SQL Básico

CURSO:
Administrador de Banco de Dados

PROFESSOR:
Denival Araújo dos Santos

CARGA HORÁRIA:
36 h/a

Índice Geral

SQL Básico

1.1) INTRODUÇÃO:

Convido você para embarcarmos neste maravilho mundo de banco de dados que nos espera de braços abertos. Se estivermos prontos, vamos lá!

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 em tabelas, criação de 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 aprendermos 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 SGBD's, alguns gratuitos, outros pagos. Listamos alguns exemplos a seguir: Postgresql, SQL Server, Oracle, Access, Firebird, dentre outros.

Figura 1 - Exemplos de SGBDs

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.

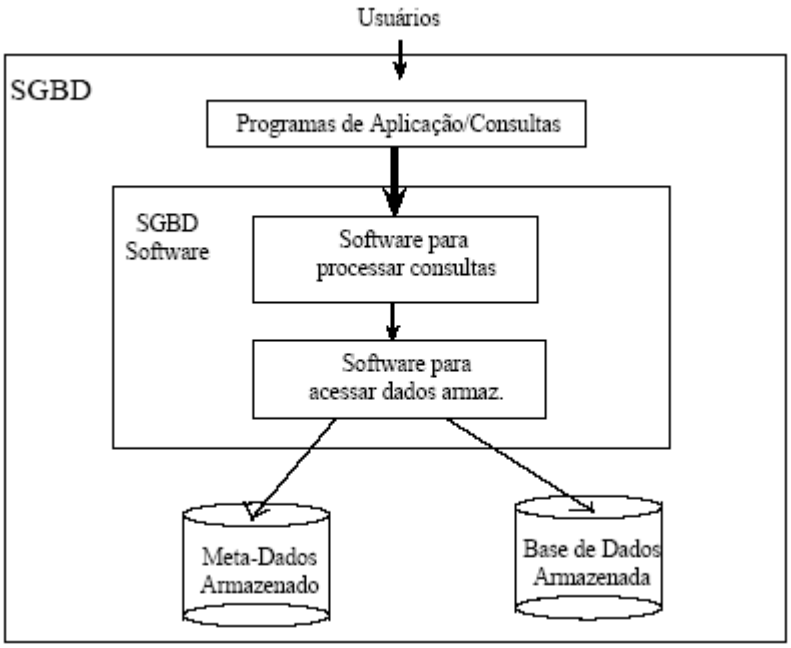

Figura 2 - Organização de um SGBD

A LINGUAGEM SQL

2.1) INTRODUÇÃO A SQL:

Você Sabia que...

O **SQL** foi desenvolvido originalmente no início dos anos 70 nos laboratórios da *IBM* em San Jose, dentro do projeto *System R*, que tinha por objetivo demonstrar a viabilidade da implementação do *modelo relacional* proposto por *E. F. Codd*. O nome original da linguagem era ***SEQUEL***, acrônimo para ***"Structured English Query Language"*** (***Linguagem de Consulta Estruturada em Inglês***), vindo daí o fato de, até hoje, a sigla, em inglês, ser comumente pronunciada *"síquel"* ao invés de *"és-kiú-él"*, letra a letra. No entanto, em português, a pronúncia mais corrente é a letra a letra: *"ése-quê-éle"*.

Fonte: Wikipédia

Structured Query Language, ou Linguagem de Consulta Estruturada ou SQL, é uma linguagem de pesquisa declarativa para banco de dados.

A linguagem SQL é um grande padrão de banco de dados. Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é uma linguagem declarativa em oposição a outras linguagens procedurais. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.

Embora o SQL tenha sido originalmente criado pela IBM, rapidamente surgiram vários "dialectos" desenvolvidos por outros produtores. Essa expansão levou à necessidade de ser criado e adaptado um padrão para a linguagem. Esta tarefa foi realizada pela American National Standards Institute (ANSI) em 1986 e ISO em 1987.

O SQL foi revisto em 1992 e a esta versão foi dado o nome de SQL-92. Foi revisto novamente em 1999 e 2003 para se tornar SQL:1999 (SQL3) e SQL:2003, respectivamente. O SQL:1999 usa expressões regulares de emparelhamento, *queries* recursivas e gatilhos (*triggers*). Também foi feita uma adição controversa de tipos não-escalados e algumas características de orientação a objeto. O SQL:2003 introduz características relacionadas ao XML, seqüências padronizadas e colunas com valores de auto-generalização (inclusive colunas-identidade).

Tal como dito anteriormente, o SQL, embora padronizado pela ANSI e ISO, possui muitas variações e extensões produzidos pelos diferentes fabricantes de sistemas gerenciadores de bases de dados. Tipicamente a

linguagem pode ser migrada de plataforma para plataforma sem mudanças estruturais principais.

As instruções da linguagem SQL podem ser divididas em dois grandes grupos: Linguagem de Manipulação de Dados (LMD ou Data Manipulation Language – DML) e a Linguagem de definição de dados (LDD ou Data Definition Language – DDL). A DML trata dos comandos de manipulação de dados, definindo os comandos para a seleção, inclusão, alteração e exclusão de dados de tabelas. Já a DDL reúne os comandos para criação e manipulação de estruturas e objetos do banco de dados.

Você Sabia que...

Os vários fornecedores de Sistemas Gerenciadores de Banco de Dados 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.

Fonte: Costa (2006)

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.

Apresentamos, a seguir, os 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

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 será criadas.

Sua sintaxe é bastante simples. Vejamos, através de exemplo, a criação de uma base de dados chamada *PraticaBD*.

```
CREATE DATABASE praticaBD;
```

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:

```
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]
```

Obs: Os campos 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) defini 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) defini 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.

Abaixo, 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));
```

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 Funcionário.

```
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))
```

Observamos que a tabela funcionário 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.

Abaixo, seguem as criações das tabelas Função e Projeto:

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

```
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))
```

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). O 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.

Atenção

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 Funcionário e Departamento, observe que o atributo *cod_dep* da tabela funcionário 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 Funcionário.

Por fim, 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 Funcionário, Projeto e Função.

```
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_trabalhal 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));
```

Na tabela Trabalha, inserimos uma restrição chamada *checa_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:

```
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]
```

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? O quadro abaixo apresenta a solução:

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

2.5.2) Adicionando uma coluna em uma tabela:

Imagine que, após criarmos a tabela funcionário e já termos cadastrados alguns registros, percebemos que esquecemos de criar a coluna *telefone* na tabela. Como resolveríamos este problema?

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

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?

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 Função. Além disso, desejamos também alterar o domínio do atributo, passado de *real* para *integer*. Para isso, observe o código abaixo:

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

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 abaixo.

```
DROP TABLE trabalha;
```

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 *funcionário* é dependente de *departamento*, pois ela recebe o atributo *cod_dep* como chave estrangeira. Para resolvermos este problema, teríamos que destruirmos a referência de chave estrangeira da tabela *funcionário*, ou mesmo, chegarmos ao ponto de destruirmos primeiro a tabela *funcionário*, 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 a seguinte:

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

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;

```
SELECT * FROM funcionario
```

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:

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

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.

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 Funcionário. Assim, precisaríamos utilizar as duas tabelas para obtermos o nosso resultado. A instrução que atende à nossa necessidade encontra-se logo abaixo:

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

Observe que utilizamos o código do departamento como “ponte” para “pularmos” da tabela funcionário para a tabela departamento. Isso aconteceu, pois a chave primária de departamento (cod_dep) é a chave estrangeira da tabela Funcionário.

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 inserirmos um ou vários registros em uma tabela do banco de dados. A sintaxe é a seguinte:

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

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:

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

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:

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

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 Funcionário, 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 funcionário, 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:

Codigo	Apelido	Data_nasc	Cod_setor	Nome_mãe
100	Joãozinho	01/01/1980	1	Francisca

200	Maricota	02/02/1979	1	Raimunda
300	Franzé	03/03/1978	1	Joanete

Bem, agora imagine que precisamos cadastrar, na tabela funcionário, todos os registros da tabela Pessoa. Como faríamos isso de maneira rápida? Como forma de agilizarmos o nosso trabalho, poderíamos executar o seguinte comando:

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

Perceba que conseguimos, através de uma única instrução, inserirmos vários registros na tabela funcionário. 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 a seguinte:

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

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. Este seleção poderá resultar em um ou vários registros. Neste caso a alteração irá ocorrer em todos os registros selecionados.

Vejamos os exemplos abaixo:

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

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;
```

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. Abaixo, apresentamos a sua sintaxe:

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

Caso desejássemos deletar os projetos que custam mais de 2000, usaríamos o seguinte comando:

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

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. Abaixo, 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. Vejamos alguns exemplos:

Caso precisássemos saber quantas projetos existem cadastrados na tabela Projeto.

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

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?

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

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:

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

2.9.3) Função Sum( ):

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

Exemplo:

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

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 abaixo 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';
```

2.9.5) Função Max( ):

A função Max obtém o maior valor dentre os elementos de uma coluna.

O exemplo abaixo 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';
```

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 seguinte exemplo:

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:

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

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 mostre, para cada código de projeto, a quantidade de funcionários que

trabalharam nele? Na realidade, o que estamos buscando está representado na tabela abaixo:

Cod_proj	Quantidade_funcionários
1	3
2	2

A solução para o nosso problema é a seguinte:

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

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 Funcionário e Departamento. Elas servirão de base para os tópicos seguintes.

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

FUNCIONÁRIO

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

DEPARTAMENTO

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 seguinte instrução:

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

O comando anterior apresentaria como resultado, a partir das tabelas propostas anteriormente, a seguinte relação resultante:

Nome	**Descr**
João	Desenvolvimento
José	Análise
Maria	Desenvolvimento
Antônio	Testes

Observe, no comando apresentado, que selecionamos o nome do funcionário a partir da tabela Funcionário 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 Funcionário 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 Funcionário, 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 Funcionário, 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 obrigatoriamente obedecem 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:

2.11.2.1) 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 abaixo:

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

A instrução anterior produzirá o seguinte resultado, a partir das tabelas propostas anteriormente:

Descr	Nome
Desenvolvimento	João
Desenvolvimento	Maria
Análise	José
Testes	Antônio
Contabilidade	

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.

2.11.2.2) 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 seguinte instrução:

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

EXERCÍCIO AULA

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 abaixo:

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.