

## **AULA 1**

### **INTRODUÇÃO A BANCO DE DADOS E VISÃO GERAL DO SQL**

#### **CONCEITUANDO BANCO DE DADOS**

Um banco de dados é uma coleção de dados (ou informações) organizadas de forma lógica, e que possui um significado próprio, cuja a interpretação é dada normalmente pela a aplicação (sistema) que está sendo desenvolvida.

A abstração e a construção de um banco de dados representa uma parte do que chamamos de mini-mundo, que a representação abstrata do mundo real, e que mormente queremos transformar em uma aplicação.

#### **MODELO RELACIONAL**

Um modelo relacional é o padrão atual para construção de ferramentas de banco de dados. Basicamente, um banco de dados relacional é composto de TABELAS ou RELAÇÕES. Uma TABELA é um conjunto não ordenado de linhas e cada linha é composto por uma série de valores chamados CAMPO ou COLUNA. Cada CAMPO é identificado por um NOME DE CAMPO e as informações ficam armazenadas nestes campos.

O modelo relacional é bastante potente, sendo que o seu principal elemento é a relação. Foi criado por Edgar Frank Codd, na década de 70, utilizando teoria matemática. A partir da década de 80, o modelo relacional começou a ser difundido, tornando-se líder de mercado a partir da década de 90.

O modelo relacional surgiu a partir de algumas necessidades como:

- aumentar a independência dos dados nos SGBDS;
- prover um conjunto de funções apoiadas em álgebra relacional para armazenamento e recuperação de dados.

Este modelo mostrou flexibilidade e passou a solucionar vários problemas a nível de concepção e implementação da base de dados. Um banco de dados possui uma estrutura organizada e funcional, que veremos a seguir.

## TERMOLOGIAS DO MODELO RELACIONAL

- a. As linhas das tabelas são conhecidas como TUPLAS e as colunas como ATRIBUTOS.
- b. O número de atributos (colunas) de uma relação (tabela) determina o GRAU DA RELAÇÃO. Portanto uma relação com quatro colunas possui grau quatro.
- c. A interseção linha X coluna de uma tabela denomina-se CÉLULA.
- d. O conteúdo de uma célula denomina-se valor de atributo .
- e. Cada célula de uma tabela relacional comporta apenas um valor de atributo, característica a qual designa-se por ATOMICIDADE (valor atômico).
- f. O conjunto de valores possíveis para um atributo de tabela denomina-se DOMÍNIO. Por exemplo, o domínio para o atributo cargo pode ser definido como: Valor numérico entre 1 e 10.

### Relação (tabelas ou entidades)

Uma relação é representada através de uma tabela. Esta tabela representa ao que no modelo entidade-relação chamávamos entidade. Esta tabela contém os atributos (colunas) e as tuplas (filas). Exemplo de uma tabela relativo ao armazenamento de livros de uma biblioteca: tabela livros

### Atributo (Coluna ou campo)

trata-se de cada uma das colunas da tabela. Vêm definidas por um nome e podem conter um conjunto de valores. Exemplo de campos relativo a um cadastro de livros: nomedolivro, autordolivro

Obs: Cada campo ou célula de uma tabela relacional comporta apenas um valor de atributo, característica a qual designa-se por ATOMICIDADE (valor atômico).

Exemplo:na tabela livros, temos as seguintes colunas, com os seguintes valores:

Entidade: livros	
Nomedolivro	Autordolivro
JAVA AVANÇADO	ALINE ROSA
SISTEMA COMPUTAÇÃO	ARMANDO GONÇALVES

### Tupla ou Registro

trata-se de cada uma das filas da tabela. É importante assinalar que não se podem ter tuplas duplicadas em uma tabela.

Duas tabelas estarão relacionadas a partir da união das chaves primária e estrangeira.

## Chaves

Cada tupla de uma tabela tem que estar associada a uma chave única que permita identificá-la. Uma chave pode estar composta por um ou mais atributos.

Uma chave tem que ser única dentro de sua tabela e não se pode descartar nenhum atributo da mesma para identificar uma fila.

Existem dois tipos de chaves:

### Chave primária (Primary Key)

É o valor ou conjunto de valores que identificam uma informação dentro de uma tabela. Este valor não pode ser nulo. Uma característica, é a não repetição de um campo que foi escolhido como chave primária. Um exemplo claro de chave primária seria o CPF, que é único para cada pessoa e não pode ser NULL.

### Chave estrangeira (Foreign Key)

Chave estrangeira é o valor ou valores de uma tabela que corresponde com o valor de uma chave primária em outra tabela. Esta chave é a que representa as relações (relacionamentos) entre as tabelas. Ou seja, em uma tabela em que existe uma chave primária e a informação contida neste campo não poderá se repetir, este mesmo campo poderá estar ligado a uma chave estrangeira e nesta outra tabela poderão haver repetições. Um exemplo seria um cliente de uma concessionária. Na tabela de clientes, o CPF poderá ser a chave estrangeira, não havendo repetições. Já na tabela vendas de carros, como o cliente poderá adquirir mais de um carro, este CPF poderá estar repetido diversas vezes.

Existem alguns tipos de relacionamentos possíveis:

- Um para um (1 para 1) - indica que as tabelas têm relação unívoca entre si. Você escolhe qual tabela vai receber a chave estrangeira;
- Um para muitos (1 para N) - a chave primária da tabela que tem o lado 1 está para ir para a tabela do lado N. No lado N ela é chamada de chave estrangeira;
- Muitos para muitos (N para N) - quando tabelas têm entre si relação n..n, é necessário criar uma nova tabela com as chaves primárias das tabelas envolvidas, ficando assim uma chave composta, ou seja, formada por diversos campos-chave de outras tabelas. A relação então se reduz para uma relação 1..n, sendo que o lado n ficará com a nova tabela criada.

## **SGBD – Sistemas de Gerenciamento de Banco de Dados**

Os SGBDS são softwares que foram desenvolvidos para facilitar o usuário. Costumam ter uma interface mais simples e facilidade nos acessos aos comandos. Muitas vezes o termo banco de dados é usado como sinônimo de SGDB. Na verdade, quando fala-se em bando de dados, trata-se aos dados que estão sendo armazenados fisicamente. Já um SGBD, é um software que irá manipular um banco de dados de forma geral, desde a sua concepção até sua manutenção.

O modelo de dados mais adotado hoje em dia é o modelo relacional, onde as estruturas têm a forma de tabelas, compostas por tuplas (linhas) e colunas.

**Exemplos de SGBD:** SQL Server, Postgresql (PGADMIN III), Mysql, Oracle

### **Porque utilizar um SGBD ?**

- Controle centralizado dos dados
- Controle de redundância
- Compartilhamento dos dados
- Independência dos dados
- Segurança
- Backup e recuperação
- Forçar restrições de integridade
- Aumento de produtividade e disponibilidade
- Padronização

### **Modelo de Dados**

Conjunto de conceitos que podem ser usados para descrever a estrutura de um banco de dado, tipos de dados, relacionamentos e restrições. Pode também incluir operações que especificam consultas e atualizações no banco de dados. Normalmente, o projetista cria modelos em forma de diagramas, que é a representação gráfica do esquema, para estabelecer como será o banco de dados, quais as tabelas que o comporão e quais os campos que esta tabela terá, e quais os relacionamentos e restrições.

### Dicionário de Dados

Um dicionário de dados, em um contexto de banco de dados e modelagem de dados, consiste na representação das ENTIDADES (TABELAS) que irão compor um banco de dados. Neste caso, é uma forma de abstração de um modelo do banco de dados que deverá ser implantado. O dicionário de dados é criado a partir de uma análise das informações que serão relevantes serem armazenadas, e demonstra como ficará as tabelas de um banco de dados quando forem gerados fisicamente.

A idéia do dicionário de dados é padronizar precisamente definições semânticas a serem adotadas. Estas definições devem ser incluídas na representação dos elementos de dados. Nesta definição estão: o nome da tabela, o nome do campo, o tipo do campo (se é numérico, caracter, data, etc), o tamanho do campo e outras informações que o analista julgar necessário.

Exemplo:

#### NOME DA TABELA

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrição
NOME1	integer		pk
NOME2	Varchar	50	Not null

### SQL – Structured Query Language

A SQL (Structured Query Language) é uma linguagem padrão para as bases de dados relacionais. Uma pesquisa, que também poderá ser chamada de query, em SQL especifica o que deve ser procurado mas não como deve ser procurado. A SQL não é propriamente dita uma linguagem de programação completa; por exemplo, não tem instruções de controle nem de interação, sendo que possui outros controles como repetição e condicionais.

A SQL foi desenvolvida no início dos anos 70, nos laboratórios da IBM. A linguagem SQL em si é um GRANDE padrão de banco de dados.

A linguagem SQL foi padronizado pela ANSI e pela ISO, sendo que mesmo assim, possui muitas variações no tocante a alguns comandos que foram implementados em sistemas gerenciadores de banco de dados, mas mesmo assim, os comandos principais são bem parecidos.

Alguns Sistemas de Banco de Dados que usam SQL:

- Firebird
- Informix
- InterBase
- Microsoft Access
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- LiteBase Mobile (dedicado à plataformas móveis como: Palm OS, Pocket PC, WinCE, Symbian)

A SQL é :

- uma linguagem de definição de dados (DDL – Data Definition Language)
- uma linguagem de manipulação de dados (DML – Data Manipulation Language)
- e uma linguagem de controle de acesso a dados (DCL – Data Control Language).

DDL – para criar/alterar/eliminar objetos (tabelas, restrições, índices, etc.) da base de dados.

DML – para manipular dados (inserir dados nas tabelas, atualizar dados existentes e eliminar dados das tabelas) e para interrogar (baseado na álgebra relacional) a base de dados, as chamadas consultas.

DCL – para tratar os aspectos relacionados com a autorização de acesso aos dados. Permite que o utilizador controle quem tem acesso para visualizar ou manipular dados dentro da base de dados.

## EXERCÍCIOS

- 1) Qual a diferença entre chave primária e chave estrangeira ?
- 2) Conceitue banco de dados.
- 3) Conceitue Sistema de Gerenciamento de Banco de Dados.

## AULA 2

### CLÁUSULAS SQL, CRIANDO BANCO DE DADOS, CRIANDO TABELAS

#### CONCEITOS : NÍVEIS DE VISÃO

Segundo Silberschatz (2006), um dos maiores benefícios de um banco de dados é proporcionar ao usuário uma visão abstrata dos dados. Isto é, o sistema acaba por ocultar determinados detalhes sobre a forma de armazenamento e manutenção desses dados.

##### Nível Lógico (ou conceitual)

O próximo nível de abstração descreve quais dados estão armazenados de fato no banco de dados e as relações que existem entre eles. Aqui o banco de dados inteiro é descrito em termos de um pequeno número de estruturas relativamente simples. Embora as implementações de estruturas simples no nível conceitual possa envolver complexas estruturas de nível físico, o usuário do nível conceitual não precisa preocupar-se com isso. O nível conceitual de abstração é usado por administradores de banco de dados, que podem decidir quais informações devem ser mantidas no BD;

Características:

- Não contém detalhes de implementação
- Independe do tipo de BD utilizado
- Ponto de partida do projeto de banco de dados

Exemplos : Diagramas de entidade e relacionamento, dicionários de dados

##### Nível físico

O nível mais baixo de abstração descreve como os dados estão realmente armazenados. No nível físico, complexas estruturas de dados de baixo nível são descritas em detalhes;

Características:

Descreve as estruturas de armazenamento e quais relacionamentos possuem

Mapeamento do modelo lógico em um esquema físico

Possibilita um esquema eficiente aos entre usuário e sistema

##### Nível de visões

O mais alto nível de abstração descreve apenas parte do banco de dados. Apesar do uso de estruturas mais simples do que no nível conceitual, alguma complexidade perdura devido ao grande tamanho do banco de dados. Muitos usuários do sistema de banco de dados não estarão interessados em todas as informações. Em vez disso precisam de apenas uma parte do banco de dados. O nível de abstração das visões de dados é definido para simplificar esta interação com o sistema, que pode fornecer muitas visões para o mesmo banco de dados.

Características:

- Os usuários visualizam um conjunto de programas de aplicação que escondem os detalhes do BD
- Não contém detalhes físicos (índices, etc)

## INSTRUÇÕES E CLÁUSULAS SQL

Instruções e cláusulas são comandos utilizados no SQL, para manipulação dos dados nas tabelas e manipulação das próprias tabelas. Como já vimos, o SQL é um padrão, havendo apenas algumas diferenças entre os diversos SGBD existentes.

### Exemplos de Instruções padrão no SQL:

ALTER TABLE, CREATE, DROP, RENAME, INSERT, UPDATE, DELETE, SELECT

### Exemplos de Cláusulas:

CONSTRAINT, FROM, GROUP BY, HAVING, ORDER BY, WHERE

### Consultas (JOIN):

RIGHT OUTER JOIN, INNER JOIN, JOIN, LEFT OUTER JOIN

## RESTRIÇÕES

Para que possamos ter integridade em nosso banco de dados, na fase de projetar o banco de dados, temos que pensar nos campos mais importantes de nossas tabelas. E uma forma de manter essa integridade é utilizarmos restrições (constraints). Por exemplo, imagine um cadastro de clientes, que contenha os seguintes dados:

- cpf
- nome
- endereço
- cep
- telefone
- email
- página na internet
- dentre outros dados.

Dos dados acima listados, quais você considera de suma importância ?

Vamos eleger cpf, nome, endereço, cep e telefone. Email e página na internet seriam apenas complemento. Os primeiros campos listados, poderíamos configurar



estes campos para preenchimento obrigatório, ou seja, o valor não poderá ser nulo (null). Isso poderia tranquilamente ser feito através do front-end (parte do sistema de software que interage diretamente com o usuário) do sistema, mas precisamos de uma restrição mais sólida. E se o front-end falhar ? O mais confiável é que estas restrições sejam realizadas diretamente no banco de dados. Nada impede que seu front-end faça o tratamento destes campos, mas a segurança maior estará na configuração do banco de dados. Ou seja, ao criarmos qualquer tabela, antes precisamos analisar bem que tipo de restrições queremos configurar. Mais para frente veremos quais restrições poderemos configurar para manter a integridade do nosso banco de dados.

**Observação: chave primária e chave estrangeiras também são restrições.**

## O QUE É UMA QUERY

Uma query é uma instrução onde são digitados todos os comandos que referem-se ao seu banco de dados. Todos os SGBS possuem a instrução query para que estes comandos sejam digitados. Por exemplo, caso queira gravar dados no seu banco de dados, é só passar os dados para a query e executar. Pronto ! Os dados serão gravados... claro se não houverem erros de sintaxe. Em uma query pode-se, além de gravar, efetuar consultas.

## CONVENÇÕES PARA A LINGUAGEM SQL

Convenção mais é do que um padrão que iremos utilizar para criar os nossos objetos (banco de dados, tabelas, campos).

Convenção	Exemplo
Letra maiúscula	Comandos: SELECT, WHERE, AND, OR, ORDER BY, HAVING, CREATE, ALTER, INSERT, UPDATE, DELETE, INNER JOIN
Letras minúsculas	Nome de tabelas e colunas, nome de funções, etc

## CRIANDO UM BANCO DE DADOS

Representa o arquivo físico de dados, onde são armazenados os dados de um sistema informatizado, para consulta pelo usuário.

Através do comando `CREATE DATABASE` podemos criar um banco de dados. Não esquecendo, que, para isso deveremos acessar algum tipo de SGBD.

Sintaxe: `CREATE DATABASE NOME_DO_BANCODEDADOS`

Exemplo para criação de um banco de dados. Vamos criar um banco de dados chamado aulas:

Na query do SGBD digite

```
CREATE DATABASE aulas
```

Verifique se o banco de dados foi criado

### **APAGANDO UM BANCO DE DADOS**

O comando para apagar um banco de dados é o `DROP DATABASE`.

Sintaxe: `DROP DATABASE NOME_DO_BANCODEDADOS`

Para apagar um banco de dados, digite o comando na query do SGBD:

```
DROP DATABASE aulas
```

O comando `DROP DATABASE` não pede confirmação, ou seja, cuidado antes de executar esse comando, para não apagar um banco de dados inteiro por engano.

### **CRIANDO A ESTRUTURA DE UMA TABELA**

Para criarmos a estrutura de uma tabela em um banco de dados, o comando chama-se `CREATE TABLE`.

Regras de nomeação para nomes de tabelas e campos:

1. Deve-se começar com letras
2. Utilizar letras em caixa baixa
3. Conter somente a\_z, 0\_9, “\_”

**COMANDO CREATE TABLE**

Sintaxe: CREATE TABLE **nome\_da\_tabela**

**Exemplo: Criar as tabelas CURSOS e ALUNOS**

No dicionário de dados (parte do nível lógico) abaixo demonstrados, sugerimos os tipos de campos, o tamanho de cada campo.

**CURSOS**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrição
codigocurso	integer		
nomedocurso	varchar	50	

```
CREATE TABLE cursos (  
    codigocurso INTEGER,  
    nomedocurso VARCHAR(50)  
)
```

**Campos de restrição do tipo NÃO NULO**

uma restrição de não-nulo especifica que uma coluna não pode conter valor nulo.

**Exemplo:**

```
CREATE TABLE cursos (  
    codigocurso INTEGER NOT NULL,  
    nomedocurso VARCHAR(50) NOT NULL  
)
```

**Para selecionar a tabela:**

```
select * from cursos
```

Desta forma poderá visualizar a estrutura da tabela e os dados após tê-los inserido

**Mais um exemplo... sendo que utilizando campos NOT NULL**

**ALUNO**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Matricula	Integer		Not null
Nomedoaluno	Varchar	50	Not null
Rua	Varchar	50	Not null
Numero	Integer		
Complemento	Varchar	10	
Bairro	Varchar	20	
Cep	Integer		

```
CREATE TABLE aluno (
  matricula integer not null,
  nomedoaluno varchar(50) not null,
  rua varchar(50) not null,
  numero integer,
  complemento varchar(10),
  bairro varchar(20),
  cep integer
)
```

**Obs: para selecionar a tabela: select \* from alunos**

**Alguns tipos de dados permitidos, Tamanho dos campos**

<i>Tipo de Campo</i>	<i>Observações</i>
integer / int	Números inteiros de -2147483648 a 2147483647 (signed) ou então de 0 a 4294967295 (unsigned).
Bigint	Números inteiros de -9223372036854775808 a 9223372036854775807 (signed) ou de 0 a 18446744073709551615 (unsigned).
bool / boolean / bit	Indica falso (zero) ou verdadeiro (qualquer número diferente de zero). float(m,d) - Números reais de -3.402823466E+38 a -1.175494351E-38 e de 1.175494351E-38 a 3.402823466E+38. M representa o tamanho do número de d representa o número de decimais.
char(m)	Uma string de tamanho fixo. m representa o tamanho da coluna. Caso o

<b>Tipo de Campo</b>	<b>Observações</b>
	dado guardado nessa coluna seja menor que m, a diferença é preenchida com espaços vazios. Caso m não seja declarado, o tamanho considerado é 1. O tamanho vai de 0 a 255.
varchar(m)	Funciona da mesma maneira que o <i>char</i> , porém o tamanho da string não é fixo (não existe o preenchimento com espaços).
text / blob	Strings com máximo de 65,535 caracteres.
Date	Datas com valor entre '1000-01-01' e '9999-12-31'. Perceba que o formato suporta é 'AAAA-MM-DD'.
Datetime	Combinação entre <i>date</i> e <i>time</i> . O formato é 'AAAA-MM-DD HH:MM:SS'. Suporta valores entre '1000-01-01 00:00:00' e '9999-12-31 23:59:59'.

## EXERCÍCIOS

Crie algumas tabelas, seguindo os dicionários de dados abaixo:

### TURN0

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
codigoturno	Integer		NOT NULL
nometurno	Varchar	50	NOT NULL

### COBRANCA

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Matricula	Integer		NOT NULL
Dtvencimento	Datetime		NOT NULL
Vlmensalidade	Money		
Dtbaixa	Datetime		
Vlbaixa	Money		

Após criar as tabelas, selecione cada uma das tabelas.

## **AULA 3**

### **INSERINDO E PESQUISANDO DADOS EM UMA TABELA**

#### **CONCEITOS**

##### **LINGUAGEM DE DEFINIÇÃO DE DADOS (DDL)**

A linguagem de definição de dados (Data Definition Language) especificará em seu conjunto toda a definição da criação do banco de dados. A compilação das instruções DDL é um conjunto de tabelas que são armazenadas no dicionário de dados. Um dicionário de dados é um arquivo que contém metadados (dados acerca dos dados), dando ao SGBD a forma e acesso de como obter os dados reais requeridos quando de uma leitura ou modificação realizada no banco de dados.

Exemplos de comandos DDL: CREATE TABLE, ALTER TABLE, DROP TABLE

##### **LINGUAGEM DE MANIPULAÇÃO DE DADOS (DML)**

A linguagem de manipulação de dados (Data Manipulation Language) é uma linguagem que permite ao usuário recuperar, inserir, remover ou modificar informações armazenadas no banco de dados.

**Exemplos de comandos DML: SELECT, INSERT, DELETE, UPDATE**

Na aula passada, trabalhamos com o comando CREATE TABLE, que pertence ao subconjunto DDL. Voltaremos a cuidar de DDL daqui a algumas aulas.

Hoje e na próxima aula, trataremos do subconjunto DML.

#### **Instrução INSERT**

A instrução INSERT é responsável por inserir dados em uma tabela. Para isso, precisamos seguir uma sintaxe básica.

##### **Sintaxe Básica:**

INSERT INTO TABELA

values (DADO1, DADO2, DADO3);

**OU**

INSERT INTO TABELA (CAMPO1,CAMPO2,CAMPO3)

values (DADO1, DADO2, DADO3);

No primeiro grupo de parênteses, descrevemos os campos da tabela que iremos gravar as informações. No segundo grupo de parênteses, descrevemos os

dados que serão gravados em cada campo. As posições deverão estar na ordem entre campos e dados:

CAMPO1 = DADO1

CAMPO2 = DADO2

CAMPO3 = DADO3

### Exemplo - Inserindo dados na tabela CURSOS:

```
CREATE TABLE cursos (  
    codigocurso INTEGER,  
    nomedocurso VARCHAR(50)  
)  
INSERT INTO cursos (codigocurso, nomedocurso)  
VALUES (1, 'DIREITO')
```

Note que algumas informações estão entre aspas e outras não. Os do tipo **inteiro ou numérico não ficam entre aspas**; já as do tipo string, sim. Já o campo data, como é tipo americano, vamos gravar invertido (ano, mês, dia).

### Exemplo - Inserindo dados na tabela TURNO:

```
CREATE TABLE turno (  
    codigoturno INTEGER,  
    nometurno VARCHAR(20)  
)  
INSERT INTO turno (codigoturno, nometurno)  
VALUES (1, 'MANHÃ')
```

- ✧ **Insira mais alguns cursos e turnos em suas respectivas tabelas**
- ✧ **Selecione os dados de cada uma das tabelas**

## CONSULTANDO DADOS EM UMA TABELA

Após a inclusão dos dados, vamos agora consultar as informações de ambas

as tabelas. Não esqueça que, para que possamos fazer vários níveis de consulta, seria interessante que vários dados estivessem lançados em ambas as tabelas.

### Instrução SELECT

A instrução SELECT é utilizada para realizar consultas em uma ou mais tabelas de um banco de dados

#### Sintaxe Básica:

```
SELECT [DISTINCT] {*, coluna [pseudônimo], ... }  
FROM   tabela  
WHERE  condição(ões)  
GROUP BY coluna(s)  
HAVING condição(ões)  
ORDER BY {coluna, expr} [ASC|DESC];
```

#### Selecionando todas as linhas e todos os campos da tabela:

Exemplos:

```
SELECT * FROM cursos  
SELECT * FROM aluno
```

Perceba que em uma consulta a uma tabela, o resultado que temos são linhas e colunas. As colunas referem-se aos campos da tabela e cada linha corresponde a um determinado registro.

#### Selecionando todas as linhas e alguns campos da tabela:

```
SELECT codigocurso,nomedocurso FROM cursos  
SELECT nometurno FROM turno
```

### ALIAS DE COLUNA

Podem ser atribuídos nomes para as entradas da lista de seleção.



Exemplo:

```
SELECT codigocurso AS codigo, nomedocurso AS curso FROM cursos
```

### CLÁUSULA ORDER BY

A cláusula ORDER BY é um elemento opcional da Instrução SELECT. Essa cláusula é utilizada para ordenar as informações de uma tabela através de um determinado campo.

**Exemplo - Selecionando todas as linhas, todos os dados da tabela, ordenando em ordem alfabética:**

```
SELECT * FROM cursos ORDER BY nomedocurso  
SELECT * FROM alunos ORDER BY nomedoaluno
```

**Exemplo - Selecionando todas as linhas, alguns dados da tabela, ordenando em ordem alfabética:**

```
SELECT codigocurso,nomedocurso FROM cursos order by nomedocurso  
SELECT matricula, nomedoaluno, sexo FROM aluno order by nomedoaluno
```

Obs: Ao utilizar a cláusula ORDER BY, mesma pode ser seguida das palavras opcionais ASC e DESC. A primeira faz ordenação ascendente e a segunda, descendente.

Exemplo:

```
SELECT codigocurso,nomedocurso FROM cursos order by codigocurso ASC  
SELECT codigocurso,nomedocurso FROM cursos order by codigocurso DESC
```

### CLÁUSULA WHERE NO SELECT

Na instrução SELECT é possível usarmos condicionais. A cláusula utilizada para uma condicional é a WHERE.

**Exemplo : selecionando um determinado turno**

```
SELECT * FROM turno WHERE codigoturno = 2
```

**Exemplo : selecionando todos os cursos cujo código seja maior ou igual a 3**

```
SELECT * FROM cursos WHERE codigocurso >= 3
```

**Os operadores de comparação habituais são:**

Operador	Descrição
<	Menor que
>	Maior que
<=	Menor que ou igual a
>=	Maior que ou igual a
=	Igual
<> ou !=	Diferente

### REMOVENDO LINHAS DUPLICADAS NA CONSULTA

Caso nenhuma regra de restrição na entrada de dados tenha sido feita, corre-se o risco que dados entrem em duplicidade, ou seja, todos os campos de duas ou mais linhas são idênticos. Para que na consulta as linhas em duplicidade se tornem uma, podemos utilizar a palavra-chave DISTINCT.

Para que o exemplo abaixo possa ser visualizado, inclua alguns dados em duplicidade na tabela cursos.

Exemplo:

```
SELECT nomedocurso FROM cursos – mostrará todas as linhas  
SELECT DISTINCT nomedocurso FROM cursos – irá retirar as duplicidades
```

***Veremos outras cláusulas da instrução SELECT em outros capítulos***

## EXERCÍCIOS

- 1) Crie uma tabela chamada **pessoas** com os seguintes campos:  
código integer not null,  
nome varchar(40) not null,  
idade integer
- 2) Insira alguns registros nesta tabela, sendo que com alguns registros idênticos
- 3) Selecione todas as colunas de todas as pessoas da tabela, ordenando pelo campo nome
- 4) Selecione os campos nome e idade de todas as pessoas com idade maior ou igual a 18 anos
- 5) Selecione o campo nome da tabela, retirando duplicidades

## AULA 4

### ALTERANDO E APAGANDO DADOS EM UMA TABELA / ALTERANDO A ESTRUTUR FÍSICA DE UMA TABELA

#### INDEPENDÊNCIA DOS DADOS

A habilidade de modificar a definição de um esquema em um nível sem afetar a definição de esquema num nível mais alto é chamada de *independência de dados*. Existem dois níveis de independência dos dados:

1) Independência física de dados: é a habilidade de modificar o esquema físico sem a necessidade de reescrever os programas aplicativos. As modificações no nível físico são ocasionalmente necessárias para melhorar o desempenho.

1. Características:

1. modificar o esquema físico sem que qualquer programa de aplicação tenha que ser reescrito (exemplo: criação de índices);
2. melhoria de desempenho;
3. oferecida em produtos modernos

2) Independência lógica de dados: é a habilidade de modificar o esquema conceitual sem a necessidade de reescrever os programas aplicativos. As modificações no nível conceitual são necessárias quando a estrutura lógica do banco de dados é alterada (por exemplo, a adição de contas de bolsas de mercado num sistema bancário).

2. Características:

1. modificar o esquema conceitual sem precisar reescrever programas de aplicação;
2. difícil de ser conseguida;
3. modificações de nível lógico são necessárias para adequar o conjunto de dados às aplicações;
4. programas de aplicação são mais fortemente dependentes da estrutura lógica dos dados

A independência lógica dos dados é mais difícil de ser alcançada do que a independência física, porém os programas são bastante dependentes da estrutura lógica dos dados que eles acessam.

O conceito de independência dos dados é similar em muitos aspectos ao conceito de *tipos abstratos de dados* em modernas linguagens de programação. Ambos escondem detalhes de implementação do usuário. Isto permite ao usuário concentrar-se na estrutura geral em vez de detalhes de baixo nível de implementação.

**ALTERANDO E APAGANDO DADOS EM UMA TABELA****Instrução UPDATE – ALTERANDO DADOS**

O comando UPDATE é utilizado para realizar alterações de valores das colunas de uma tabela em um banco de dados, em todas as linhas que satisfazem uma determinada condição. Somente precisam ser mencionadas na cláusula SET as colunas que serão modificadas; as colunas que não serão modificadas manterão seus valores atuais.

Por padrão, o comando UPDATE atualiza linhas na tabela especificada e nas suas tabelas descendentes.

Vamos fazer uma alteração na tabela CURSO, onde temos as seguintes linhas e colunas gravadas.

<b>Codigocurso</b>	<b>Nomecurso</b>
1	ADMINISTRAÇÃO
2	CONTÁBEIS
3	TADS

Vamos supor que um dos cursos teve o nome inserido errado na tabela. E precisamos alterar.

```
CREATE TABLE cursos (  
    codigocurso INTEGER,  
    nomedocurso VARCHAR(50)  
)  
INSERT INTO cursos (codigocurso, nomedocurso)  
VALUES (1, 'DIREITO')  
*** insira mais cursos
```

O código do curso a ser alterado será o 3 (TADS). Supomos que o conteúdo do campo nomecurso, está errado. Vamos alterar o nomecurso do codigocurso 3 para TECNOLOGIA EM PROCESSAMENTO DE DADOS. Mas lembre que vamos alterar SOMENTE o curso de código igual a 3.

Exemplo:

```
UPDATE cursos SET
    Nomecurso = 'TECNOLOGIA EM ANÁLISE E DES.DE SISTEMAS'
WHERE
    codigocurso = 3
```

### Cláusula WHERE para alterar dados

A cláusula WHERE é uma parte opcional da Expressão X Seleção da Instrução DELETE e da Instrução UPDATE. A cláusula WHERE permite selecionar linhas baseado em uma expressão booleana. Somente as linhas para as quais a expressão é avaliada como TRUE são retornadas no resultado, ou no caso da instrução DELETE, excluídas, ou no caso da instrução UPDATE, atualizadas. Também podemos utilizar a cláusula WHERE em outros comandos, como SELECT.

No caso do exemplo apresentado, nós queremos alterar somente o código do curso 3. Caso não tivéssemos colocado a cláusula **WHERE**, todas as linhas da tabela teriam o campo nomecurso alterado, o que seria um transtorno, pois todas as linhas da tabela ficariam com o nome nome do curso.

### INSTRUÇÃO DELETE

A instrução DELETE é utilizada para apagar registros de uma tabela.

**Podemos apagar todos os registros de uma tabela:**

```
DELETE FROM nome_da_tabela
```

**Observação:** cuidado tanto com a cláusula DELETE quanto com a UPDATE, caso não esteja utilizando a cláusula WHERE. Se não estiver especificando o campo que deseja alterar ou apagar, TODOS os registros serão afetados, gerando assim inconsistência de informações.

### Criando tabela e inserindo registros

```
CREATE TABLE turno (
    codigoturno INTEGER,
    nometurno VARCHAR(20)
)
INSERT INTO turno (codigoturno, nometurno)
```

```
VALUES (1, 'MANHÃ')
```

\*\*\* Insira mais turnos

**Podemos apagar somente o registro desejado:**

```
DELETE FROM turno WHERE codigoturno = 3;
```

## OUTRAS CLÁUSULAS DA INSTRUÇÃO SELECT

### BETWEEN

A construção especial BETWEEN faz o papel de um operador de comparação.

Exemplo:

```
SELECT codigocurso WHERE codigocurso BETWEEN 1 AND 5
```

### IS NULL / IS NOT NULL

Verifica se um valor de um campo da tabela é nulo ou não.

Exemplo:

```
SELECT * FROM cursos WHERE nomecurso IS NOT NULL  
SELECT * FROM cursos WHERE nomecurso IS NULL
```

### CONDIÇÃO ILIKE

A condição ILIKE é usada para executar buscas de valores válidos de uma string. As condições da busca podem conter caracteres, sendo que a busca pode ser parte dele.

Exemplo:

```
SELECT * FROM cursos WHERE nomedocurso ILIKE '%ANAL%'  
SELECT * FROM cursos WHERE nomedocurso ILIKE 'ANAL%'
```

Observação : os comandos UPDATE, DELETE e SELECT pertencem ao subconjunto DML.

**ALTERANDO A ESTRUTURA FÍSICA DE UMA TABELA**

Vamos criar a tabela abaixo para trabalharmos com esta estrutura

**ALUNO**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Matricula	Integer		
Nomedoaluno	Varchar	50	
Rua	Varchar	50	
Numero	Integer		
Complemento	Varchar	10	
Bairro	Varchar	20	
Cep	Integer		

```
CREATE TABLE aluno (  
    matricula integer,  
    nomedoaluno varchar(50),  
    rua varchar(50),  
    numero integer,  
    complemento varchar(10),  
    bairro varchar(20),  
    cep integer  
)
```

**Instrução ALTER TABLE**

Após termos criado fisicamente uma tabela em um banco de dados (esta tabela ter uma nome, campos etc), pode ser que tenhamos a necessidade de alterar esta estrutura, acrescentando ou alterando campos desta tabela.

**A instrução ALTER TABLE permite:**

- Adicionar coluna à tabela
- Adicionar restrição à tabela
- Apagar uma coluna
- Alterar o tamanho de uma coluna

Uma alteração de estrutura de uma tabela pode afetar os dados já cadastrados.



**Adicionar colunas novas**

Exemplos:

```
ALTER TABLE aluno ADD sexo varchar(1)
ALTER TABLE aluno ADD telefone integer
ALTER TABLE aluno ADD celular integer
ALTER TABLE aluno ADD email integer
```

**Apagar uma coluna**

Exemplo:

```
ALTER TABLE aluno DROP COLUMN celular
```

Obs: Ao apagar uma coluna, é óbvio que todos os dados sejam apagados.

**Alterar o tamanho de uma coluna**

Exemplo:

```
ALTER TABLE aluno
  ALTER COLUMN rua varchar(80)
```

**INSTRUÇÃO DROP TABLE**

A instrução DROP TABLE é usada para apagar uma tabela de um banco de dados.

Exemplo:

```
DROP TABLE aluno
```

***Muito cuidado com esta instrução. Após apagar uma tabela de um banco de dados, não será possível obtê-la novamente a não ser que tenha backup.***

---

**EXERCÍCIO 1**

1) Crie a seguinte tabela:

Tabela: notas

CAMPO	TIPO	TAMANHO	RESTRIÇÃO
Matricula	Integer		
Nome	Varchar	40	
Nota	Float	4,2	

2) Insira os seguintes dados na tabela:

MATRICULA	NOME	NOTA
500	Sandra	5,5
600	Pedro	8
700	Julia	9
800	Carlos	4,5
900	Joana	7

3) Selecione todos os campos da tabela e somente os nomes que iniciem com J.

4) Altere a nota da matricula 800 para 7,5

5) Altere nota do Carlos para NULL.

6) Selecione as linhas cuja as notas sejam maiores ou iguais a 5

7) Selecione as linhas cuja as matriculas estejam no intervalo 600 a 800

**EXERCÍCIO 2**

1) Crie a seguinte tabela, seguindo exatamente o proposto:

**VEÍCULO**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigoveiculo	Integer		Not null
Descricaoveiculo	Varchar	60	
Anofabricacao	integer		

2) Adicione a seguinte coluna na tabela VEÍCULO:

**placa varchar(10) not null**

3) Inclua as seguintes restrições nos respectivos campos da tabela:

- **No campo descricaoveiculo, o campo deverá ser NOT NULL**
- **no campo anofabricacao, não aceitar ano menor que 1980**
- **o campo descricaoveiculo deverá ter restrição UNIQUE**

4) Insira alguns registros na tabela

5) Selecione os dados da tabela

## AULA 5

### INTEGRIDADE DOS DADOS - CRIANDO RESTRIÇÕES

Aplicar a integridade de dados garante a qualidade dos dados do banco de dados. Por exemplo, se um aluno for inserido com um valor de matrícula 1234, o banco de dados não deverá permitir que outro aluno tenha uma mesma matrícula com o mesmo valor. Se você tiver uma coluna qualquer destinada a conter valores que se estendem de 1 a 5, o banco de dados não deverá aceitar um valor fora desse intervalo. Se a tabela tiver uma coluna `cod_depto` que armazene o número do departamento de um funcionário, o banco de dados deverá permitir apenas os valores que sejam válidos para os números de departamento que empresa possui cadastrada no sistema.

Duas importantes etapas do planejamento de tabelas são a identificação de valores válidos para a coluna e a decisão sobre como aplicar a integridade de dados à coluna. A integridade de dados se encaixa nas seguintes categorias:

- **Integridade de entidade**
- **Integridade de domínio**
- **Integridade referencial**
- **Integridade definida pelo usuário**

#### **Integridade de entidade**

A integridade de entidade define uma linha como entidade exclusiva de determinada tabela. A integridade de entidade aplica a integridade das colunas do identificador ou da chave primária de uma tabela por meio de índices UNIQUE, restrições UNIQUE ou restrições PRIMARY KEY.

#### **Exemplo de uma tabela sem restrição:**

```
CREATE TABLE pessoa (  
    codigo integer,  
    nome varchar(50)  
)  
INSERT INTO pessoa VALUES (1, 'Paula');  
INSERT INTO pessoa VALUES (1, 'Regina');  
INSERT INTO pessoa VALUES (2, 'Andre');
```

Veja que, sem restrição, permite-se que uma informação que não poderia haver duplicidade seja cadastrada na tabela.

### Exemplo de uma tabela utilizando chave primária (Primary key):

*"Chaves primárias" (em inglês, "Primary Keys" ou "PK"), sob o ponto de vista de um [banco de dados relacional](#), referem-se aos conjuntos de um ou mais campos, cujos valores, considerando a combinação de valores de todos os campos da tupla, nunca se repetem e que podem ser usadas como um índice para os demais campos da tabela do banco de dados. Em chaves primárias, não pode haver valores nulos nem repetição de tuplas.*

*Origem: Wikipédia*

*\* tuplas são linhas ou registros da tabela (entidade)*

**\*\* ao criarmos uma chave primária este campo poderá ter ligação com outra tabela através da chave estrangeira (próxima aula)**

### Exemplo Adicionando chave primária em uma tabela que já foi criada

Comando:

```
ALTER TABLE pessoa ADD CONSTRAINT codpessoa PRIMARY KEY (codigo)
```

**\*\*\*** No comando acima, a estrutura física da tabela pessoa está sendo alterada e uma chave primária está sendo acrescentada. Para que o comando seja executado com sucesso, não poderá haver duplicidade de informações inseridas na tabela.

### Exemplo Criando uma tabela já com a chave primária

Vamos apagar a tabela e recriá-la elegendo o campo código como chave primária da tabela pessoa.

```
DROP TABLE pessoa
CREATE TABLE pessoa (
    codigo integer PRIMARY KEY,
    nome varchar(50)
)
INSERT INTO teste VALUES (1, 'Paula');
INSERT INTO teste VALUES (1, 'Regina');
INSERT INTO teste VALUES (2, 'Andre')
```

**Exemplo de uma tabela utilizando unicidade (Unique):**

No exemplo abaixo vamos criar uma tabela acrescentando a restrição UNIQUE. Esta restrição simplesmente impede que informações de um campo sejam cadastradas mais de uma única vez.

```
CREATE TABLE pessoaunique (
    codigo integer PRIMARY KEY,
    nome varchar(50) NOT NULL UNIQUE
)
INSERT INTO teste VALUES (1, 'Paula');
INSERT INTO teste VALUES (1, 'Regina');
INSERT INTO teste VALUES (2, 'Andre');
INSERT INTO teste VALUES (3, 'Andre')
```

**EXERCÍCIOS**

**1) Crie a tabela abaixo, seguindo exatamente o proposto nos dicionários de dados:**

**TABELA : FABRICANTE**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigodofabricante	integer		
Nomedofabricante	Varchar	10	

1.1) Adicione uma chave primária (codigodofabricante) alterando a estrutura da tabela

1.2) Insira alguns registros

1.3) Selecione os dados

**2) Apague a tabela (drop) e a recrie, seguindo exatamente o proposto nos dicionários de dados abaixo:**

**TABELA : FABRICANTE**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigodofabricante	integer		PK
Nomedofabricante	Varchar	10	NOT NULL UNIQUE

2.1) insira alguns registros

2.2) Selecione os dados

## AULA 6

### INTEGRIDADES DOS DADOS - CRIANDO RESTRIÇÕES

#### **Integridade de domínio**

A integridade de domínio é a validade de entradas para uma coluna específica. É possível aplicar a integridade de domínio para restringir o tipo usando tipos de dados; restringir o formato usando restrições e regras CHECK ou restringir o intervalo de valores possíveis usando as restrições FOREIGN KEY, restrições CHECK, definições DEFAULT, definições NOT NULL e regras.

Restrições de domínio são a forma mais elementar de restrições de integridade. Estas testam valores inseridos no Banco de Dados, e testam (efetua) consultas para assegurar que as comparações façam sentido.

No exemplo que estamos utilizando hoje, a tabela aluno possui vários campos que poderiam ter restrições de entrada.

Podemos adicionar uma restrição de duas formas:

- Adicionando a restrição após a tabela ter sido criada
- Adicionar a restrição no momento da criação da tabela

#### **Criando tabela alunos**

```
CREATE TABLE aluno (  
  matricula varchar(9) PRIMARY KEY,  
  nome varchar(40) NOT NULL  
)
```

#### **Adicionando a restrição após a tabela ter sido criada**

Exemplo 1: Vamos incluir uma restrição no campo sexo da tabela aluno, para que só aceitei F ou M.

```
ALTER TABLE aluno ADD CHECK (sexo='F' or sexo='M')
```

Exemplo 2: Vamos incluir uma restrição no campo nome, para que não aceite valor nulo (null).

```
ALTER TABLE aluno ADD CHECK (nomedoaluno <> '')
```

## **Criar a tabela já com a restrição CHECK**

### **Criando tabela alunos2**

```
CREATE TABLE aluno2 (  
  matricula varchar(9) PRIMARY KEY,  
  nome varchar(40) NOT NULL,  
  sexo varchar(1) CHECK (sexo='F' OR sexo='M')  
)
```

### **Integridade referencial**

A integridade referencial preserva as relações definidas entre tabelas quando linhas são digitadas ou excluídas. No SQL, a integridade referencial baseia-se nas relações entre chaves estrangeiras e chaves primárias ou entre chaves estrangeiras e chaves exclusivas, por meio de restrições FOREIGN KEY e CHECK. A integridade referencial assegura que os valores chave permaneçam consistentes em todas as tabelas. Esse tipo de consistência requer que não haja referências a valores não existentes e que se um valor chave é alterado, todas as referências a ele são consistentemente alteradas em todo o banco de dados.

Quando uma integridade referencial é aplicada, o SQL impede que os usuários façam o seguinte:

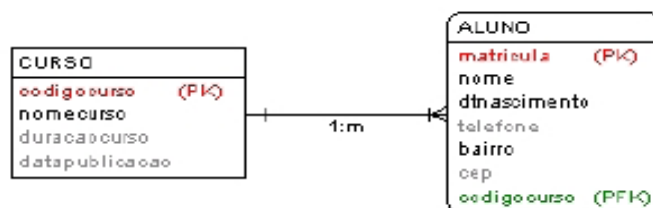
- Adicionar ou alterar linhas a uma tabela relacionada se não houver nenhuma linha associada na tabela primária.
- Alterar valores em uma tabela primária que causa linhas órfãs em uma tabela relacionada.
- Excluir linhas de uma tabela primária se houver linhas relacionadas correspondentes.

## **CRIANDO RESTRIÇÕES REFERENCIAIS**

### **CHAVE ESTRANGEIRA**

#### **Relacionamentos entre tabelas**

Um relacionamento é baseado nos campos em comum que duas ou mais tabelas possuem. Estes relacionamentos são importantes para se manter a integridade das informações. Em modelagem pode ser representado da seguinte forma:



### Adicionando um Relacionamento

Vamos relacionar o `codigocurso` da tabela `CURSOS` com o `codigocurso` da tabela `ALUNO`. Ou seja, criar uma Constraint que referencia a chave estrangeira entre Aluno e Curso.

Para podermos exemplificar, vamos criar as seguintes tabelas:

```
CREATE TABLE cursos (
    codigocurso INTEGER primary key,
    nomedocurso varchar(50) NOT NULL
)

CREATE TABLE aluno (
    matricula integer PRIMAY KEY,
    nomedoaluno varchar(50) NOT NULL,
    codigocurso integer REFERENCES cursos(codigocurso)
)
```

Para cada aluno que for cadastrado, OBRIGATORIAMENTE deverá ter um código de curso cadastrado na tabela `cursos`.

O comando **REFERENCES** está referenciando o campo `codigocurso` da tabela `aluno`, que é a chave estrangeira, com o campo `codigocurso` da tabela `cursos`, que nesta é a chave primária. Ou seja, para se criar uma chave estrangeira, obrigatoriamente deverá existir uma chave primária.



## ADICIONANDO UMA CHAVE ESTRANGEIRA

### Caso tenha esquecido de criar a tabela com a chave estrangeira...

```
DROP TABLE aluno  
ALTER TABLE aluno  
ADD CONSTRAINT cursosalunos FOREIGN KEY (codigocurso) REFERENCES  
cursos(codigocurso)
```

A partir da efetivação deste relacionamento, ao tentarmos cadastrar um aluno com um código de curso que não esteja cadastrado na tabela CURSO, haverá restrição nesta inclusão.

Exemplo: Teremos os seguintes cursos cadastrados na tabela CURSOS

1 – ADMINISTRAÇÃO

2 – CONTÁBEIS

3 – TECNOLOGIA EM PROC.DE DADOS

Com o comando INSERT, insira os dados acima.

Tentaremos inserir na tabela aluno, um curso que não esteja cadastrado:

Exemplo:

```
INSERT INTO ALUNO (matricula, nomedoaluno, codigocurso)  
VALUES  
(200810098, 'ALLAN DA SILVA', 99)
```

O curso 99 não existe na tabela cursos. Neste caso, o registro não será incluído e irá aparecer uma mensagem de erro.

Caso tentemos incluir o aluno com um curso já cadastrado, não teremos problemas:

Exemplo (correto):

```
INSERT INTO ALUNO (matricula, nomedoaluno, codigocurso)  
VALUES  
(200810098, 'ALLAN DA SILVA', 2);
```

**EXERCÍCIOS**

**1) Crie as seguintes tabelas, seguindo exatamente o proposto nos dicionários de dados:**

**FABRICANTE**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigodofabricante	integer		PK
Nomedofabricante	Varchar	10	NOT NULL

**VEÍCULO**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigodoveiculo	integer		NOT NULL
Descricaodoveiculo	Varchar	60	NOT NULL
Anodefabricacao	Integer		>= 1980
Placaveiculo	Varchar	10	NOT NULL
Codigodofabricante	Integer		FK

**2) Adicione a chave primária através do ALTER TABLE na tabela VEÍCULO. A chave primária será codigodoveiculo**

**3) Insira dados em ambas as tabelas, tentando incluir dados inconsistentes, como cadastrar um veículo cujo código do fabricante não exista**

**4) Selecione todos os veículos que o ano de fabricação seja maior ou igual a**

**5) Selecione todos os fabricantes que tenha a letra “F” no campo nomedofabricante.**

## AULA 7

### FUNÇÕES AGREGADAS

#### Funções Agregadas

São funções aquelas que tomam uma coleção (um conjunto ou subconjunto) de valores como entrada, retornando um único valor. O número de funções agregadas na linguagem SQL é vasta. Por isso, veremos alguns exemplos e abaixo, estamos passando um link para que acessem mais funções.

Importante: dependendo do SGBD, poderá haver diferentes funções que executem a mesma coisa entre eles, e até mesmo a sintaxe de funções similares poderão ser diferentes. Por isso, é importante que consultem o manual ou o tutorial do SGBD que resolverem utilizar.

**Para ver mais funções, veja nos links:**

#### Manual PostgreSQL

Funções matemáticas: <http://pgdocptbr.sourceforge.net/pg80/functions-math.html>

Funções caracteres: <http://pgdocptbr.sourceforge.net/pg80/functions-string.html>

#### Manual Mysql

<http://dev.mysql.com/doc/refman/4.1/pt/functions.html>

**Para auxiliar nos exemplos abaixo, vamos criar uma tabela.**

#### VENDEDOR

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Regras
Codigo	integer		PK
Nome	Varchar	40	NOT NULL
Comissao	Float	8,2	NOT NULL

**Inserir alguns registros**

### Funções do SQL

Os exemplos de funções abaixo apresentados fazem parte da linguagem SQL. Funções são comandos prontos para executar determinada tarefa.

#### **COUNT()**

Contagem do número de registros agregados de uma determinada tabela ou junção entre tabelas.

Exemplo:

```
SELECT count(*) FROM vendedor
```

#### **SUM()**

Somatória do conjunto de valores do campo passado como parâmetro.

Exemplo:

```
SELECT sum(comissao) FROM vendedor
```

#### **MIN()**

Retorna o menor valor do conjunto de valores do campo passado como parâmetro.

Exemplo:

```
SELECT min(comissao) FROM vendedor
```

#### **MAX()**

Retorna o maior valor do conjunto de valores do campo passado como parâmetro.

Exemplo:

```
SELECT min(comissao) FROM vendedor
```

### FUNÇÕES PARA STRING

#### **LOWER( string )**

A função LOWER converte string maiúsculas para minúsculas.

Exemplo:

```
SELECT lower(nome) FROM vendedor
```

**UPPER( string )**

A função UPPER converte string de minúsculas para maiúsculas.

Exemplo:

```
SELECT upper( nome) FROM vendedor
```

**SUBSTRING( string, inteiro, inteiro )**

A função SUBSTRING obtém parte de uma string completa.

Exemplo:

```
SELECT substring(nome,4,6) FROM vendedor
```

**FUNÇÕES DE DATA****NOW( ) (POSTGRESQL)****GETDATE ( ) (SQL SERVER)**

Retorna a data e a hora atuais do sistema (timestamp). Podemos, no Query Analyzer , digitar o seguinte comando para obter a data e hora atuais:

```
SELECT now() FROM produtos
```

Exemplo: SELECT GETDATE ( ) → SQL SERVER

NOW() → POSTGRESQL

**DATE\_PART ( unidade , data ) (POSTGRESQL)****DATEPART ( unidade , data ) (SQL SERVER)**

Retorna a parte especificada de uma data como um inteiro. Observe os Exemplos:

POSTGRESQL

```
SELECT DATE_PART('YEAR', now())
```

Resposta: 2012 (para mês MONTH, para dia DAY)

SQL SERVER

```
SELECT DATEPART ( YEAR , '02/01/2004' )
```

Resposta: 2004 (para mês MONTH, para dia DAY)

**DATEDIFF ( unidade , data1,data2 ) → SQL SERVER**

**AGE (timestamp, timestamp) → POSTGRESQL**

Calcula a diferença entre as datas data2 e data1 , retornando o resultado como um inteiro, cuja unidade é definida pelo valor unidade . Observe os exemplos:

Exemplo

**Postgresql:**

```
SELECT age('2001-04-10', timestamp '1957-06-13')
```

SQL SERVER:

```
SELECT DATEDIFF ( DAY , '02/01/2004' , '05/25/2004' )
```

## EXERCÍCIOS

1) Crie a seguinte estrutura de tabela:

### PRODUTOS

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codigoproducto (PK)	integer		PK
Nomedoproducto	Varchar	50	NOT NULL
Dataprimera compra	Datetime		
Valorunitario	Money		

2) Insira as seguintes informações na tabela (caso queira inclua mais produtos):

Codigoproducto	Nomeproduto	dataprimera compra	valorunitario
1	SABONETE	2002-05-05	1.10
2	XAMPU	2001-06-12	7.00
3	CREME DENTAL	2004-03-21	2.50

Através das funções agregadas do sql, faça:

3) Conte quantos produtos temos cadastrados na tabela produtos

4) Some o campo valorunitario da tabela produtos

5) Verifique qual o maior valor unitário

6) Verifique qual o menor valor unitário

7) Verifique a quantos dias estes produtos foram comprados a partir da data da primeira compra

8) Verifique a quantos dias o produto creme dental foi comprado a partir da data da primeira compra

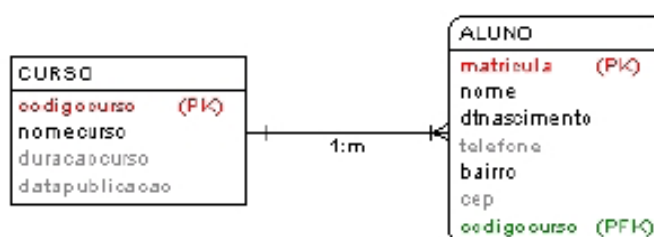
9) Selecione o nome do produto convertendo para minúsculo.

10) Formate o campo data da primeira compra para o formato DD/MM/YYYY

## AULA 8 – CONSULTAS UNINDO DUAS OU MAIS TABELAS PARTE 1

Uma situação muito comum é, ao construirmos uma consulta (select), termos que selecionar dados de duas ou mais tabelas ao mesmo tempo.

Nos bancos de dados relacionais, a maior parte das tabelas possuem ligações entre campos em comum, e para maior segurança, utilizamos para esta ligação chaves primárias e estrangeiras.



Por exemplo, se quisermos uma listagem com as seguintes informações: nome do curso, matrícula do aluno e nome do aluno. Sendo que o nome do curso está na tabela CURSOS e as demais informações estão na tabela ALUNO. O campo que ligará as duas tabelas será o código do curso (codigocurso).

## CLÁUSULA JOIN

Para efetuarmos a junção entre duas ou mais tabelas durante uma consulta (select) utilizamos a cláusula JOIN. Um Join é uma operação que nos permite selecionar dados de duas ou mais tabelas, através de um operador de comparação. O JOIN é efetuado com base em uma coluna que seja comum a estas tabelas, ou seja, serão comparados os valores de colunas provenientes de tabelas associadas.

## TIPOS DE JOIN

Existem três tipos de JOIN: INNER JOIN, LEFT OUTER JOIN e RIGHT OUTER JOIN.



## INNER JOIN

O INNER JOIN é o JOIN padrão. Neste tipo, somente serão retornados os registros que têm valores coincidentes nas duas ou mais tabelas. Veja o exemplo a seguir.

Query - CDPROSANE.aulasx.FEFIS2003\rosane - Untitled1\*

```
SELECT nomedocurso, matricula, nomedoaluno
FROM cursos INNER JOIN aluno ON cursos.codigocurso=aluno.codigodoc
```

	nomedocurso	matricula	nomedoaluno
1	ADMINISTRAÇÃO	200810001	ALDAIR SOUZA
2	CONTÁBEIS	200810002	LUIZA DE ANDRADE
3	TPD	200810003	MARCELLE FERREIRE
4	CONTÁBEIS	200810098	ALLAN DA SILVA
5	ADMINISTRAÇÃO	200810099	ALDAIR SOUZA
6	TPD	200950001	LUIZ CLAUDIO DE OLIVEIRA

## LEFT OUTER JOIN

Este tipo retorna **todos** os registros da primeira tabela (tabela à esquerda) e os registros relacionados da segunda tabela. Vamos utilizar o mesmo exemplo entre CURSOS e ALUNO. Temos que ter cadastrado na tabela CURSOS um curso que não possua alunos associados na tabela ALUNO. No caso deste exemplo, é o curso de geografia que não possui nenhum aluno.

Query - CDPROSANE.aulasx.FEFIS2003\rosane - Untitled1\*

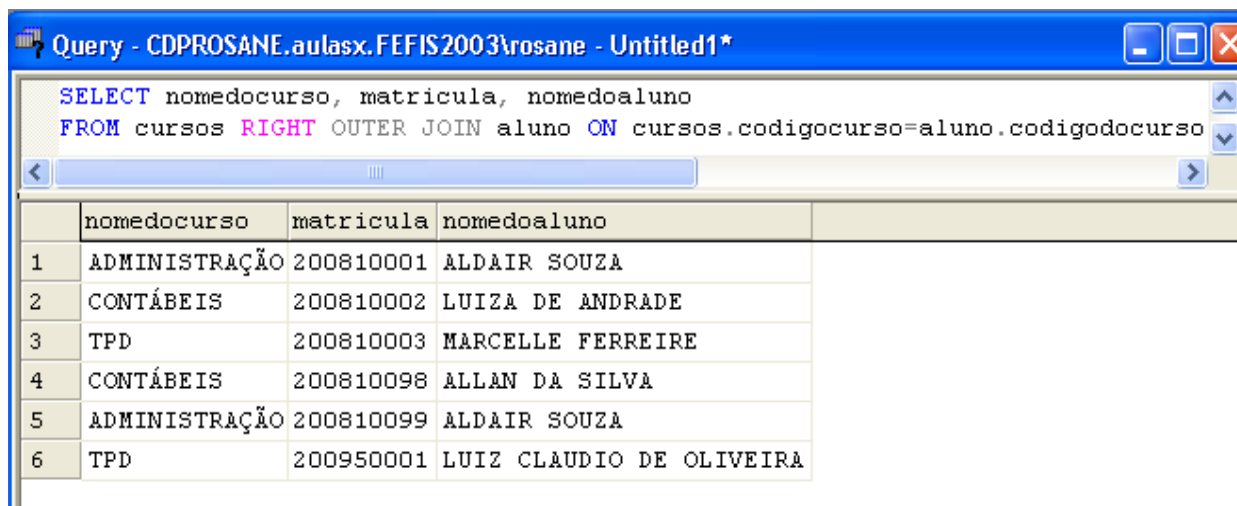
```
SELECT nomedocurso, matricula, nomedoaluno
FROM cursos LEFT OUTER JOIN aluno ON cursos.codigocurso=aluno.codigodoc
```

	nomedocurso	matricula	nomedoaluno
1	ADMINISTRAÇÃO	200810001	ALDAIR SOUZA
2	ADMINISTRAÇÃO	200810099	ALDAIR SOUZA
3	CONTÁBEIS	200810002	LUIZA DE ANDRADE
4	CONTÁBEIS	200810098	ALLAN DA SILVA
5	TPD	200810003	MARCELLE FERREIRE
6	TPD	200950001	LUIZ CLAUDIO DE OLIVEIRA
7	GEOGRAFIA	NULL	NULL

Note que o curso de geografia aparece na consulta, mas a matrícula e o nome estão como nulos.

## RIGHT OUTER JOIN

Este tipo retorna todos os registros da segunda tabela (tabela à direita) e os registros selecionados da primeira tabela.



Query - CDPROSANE.aulasx.FEFIS2003\vrosane - Untitled1\*

```
SELECT nomedocurso, matricula, nomedoaluno
FROM cursos RIGHT OUTER JOIN aluno ON cursos.codigocurso=aluno.codigodocurso
```

	nomedocurso	matricula	nomedoaluno
1	ADMINISTRAÇÃO	200810001	ALDAIR SOUZA
2	CONTÁBEIS	200810002	LUIZA DE ANDRADE
3	TPD	200810003	MARCELLE FERREIRE
4	CONTÁBEIS	200810098	ALLAN DA SILVA
5	ADMINISTRAÇÃO	200810099	ALDAIR SOUZA
6	TPD	200950001	LUIZ CLAUDIO DE OLIVEIRA

Se tivéssemos algum aluno não relacionado com algum curso, iria aparecer a matrícula e o nome do aluno, e o nome do curso iria aparecer NULL.

## EXERCÍCIOS

### Criar as duas tabelas abaixo

TABELA: FUNCIONARIO (chave primária = codfun)

<i>Campo</i>	<i>Tipo</i>	<i>Tamanho</i>	<i>Restrições</i>
Codfun	integer		PK
Nomefun	Varchar	40	NOT NULL

Inserir na tabela FUNCIONARIOS as seguintes informações:

<i>Codfun</i>	<i>Nomefun</i>
1	Valéria
2	Andressa
3	Marcos
4	Vinicius
5	Viviane
6	Carlos

TABELA: DEPENDENTES (chave estrangeira = coddep)

<i>Campo</i>	<i>Tipo</i>	<i>Tamanho</i>	<i>Descrição do campo</i>
Coddep	Integer		PK

<b><i>Campo</i></b>	<b><i>Tipo</i></b>	<b><i>Tamanho</i></b>	<b><i>Descrição do campo</i></b>
Nomedep	Varchar	40	NOT NULL
Idade	Integer		NOT NULL
Codfun	Integer		FK

**Inserir na tabela DEPENDENTES as seguintes informações:**

<b><i>Coddep</i></b>	<b><i>Nomedep</i></b>	<b><i>Idade</i></b>	<b><i>Codfun</i></b>
1	Victor	5	2
2	André	2	3
3	Vitória	12	3
4	Ana Clara	1	5

\*\*\* Os códigos de funcionários 1 e 6 ficarão de propósito sem dependentes vinculados

**PEDE-SE:**

- Criar as tabelas de acordo com o dicionário de dados apresentado
- Criar os relacionamentos (chave estrangeira) entre ambas as tabelas
- Inserir as informações em ambas as tabelas, de acordo com o apresentado
- Fazer consultas unindo ambas as tabelas (JOIN), mostrando código do funcionário, nome do funcionário, código do dependente, nome do dependente e idade do dependente. Pede-se:
- Uma consulta que mostre somente os funcionários que possuem dependentes
- Uma consulta que mostre todos os funcionários, independente de possuírem dependentes ou não.

**AULA 9 – CONSULTAS UNINDO DUAS OU MAIS TABELAS – PARTE 2**

Na aula passada, uma situação muito comum é, ao construirmos uma consulta (select), termos que selecionar dados de duas ou mais tabelas ao mesmo tempo.

Para essa finalidade, utilizamos a cláusula JOIN para estas junções entre tabelas.

Para reforçarmos o conceito de JOIN, vamos fazer alguns exercícios, com base em uma complexidade maior de tabelas.

**Exemplo 1: BANCO DE DADOS PARA CONTROLE DE ESTOQUE**

Enunciado : Criar um banco de dados para controle de entradas e saídas de um estoque.

Criar 3 (três) tabelas, com a seguintes estrutura:

TABELA: CLASSIFICACAO

<b><i>Campo</i></b>	<b><i>Tipo</i></b>	<b><i>Tamanho</i></b>	<b><i>Restrições</i></b>
codigocla	integer		PK
nomecla	Varchar	40	NOT NULL

TABELA: MATERIAIS

<b><i>Campo</i></b>	<b><i>Tipo</i></b>	<b><i>Tamanho</i></b>	<b><i>Restrições</i></b>
codigomat	integer		PK
nomemat	Varchar	40	NOT NULL
quantinicial	Integer		
codigo	Integer		FK (classificacao)

TABELA: MOVIMENTO

<b><i>Campo</i></b>	<b><i>Tipo</i></b>	<b><i>Tamanho</i></b>	<b><i>Restrições</i></b>
Idmov	serial		PK
codigomat	Integer		FK (materiais)
datamov	Date		NOT NULL
quantmov	Integer		NOT NULL
Tipomov	Varchar	1	(E) OU (S) ***

\*\*\* Restringir somente E (para as entradas) e S (para as saídas)

**PEDE-SE:**

- Criar as tabelas de acordo com o dicionário de dados apresentado e suas respectivas chaves primárias
- Criar os relacionamentos (chave estrangeira) entre nas tabelas indicadas acima.
- Inserir as informações nas tabelas, de acordo com o apresentado

Pede-se:

- Fazer consultas unindo as 3 tabelas (INNER JOIN, LEFT JOIN E RIGHT JOIN), sendo que:
  - Uma consulta que mostre somente as entradas
  - Uma consulta que mostre somente as saídas

\*\*\*\* Resolução em sala

**CRIANDO AS TABELAS:**

```
CREATE TABLE classificacao (  
    codigocla integer PRIMARY KEY,  
    nomecla VARCHAR(40) NOT NULL  
  
)
```

```
CREATE TABLE materiais (  
    codigomat integer PRIMARY KEY,  
    nomemat VARCHAR(40) NOT NULL,  
    quantinicial integer,  
    codigocla integer REFERENCES classificacao(codigocla)  
  
)
```

```
CREATE TABLE movimento (  
    idmov SERIAL PRIMARY KEY,  
    codigomat integer REFERENCES materiais (codigomat),  
    datamov date NOT NULL,  
    quantmov integer NOT NULL,  
    tipomov varchar(1) CHECK (tipomov='E' or tipomov='S')  
  
)
```

**INSERINDO DADOS**

SET DATESTYLE TO SQL, DMY

```
INSERT INTO classificacao VALUES (1, 'ESCRITÓRIO');  
INSERT INTO classificacao VALUES (2, 'MANUTENÇÃO');  
INSERT INTO classificacao VALUES (3, 'REFEITÓRIO');  
INSERT INTO classificacao VALUES (4, 'INFORMÁTICA');
```

select \* from classificacao

```
INSERT INTO materiais VALUES (1, 'CANETA', 20, 1);  
INSERT INTO materiais VALUES (2, 'ARROZ', 30, 3);  
INSERT INTO materiais VALUES (3, 'TECLADO', 10, 4);  
INSERT INTO materiais VALUES (4, 'MOUSE', 10, 4);  
INSERT INTO materiais VALUES (5, 'RESMA PAPEL A4', 20, 1);  
INSERT INTO materiais VALUES (6, 'CARNE', 10, 3);
```

SELECT \* FROM materiais

```
INSERT INTO movimento VALUES (default, 1, now(), 5, 'S');  
INSERT INTO movimento VALUES (default, 5, now(), 3, 'S');  
INSERT INTO movimento VALUES (default, 2, now(), 15, 'E');  
INSERT INTO movimento VALUES (default, 6, now(), 5, 'E')
```

SELECT \* FROM movimento

**REALIZAR OS SELECTS UTILIZANDO JOIN**

1. Selecionar os dados das tabelas classificacao e materiais, mostrando somente os dados que possuem junção
2. Selecionar os dados das tabelas classificacao e materiais, mostrando os dados que possuem ou não junção, a partir da tabela da direita
3. Selecionar os dados das tabelas classificacao e materiais, mostrando os dados que possuem ou não junção, a partir da tabela da esquerda
4. Em um único SELECT, selecionar os dados das tabelas classificacao, materiais e movimento, mostrando somente os dados que possuem junção entre as 3 tabelas
5. Em um único SELECT, selecionar os dados das tabelas classificacao, materiais e movimento, mostrando os dados que possuem junção entre as tabelas classificação e materiais, e os dados que possuem ou não junção entre as tabelas materiais e movimento, a partir da tabela da direita
6. Em um único SELECT, selecionar os dados das tabelas classificacao, materiais e movimento, mostrando os dados que possuem junção entre as tabelas classificação e materiais, e os dados que possuem ou não junção entre as tabelas materiais e movimento, a partir da tabela da esquerda

**RESPOSTAS DO EXERCÍCIO**

**SELECT \* FROM classificacao cla**

**INNER JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**ORDER BY** nomecla,nomemat

**SELECT \* FROM** classificacao cla  
**LEFT JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**ORDER BY** nomecla,nomemat

**SELECT \* FROM** classificacao cla  
**RIGHT JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**ORDER BY** nomecla,nomemat

**SELECT \* FROM** classificacao cla  
**INNER JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**INNER JOIN** movimento mov ON mat.codigomat=mov.codigomat  
**ORDER BY** nomecla,nomemat

**SELECT \* FROM** classificacao cla  
**INNER JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**LEFT JOIN** movimento mov ON mat.codigomat=mov.codigomat  
**ORDER BY** nomecla,nomemat

**SELECT \* FROM** classificacao cla  
**LEFT JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**LEFT JOIN** movimento mov ON mat.codigomat=mov.codigomat  
**ORDER BY** nomecla,nomemat

--- **GROUP BY** : A cláusula GROUP BY é usada para agrupar (agregar) as linhas da tabela segundo um critério escolhido pelo utilizados. Utilizado com funções count(\*) e sum()

RETORNANDO A QUANTIDADE DE MATERIAIS POR CLASSIFICAÇÃO  
**SELECT** nomecla, count(\*) **FROM**  
classificacao cla  
**INNER JOIN** materiais mat ON cla.codigocla=mat.codigocla  
**GROUP BY** nomecla



## AULA 10 – CRIANDO VIEWS

### VIEWS (VISÕES)

VIEW é uma instrução SQL que retorna dados e é salva no banco de dados com um nome, ou seja, passa a ser um objeto do banco de dados. Quando uma View é executada, a mesma retorna um conjunto de dados no formato de uma tabela. Uma View pode retornar dados de uma ou mais tabelas.

Uma visão [view] é uma forma alternativa de olhar os dados contidos em uma ou mais tabelas. Para definir uma visão, usa-se um comando SELECT que faz uma consulta sobre as tabelas. A visão aparece depois como se fosse uma tabela.

Visões têm as seguintes vantagens:

- Uma visão pode restringir quais as colunas da tabela que podem ser acessadas (para leitura ou para modificação), o que é útil no caso de controle de acesso.
- Uma consulta SELECT que é usada muito freqüentemente pode ser criada como visão. Com isso, a cada vez que ela é necessária, basta selecionar dados da visão.
- Visões podem conter valores calculados ou valores de resumo, o que simplifica a operação.
- Uma visão pode ser usada para exportar dados para outras aplicações. Geralmente as views são criadas temporariamente.

### Criando uma Visão

Para criar uma View, utilizamos o comando CREATE VIEW. Vamos criar uma View a partir de uma tabela que já existe.

Exemplo:

```
CREATE VIEW vw_curso as  
SELECT * from CURSOS
```

Para executarmos a View, basta darmos um select \* nome\_da\_view.

Para fazermos exemplos de VIEWS, vamos utilizar as tabelas da aula 9.

Exemplo 1 – Utilizando uma tabela

```
CREATE VIEW consulta as  
SELECT * FROM materiais  
CONSULTANDO A VIEW – SELECT * FROM consulta
```

Exemplo 2 – Utilizando duas tabelas

```
CREATE VIEW consmaterial1 as  
SELECT nomecla, nomemat, quantinicial  
FROM classificacao cla  
INNER JOIN materiais mat ON cla.codigocla=mat.codigocla  
ORDER BY nomecla,nomemat
```

```
CREATE VIEW consmaterial2 as  
SELECT nomecla, nomemat, quantinicial  
FROM classificacao cla  
LEFT JOIN materiais mat ON cla.codigocla=mat.codigocla  
ORDER BY nomecla,nomemat
```

Exemplo 3 – Utilizando três tabelas

```
CREATE VIEW consmovimento1 as  
SELECT nomecla, nomemat,quantinicial, datamov, quantmov  
FROM classificacao cla  
INNER JOIN materiais mat ON cla.codigocla=mat.codigocla  
INNER JOIN movimento mov ON mat.codigomat=mov.codigomat  
ORDER BY nomecla,nomemat
```

```
CREATE VIEW consmovimento2 as  
SELECT nomecla, nomemat,quantinicial, datamov, quantmov  
FROM classificacao cla  
LEFT JOIN materiais mat ON cla.codigocla=mat.codigocla  
LEFT JOIN movimento mov ON mat.codigomat=mov.codigomat  
ORDER BY nomecla,nomemat
```

Criando uma View utilizando GROUP BY

--- **GROUP BY** : A cláusula GROUP BY é usada para agrupar (agregar) as linhas da tabela segundo um critério escolhido pelo utilizadores.

**CREATE VIEW mapamov as**

**SELECT nomemat, sum(quantmov) FROM**

**materiais mat**

**INNER JOIN movimento mov ON mat.codigomat=mov.codigomat**

**GROUP BY nomemat**

--- **HAVING** : Cláusula que estabelece condições para listar os grupos. Funciona como a cláusula WHERE.

**CREATE VIEW mapamov2 as**

**SELECT nomemat, sum(quantmov) FROM**

**materiais mat**

**INNER JOIN movimento mov ON mat.codigomat=mov.codigomat**

**GROUP BY nomemat**

**HAVING sum(quantmov) >= 5**

--- **AVG** : Função que calcula a média dos registos do campo informado

**SELECT codigomat FROM**

**movimento**

**GROUP BY codigomat**

**HAVING avg(quantmov) >= 5**

### **Excluindo uma visão**

Para excluir uma visão é só digitar o comando .

Exemplo:

**DROP VIEW consmaterial1**

**EXERCÍCIOS**

- Crie as seguintes tabelas, com o objetivo de controlar músicas de Cds:

CDs

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codigocd (PK)	integer		PK
Nomecd	Varchar	50	NOT NULL
DataCompra	Date		

MUSICAS

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codigomusica	integer		PK
Nomemusica	Varchar	50	NOT NULL
Artista	Varchar	50	
Tempomusica	float		
Codigocd	Integer		FK

- Insira algumas informações nas tabelas, de forma que fiquem com os dados relacionados.
- Crie uma View com os seguintes campos: CODIGOCD, NOMECD, NOMEMUSICA e TEMPO. Mostre todos os CDS de músicas cadastrados, tenha música ou não.
- Crie uma View com os seguintes campos: CODIGOCD, NOMECD, NOMEMUSICA e TEMPO. Mostre somente os CDS de músicas cadastrados, que tenham música.
- Crie uma View que selecione a quantidade de música por CD, utilizando a cláusula GROUP BY.

## AULA 11 – OPERADORES ARITMÉTICOS

Operadores aritméticos, lógicos e comparação são utilizados em consultas SQL para realizar tarefas como somar ou agrupar valores, comparar valores do banco de dados com constantes e variáveis, entre outras. São amplamente usados nos principais banco de dados relacionais.

### Usando operadores aritméticos

Para criar expressões aritméticas em uma consulta SQL usamos os operadores abaixo:

+(somar)

-(subtrair)

\*(multiplicar)

/(dividir)

Os operadores acima podem ser usados apenas em colunas do tipo numérico.

Você poderá usar operadores aritméticos em qualquer cláusula, exceto na cláusula FROM.

### Exemplos:

No exemplo abaixo, estamos verificando a diferença entre o valor da mensalidade e o valor que o aluno pagou.

```
SELECT matricula, vmensalidade, vlbaixa, vmensalidade-vlbaixa as diferenca  
FROM cobranca
```

No exemplo abaixo, estamos calculando 2% de juros do valor da mensalidade.

```
SELECT matricula, vlmensalidade, vlmensalidade*0.02 as multa FROM cobranca
```

### **Precedência de operadores**

Quando usamos vários operadores em uma consulta é importante observarmos qual será a precedência dos operadores. Quando isso ocorrer, vale a pena utilizar parênteses para que determinada operação seja realizada.

#### **Exemplos:**

No exemplo abaixo, vamos calcular os 2% de multa e somar ao valor da mensalidade. Os parênteses ditam as normas de cálculo. Primeiramente, os 2% serão calculados, para somente depois serem somados ao valor da mensalidade.

```
SELECT matricula, vlmensalidade, (vlmensalidade*0.02)+vlmensalidade as multacalculada FROM cobranca
```

### **Operadores de comparação**

Os operadores de comparação são usados em condições que comparam uma expressão a outro valor ou expressão. A tabela abaixo mostra os operadores:

= Igual a

> Maior que

>= Maior ou igual a que

< Menor que

<= Menor ou igual a que

<> Diferente de

**Exemplos:**

Listar todos os alunos, separando sexo igual a F (feminino) ou M (masculinho):

**SELECT matricula, nomedoaluno, sexo FROM aluno WHERE sexo = 'F'**

**SELECT matricula, nomedoaluno, sexo FROM aluno WHERE sexo = 'M'**

Listar todas as cobranças após uma determinada data:

**SELECT matricula FROM cobranca WHERE dtvencimento > '2009-05-12'**

Os operadores de comparação também poderão ser utilizados juntamente com os comandos UPDATE e DELETE.

**EXERCÍCIOS**

1) Crie as seguintes tabelas, objetivando controlar o estoque de alguns produtos:

**ESTOQUE**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codigoproduto (PK)	integer		PK
Nomedoproduto	Varchar	50	NOT NULL
Quantidade	Integer		
Valorunitario	Money		

**ESTOQUEMOV**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Itemmovimento	Integer		PK
Codigoproduto	integer		NOT NULL
Datavenda	Datetime		
Quantidadevenda	Integer		

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Valorvenda	Money		
Codigoproducto	Integer		NOT NULL

3) Crie um FOREIGN KEY entre ambas as tabelas.

4) Inclua os seguintes produtos:

Codigoproducto	Nomeproduto	Quantidade	Valorunitario
1	SABONETE	100	1.10
2	XAMPU	135	7.00
3	CREME DENTAL	141	2.50
4	TALCO	98	4.50

Itemmovimento	Codigoproducto	Datavenda	Quantidadevenda	Valorvenda
1	1	2009-05-02	2	1.50
2	3	2009-05-02	4	9.00
3	1	2009-05-03	3	3.00
4	4	2009-05-03	10	7.00

5) Selecione os seguintes campos da tabela ESTOQUE: nomeproduto, quantidade, valorunitario. Inclua mais uma coluna multiplicando a quantidade ao valor unitário para sabermos o valor que temos em estoque de cada produto

6) Selecione somente os produtos da tabela ESTOQUE que possuem quantidade menor ou igual a 100



## AULA 12 – AGRUPANDO E UNINDO DADOS

Muitas vezes, temos a necessidade de, além de consultar dados de forma tradicional em uma tabela, precisamos também agrupar dados com base nos valores de uma determinada coluna (para depois aplicar alguma função de agregação como COUNT, AVG ou SUM, funções agregadas do SQL). Por exemplo, temos o preço de vários produtos e queremos o somatório destes produtos, sendo que este somatório deverá ser por blocos, que poderá ser um setor, um cliente, um vendedor. Ou seja, poderemos obter como resultado subtotais de alguma coluna de nossa tabela.

### Cláusula GROUP BY

A cláusula GROUP BY agrupa e organiza linhas baseado em semelhanças entre elas. Podemos, por exemplo, agrupar todas as pessoas que nasceram em um determinado mês do ano. O resultado seria o ano, que poderíamos obter da data de nascimento da pessoa, pegarmos somente o ano desta data e aplicarmos a função agregada count(\*). A Partir daí poderemos obter dados estatísticos que poderão ser transformados em relatórios.

A partir da explicação acima, podemos sempre adicionar funções de agregação, tais como COUNT (CONTAR) e SUM (SOMAR), a um bloco selecionado com a utilização de uma cláusula GROUP BY, informações podem ser agregadas. Agregar significa que você pergunta não pelos valores individuais, mas por somas, médias, frequências e subtotais.

Se quisermos o total geral de matriculados em uma tabela aluno, fazemos a seguinte seleção:

```
SELECT count(*) Alunos FROM aluno
```

Sendo que queremos outros tipos de consulta, com subtotais

Exemplo 1 - Vamos exemplificar o exemplo citado acima. Vamos contar quantos alunos temos cadastrados na nossa tabela aluno, agrupando pelo mês da data de nascimento. Assim teremos como resultado o número de aniversariantes em cada

mês:

```
SELECT DATEPART ( MONTH , datanascimento ) AS Mês, count(*) Alunos FROM  
aluno  
GROUP BY  
DATEPART ( MONTH , datanascimento )
```

Note que utilizamos duas funções agregadas: a primeira para selecionar o mês de nascimento e a segunda para contar quantos alunos estão associados aqueles determinado mês. O que esta seleção está fazendo é um agrupamento de registros.

Exemplo 2 – Vamos selecionar todos os alunos, totalizando a quantidade por sexo.

```
SELECT sexo AS Sexo, count(*) Alunos FROM aluno  
GROUP BY  
sexo
```

Exemplo 3 – No exemplo abaixo, seguindo a mesma linha de raciocínio acima, vamos somar os valores de mensalidade, agrupando os totais pelo mês de vencimento. Assim, teremos uma espécie de previsão de receita dentro de cada Mês:

```
SELECT DATE_PART ( 'MONTH' , dtvencimento ) as Mês, SUM(vlmensalidade) as  
Mensalidade FROM cobranca  
GROUP BY  
DATE_PART ( 'MONTH' , dtvencimento )
```

Aqui, não estamos contando como no primeiro exemplo e sim somando o resultado do campo vlmensalidade, através da função agregada SUM().

### **Cláusula HAVING**

Algumas vezes faz-se necessário definir condições em uma determinada seleção e aplicá-las a grupos já definidos (group by). Nesses casos, utilizamos a cláusula having.

Exemplo: Utilizando o exemplo no qual obtemos o somatório de alunos por sexo. A condição para seleção será selecionar somente a quantidade de alunos acima de 2.

```
SELECT sexo AS Sexo, count(*) AS Alunos FROM aluno
GROUP BY
sexo
HAVING count(*) > 2
```

Vamos dizer que, em uma seleção onde temos o total de receitas por mês (exemplo 3 página 1), queremos somente os meses cuja a receita ultrapasse 200.00, por exemplo.

```
SELECT DATE_PART ( 'MONTH' , dtvencimento ) as Mês, SUM(vlmensalidade) as
Mensalidade FROM cobranca
GROUP BY
DATE_PART ( 'MONTH' , dtvencimento )
HAVING SUM(vlmensalidade) > 200.00
```

Vemos então que a cláusula HAVING atua diretamente em uma função agregada como parâmetros para sua execução.

## EXERCÍCIOS

1) Crie as seguintes tabelas no seu banco de dados:

### VENDEDOR

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codvendedor (PK)	integer		PK
Nomvendedor	Varchar	50	NOT NULL
Metavenda	Integer		

**VENDAS**

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Itemvenda	Integer		PK
Codvendedor	integer		NOT NULL
Datavenda	Datetime		
Quantvenda	Integer		
Valorvenda	Money		
Codvendedor	Integer		NOT NULL

3) Crie uma chave estrangeira (FOREIGN KEY) entre ambas as tabelas.

4) Inclua os seguintes vendedores:

Codvendedor	Nomvendedor	Metavenda
1	ROBSON BARROSO	10
2	ABEL	13
3	ALINE	14
4	MICHELON	9

Itemmovimento	Codvendedor	Datavenda	Quantvenda	Valorvenda
1	1	2009-05-02	2	1.50
2	2	2009-05-02	4	9.00
3	3	2009-05-03	3	3.00
4	4	2009-05-03	10	7.00
5	1	2009-05-12	2	1.50
5	2	2009-05-12	4	9.00
6	3	2009-05-23	3	3.00
7	4	2009-05-23	10	7.00

5) Selecione somente a quantidade vendida por cada vendedor

6) Selecione somente o valor vendido por cada vendedor

## AULA 13 – TRANSAÇÕES

*Grande parte dos vários processos existentes no mundo real, como transferências de valores em instituições bancárias, compras em comércio eletrônico e outros serviços utilizam transações que são um atributo ou artifício comum em bancos de dados comerciais.*

*“Transação é uma unidade lógica de trabalho, envolvendo diversas operações de bancos dados.”*

*(C. J. Date – Introdução a Sistemas de Bancos de Dados –  
página 63)*

Imaginem transações como sendo ações referentes a pagamento de contas pela internet nos sites dos bancos, compras em comércio eletrônico, dentre outros serviços. Caso em um determinado momento em que a transação não é aquilo que você deseja realizar e caso ainda não tenha sido finalizada, podemos voltar atrás e cancelar a mesma.

No caso das transações no SQL, seguem este mesmo princípio. Quando precisamos inserir, alterar ou excluir um ou mais dados de uma tabela no banco de dados, temos que ter a certeza que estes dados estão íntegros e consistentes. Ou grava-se os dados de forma correta e consistente ou o melhor é que os mesmos não sejam gravados. Este tipo de transação também é chamado de two phase commit (commit de duas fases, pois ou tem os dados, ou não tem nada).

Basicamente as transações no SQL Server são feitas utilizando três comandos:

- BEGIN TRANSACTION – Marca o início da transação
- COMMIT – Marca o término da transação, caso não haja erros de sintaxe na execução da mesma.
- ROLLBACK – Reverte uma transação.

A sintaxe da utilização de uma transação ficaria mais ou menos assim:

Exemplo 1: Inicia transação e cancela

BEGIN TRANSACTION

\* COMANDOS

ROLLBACK

Exemplo2: Inicia transação e dá continuidade

```
BEGIN TRANSACTION
```

```
* COMANDOS
```

```
COMMIT
```

Abaixo um exemplo de código utilizando uma transação:

```
BEGIN TRANSACTION
```

```
UPDATE TABELA SET CAMPO1 = 'NOVO VALOR'
```

```
WHERE CAMPO2 = 35
```

```
ROLLBACK
```

Quanto iniciamos um comando SQL com um BEGIN TRANSACTION, toda a ação somente será executada em definitivo após o comando COMMIT. Caso exista algum erro de sintaxe no processamento, o comando ROLLBACK não finaliza o comando e tudo fica como antes da transação.

Perceba que no exemplo acima, iniciamos a transação com o BEGIN TRANSACTION. Vamos supor que a tabela possua 1000 registros que satisfaçam a condição e que por algum motivo, no registro 61 houve um problema, e o mesmo foi percebido pelo usuário. Neste caso, podemos executar o comando ROLLBACK para reverter a situação. Imaginem se não houvesse uma forma de reverter a transação, e isso fosse influenciar uma instituição financeira, por exemplo ?

No caso de termos certeza de que o comando está correto e de que as informações que estão sendo alteradas são mesmo aquelas, podemos finalizar a transação com o comando COMMIT.

```
BEGIN TRANSACTION
```

```
UPDATE TABELA SET CAMPO1 = 'NOVO VALOR' WHERE CAMPO2 = 35
```

```
COMMIT
```

Segundo Mauro Pichialini, algumas dicas interessantes quanto ao uso de transações:

1. Mantenha transações curtas, ou seja, não coloque muitas instruções SQL entre o BEGIN TRANSACTION e o COMMIT. Apesar de outros usuários enxergarem os dados de uma transação que ainda não fez o COMMIT, os dados NÃO ficam gravados no SQL e sim em arquivos temporários do Windows (os famosos arquivos .tmp)

2. Vamos supor que alguém resolva utilizar a mesma tomada do computador servidor em que o SQL Server está rodando para ligar uma cafeteira, quando uma transação foi iniciada mais ainda não fez o COMMIT. Neste caso, quando o serviço do SQL Server foi reiniciado, todas as transações que ainda não executaram o COMMIT voltarão ao seu estado inicial antes do BEGIN TRANSACTION.

3. Procure sempre dar nomes as transações. Isso obriga o utilização do TRANSACTION no COMMIT e no ROLLBACK. Exemplo:

```
BEGIN TRANSACTION TRAN_01
DELETE FROM TABELA1
IF @@ERROR <> 0
ROLLBACK TRANSACTION TRAN_01
ELSE
COMMIT TRANSACTION TRAN_01
```

Podemos utilizar os comandos de transação nas cláusulas INSERT, UPDATE e DELETE.

A utilização de transações durante a execução de comandos SQL traz uma certa segurança, no sentido de finalizar a execução somente se a mesma for válida.

## EXERCÍCIOS

1) Crie uma tabela com os seguintes campos :

### BOATE

NOME DO CAMPO	TIPO DO CAMPO	TAMANHO	Restrições
Codigo	integer		NOT NULL
Nome	Varchar	60	NOT NULL
Sexo	Varchar	1	
Ingresso	Float	5,2	

2) Insira os seguintes dados:

<b>Codigo</b>	<b>Nome</b>	<b>Sexo</b>	<b>Ingresso</b>
101	Amanda	F	25,00
102	Fernanda	F	25,00
103	Flávio	M	15,00
104	Marcio	M	15,00

3) Utilizando transação, altere todos os valores de ingresso, do sexo masculino, para 38,00. Faça um select da tabela. Depois cancele a operação.

4) Utilizando transação, altere todos os valores de ingresso, do sexo masculino, para 42,00. Faça um select da tabela. Depois confirme a operação.



## **AULA 14 – INDEXAÇÃO DE TABELAS**

Quando pensamos em desenvolver uma aplicação que irá acessar informações em um banco de dados, várias situações deve-se ter em mente: código fonte, formulários, menu de acesso, módulos, etc.

Além das situações citadas acima, devemos também pensar na performance deste banco de dados, principalmente se este tiver um grande número de acesso.

A indexação de tabelas é uma forma de otimizar uma aplicação com acesso a banco de dados, melhorando assim sua performance. Quando utiliza-se indexação no seu banco de dados, pode-se aumentar e muito a velocidade de acesso às informações em um banco de dados.

### **USO DE ÍNDICES E SEUS BENEFÍCIOS**

Como foi dito anteriormente, o uso de índices em tabelas melhora a performance do banco de dados. Quando uma consulta é realizada e existe uma demora no retorno dos dados consultados, é sinal de que é necessário criar um índice. É claro que, cada caso é um caso. O uso de índices pode sim, melhorar o desempenho do banco de dados, mas quando este não resolve, deve-se buscar outros meios para melhoria da performance. O uso da indexação pode ser utilizada como primeira alternativa para ganho de performance. O benefício da utilização de índices pode ser mais percebida quando precisamos executar uma consulta envolvendo tabelas ou mais.

Exemplo:

Imaginemos três tabelas: tabela1, tabela2 e tabela3, cada uma contendo 1 coluna cada: coluna1, coluna2 e coluna3, e estas são os campos que unirão as três tabela. Além do mais, cada tabela possui mais de 1000 registros.

A consulta para unir as três tabelas, ficaria mais ou menos assim:

```
SELECT tab1.coluna1, tab2.coluna2, tab3.coluna3 FROM tabela 1 as tab1,
tabela2 as tab2, tabela3 as tab3 WHERE tab1.coluna1=tab2.coluna2 AND
tab2.coluna2=tab3.coluna3
```

Quando temos um tipo de consulta semelhante a exemplificada acima, com uma quantidade bem grande de registros armazenados, o banco de dados faz várias tentativas de combinações para encontrar os registros que combinam com a condição da cláusula WHERE. Ou seja, haverá um grande esforço do banco de dados para retornar as informações. Imagine como ficará a performance do banco de dados, conforme o número de registros for aumentando e, na mesma proporção, aumentar o número de acessos por parte dos usuários executando esta mesma consulta.

Ao criar-se um índice, o banco de dados não ficará tão sobrecarregado efetuando as combinações entre registros, melhorando significativamente a velocidade no retorno dos dados em uma consulta.

## **COMO ESCOLHER OS ÍNDICES**

Deve-se escolher colunas para serem utilizadas como índice, aquelas ligadas a pesquisas, ordenação ou agrupamento. Ou seja, as colunas candidatas são aquelas que aparecem nas cláusulas WHERE, JOIN, ORDER BY ou GROUP BY.

O uso de índices em excesso também não é benéfico. Utilize índices caso exista realmente necessidade. Ao criar-se um índice, na tabela é criada para armazená-los, e espaço adicional em disco é gerado.

Outra dica, os índices devem ser atualizados e reorganizados periodicamente, principalmente quando o conteúdo de uma das tabelas forem modificados.

## **COMANDO CREATE INDEX**

O comando CREATE INDEX é utilizado na linguagem sql para gerar um índice na tabela desejada.

O comando CREATE INDEX é encontrado nos SGBDS MySQL, PostgreSQL e SQL SERVER. A sintaxe e os argumentos podem variar de SGBD.

Exemplo:

No exemplo abaixo, vamos criar um índice através de uma sintaxe bem básica.

Sintaxe básica para criação de índices:

```
CREATE INDEX nome_indice ON tabela_nome (coluna_nome);
```

Vamos imaginamos uma tabela onde estão armazenados dados de alunos. Uma das pesquisas mais utilizadas é a por nome. A criação de um índice para esta tabela ficaria assim:

```
CREATE INDEX idx_nome ON alunos (nome)
```

Onde:

Idx\_nome -> nome da tabela que armazenará os índices

Alunos -> nome da tabela

(nome) -> nome do campo (coluna) a ser indexado

Pode-se também criar um índice para evitar dados duplicados. Para isso, deve-se utilizar o parâmetro UNIQUE. Este parâmetro verifica dados duplicados em uma tabela quando o índice é criado. Toda vez que houver tentativa de entrada de uma informação duplicada, resultará em erro.

Exemplo:

```
CREATE UNIQUE INDEX idx_nome ON alunos (nome)
```

*Nota: No caso de nome, não é usado índice único, visto que existem homônimos.*

## EXERCÍCIOS

- 1) Qual a vantagem de criar-se índices ?
- 2) Qual o parâmetro utilizado para tornar o índice único ?
- 3) Escreva o código para a criação de um índice, referente a uma tabela chamada livros, cuja coluna a ser indexada chama-se título.

## AULA 15 – CRIANDO FUNÇÕES

As funções de banco de dados mais comuns são as funções de agregação `sum()`, `count()`, `avg()`, `min()` e `max()`, utilizadas para operações simples como soma, contagem, média, mínimo e máximo.

O PostgreSQL possui também uma gama de funções diversas para operações aritméticas, conversões entre tipos de dados, funções de formatação e geográficas, entre outras. Mas além dessas funções pode-se criar uma infinidade de outras, definidas pelo usuário, permitindo encapsular código requerido para tarefas comuns. Tais funções podem ser criadas em linguagem C, SQL, PL/PGSQL, PL/TCL, PL/PERL e até PHP (com suporte experimental).

Sintaxe básica para criar funções:

```
CREATE FUNCTION nomedafuncao(variavel1, variavel2, ...)
returns varchar as
comandos
language 'SQL';
```

Explicando os comandos:

**CREATE FUNCTION** – comando para criar uma função

**nomedafuncao(variaveis)** – toda função precisa ter um nome, e pode ser que necessite de parâmetros. Estes parâmetros, caso a função tenha, fica entre parênteses.

Vamos criar uma função para selecionar o nome de um aluno na tabela `alunos` (aulas passadas), sendo que iremos selecionar somente os alunos de um determinado curso.

Crie a tabela (veja na aula 2) e insira alguns registros.

```
CREATE FUNCTION selecionacurso(int)
returns varchar as
'select c_nomealuno from alunos where n_codcurso=$1'
language 'SQL';
```

Após ter criado a função, poderemos executá-la:

```
select selecionacurso(4)
```

Vamos criar a seguinte tabela:

```
create table pessoas (
nome varchar(40),
idade integer
)
```

Vamos agora inserir algumas informações, diferenciando as idades.

Agora, vamos criar uma função de irá selecionar somente os nomes cuja as idades que forem maiores ou iguais as que solicitarmos a função.

```
CREATE FUNCTION selecionapessoasmaiores(int)
returns varchar as
'select nome from pessoas where idade>=$1'
language 'SQL';
```

Vamos selecionar somente pessoas com idade maior ou igual a 18:

```
select selecionapessoasmaiores(18)
```

**EXERCÍCIOS**

1. Criar a seguinte tabela:

```
create table pessoas (  
  nome varchar(40),  
  idade integer,  
  sexo varchar(2)  
)
```

2. Insira alguns registros

3. Criar uma função que selecione e retorne os alunos de um determinado sexo, dependendo do parâmetro passado (F ou M)

## AULA 16 – TRIGGER

Ao trabalhar com base de dados Cliente/Servidor como SQL Server , Oracle , Informix , dentre outras, podemos usar um recurso muito poderoso chamado Trigger.

Um Trigger é bloco de comandos Transact-SQL. Quando um comando INSERT , DELETE ou UPDATE for executado em uma tabela do banco de dados, o Trigger é executado de forma automática.

Os Triggers são amplamente utilizados para a realização de tarefas relacionadas com validações, restrições de acesso , rotinas de segurança e consistência de dados; assim, estes controles deixam de ser executados pela aplicação e passam a ser executados pelos Triggers em determinadas situações, de acordo com Macoratti :

- Mecanismos de validação envolvendo múltiplas tabelas
- Criação de conteúdo de uma coluna derivada de outras colunas da tabela
- Realizar análise e e atualizações em outras tabelas com base em alterações e/ou inclusões da tabela atual

A criação de um Trigger envolve duas etapas :

- 1) Um comando SQL que vai disparar o Trigger ( INSERT , DELETE , UPDATE)
- 2) A ação que o Trigger vai executar ( Geralmente um bloco de códigos SQL )

A utilização de triggers possuem algumas limitações, também segundo

Macoratti:

- Não é possível criar um Trigger para uma visão
- O resultado da execução de um Trigger é retornado para a aplicação que o chamou.
- O comando WRITETEXT não ativa um Trigger
- O comando TRUNCATE TABLE não pode ser reconhecido por um Trigger
- Não podemos usar em um Trigger os seguintes comandos SQL :
  - *ALTER DATABASE , ALTER TRIGGER , ALTER PROCEDURE , ALTER*

*TABLE , ALTER VIEW . CREATE DATABASE , CREATE INDEX , CREATE PROCEDURE, CREATE SCHEMA, CREATE TABLE , DROP DATABASE, DROP TABLE , DROP PROCEDURE, DROP TRIGGER, DROP INDEX, GRANT , LOAD DATABASE, REVOKE, RESTORE DATABASE, TRUNCATE TABLE.*

A diferença da trigger para stored procedure, é a forma de execução. Uma trigger é disparada no momento da execução de um UPDATE, INSERT OU DELETE.

A sintaxe para a criação de uma trigger é apresentada abaixo:

```
CREATE TRIGGER nome { BEFORE | AFTER } { evento [ OR ... ] }  
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE  
nome_da_função ()
```

Exemplo:

No exemplo abaixo, estamos calculando o valor de bolsa para um determinado valor. O objetivo da trigger será calcular a bolsa, que será de 5%, de forma automática, no momento que inserir a informação na tabela.

Primeiramente, vamos criar uma tabela bem pequena e simples:

```
create table cob (  
valor numeric(10,2),  
bolsa numeric(10,2)  
)
```

Agora, vamos criar a função para gerarmos a trigger.

```
CREATE OR REPLACE FUNCTION bolsa()  
  
RETURNS trigger  
  
AS $$  
  
BEGIN  
  
update cob set bolsa = valor*0.05;
```



```
RETURN new;  
END;  
$$ language 'plpgsql';
```

Agora vamos “embutir” a função na tabela, criando uma trigger e vinculando a inclusão de valores ao cálculo da bolsa.

```
CREATE TRIGGER insert_bolsa AFTER INSERT  
ON cob FOR EACH ROW  
EXECUTE PROCEDURE bolsa();
```

Agora, vamos inserir um valor qualquer na tabela

```
insert into cob (valor) values (10.00)
```

Agora, verifique o resultado:

```
select * from cob
```

## EXERCÍCIO

Para fazer o exercício abaixo, crie uma tabela (ou utilize de exercícios anteriores) chamada receita, com as colunas dtvencimento (data de vencimento), dtbaixa (data da baixa), vlapagar (valor a pagar) e vldesconto (valor do desconto). Insira alguns dados, preenchendo somente as colunas vlmensalidade e dtvencimento, com valores e datas diversificadas.

Crie um TRIGGER para vamos gravar o valor de desconto na tabela cobrança. Iremos precisar do seguinte:

O desconto a ser aplicado deverá ser de 10% sobre o valor a pagar. Grave o valor calculado na coluna vldesconto.

O desconto somente poderá ser aplicado caso a data da baixa for menor do que a data do vencimento