE USER aluno_EAD;
```

Figura 4.6: Comando *create user*
Fonte: Manual de Referência do MySQL, 1997

Temos também a possibilidade de criar o novo usuário no MySQL já com senha cada um:

```
CREATE USER alunoEAD IDENTIFIED BY '2010';
```

Figura 4.7: Comando *create user* com senha
Fonte: Manual de Referência do MySQL, 1997

Crie um usuário EADaluno com senha 2011. Poste no ambiente da disciplina o arquivo de sua atividade

Depois de criarmos um usuário com senha no MySQL, temos a possibilidade de alterarmos a senha desse usuário, conforme a sintaxe:

SET PASSWORD FOR usuario PASSWORD ('nova senha');

Veremos como alterar a senha do usuário alunoEAD, exemplificado no exemplo da Figura 4.8.

```
SET PASSWORD FOR alunoEAD = PASSWORD ('2014');
```

Figura 4.8: Comando *set password*
Fonte: Manual de Referência do MySQL, 1997

Altere a senha do usuário EADaluno para EAD. Poste no ambiente da disciplina o arquivo de sua atividade.

Caso se queira, renomear um usuário, a sintaxe é:

RENAME USER usuario_velho TO usuario_novo;

Abaixo, um exemplo de renomeação de usuário:

```
RENAME USER alunoEAD TO aluno_E_A_D;
```

Figura 4.9: Comando*rename user*
Fonte: Manual de Referência do MySQL, 1997

E para remover um usuário no MySQL, a sintaxe é:

DROP USER usuário;

A exemplificação de como remover um usuário no MySQL, a seguir.

```
DROP USER aluno_E_A_D;
```

Figura 4.10: Comando *drop user*
Fonte: Manual de Referência do MySQL, 1997

Definido no MySQL, os usuários precisamos definir quais privilégios eles terão direito.

O sistema de privilégios do MySQL faz a autenticação de um usuário a partir de uma determinada máquina e associa este usuário com privilégio ao banco de dados como usar os comandos *select, insert, update* ou *delete*. Isso garante que os usuários só possam fazer aquilo que lhes é permitido.

Quando nos conectamos a um servidor MySQL, nossa identidade é determinada pela máquina de onde nos conectamos e nosso nome de usuário que foi especificado. O MySQL concede os privilégios de acordo com a identidade e com o que se deseja fazer no SGBD. O controle de acesso do MySQL é realizado em dois passos:

Durante o controle de acesso, no passo 2 é preciso que seja feita a verificação da requisição, conforme consta no *link:* http://dev.mysql.com/doc/refman/4.1/pt/request-access.html

- **1º Passo:** É feita a conferência para saber se o usuário pode ter ou não acesso. Nesse caso, o usuário tenta se conectar, e o MySQL aceita ou rejeita a conexão baseado na identidade do usuário e no uso de sua senha correta.

- **2º Passo:** Caso o usuário posso se conectar, o SGBD verifica da solicitação feita pra saber se o usuário tem ou não privilégios para realizar cada solicitação requisitada. Tome como exemplo, o caso de um usuário que tenta altera a linha de uma tabela em um banco de dados ou uma tabela do mesmo banco, o MySQL irá se certificar que o usuário tem o privilégio *update* para a tabela ou o privilégio DROP.

4.3 Privilégios

Quando nos conectamos a um servidor MySQL, nossa identidade é determinada pela máquina de onde nos conectamos e nosso nome de usuário que foi especificado.

O MySQL concede os privilégios de acordo com a identidade e com o que se deseja fazer no SGBD.

Os privilégios fornecidos pelo MySQL são descritos no Quadro 4.3 a seguir.

Quadro 4.3: Privilégios MySQL

SELECT – INSERT – UPDATE – DELETE – INDEX – ALTER – CREATE – DROP – GRANT OPTION – RELOAD
PROCESS - FILE - ALL - SHUTDOWN - ALL

Fonte: Manual de Referência do MySQL, 1997

Existem mais privilégios que deve ser concedido apenas a administradores, são eles o *replication client, replication slave, super, create user, drop user, rename user*. Os privilégios são armazenados em quatro tabelas interna, no banco MySQL, a saber: **mysql.user**: privilégios globais aplicam-se para todos os banco de dados em um determinado servidor;

mysql.db: privilégios de banco de dados aplicam-se a todas as tabelas em um determinado banco de dados;

mysql.tables_priv: privilégios de tabelas que aplicam-se a todas as colunas em uma determinada tabela;

mysql.colums_priv: privilégios de colunas aplicam-se a uma única coluna em uma determinada tabela.

Os privilégios *select, insert, update* e *delete* permitem que sejam realizadas operações nos registros das tabelas existentes no SGBD.

O privilégio *index* permite a criação ou remoção de índices.

O privilégio *alter* permite que se altere tabelas (adicione colunas, renomeie colunas ou tabelas, altere tipos de dados de colunas), aplicado a tabelas.

Os privilégios *create* e *drop* permitem a criação de banco de dados e tabelas, ou a remoção de banco de dados e tabelas existentes.

O privilégio *grant option* permite que um usuário possa fornecer a outros usuários os privilégios que ele mesmo possui.

Os privilégios *reload, shutdown, process, file* e *all* são usados para operações administrativas que são realizadas pelo DBA (Administrador do Banco de Dados).

Deve-se ter muito cuidado ao conceder certos privilégios. Por exemplo, o privilégio *grant option* permite que usuários possam repassar seus privilégios a outros usuários, portanto dois usuários com diferentes privilégios e com privilégio *grant option* conseguem combinar seus privilégios, o que pode ser um risco numa organização, dependo do tipo de privilégio cedido a cada usuário desses. O privilégio *alter* pode ser usado para subverter o sistema de privilégios, quando o usuário pode renomear todas as tabelas do banco de dados.

No MySQL, ou em qualquer SGBD, o ideal é conceder privilégios somente para usuários que realmente necessitem.

Após o entendimento, do que sejam os privilégios, vamos vê como atribuir os privilégios a um usuário sua sintaxe é:

GRANT privilegios [(coluna)] [, privilégio [(coluna)] ...]
ON {nome_tabela – função - procedimento | * | *.* | nome_banco.*}
TO nome_usuario [IDENTIFIED BY 'senha']
[, nome_usuario [IDENTIFIED BY 'senha'] ...]
[WITH GRANT OPTION]

Onde:

privilegios: representa qual privilégio será concedido.
colunas: é opcional e especifica as colunas a que os privilégios se aplicam. Pode ser uma única coluna ou várias separadas por vírgula;
nome-tabela, **função** ou **procedimento**, *****, ***.*** , **nome_banco.***: é o banco de dados, tabela, função ou procedimento a que os privilégios se aplicam. Pode ser especificado como: *.* – neste caso os privilégios aplicam-se a todos os bancos de dados; * – os privilégios aplicam-se se não estiver utilizando nenhum banco de dados em particular; nome_banco.*- neste caso, os privilégios aplicam-se a todas as tabelas do banco; banco.tabela – neste caso os privilégios aplicam-se a uma determinada tabelas. Você pode, ainda especificar alguma(s) coluna(s) sem particular inserindo esta(s) em colunas;
nome-usuario: especifica um usuário do MySQL. Para preservar a segurança este usuário não deve ser o mesmo usuário do sistema;
senha: senha do usuário para acessar o servidor MySQL;
WITH GRANT OPTION: se for especificado, o usuário poderá conceder privilégios a outros usuários.

Vejamos, a seguir, alguns exemplos de privilégios concedidos a alguns usuários.

Nesse primeiro exemplo, temos a concessão de todos os privilégios ao usuário **aluno_E_A_D**, incluindo o privilégio que permite que o usuário possa conceder privilégios a outros usuários, sintaxe a seguir.

```
GRANT ALL PRIVILEGES ON *.* TO aluno_E_A_D WITH GRANT OPTION;
```

Figura 4.11: Comando *grant all*
Fonte: Manual de Referência do MySQL, 1997

No próximo exemplo, o usuário **aluno_E_A_D** terá os privilégios de realizar as operações SELECT, INSERT, UPDATE e DELETE no nosso banco de dados **praticabd**, podemos constatar que diferente do exemplo anterior, o usuário agora não poderá conceder privilégios para os outros usuários, a sintaxe a seguir.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON praticabd.* TO aluno_E_A_D;
```

Figura 4.12: Comando *grant select*
Fonte: Manual de Referência do MySQL, 1997

E para revogar direitos aos usuários do MySQL, usamos o *revoke*, sua sintaxe a seguir:

REVOKE privilegio [(coluna)] [, privilegio [(coluna)] ...]
ON {nome_tabela | * | *.* | nome_banco.*}
FROM nome_usuario [, nome_usuario ...]

ou

REVOKE ALL PRIVILEGES, GRANT OPTION FROM nome_usuario

Onde:

privilegios: representa qual privilégio será removido;
colunas: é opcional e especifica as colunas a que os privilégios serão removidos. Pode ser uma única coluna ou várias separadas por vírgula;
nome-tabela, *****, ***.*** , **nome_banco.*:** é o banco de dados ou tabela a que os privilégios se aplicam. Pode ser especificado como: *.* – neste caso os privilégios serão removidos de todos os bancos de dados; * – os privilégios removidos aplicam-se se não estiver utilizando nenhum banco de dados em particular; nome_banco.*- neste caso, os privilégios removidos aplicam-se a todas as tabelas do banco;
nome-usuario: especifica um usuário do MySQL que terá seus privilégios removidos;

Ou:

ALL PRIVILEGES: para remover todos privilégios do usuário;
GRANT OPTION: para remover o privilégio do usuário de conceder privilégio a outros usuários;
nome-usuario: especifica um usuário do MySQL que terá seus privilégios removidos;

Temos a seguir, a remoção dos privilégios do usuário **aluno_E_A_D**. Esse usuário tinha como privilégios manipular as operações de SELECT, INSERT, UPDATE e DELETE, no banco de dados **praticabd**, a sintaxe é:

```
REVOKE SELECT, INSERT, UPDATE, DELETE ON praticabd.* FROM aluno_E_A_D;
```

Figura 4.13: Comando *revoke*
Fonte: Manual de Referência do MySQL, 1997

No próximo exemplo temos a revogação de todos os privilégios concedidos ao usuário **aluno_E_A_D**, inclusive o privilégio de conceder privilégio a outros usuários, a sintaxe é:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM aluno_E_A_D;
```

Figura 4.14: Comando *revoke all*
Fonte: Manual de Referência do MySQL, 1997

Assista à vídeo-aula da disciplina de Prática de Banco de Dados disponível em http://cefetpi.nucleoead.net/etapi/. Aproveite para revisar os conteúdos da aula sobre **gatilhos (*triggers*)** e **controle de acesso**

Resumo

Nesta aula 4, conceituamos os gatilhos (*triggers*) que são uma adição valiosa ao banco de dados, podem ser utilizados para impor restrições de dados que são muito mais complexas do que restrições referenciais normais de integridade. Na verdade, os gatilhos (*triggers*) são executados quando um evento ocorre podendo ser uma inclusão, alteração ou exclusão ocorre em uma tabela do SGBD. Quando nos referimos a prática de banco de dados, não podemos deixar de focar no controle de acesso. No MySQL é necessário saber manipular o usuário do banco de dados, desde sua geração até saber como deletá-lo. Um usuário deve ter nível de permissões ou privilégios no SGBD para que possa executar suas tarefas. O ideal é que se um usuário só precisa escrever numa determinada tabela, dê a ele somente permissão pra escrever, se ele só precisa lê, então só dê a ele privilégio de lê. Portanto, quanto menos privilégio o usuário tiver para executar alguma tarefa, melhor para a segurança do MySQL.

Atividades de aprendizagem

1. Qual a vantagem de se usar um gatilho, em vez de se usar um procedimento ou uma função num SGBD?

2. No que se refere à utilização de gatilhos no MySQL, escolha a alternativa que contém o comando utilizado para se que possa mudar as características do gatilho:

   a) CREATE TRIGGER

   b) DROP TRIGGER

   c) ALTER TRIGGER

   d) CREATE TRIGGER FOR EACH ROW

   e) Nenhuma das alternativas anteriores

3. Utilizando o banco de dados **praticabd** da "empresa de desenvolvimento de projetos" da nossa apostila de Prática de Banco de Dados, elabore um gatilho. Explique a funcionalidade dele para a empresa. O código do gatilho deverá ser compilado e executado no MySQL, de acordo com o que for proposto pelo aluno.

4. A integridade de um Sistema de Banco de Dados depende muito do tipo de segurança implementada no SGBD, por isso tem que se saber como implementar o controle de acesso, para c, para que se possa melhorar a segurança no banco de dados. Então, analise as alternativas a seguir, e coloque (V) para verdadeiro e (F) para falso:

   ( ) Depois que criamos um usuário no banco de dados (MySQL) não podemos mais alterar a senha desse usuário.

   ( ) O comando DROP ALL PRIVILEGES, serve para revogar todos os privilégios do usuário do banco de dados (MySQL).

   ( ) O comando DROP USER nome-do-usuário serve para remover um usuário do banco de dados (MySQL).

( ) Existem alguns privilégios que são usados para operações administrativas que são realizadas pelo DBA (Administrador de Banco de Dados), são exemplo desse tipo: RELOAD, SHUTDOWN, PROCESS, FILE e ALL.

( ) O privilégio GRANT OPTION permite que um usuário possa fornecer a outros usuários os privilégios que ele mesmo possui.

5. Tratando sobre Controle de Acesso no MySQL, responda:

a) Criar um usuário novo sem senha;

b) Criar um usuário novo com a senha '**aba500**', o nome do usuário deve ser o seu 1º nome junto com sua idade. Por exemplo: Maria20;

c) Alterar a senha do usuário, criado na alternativa anterior, para '**500cba**';

d) Alterar o nome de seu usuário criado na alternativa b para seu sobrenome;

e) Remova o usuário criado na alternativa b.

6. Qual a importância do Controle de Acesso de Usuários num SGBD?

Poste suas respostas no AVEA.

Referências

COSTA, Rogério Luis de C. **SQL: Guia Prático.** São Paulo: Brasport, 2006.

DESCOBRE. **Tipos de Linguagem de Programação.** Disponível em: http://www.descobre.com/forum/showthread.php?t=697. Acesso em 23 mar. 2010.

LOUCOPOULOS, P. **Conceptual Modeling, in: P. Loucopouls, Conceptual Modeling, Databases, and CASE.** New York: John Wiley & Sons, 1992.

Manual MySQL 5.1. Disponível em: http://dev.mysql.com/doc/refman/5.1/en/index.html. Acesso em 24 mar. 2010.

NETO, Álvaro Pereira. **PostgreSql: técnicas avançadas.** São Paulo. Érica, 2003.

SQL. Disponível em: http://pt.wikipedia.org/wiki/Sql Acesso em 24 mar. 2010.

Currículo dos professores-autores

Thiago Alves Elias da Silva

Possui graduação em Tecnologia em Processamento de Dados pela Faculdade Piauiense de Processamento de Dados (2000), graduação em Engenharia Civil pela Universidade Federal do Piauí (2005), Especialização em Análise de Sistemas pela Universidade Estadual do Piauí (2001) e Mestrado em Engenharia de Produção - Tecnologia da Informação pela Universidade Federal do Rio Grande do Norte (2007). Atualmente doutorando em Ciência da Computação pela Universidade Federal Fluminense. Hoje é professor do Instituto Federal de Educação, Ciência e Tecnologia do Piauí. Tem experiência na área de Ciência da Computação, com ênfase em Ciência da Computação. *Curriculum Lattes*: **http://lattes.cnpq.br/7251217977577843**

Nádia Mendes dos Santos

Bacharel em Ciências Contábeis pela Universidade Federal do Piauí, Campus Ministro Reis Velloso, em Parnaíba, e Tecnólogo em Processamento de Dados pela Universidade Estadual do Piauí, Campus de Parnaíba. Pós-graduação "*Lato Sensu*" em Banco de Dados, pelo Centro Federal de Educação do Piauí. Mestre em Engenharia Elétrica pela Universidade Federal do Ceará. E atualmente faz Doutorado em Computação pela Universidade Federal Fluminense, linha de pesquisa em Inteligência Artificial. Desde 2006, ela é professora do quadro efetivo do Instituto Federal de Ciência e Tecnologia do Piauí, Campus Angical do Piauí, na categoria de Dedicação Exclusiva. Seus principais interesses de pesquisa e atuação são inteligência artificial, estrutura de dados, eletrônica digital, microprocessadores digitais e linguagens de programação. E faz parte da Equipe de Educação à Distância do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, atuando como tutora à distância do polo de Monsenhor Gil. *Curriculum Lattes*: **http:// lattes.cnpq.br/7392557956968627**

Wilson de Oliveira Junior

Graduado em Processamento de Dados pela Associação Piauiense de Ensino Superior do Piauí, com especialização em redes de computadores. Atua há mais de dez anos na área, desenvolvendo trabalhos e prestando consultoria. Possui vasta experiência acadêmica atuando como professor do ensino superior há mais de oito anos. Atualmente desenvolve pesquisas em redes PLC e redes *Wireless*.

rede e-Tec Brasil

ISBN 978-85-67082-07-3