PROGRAMAÇÃO E DESENVOLVIMENTO DE BANCO DE DADOS
Unidade I – Repositório de Dados

Anotações:

Linguagem de Definição de Dados (**DDL**, do inglês Data Definition Language)
Conjunto de instruções SQL para definição dos dados e sua estrutura.

- CREATE – cria banco de dados, tabelas, colunas.
- DROP – exclui banco de dados, tabelas, colunas.
- ALTER – altera banco de dados, tabelas, colunas.
- TRUNCATE – esvazia toda a tabela.

Linguagem de Manipulação dos Dados (**DML**, do inglês Data Manipulation Language)
Conjunto de instruções SQL para inserção e manutenção dos dados.

- INSERT – insere dados em uma tabela.
- UPDATE – atualiza os dados existentes em uma tabela.
- DELETE – exclui registros de uma tabela.

Linguagem de Consulta a Dados (**DQL**, do inglês Data Query Language)
Conjunto de instruções SQL para consulta de todos os dados armazenados e suas relações, e ajuda para comandos de sintaxe.

- SELECT – principal instrução de consulta do SQL.
- SHOW – exibe todas as informações além dos dados (metadata).
- HELP – exibe informações do manual de referência do MySQL.

Linguagem de Controle de Dados (**DCL**, do inglês Data Control Language)
Conjunto de instruções SQL para controle de autorizações de acesso e seus níveis de segurança.

- GRANT – essa instrução concede privilégios às contas de usuário.
- REVOKE – essa instrução permite revogar os privilégios da conta de usuário.

Linguagem de Transação de Dados (**DTL**, do inglês Data Transaction Language)
Conjunto de instruções para o controle de transações lógicas que são agrupadas e executadas pela DML.

- START TRANSACTION – inicia uma nova transação.
- SAVEPOINT – identifica um determinado ponto em uma transação.
- COMMIT – é uma instrução de entrega ao SGBD, fazendo com que todas as alterações sejam permanentes.
- ROLLBACK [TO SAVEPOINT] – é uma instrução ao SGBD para reverter toda a transação, cancelando todas as alterações ou até determinado ponto da transação.
- RELEASE SAVEPOINT – instrução para remoção de um SAVEPOINT.

Exemplo de consulta precisa:

- SELECT nome, nascimento, cpf FROM clientes WHERE cpf = ‘12345678901’;

Exemplo de consulta imprecisa:

- SELECT * FROM clientes;

Supondo que você deseja acessar o banco chamado “mundo”, para isso utilize primeiramente a instrução:

- USE mundo;

Para visualizar todas as definições de uma determinada tabela, como a tabela “cidade”, utilize essa instrução:

- SHOW COLUMNS FROM cidade;

Sempre que for necessário, utilize o **DISTINCT** para remover duplicidades em suas consultas:

- SELECT DISTINCT Nome
- FROM cidade;

Se você deseja que as duplicidades não sejam removidas:

- SELECT ALL Nome
- FROM cidade;

Outra opção é utilizar “*” para indicar “todos os campos”. Um exemplo está na sintaxe:

- SELECT *
- FROM cidade;

Na cláusula **SELECT**, você pode também incluir expressões aritméticas utilizando os operadores +, -, * e /. Como, por exemplo, na sintaxe:

- SELECT Nome, Populacao / 2
- FROM cidade;

“Encontre todos os nomes de cidades que tenham uma população inferior a 100000”. Para isso execute a sintaxe:

- SELECT Nome, Populacao
- FROM cidade
- WHERE Populacao < 100000;

Os conectivos **AND, OR e NOT** podem também ser utilizados como subqualificadores para o qualificador **WHERE**. Também pode- se utilizar operadores de comparação <, <= , > , >= , = e < > .

Para simplificar algumas operações utilizado o **WHERE**, existe o **BETWEEN**:

- SELECT Nome, Populacao
- FROM cidade
- WHERE Populacao BETWEEN 90000 AND 100000;

Em vez de:

- SELECT Nome, Populacao
- FROM cidade
- WHERE Populacao >= 90000 AND Populacao <= 100000;

Podemos, ainda, usar o operador de comparação **NOT BETWEEN**:

- SELECT Nome, Populacao
- FROM cidade
- WHERE Populacao NOT BETWEEN 60000 AND 70000;

Em uma consulta SQL, o resultado de uma consulta sem condições lógicas para delimitação envolvendo estas duas tabelas provocará a exibição de 1.000.000 de registros como resultado:

- SELECT * FROM cidade, pais;

A cláusula **FROM**

- SELECT cidade.Nome, cidade.Populacao, linguapais.Linguagem
- FROM cidade, pais, linguapais
- WHERE cidade.CodigoPais = pais.Codigo AND pais.Codigo = linguapais.CodigoPais;

- SELECT cidade.Nome, cidade.Populacao, linguapais.Linguagem
- FROM cidade, pais, linguapais
- WHERE cidade.CodigoPais = pais.Codigo AND pais.Codigo = linguapais.CodigoPais AND linguaPais.Linguagem = "Português";

Os campos podem ser renomeados e, para isso, usa-se a cláusula AS, da seguinte forma:
nome-antigo AS nome-novo

- SELECT Nome, Populacao as PopulacaoDaCidade
- FROM cidade;

- SELECT C.Nome, C.Populacao, L.Linguagem
- FROM cidade as C, pais as P, linguapais as L WHERE C.CodigoPais = P.Codigo
- AND P.Codigo = L.CodigoPais;

Operações de String e Ordenação

O operador LIKE determina a correspondência de padrões, que são descritos usando caracteres especiais:

1. porcentagem (%): corresponde a qualquer substring.
2. sublinhado (_): corresponde a qualquer

Para ilustrar, considere os seguintes exemplos:

1. ‘Sor%’ localizará qualquer string iniciando com “Sor”.
2. ‘%or%’ localizará qualquer string iniciando contendo “or”.
3. ‘ ’ localizará qualquer string com exatamente três
4. ‘ %’ localizará qualquer string com pelo menos três

- SELECT Nome
- FROM cidade
- WHERE Nome like ‘Sor%’;

“Encontre os nomes de todas as cidades na tabela cidade, ordenadas por nome”:

- SELECT Nome
- FROM cidade
- ORDER BY Nome;

“Encontre os nomes de todas as cidades na tabela cidade ordenados por nome em ordem decrescente”:

- SELECT Nome
- FROM cidade
- ORDER BY Nome DESC;

SP1:

- USE world;
- SHOW TABLES;
- SHOW COLUMNS FROM nome-tabela;
- SHOW COLUMNS FROM city;

“Encontre os nomes de todas as cidades na tabela cidade com nomes iniciados por ‘Sor’”:

- SELECT Name
- FROM city
  WHERE Name LIKE ‘Sor%’;

“Encontre os nomes e a população de todas as cidades com nomes iniciados por ‘Sor’”:

- SELECT Name, Population
- FROM city
- WHERE Name LIKE ‘Sor%’;

“Encontre os nomes, sua população e os países em que se encontram, para todas as cidades com nomes iniciados por ‘Sor’”:

- SELECT city.Name, city.Population, country.
- Name
- FROM city, country
- WHERE city.Name LIKE 'Sor%' AND city. CountryCode = country.Code;

Conjunto de caracteres (**CHARSET**) e agrupamentos (**COLLATION**)

1. A cláusula CHARSET, como o próprio nome indica, designa um conjunto de símbolos e codificações e como eles são representados binariamente.
2. Já a cláusula COLLATION é o conjunto de regras para comparação de caracteres em um conjunto de caracteres.

Para verificar quais conjuntos de caracteres estão instalados no seu MySQL, utilize a instrução:

- SHOW CHARACTER SET;

Essa instrução exibirá todos os conjuntos de caracteres, mas vamos analisar especificamente dois deles, o “latin1” e o UTF-8.

- SHOW CHARACTER SET WHERE charset LIKE ‘latin1’;
- SHOW CHARACTER SET WHERE charset LIKE ‘utf8’;

Para você exibir todas as instruções instaladas, utilize a seguinte instrução:

- SHOW COLLATION;

Para você visualizar o conjunto de regras do UTF-8, utilize a instrução:

- SHOW COLLATION WHERE collation LIKE 'utf8_ general_ci';

1. As letras “CI” (case insensitive) demonstram que, dentre todas as suas regras, não há distinção entre maiúsculas e minúsculas.
2. Já as letras “CS” (case sensitive) demonstram que, dentre todas as suas regras, há distinção entre maiúsculas e minúsculas.

Para criarmos um banco de dados, deveremos utilizar as instruções da classe da linguagem de definição de dados (**DDL**). A instrução a seguir tem essa finalidade:

- CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
- [create_specification] ... create_specification:
- [DEFAULT] CHARACTER SET [=] charset_name
- | [DEFAULT] COLLATE [=] collation_name

Para criarmos um banco de dados chamado “mundo”, com o **CHARSET UTF-8** e **COLLATION “utf8_general_ci”**, a instrução será a seguinte:

- CREATE DATABASE mundo
- DEFAULT CHARSET = utf8
- DEFAULT COLLATE = utf8_general_ci;

Se você utilizar a instrução de criação de banco de dados e o mesmo já existir, visualizará uma mensagem de erro:

- CREATE DATABASE IF NOT EXISTS mundo
- DEFAULT CHARSET = utf8
- DEFAULT COLLATE = utf8_general_ci;

Supondo que você deseja alterar o **CHARSET** ou **COLLATION** de seu banco de dados, você deve usar a instrução:

- ALTER {DATABASE | SCHEMA} [db_name]
- alter_specification ... alter_specification:
- [DEFAULT] CHARACTER SET [=] charset_name
- | [DEFAULT] COLLATE [=] collation_name

Para alterar o **CHARSET** do banco de dados mundo para “latin1”:

- ALTER DATABASE mundo CHARSET = latin1;

Apagará seu banco de dados:

- DROP {DATABASE | SCHEMA} [IF EXISTS] db_name

A cláusula **IF EXISTS** previne que não seja gerado um erro se a base não existir. A instrução com a cláusula no banco de dados world ficaria assim:

- DROP DATABASE IF EXISTS mundo;

A instrução para a criação de tabelas e sua estrutura é a **CREATE TABLE**:

- CREATE TABLE [IF NOT EXISTS] nome_tabela (
  Lista_campos
  );

Cria a tabela com o nome especificado. Se esta já existir e a cláusula **IF NOT EXISTS** for utilizada, não ocasionará um erro, apenas um alerta:

- nome_campo tipo_campo[tamanho] [NOT NULL|NULL] [DEFAULT valor] [AUTO_INCREMENT] [PRIMARY KEY]

MySQL: Tipos de dados

1) Tipos Numéricos

- SMALLINT [(M)] [UNSIGNED] [ZEROFILL] - Um número inteiro no intervalo de -32768 a O intervalo sem sinal é de 0 a 65535.
- MEDIUMINT [(M)] [UNSIGNED] [ZEROFILL] - Um número inteiro no intervalo de -8388608 a 8388607. O intervalo sem sinal é de 0 a 16777215.
- INT [(M)] [UNSIGNED] [ZEROFILL] - Um número inteiro no intervalo de -2147483648 a 2147483647. O intervalo sem sinal é de 0 a INTEGER é um sinônimo de INT.
- BIGINT [(M)] [UNSIGNED] [ZEROFILL] - Um número inteiro no intervalo de -9223372036854775808 a 9223372036854775807. O intervalo sem sinal é de 0 a 18446744073709551615.
- FLOAT [(M, D)] [UNSIGNED] [ZEROFILL] - Um número de ponto flutuante, de precisão Os valores admissíveis são -3,402823466E+38 a -1,175494351E-38, 0 e 1,175494351E-38 a 3,402823466E+38.
- DOUBLE [(M, D)] [UNSIGNED] [ZEROFILL] - Um número de ponto flutuante de precisão dupla. Os valores admissíveis são -1,7976931348623157E+308 a 2,2250738585072014E-308,0 e 2,2250738585072014E- 308 a 1,7976931348623157E+308. No lugar de DOUBLE, você também pode utilizar o sinônimo DOUBLE PRECISION.

2) Tipo Data e hora

- DATE - O intervalo suportado é '1000-01-01' a '9999-12-31'. O MySQL exibe valores DATE no formato 'YYYY-MM-DD', mas permite a atribuição de valores a colunas DATE usando *strings* ou números.
- DATETIME [(fsp)] - Uma combinação de data e hora. O intervalo suportado é '1000-01-01 00: 00: 000000' a '9999- 12-31 23: 59: 59.999999'. O MySQL exibe valores de DATETIME no formato 'AAAA-MM-DD HH: MM: SS [fração]', mas permite a atribuição de valores a colunas DATETIME usando *strings* ou números. Um valor ‘fsp’ opcional no intervalo de 0 a 6 pode ser dado para especificar a precisão dos segundos fracionários. Um valor de 0 significa que não há parte fracionária. Se o qualificador ‘fsp’ for omitido, a precisão padrão é 0.

- TIMESTAMP [(fsp)] - O intervalo é '1970-01-01 00: 00: 000000' UTC para '2038-01-19 03: 14: 07.999999' UTC. Os valores de TIMESTAMP são armazenados como o número de segundos desde a época ('1970-01-01 00:00:00' UTC).
- TIME [(fsp)] - O intervalo é '-838: 59: 59.000000' para '838: 59: 59.000000'. O MySQL exibe valores “TIME” no formato 'HH: MM: SS [fração]', mas permite a atribuição de valores a colunas “TIME” usando *strings* ou números.
- YEAR [(4)] - Um ano no formato de quatro dígitos. O MySQL exibe valores YEAR (ano) no formato YYYY, mas permite a atribuição de valores a colunas YEAR usando *strings* ou números. Os valores são exibidos de 1901 a 2155 e de 0000.

3) Tipo Texto

- CHAR [(M)] [CHARACTER SET charset_name] [COLLATE collation_name] - Uma cadeia de comprimento fixo que é sempre preenchida à direita com espaços para o comprimento especificado quando ‘M’ representa o comprimento da coluna em caracteres. O intervalo de M é de 0 a 255. Se M for omitido, o comprimento é 1.
- VARCHAR (M) [CONJUNTO DE CARACTERES charset_ name] [COLLATE collation_name] - Uma cadeia de comprimento variável. M representa o comprimento máximo da coluna em caracteres. O intervalo de M é de 0 a 65.535. O comprimento máximo efetivo de um VARCHAR está sujeito ao tamanho máximo da linha (65.535 bytes, que é compartilhado entre todas as colunas) e ao conjunto de caracteres utilizando.
- BINARY [(M)] - O tipo BINARY é semelhante ao tipo CHAR, mas armazena cadeias de bytes binários em vez de cadeias de caracteres não binários. Um comprimento opcional M representa o comprimento da coluna em Se omitido, M é padronizado como 1.
- VARBINARY (M) - O tipo VARBINARY é semelhante ao tipo VARCHAR, mas armazena cadeias de bytes binários em vez de cadeias de caracteres não binários. M representa o comprimento máximo da coluna em bytes.
- TINYBLOB - Uma coluna “BLOB” com um comprimento máximo de 255 (28 - 1) bytes. Cada valor TINYBLOB é armazenado usando um prefixo de comprimento de 1 byte, que indica o número de bytes no valor.

- TINYTEXT [CHARACTER SET charset_name] [COLLATE collation_name] - Uma coluna TEXT com um comprimento máximo de 255 (28 - 1) O comprimento máximo efetivo é menor, se o valor contiver caracteres multibyte. Cada valor TINYTEXT é armazenado usando um prefixo de comprimento de 1 byte, que indica o número de bytes no valor. BLOB [(M)] - Uma coluna “BLOB” com um comprimento máximo de 65.535 (216 - 1) bytes. Cada valor BLOB é armazenado usando um prefixo de comprimento de 2 bytes, que indica o número de bytes no valor.
- TEXT [(M)] [CHARACTER SET charset_name] [COLLATE collation_name] - Uma coluna TEXT (texto) com um comprimento máximo de 65.535 (216 - 1) caracteres. O comprimento máximo efetivo é menor se o valor contiver caracteres Cada valor “TEXT” é armazenado usando um prefixo de comprimento de 2 bytes, que indica o número de bytes no valor.
- MEDIUMBLOB - Uma coluna “BLOB” com um comprimento máximo de 777.215 (224 - 1) bytes. Cada valor “MEDIUMBLOB” é armazenado usando um prefixo de comprimento de 3 bytes, que indica o número de bytes no valor.
- MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name] - Uma coluna TEXT com um comprimento máximo de 777.215 (224 - 1) caracteres. O comprimento máximo efetivo é menor se o valor contiver caracteres multibyte. Cada valor MEDIUMTEXT é armazenado usando um prefixo de comprimento de 3 bytes, que indica o número de bytes no valor.
- LONGBLOB - Uma coluna “BLOB” com um comprimento máximo de 4.294.967.295 ou bytes de 4 GB (232 - 1). O comprimento máximo efetivo das colunas “LONGBLOB” depende do tamanho máximo de pacote configurado no protocolo cliente / servidor e da memória disponível.
- LONGTEXT [CHARACTER SET charset_name] [COLLATE collation_name] - Uma coluna TEXT com um tamanho máximo de 294.967.295 ou 4 GB (232 - 1) caracteres.
- ENUM ('valor1', 'valor2', ...) [CHARACTER SET charset_name] [COLLATE collation_name] - Um objeto de *string* que pode ter apenas um valor, escolhido na lista de valores 'valor1', 'valor2', [...], NULO, ou o valor de erro especial ''. Valores ENUM são representados internamente como números inteiros.
- SET ('valor1', 'valor2', ...) [CHARACTER SET charset_name] [COLLATE collation_name] - Um objeto de *string* que pode ter zero ou mais valores, e cada

um dos quais deve ser escolhido na lista de valores 'valor1', 'valor2', [...]. Valores SET são representados internamente como números inteiros.

Para a criação de um campo tipo JSON, a instrução SQL poderia ser:

- CREATE TABLE cinema
- (id INT AUTO_INCREMENT PRIMARY KEY,
  filme JSON
  );

O campo **filme** poderá armazenar conteúdos JSON como:

- [
  {
  “titulo”: “JSON Zorro”,
  “resumo”: “a história da lenda do velho oeste”, “ano”: 2014,
  “genero”: [“ação”, “ficção”]
  }
  ]

Para exemplificar a criação de uma tabela, vamos criar uma chamada de “convidado”:

- CREATE TABLE convidados (
  id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(30) NOT NULL,
  sobrenome VARCHAR(30) NOT NULL, email VARCHAR(50),
  data_reg DATETIME, nascimento DATE
  );

A criação de todas as tabelas necessárias para a definição de seu banco de dados determinará a sua estrutura, índices e relacionamentos. Quando tudo isso estiver finalizado, você terá criado o modelo de dados ou **metadados**:

```
1    CREATE TABLE IF NOT EXISTS pais (
2    id INT(11) NOT NULL AUTO _ INCREMENT PRIMARY KEY,
3    nome VARCHAR(50) NOT NULL DEFAULT '',
4    continente ENUM('Ásia', 'Europa', 'América', 'África',
```

```
5       'Oceania', 'Antártida')
6       NOT NULL DEFAULT 'América',
7       codigo CHAR(3) NOT NULL DEFAULT '',
8       );
9       CREATE TABLE IF NOT EXISTS estado (
10      id INT(11) NOT NULL AUTO _ INCREMENT PRIMARY KEY,
11      nome VARCHAR(50) NOT NULL DEFAULT '',
12      sigla CHAR(2) NOT NULL DEFAULT '',
13      );
14      CREATE TABLE IF NOT EXISTS cidade (
15      id INT(11) NOT NULL AUTO _ INCREMENT PRIMARY KEY,
16      nome VARCHAR(50) NOT NULL DEFAULT '',
17      populacao INT(11) NOT NULL DEFAULT '0',
18      );
19      CREATE TABLE IF NOT EXISTS ponto _ tur (
20      id INT(11) NOT NULL AUTO _ INCREMENT PRIMARY KEY,
21      nome VARCHAR(50) NOT NULL DEFAULT '',
22      populacao INT(11) NOT NULL DEFAULT '0',
23      tipo ENUM('Atrativo', 'Serviço', 'Equipamento',
24      'Infraestrutura', Instituição', 'Organização'),
25      publicado ENUM('Não', 'Sim') NOT NULL DEFAULT 'Não'
26      );
```

Para armazenar as coordenadas de GPS:

- CREATE TABLE IF NOT EXISTS coordenada (latitude FLOAT(10,6), longitude FLOAT(10,6)
  );

UNIDADE II – Manipulação de dados e estruturas

Anotações:

Linguagem de manipulação de dados (**DML**)

A cláusula **SELECT**

- SELECT * FROM convidado;

  Toda a tabela será exibida.

- SELECT * FROM convidado WHERE nome LIKE ‘A%’;

  Exibirá todos os convidados com o nome iniciando com ‘A’.

Instrução de Inserção (**INSERT**)

- INSERT

  [INTO] nome_tabela

  [(nome_coluna [, nome_coluna] ...)]

  {VALUES | VALUE} (lista_valores) [, (lista_ valores)] ...

A declaração **VALUES** especificará os valores explícitos para serem inseridos. Caso a instrução seja utilizada sem parâmetros:

- INSERT INTO nome_tabela () VALUES ();

Se a lista de colunas e a lista **VALUES** estiverem vazias, a instrução determinará que o MySQL crie uma linha com cada coluna configurada com seus valores padrões.

- INSERT INTO nome_tabela (col1, col2) VALUES(15, col1*2);

Você também pode utilizar a declaração **VALUES** para inserir múltiplas linhas.

- INSERT INTO nome_tabela (a,b,c)
- VALUES(1,2,3),(4,5,6),(7,8,9);

Para exemplificar algumas instruções, imagine uma tabela criada com a seguinte instrução:

- CREATE TABLE IF NOT EXISTS convidado (

  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,

  nome VARCHAR(50) NOT NULL DEFAULT '',

  nascimento DATE,

  estudante ENUM('Não', 'Sim') NOT NULL DEFAULT 'Não'

  );

Ao utilizar a instrução abaixo, você estará inserindo uma linha à tabela “convidado”, incluindo nas colunas “nome”, “nascimento” e “estudante” os respectivos valores da cláusula **VALUES**.

- INSERT INTO CONVIDADO (nome, nascimento, estudante)
- VALUES ('Dani Moura', '1979-03-28', 'Sim');

Para verificar o resultado, utilize a sintaxe a seguir. O resultado será visualizado conforme mostra a figura.

- SELECT * FROM convidado;

Para atribuir explicitamente um valor nulo, a instrução deverá ser:

- INSERT INTO CONVIDADO (nome, nascimento, estudante)
- VALUES ('Rui Albuquerque', null , 'Sim');

Instrução de Atualização (**UPDATE**)

UPDATE é uma instrução DML que altera ou atualiza linhas em uma tabela.

- UPDATE tabela_referência
    - SET lista_atribuição
    - [WHERE condição]
    - [ORDER BY ...]
    - [LIMIT quantidade_linhas]
- value:
    - {expr | DEFAULT}
- assignment:
    - nome_coluna = valor
- lista_atribuições:
    - atribuição [, atribuição] ...

A instrução **UPDATE** atualiza colunas de linhas existentes na tabela nomeada com novos valores. Já a cláusula **SET** indica quais colunas modificar e os valores que devem ser fornecidos.

- UPDATE convidado
    - SET estudante = 'Sim'
    - WHERE nome = 'Lebrencio Grulher'
    - AND nascimento = '08 Jul-1990';

Se fornecida, a cláusula **WHERE** especifica as condições que identificam quais linhas devem ser atualizadas. Fique atento, pois se não for apresentada nenhuma cláusula **WHERE**, todas as linhas serão atualizadas.

A instrução a seguir exemplifica a cláusula **ORDER BY**:~

- UPDATE convidado
    - SET estudante = 'Sim'
    - WHERE nascimento < '08-Jul-1990'
    - ORDER BY nome;

Quando adicionamos um limite por meio da cláusula **LIMIT**, o que teremos é uma limitação nas atualizações.

- UPDATE convidado
    - SET estudante = 'Sim'
    - WHERE nascimento < '08-Jul-1990'
    - LIMIT 10
    - ORDER BY nome;

Se você acessar uma coluna da tabela para ser atualizada em uma expressão, **UPDATE** usará o valor atual da coluna.

- UPDATE tabela_ref SET col1 = col1 + 1;

ORDER BY para fazer com que as linhas com valores de ID maiores sejam atualizadas antes daquelas com valores menores:

- UPDATE tabela SET ID = ID + 1 ORDER BY ID
    - DESC;

Você também pode executar operações **UPDATE** abrangendo várias tabelas. No entanto, você não pode usar **ORDER BY** ou **LIMIT** com um **UPDATE** de várias tabelas.

- UPDATE lista, produto SET lista.preco = produto.preco
    - WHERE lista.id = produto.id;

Instrução de Exclusão (DELETE)

DELETE é uma instrução DML que exclui linhas de uma tabela. Sua sintaxe geral é dada por:

- DELETE FROM nome_tabela
    - [WHERE condição]
    - [ORDER BY ...]
    - [LIMIT quantidade_linhas]

Assim como vimos antes para a instrução **UPDATE**, as condições na cláusula **WHERE** irão identificar quais linhas serão excluídas.

- DELETE FROM convidados
    - WHERE estudante = 'Sim'
    - ORDER BY nome
    - LIMIT 10;

Se a instrução DELETE incluir uma cláusula ORDER BY, as linhas serão excluídas na ordem especificada pela cláusula.

- DELETE FROM log_usuario WHERE usuario = 'rm'
    - ORDER BY datahora_acao LIMIT 1;

Alteração de tabelas (ALTER TABLE)

A sintaxe da instrução é:

- ALTER TABLE nome_tabela
    - [especificação_alteração [,especificação_alteração] ...]

No quadro a seguir, você tem os principais qualificadores de especificação_alteração que podem ser utilizados:

- opções _ tabela
    - ADD [COLUMN] (nome _ coluna column _ definition,...)
    - ADD [COLUMN] nome _ coluna column _ definition
    - [FIRST | AFTER nome _ coluna]
    - ADD {INDEX|KEY} [índice _ nome]
    - [índice _ tipo] (índice _ nome _ coluna,...) [índice_opção] ...
    - ALTER [COLUMN] nome _ coluna {SET DEFAULT literal | DROP
    - DEFAULT}
    - ALTER INDEX índice _ nome {VISIBLE | INVISIBLE}
    - CHANGE [COLUMN] old _ nome _ coluna new _ nome _ coluna column _ definition
    - [FIRST|AFTER nome _ coluna]
    - [DEFAULT] CHARACTER SET [=] charset _ nome [COLLATE [=] collation nome]
    - CONVERT TO CHARACTER SET charset _ nome [COLLATE collation _ nome]
    - {DISABLE|ENABLE} KEYS
    - {DISCARD|IMPORT} TABLESPACE
    - DROP [COLUMN] nome _ coluna
    - DROP {INDEX|KEY} índice _ nome
    - DROP PRIMARY KEY
    - MODIFY [COLUMN] nome _ coluna column _ definition
    - [FIRST | AFTER nome _ coluna]
    - ORDER BY nome _ coluna [, nome _ coluna] ...
    - RENAME COLUMN old _ nome _ coluna TO new _ nome _ coluna
    - RENAME {INDEX|KEY} old _ índice _ nome TO new _ índice _nome
    - RENAME [TO|AS] new _ tbl _ nome

Por exemplo, para descartar várias colunas em uma única instrução, execute o seguinte comando:

- ALTER TABLE cliente DROP COLUMN parentesco, DROP COLUMN telefones;

Para você alterar o valor do campo auto incremento de uma tabela, você utilizará a instrução:

- ALTER TABLE cliente AUTO_INCREMENT = 13;

Após a execução dessa instrução, todas as tabelas criadas estarão utilizando o charset UTF-8, com exceção da tabela “pessoas”.

- ALTER TABLE pessoas CHARACTER SET = latin1;

Adicionando e excluindo colunas (ADD e DROP)

A instrução para adicionar um campo, por exemplo o campo nome como o tipo especificado na tabela “pessoas”, será realizada por:

- ALTER TABLE pessoas ADD nome VARCHAR(50);

Para excluir o campo “sobrenome” da mesma tabela a instrução seria:

- ALTER TABLE pessoas DROP COLUMN sobrenome;

Renomeando, redefinindo e reordenando colunas (CHANGE, MODIFY, RENAME COLUMN e ALTER)

Para alterar uma coluna a fim de alterar seu nome e sua definição, use **CHANGE**, especificando os nomes antigos e novos e a nova definição.

- ALTER TABLE pessoas CHANGE antigo novo BIGINT NOT NULL;

MODIFY é mais conveniente para alterar a definição sem alterar o nome por que requer o nome da coluna apenas uma vez.

- ALTER TABLE pessoas MODIFY novo INT NOT NULL;

RENAME COLUMN: pode alterar o nome de uma coluna, mas não sua definição, sendo um comando mais conveniente do que **CHANGE** para renomear uma coluna sem alterar sua definição pois exige apenas os nomes antigos.

- ALTER TABLE pessoas RENAME COLUMN novo TO antigo;

Usando restrições (constraints)

Você pode identificar, numa cláusula da instrução CREATE TABLE ou ALTER TABLE, um campo como PRIMARY KEY.

- CREATE TABLE pessoa(

  id int NOT NULL PRIMARY KEY,

  nome varchar(255) NOT NULL,

  sobrenome varchar(255), idade int

  );

Agora, vamos supor que você deseja alterar sua chave primária, ela conterá duas colunas e não apenas uma. Primeiro você deverá retirar a chave primária declarada:

- ALTER TABLE pessoa
  - DROP PRIMARY KEY;

Após isso, será necessário a nomeação de uma restrição PRIMARY KEY, e para defini-la em várias colunas, use a seguinte sintaxe SQL:

- ALTER TABLE pessoa
  - ADD CONSTRAINT PK_pessoa PRIMARY KEY (id, sobrenome);

A sintaxe essencial para uma definição de restrição de chave estrangeira em uma instrução CREATE TABLE ou ALTER TABLE é semelhante à apresentada no quadro:

- ALTER TABLE tbl_name
  - ADD [CONSTRAINT [símbolo]] FOREIGN KEY

    [index_nome] (index_col_nome, ...) REFERENCES

    tbl_nome (index_col_nome,...)

    [ON DELETE referências] [ON UPDATE

    referências]
- referências:
  - RESTRICT | CASCADE | SET NULL | NO ACTION

Na tabela de referência, deve haver um índice em que as colunas de chave estrangeira sejam listadas como as primeiras colunas na mesma ordem.

- CREATE TABLE pai (
  id INT NOT NULL,

nome VARCHAR(50),
PRIMARY KEY (id)
);

- CREATE TABLE filha (
  id INT PRIMARY KEY,
  parente_id INT,
  nome VARCHAR(50)
  );

Por exemplo, vamos criar uma integridade referencial entre as tabelas, utilizando esse relacionamento por meio da seguinte instrução:

- ALTER TABLE filha
  - ADD CONSTRAINT FK_parente
  - FOREIGN KEY (parente_id) REFERENCES pai(id);

O MySQL suporta opções sobre a ação a ser tomada, listadas aqui:

CASCADE: esse qualificador exclui ou atualiza a linha da tabela pai e exclui ou atualiza automaticamente as linhas correspondentes na tabela filha. Tanto o qualificador **ON DELETE CASCADE** como o **ON UPDATE CASCADE** são suportados. Entre duas tabelas, não devem ser definidas várias cláusulas ON **UPDATE CASCADE** para atuarem na mesma coluna na tabela pai ou na tabela filha.

SET NULL: esse qualificador exclui ou atualiza a linha da tabela pai e define como **NULL** a coluna ou colunas de chave estrangeira na tabela filha. Ambas as cláusulas **ON DELETE SET NULL** e **ON UPDATE SET NULL** são suportadas. Você deve ter o cuidado de certificar-se de que, ao especificar uma ação **SET NULL**, não tenha declarado as colunas na tabela filha como **NOT NULL**.

RESTRICT: rejeita a operação de exclusão ou atualização da tabela pai. Especificar **RESTRICT** (ou **NO ACTION**) é o mesmo que omitir a cláusula **ON DELETE** ou **ON UPDATE**.

NO ACTION: essa é uma palavra-chave do SQL padrão. No MySQL, a equivalente é a **RESTRICT**. O servidor MySQL rejeita a operação de exclusão ou atualização para a tabela pai, se houver um valor de chave estrangeira relacionado na tabela Alguns sistemas de banco de dados possuem cheques diferidos, e **NO ACTION** é um cheque adiado. No MySQL, as restrições de chave estrangeira são verificadas imediatamente, portanto, **NO ACTION** é o mesmo que **RESTRICT**.

Incluir, alterar e excluir dados

A restrição **FOREIGN KEY** é usada para impedir ações que destruam links entre tabelas, além de impedir que dados inválidos sejam inseridos na coluna de chave estrangeira, já que ela deve ser um dos valores contidos na tabela para a qual ela aponta.

No quadro temos essa estrutura criada com algumas instruções.

- CREATE TABLE aluno (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  nome CHAR(50) NOT NULL
  );
- CREATE TABLE curso (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  nome CHAR(50) NOT NULL
  );
- CREATE TABLE nota (
  aluno_id INT NOT NULL,
  curso_id INT NOT NULL,
  dataavaliacao DATE NOT NULL,
  nota DOUBLE NOT NULL,
  PRIMARY KEY(aluno_id, curso_id, dataavaliacao),
  INDEX i2 (curso_id),
  FOREIGN KEY (aluno_id) REFERENCES aluno(id) ON DELETE CASCADE,
  FOREIGN KEY (curso_id) REFERENCES curso(id) ON DELETE RESTRICT
  );

Ao ser executada a instrução abaixo, será criada uma tabela de nome “nota”, com exibição dos resultados:

- SHOW CREATE TABLE nota;

Na tela referenciada na figura acima, ao clicar com o botão direito do mouse sobre a linha acima, na opção **COPY FIELD UNQUOTED** teremos como resultado a instrução no quadro, apresentado a seguir:

- CREATE TABLE `nota` (
  `aluno_id` int(11) NOT NULL,
  `curso_id` int(11) NOT NULL,
  `dataavaliacao` date NOT NULL,

```
`nota` double NOT NULL,
PRIMARY KEY (`aluno_id`,`curso_id`,`dataavaliacao`),
KEY `i2` (`curso_id`),
CONSTRAINT `nota_ibfk_1` FOREIGN KEY (`aluno_id`)
REFERENCES
`aluno` (`id`) ON DELETE CASCADE,
CONSTRAINT `nota_ibfk_2` FOREIGN KEY (`curso_id`)
REFERENCES
`curso` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Instrução DROP TABLE

Quando você deseja remover uma ou mais tabelas, a instrução **DROP TABLE** deve ser utilizada.

- DROP TABLE [IF EXISTS] nome_tabela [, nome_ tabela] ...

Se você utilizar a cláusula **IF EXISTS** em vez de um erro, um alerta é gerado para cada tabela inexistente. Para utilizar a cláusula (**IF EXISTS**) em nosso exemplo, averiguando a existência da tabela “aluno” antes de executar o comando **DROP**, poderemos utilizar a instrução a seguir:

- DROP TABLE IF EXISTS aluno;

Para instruir o MySQL para excluir a restrição da tabela “nota”, que faz referência à tabela “aluno” (pai), você deve executar:

- ALTER TABLE nota DROP FOREIGN KEY nota_ibfk_1;

A instrução para esvaziar uma tabela completamente é a **TRUNCATE TABLE**, e sua sintaxe é:

- TRUNCATE [TABLE] nome_tabela;

A instrução **TRUNCATE TABLE** falha para uma tabela, se houver alguma restrição **FOREIGN KEY** de outras tabelas que a referenciam, já restrições de chaves estrangeiras entre colunas da mesma tabela são permitidas.

O campo **AUTO_INCREMENT** é o campo que vai automaticamente sendo incrementado em cada registro, funcionando naturalmente como chave primária, caso o usuário deseje. Qualquer valor **AUTO_ INCREMENT** é redefinido para seu valor inicial.

- CREATE TABLE IF NOT EXISTS ponto_tur (
  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(50) NOT NULL DEFAULT '',
  populacao INT(11) NOT NULL DEFAULT '0',
  tipo ENUM('Atrativo', 'Serviço', 'Equipamento', 'Infraestrutura', 'Instituição', 'Organização'),
  publicado ENUM('Não', 'Sim') NOT NULL DEFAULT 'Não'
  );

Ao executar essa instrução, o MySQL irá verificar antes da exclusão quais são as restrições impostas na estrutura do banco de dados:

- SET FOREIGN_KEY_CHECKS = 1;

Ao executar a instrução **SET FOREIGN_KEY_CHECKS = 0;,** o MySQL está sendo instruído a ignorar quaisquer restrições que estejam criadas. Em seguida, executando a instrução **DROP TABLE IF EXISTS curso;**, a exclusão da tabela “curso” ocorrerá com sucesso.

UNIDADE 3 - Consultas avançadas

Anotações:

Banco de dados do tipo relacional

Segundo Silberschatz (2010), as condições para se efetuar uma junção dependem diretamente do tipo de junção e de uma condição de junção, dessa forma, com o SQL, será possível retornar relações como resultados.

Para isso, temos que:

- **tipo de junção**: define/trata as tuplas em cada uma das relações que não corresponda a alguma das tuplas da outra relação, sendo dividido em relação interna, com o comando INNER JOIN, e relações externas, LEFT JOIN, RIGHT JOIN e FULL JOIN.
- **condição de junção**: define se as tuplas nas duas relações são correspondentes, garantindo que os atributos utilizados em ambas as tabelas estejam presentes tanto na sintaxe SQL como nos seus resultados.

Parâmetro JOIN

Segundo Milani (2007), com a utilização do comando JOIN (Junção) é possível, por meio do SELECT, unir duas ou mais tabelas, ao se apontar os campos correspondentes entre elas.

A sintaxe utilizada para se efetuar junções nas consultas em SQL é definida como:

- SELECT[campo] FROM [tabela_1>**JOIN**<tabela_2] ON

  [tabela_1].[chave_primária] =

  [tabela_2].[chave_estrangeira]

  WHERE [condição];

INNER JOIN

No exemplo utilizado ao longo desta aula, em que temos a relação entre categorias e produtos, se quisermos efetuar uma consulta que nos retorne o nome da categoria e seus respectivos nomes dos produtos, deve ser utilizado o seguinte comando:

- SELECT categoria.nome, produto.nome FROM Catego- ria **INNER JOIN**

  Produto

  ON Categoria.Id = Produto.Id_Categoria;

LEFT JOIN

Conforme afirma Silberschatz (2010), no comando LEFT JOIN, as linhas da tabela da esquerda são projetadas na seleção juntamente com as linhas não combinadas da tabela da direita. Ou seja, como resultado dessa seleção, algumas linhas em que não haja relacionamento entre as tabelas da esquerda para a direita retornarão o valor nulo (NULL).

Para isso, o SQL utiliza a seguinte sintaxe:

- SELECT [campo] FROM [tabela_1] **LEFTJOIN** [tabela_2] ON [tabela_1].[chave_primária] =

  [tabela_2].[chave_estrangeira]

  WHERE [condição];

RIGHT JOIN

Conforme afirma Silberschatz (2010), similar ao comando LEFT JOIN, com o comando RIGHT JOIN as linhas da tabela da direita são projetadas na seleção juntamente com as linhas não combinadas da tabela da esquerda. Para isso, o SQL utiliza a seguinte sintaxe:

- SELECT [campo] FROM [tabela_1] **RIGHTJOIN** [tabela_2]

  ON [tabela_1].[chave_primária] =

  [tabela_2].[chave_estrangeira]

  WHERE [condição];

As funções de agregação em SQL

Date (2000) diz que os resultados das seleções podem ser organizados em grupos, baseados no conteúdo existente em uma ou mais colunas.

Para isso, deve ser utilizada a palavra GROUP BY na seguinte sintaxe SQL:

- SELECT [coluna]

  FROM [tabela]

  GROUP BY [coluna];

É possível, ainda, utilizar a clausula WHERE com o agrupamento, com a seguinte sintaxe SQL:

- SELECT [coluna]

  FROM <tabela]

  WHERE [condição]

  GROUP BY [coluna];

COUNT

Segundo Silberschatz (2010), a função agregada **COUNT** permite que se possa contar o número de registros de uma relação. A sintaxe SQL utilizada para tal função é descrita como:

- SELECT **COUNT**(*) FROM <tabela>;

Temos que tomar cuidado ao utilizar a função COUNT, pois o contador ignora os registros em que haja valor nulo (NULL)

Outro problema encontrado está na utilização da função COUNT para contar a quantidade de marcas de carro cadastrados. Certamente o resultado retornado seria doze, ou seja, a quantidade de veículos registrados na tabela.

Para isso, o SQL possui a função **DISTINCT**, que é utilizada juntamente com o COUNT, e, no exemplo acima, teríamos a seguinte sintaxe:

- SELECT COUNT(**DISTINCT** MARCA) FROM Veiculos;

MINIMUM (MIN)

Segundo Silberschatz (2010), a função agregada MIN permite que se possa determinar o menor valor de registro em uma coluna. A sintaxe SQL utilizada para tal função é descrita como:

- SELECT MIN(<coluna>) FROM <tabela>;
- SELECT Marca, Modelo, MIN(Valor) as “Menor Valor” FROM Veiculos;

MAXIMUM (MAX)

Segundo Silberschatz (2010), analogamente à função MIN, o recurso MAX permite que se possa determinar o maior valor de registro em uma coluna. A sintaxe SQL utilizada para tal função é descrita como:

- SELECT MAX(<coluna>) FROM <tabela>;
- SELECT Marca, Modelo, MAX(Valor) as “Maior Valor” FROM Veiculos;

AVERAGE (AVG)

Segundo Silberschatz (2010), a função AVG (abreviação do termo em inglês *average*, que quer dizer média) retorna a média dos valores em uma determinada coluna. Para isso, o SQL faz a somatória dos valores (obrigatoriamente numéricos) e divide o resultado pelo número de registros diferentes de nulo (NULL).

A sintaxe SQL utilizada para tal função é descrita como:

- SELECT AVG(<coluna>) FROM <tabela>;
- SELECT AVG(Valor) as “Valor Médio” FROM Veiculos;

A função de agregação AVG permite que o qualificador GROUP BY seja utilizado em conjunto. Para isso, vamos tomar um exemplo em que se deseja selecionar as marcas e o valor médio dos veículos conforme as marcas, agrupando as informações, com o comando SQL descrito a seguir:

- SELECT Marca, AVG(Valor) as “Valor Médio”
- FROM Veiculos
- GROUP BY Marca;

TOTAL (SUM)

Segundo Silberschatz (2010), a função SUM retorna o somatório dos valores em uma determinada coluna. Para isso, o SQL faz o somatório dos valores (obrigatoriamente numéricos). A sintaxe SQL utilizada para tal função é descrita como:

- SELECT SUM(<coluna>) FROM <tabela>;
- SELECT SUM(Valor) as “Total” FROM Veiculos;

Técnicas de subconsultas

Os atributos com **sublinhado simples**, são as chaves primária das tabelas. Já as colunas com **duplo sublinhado**, são as chaves estrangeiras da tabela.

A sintaxe presente no SQL permite que sejam feitas consultas usando diversas relações entre inúmeras tabelas presentes nos bancos de dados, bastando a utilização dos conectivos. Para isso, temos o conectivo **IN**, que efetua o teste no conjunto de dados, que é fruto de uma coleção de valores produzidos por meio de um SELECT, e o conectivo **NOT IN,** que permite efetuar a ausência em um conjunto de valores.

A sintaxe utilizada com o conectivo IN pode ser definida como:

- SELECT [campo]

  FROM [tabela]

  WHERE [campo] **IN** (SELECT [campo] FROM [tabela]);

- SELECT aluno.nome

  FROM aluno

  WHERE aluno.RA **IN** (SELECT aluno_RA FROM restrição);

O conectivo **NOT IN** é utilizado de maneira semelhante, conforme pode ser observado na sintaxe a seguir:

- SELECT [campo]

  FROM [tabela]

  WHERE [campo] **NOT IN** (SELECT [campo] FROM [tabela]);
- SELECT aluno.nome as “ALUNO”

  FROM aluno

  WHERE aluno.RA NOT IN (SELECT aluno_RA FROM

  emprestimo);

Comparação de Conjuntos

Segundo Date (2004), a sintaxe SQL permite o desenvolvimento de subconsultas aninhadas, em que é possível fazer a comparação entre conjuntos de dados, utilizando condições (WHERE).

Operador Matemático	SELECT com WHERE SQL	Subconsulta SQL
=	WHERE campo = condição	WHERE campo = some (SELECT)
≠	WHERE campo <> condição	WHERE campo <> some (SELECT)
>	WHERE campo > condição	WHERE campo > some (SELECT)
≥	WHERE campo >= condição	WHERE campo >= some (SELECT)
<	WHERE campo < condição	WHERE campo < some (SELECT)
≤	WHERE campo <= condição	WHERE campo <= some (SELECT)

- select nome as "Livro", secao as "Seção"

  from livro

  where nome> some (select nome from livro where secao = “esoterismo”);

UNIDADE 4 - Recursos avançados e automação de processos

Visões e índices

Uma das técnicas que podem auxiliar na diminuição na carga de processamento é a **VIEW**. Vamos entender o porquê? Utilizar uma **VIEW** para efetuar seleção de dados, torna as consultas mais rápidas e exige uma carga de processamento menor. Isso ocorre por que uma **VIEW** não necessita fazer o retrabalho ao executar um **SELECT**, pois a seleção já está pré-armazenada.

A sintaxe utilizada para desenvolver uma **VIEW** está descrita a seguir:

- CREATE VIEW [nome_da_VIEW] AS

  SELECT [coluna]

  FROM [tabela]

  WHERE [condições];

Exemplo:

- CREATE VIEW v_select1 AS

  SELECT veiculo.nome as “Veiculo”, fabricante.

  marca as “Marca”, veiculo.cor as “Cor”, veiculo.

  preco as “Valor”

  FROM veiculo INNER JOIN fabricante

  WHERE veiculo.fabricante_Codigo = fabricante.

  Codigo AND veiculo.preco<= 50000;

Para utilizarmos uma **VIEWS** para exibir uma consulta, devemos utilizar a sintaxe a seguir:

- SELECT * FROM [nome_da_VIEW];

Em nosso exemplo desenvolvido anteriormente, teríamos:

- SELECT * FROM v_select1;

Para excluir uma VIEW, a sintaxe deve ser:

- DROP VIEW [nome_da_VIEW];

Índices em banco de dados relacional

Segundo Silberschatz (2010), o recurso de índice (**INDEX**, no MySQL) não era admitido até a versão SQL:1999. Após isso, os engenheiros buscaram um recurso para diminuir a taxa de processamento nas buscas nas tabelas e para imposição das restrições de integridade.

Para isso, por meio da palavra reservada **INDEX** devemos utilizar as seguintes sintaxes:

Declarar um índice, quando no desenvolvimento da tabela:

- CREATE TABLE [nomeDaTabela] (

  Campo1 tipo(tamanho),

  Campo2 tipo(tamanho),

  INDEX(Campo1)

  );

Declarar um índice, em uma tabela existente no BD:

- CREATE TABLE [nomeDoIndice] ON

  [nomeDaTabela](Campo);

Exemplo:

- create index idx_Renavam ON

  veiculo(RENAVAN);

Embora o MySQL retorne a mensagem "*Query OK, 0 rows affected*", para nos certificarmos que os índices foram criados, é necessário utilizar a sintaxe a seguir:

- SHOW INDEX FROM [nomeDaTabela];

Para utilizar um índice, a sintaxe necessária pode ser observada a seguir:

- SELECT [coluna] FROM [nomeDaTabela]

  USE INDEX (nomeDoIndice)

  WHERE [condições];

No exemplo desenvolvido:

- SELECT nome AS "Veiculo", cor AS "Cor",

  Preço AS "Valor" FROM veiculo

  USE INDEX(idx_Renavam)

  WHERE preco <= 50000;

FULLTEXT em banco de dados relacional

Silberschatz (2010) afirma que esse recurso tem a capacidade de buscar um trecho dentro de várias *strings*, assim como a função “localizar” existente nos navegadores de internet, editores de texto, etc. Para isso, é utilizada a sintaxe:

- ALTER TABLE [nome_tabela] ADD FULLTEXT (nome_ da_coluna);

Para utilizar esse recurso, deve-se utilizar a sintaxe descrita a seguir:

- SELECT [coluna] from nome_da_tabela

  WHERE MATCH(coluna) AGAINST(‘palavra_ desejada’);

Controle transacional

Controle transacional com o comando COMMIT

É possível que as confirmações sejam determinadas pelo administrador do banco de dados. Para isso, o seguinte comando dever ser digitado no SGBD:

- SET AUTOCOMMIT=0;

Para que o **UPDATE** feito na tabela fique gravado (persistência), após a transação ser confirmada, é necessário utilizar o **COMMIT**.

O comando é simples e direto:

- COMMIT;

Vale ressaltar que, para os controles transacionais **SAVEPOINT** e **ROLLBACK** funcionarem, deve-se alterar o valor do **AUTOCOMMIT** para zero, pois a grande maioria dos sistemas de gerenciamento de banco de dados vem como valor padrão do **AUTOCOMMIT** igual a um, isso significa que, quando ocorre uma exclusão de dados, alteração de dados, inserção de dados ou criação de **SAVEPOINT**, o **COMMIT** grava as alterações no banco e apaga o ponto criado (**SAVEPOINT**).

Para alterar o valor do **AUTOCOMMIT**, deve ser utilizada a sintaxe SQL:

- SET AUTOCOMMIT=0;

Controle transacional com o comando ROLLBACK

Segundo Date (2012), para que uma transação feita no banco de dados possa ser revertida, é possível utilizar o recurso de **ROLLBACK** para retornar ao estado anterior da transação.

Para que possamos retornar a determinado ponto com o **ROLLBACK**, o SQL permite a criação de pontos de restauração, por meio da sintaxe:

- SAVEPOINT [nomeDoPonto];

Posteriormente, para utilizar esse ponto, deve ser usada a sintaxe:

- ROLLBACK TO SAVEPOINT [nomeDoPonto];

Com isso, vamos criar um ponto de salvação, por meio do comando:

- SET AUTOCOMMIT=0;
- SAVEPOINT status1;

Para retornar o nome do enfermeiro, o ponto de salvação deve ser retornado por meio do comando:

- ROLLBACK to savepoint status1;

Procedimentos e funções

Funções (FUNCTION)

A sintaxe SQL para criar uma função é definida como **FUNCTION** (DATE, 2012), e para ser estruturada em um banco de dados, utiliza- se a estrutura SQL representada a seguir:

- CREATE FUNCTION nome_da_funcao (x tipo, y tipo)

  RETURNS tipo

  RETURN (função);

Para a compreensão da técnica, vamos desenvolver uma função no banco de dados do exemplo para o cálculo da média final, em que: Média Final = (NotaP1 * 0,4) + (NotaP2 * 0,6).

Para isso foi desenvolvia a sintaxe SQL:

- CREATE FUNCTION fn_media(x DECIMAL(3,1), y DECIMAL(3,1))

  RETURNS DECIMAL(3,1)

  RETURN (x * 0.4) + (y * 0.6);

A sintaxe SQL para utilizar uma função desenvolvida em uma tabela deve ser estruturada como demonstrado a seguir:

- SELECT nome_da_funcao (parâmetro x, parâmetro y)

  FROM nome_da_tabela

  WHERE nome_da_coluna (condição);

Dessa forma, para utilizarmos a **FUNCTION** desenvolvida no exemplo (fn_media), com que se deseja selecionar o nome do aluno, a disciplina, e a média final dos alunos de exame (com notas entre 4,1 e 6,9 inclusive), deve ser utilizada a seguinte sintaxe SQL:

- SELECT Aluno.Nome, disciplina.Nome AS “Disiciplina”,

  fn_media(NotaP1, NotaP2)AS “Média Final” FROM Notas INNER JOIN Aluno

  ON Notas.AlunoRA = Aluno.RA INNER JOIN Disciplina

  ON Notas.DisciplinaId = Disciplina.Id WHERE **fn_media(NotaP1, NotaP2) >= 4.0**

  AND fn_media(NotaP1, NotaP2) <= 6.9;

Para exibir todas as funções desenvolvidas, deve ser utilizada a seguinte sintaxe SQL:

- SHOW FUNCTION STATUS;

Para exibir a estrutura de uma função, deve ser utilizada a seguinte sintaxe SQL:

- SHOW CREATE FUNCTION nome_da_funcao;

Para demonstração, no exemplo foi desenvolvido o comando SQL a seguir:

- SHOW CREATE FUNCTION fn_Media;

Para excluir uma função, deve ser utilizada a seguinte sintaxe SQL:

- DROP FUNCTION nome_da_funcao;

Procedimento armazenado (PROCEDURE)

No SGBD MySQL, os procedimentos armazenados (**PROCEDURES**) são inseridos em uma tabela chamada **ROUTINES**, no banco de dados chamado **INFORMATION_SCHEMA**, disponível no dicionário de dados.

A sintaxe SQL para se criar um procedimento armazenado é definida por meio da palavra-chave **PROCEDURE** (DATE, 2012). Para ser estruturada em um banco de dados, deve ser utilizado o comando SQL representado a seguir:

- CREATE PROCEDURE nome_do_procedure (var_nome tipo)

  {Declarações.}

Para isso, foi desenvolvida a sintaxe SQL:

- CREATE PROCEDURE proc_MediaExame (var_ DisciplinaId int)

  SELECT AVG(fn_media(NotaP1, NotaP2)) AS “Média Exame”

  FROM Notas

  WHERE DisciplinaId = var_DisciplinaId

  AND (fn_media(NotaP1, NotaP2) >= 4.0

  AND fn_media(NotaP1, NotaP2) <= 6.9);

Para utilizar um procedimento armazenado, a seguinte sintaxe SQL deve ser utilizada:

- CALL nome_do_procedure (var_nome);

Para exibir todos os procedimentos desenvolvidos que estejam armazenados, deve ser utilizada a seguinte sintaxe SQL:

- SHOW PROCEDURE STATUS;

O primeiro registro demonstra a PROCEDURE desenvolvida no exemplo. Para excluir um procedimento armazenado, deve ser utilizada a seguinte sintaxe SQL:

- DROP PROCEDURE nome_do_procedure;