**Unidade de Engenharia e Computação**
Disciplina Banco de Dados - Profª. Eliana Caus Sampaio

# Introdução à linguagem

# SQL

# ÍNDICE

# Introdução

Esta apostila está permanentemente em elaboração. Por isso, muitos ajustes deverão ser realizados até que alcance um nível razoável de corretude e satisfação.

*Eliana Cáus Sampaio*

# 1. Histórico da Linguagem SQL

SQL é uma abreviação de *Structured Query Language* (Linguagem Estruturada de Consulta). É uma linguagem padrão de definição e acesso de bancos de dados relacionais. A maioria dos servidores de bancos de dados utilizados em aplicativos cliente/servidor trabalha com SQL.

A SQL não é uma linguagem especificamente criada para desenvolver sistemas, como por exemplo COBOL, VISUAL BASIC, PASCAL, mas sim uma linguagem de consulta, ou seja, foi criada para extrair, organizar e atualizar informações em bancos de dados relacionais (MANZANO, 2002).

Possivelmente, o leitor deste material já deverá estar familiarizado com conceitos relacionados a banco de dados, porém, considerando a não familiaridade quanto a estes assuntos, cabe aqui um breve comentário sobre a origem da SQL. Não podemos falar de SQL sem citarmos a sua finalidade, que é a de manipular estruturas armazenadas dentro de um Banco de Dados Relacional. Ao longo do processo de desenvolvimento de mecanismos mais eficientes de armazenamento de dados, em 1970, E. F. Codd, cientista da IBM, tomando por base os conhecimentos da matemática, e dentro desta, da teoria de conjuntos, propôs uma nova forma de gerenciar o armazenamento de dados, denominado Sistema Relacional. O termo relacional foi usado para denominar essa nova linha de produto, dado a sua base ter sido gerada a partir dos fundamentos da Álgebra Relacional. Os SGBD (Sistemas Gerenciadores de Bancos de Dados) construídos sobre esta filosofia denominam-se SGBDR (Sistemas Gerenciadores de Bancos de Dados Relacionais), no qual, os dados são sempre “vistos” pelos usuários como se fossem tabelas. Tabelas são estruturas bidimensionais, onde na vertical temos as colunas e na horizontal as linhas. O primeiro produto relacional construído foi o System/R da IBM.

Em 1974, Donald Chamberlin e outros profissionais da IBM propuseram uma das primeiras versões de uma linguagem para interação com Bancos de Dados Relacionais. Na época essa linguagem chamava-se ainda SEQUEL (*Structured English Query Language*). A SEQUEL foi implementada pela primeira vez em grande escala no SYSTEM/R. Após esta primeira aparição, houveram outras versões da SEQUEL e somente em 1976 foi denominada SQL (Structured Query Language) (OLIVEIRA, 2002 e MANZANO, 2002).

A partir daí várias outras empresas lançaram os seus SGBDR, como a ORACLE, SYBASE. Isso gerou uma grande variedade de empresas criando e modificando a linguagem SQL originalmente concebida. Nesse cenário, o ANSI (*American National, Standards Institute*) estabeleceu em 1982 normas e critérios técnicos para a linguagem SQL, denominada ANSI/SQL. Este trabalho estendeu-se até 1986 quando efetivamente surgiu o primeiro padrão para a linguagem SQL denominado “SQL/86”. Posteriormente, a partir da parceria ANSI e o ISO, novos incrementos foram feitos ao padrão anteriormente concebido, surgindo em 1989 o “SQL/89”. Novas revisões foram feitas dando origem em 1982 ao “SQL/92”. E por fim, em 1999 foi feita a quarta revisão do padrão, surgindo a “SQL/99” também conhecida como SQL3. Dessa maneira, distingue-se 5 momentos da linguagem (OLIVEIRA, 2002, p.18 e MANZANO, 2002):

1º) SEQUEL (1974),
2º) SQL(1976),
3º) SQL/86(1982 – 1986),

4º) SQL/92 (1992),
5º) SQL/99 (1999).

No presente momento existe uma nova edição do padrão referenciada por SQL/2003.

Para concluir, embora a SQL/ANSI seja o padrão para os fabricantes de SGBD, cada um destes criou extensões da linguagem para explorar elementos específicos do seu produto, produzindo assim, variações da SQL dentro de cada SGBD. Assim, encontramos a Transact/SQL da Sybase e Microsoft, PL/SQL da Oracle e DB2 da IBM, como algumas das mais conhecidas variações da SQL (RAMALHO, 1999).

> Embora a sintaxe básica de uma declaração SQL seja a mesma, os bancos de dados interpretam de forma diferente uma mesma declaração. Alguns exigem uma clausula enquanto outros a consideram opcional. O SQL Server 7 da Microsoft aceita o uso do comando INSERT com ou sem a clausula INTO. Já o ORACLE e o SQLBase exigem a clausula INTO. (RAMALHO, p.41, 1999)

O padrão SQL-92 divide-se em: *Entry* (básico)*, Transational* (em evolução)*, Intermediate* (intermediário) *e Full* (completo)*.* A maior parte dos bancos de dados utilizados atualmente atende ao nível básico. Mesmo existindo outras versões mais recentes do padrão, a maior parte dos bancos de dados ainda utiliza, de forma básica o padrão anterior. Alguns comandos, contudo, atingem os níveis intermediário e completo (OLIVEIRA, 2002, p.18). Este é o padrão para os comandos testados nesse material.

Diz-se que a maioria dos produtos Relacionais disponíveis no mercado, utilizam um super-conjunto de um sub-conjunto do padrão SQL. Em outras palavras, os produtos existentes não chegam a utilizar todos os recursos oferecidos pelo padrão, mas em alguns aspectos, os recursos oferecidos estendem em muito ao definido pelo padrão, alcançando características dos padrões seguintes.

# 2. Convenções Adotadas

Falar de pontuação não é uma tarefa fácil, pois ela tem significados em SQL diferentes do português ou de qualquer outra linguagem falada. Os exemplos de pontuação recomendados abaixo podem ser adotados em qualquer SGBD, mas são na verdade integralmente suportados pelo Oracle e o Access (PATRICK, 2002, p.95 a p.113).

- **Espaços em nomes** → Evite-os sempre que puder. Nomes de tabelas, colunas ou de qualquer objeto do banco de dados, caso tenham mais do que um termo devem ser ligados por um underscore.
- **Vírgula** → Vírgulas separam os itens de uma lista. Não deve ser usada para números para a definição do ponto decimal. Nesse caso, utilizar o ponto.
- **Apóstrofos** → Também chamado de aspas simples são usados em strings de texto e de datas. Em alguns SGBD (Access por exemplo), os campos de data são envolvidos por um sustenido (#) e não por apóstrofos. Para utilizar o apóstrofo dentro do texto, basta digitar dois apóstrofos um após o outro.
- **Aspas** → Em alguns SGBDs, apóstrofos e aspas possuem o mesmo significado, mas no Oracle possuem significado diferente. No ORACLE, as aspas são utilizadas para envolver um alias de coluna que contém um caracter especial ou um espaço.
- **Ponto_e_virgula** → Marca o fim de uma declaração SQL
- **Hífen duplo**→ Em alguns SGBD, o hífen duplo significa comentário.
- **Ponto**→ Em alguns SGBD, o ponto é utilizado entre o nome de uma coluna e o nome de uma tabela para indicar que a coluna pertence aquela tabela. Também denominado ***Dot Notation***. No Access, o ponto de exclamação cumpre o mesmo papel.
- **Curingas** → os curingas são utilizados em uma clausula WHERE com uma condição LIKE.

| Caracter | Significado |
|---|---|
| % (por cento) | é usado para indicar um numero qualquer de caracteres ou então nenhum caracter. Em alguns SGBD, o * (asterisco) cumpre esse mesmo papel. |
| _ (underscore) | é usado para indicar exatamente um caractere desconhecido. Em alguns SGBDs, a ? (interrogação) cumpre esse mesmo papel |
| # (sustenido) | é usado para indicar exatamente um digito de 0 a 9. |
| [a-d] (colchetes) | são usados para indicar um intervalo de caracteres. Nesse caso, o intervalo é de a a d |
| [*] (colchetes fechando um caracter curinga) | indica o próprio caractere sem suas propriedades de curinga |

Tabela 1 – Caracteres especiais utilizados em comandos SQL

# 3. Panorâmica da Linguagem SQL

A linguagem SQL surgiu originalmente para dar suporte somente a leitura dos dados armazenados. Posteriormente, foi incrementada para dar suporte também as demais funções de criação e manipulação de dados, possibilitando assim a construção, alteração e destruição das estruturas de dados, bem como a alimentação e consulta dos dados dentro desta estrutura. Dessa maneira, linguagem SQL pode ser dividida em:

1) **DDL (Data Definition Language)**: Linguagem de Definição de Dados. Refere-se ao conjunto de instruções que nos permitem criar/modificar as estruturas de dados dentro de um SGBD. O resultado de uma instrução DDL afetará o Dicionário de Dados ou Catálogo do Banco de Dados.

2) **DML (Data Manipulation Language)**: Linguagem de Manipulação de Dados. É uma linguagem de consulta baseada na Álgebra Relacional e no Cálculo Relacional de Tupla. Além da consulta, compreende também os comandos para inserção, alteração e exclusão dos dados armazenados.

| **Linguagem SQL** | | | |
|---|---|---|---|
| **DDL** | | **DML** | |
| **Comando** | **Função** | **Comando** | **Função** |
| CREATE | Cria o objeto | SELECT | Lê os dados de uma tabela/visão |
| DROP | Destrói o objeto | INSERT | Insere linhas em uma tabela/visão |
| ALTER | Altera o objeto | UPDATE | Altera linhas em uma tabela/visão |
| GRANT | Concede privilégios | DELETE | Exclui linhas em uma tabela/visão |
| REVOKE | Retira privilégios | COMMIT | Confirma a transação |
| | | ROLLBACK | Desfaz a transação |

Tabela 2 – Comandos da linguagem SQL

Em algumas bibliografias pode-se encontrar uma classificação um pouco diferente. DDL, DML, DCL e DQL, correspondendo a *Data Definition Language* (criação das estruturas), *Data Manipulation Language* (inserir, alterar e excluir dados na estrutura), *Data Query Language* (leitura) e *Data Control Language* (concessão e retirada de privilégios) (STEPHENS, PLEW, 2003).

A figura 1.1 demonstra os passos que uma instrução DDL ou DML percorre dentro da estrutura do SGBD. Possivelmente o leitor dessa apostila já deverá estar familiarizado com os conceitos relativos a Estrutura de um SGBD. Por isso, tornar a abordar tais tópicos, foge ao escopo. No entanto informações mais detalhadas sobre tal assunto poderão ser encontradas em notas de aula das disciplinas de Banco de Dados, disponíveis na página do curso.

A execução de cada uma das operações vindas do mundo externo (DBAs, Usuários, Programadores, etc) requer do SGBD o uso de diversos programas ou componentes funcionais (processamento de consulta ou gerenciamento de memória) bem como de diversos arquivos (estruturas de dados) conforme demonstra a figura 1 abaixo.

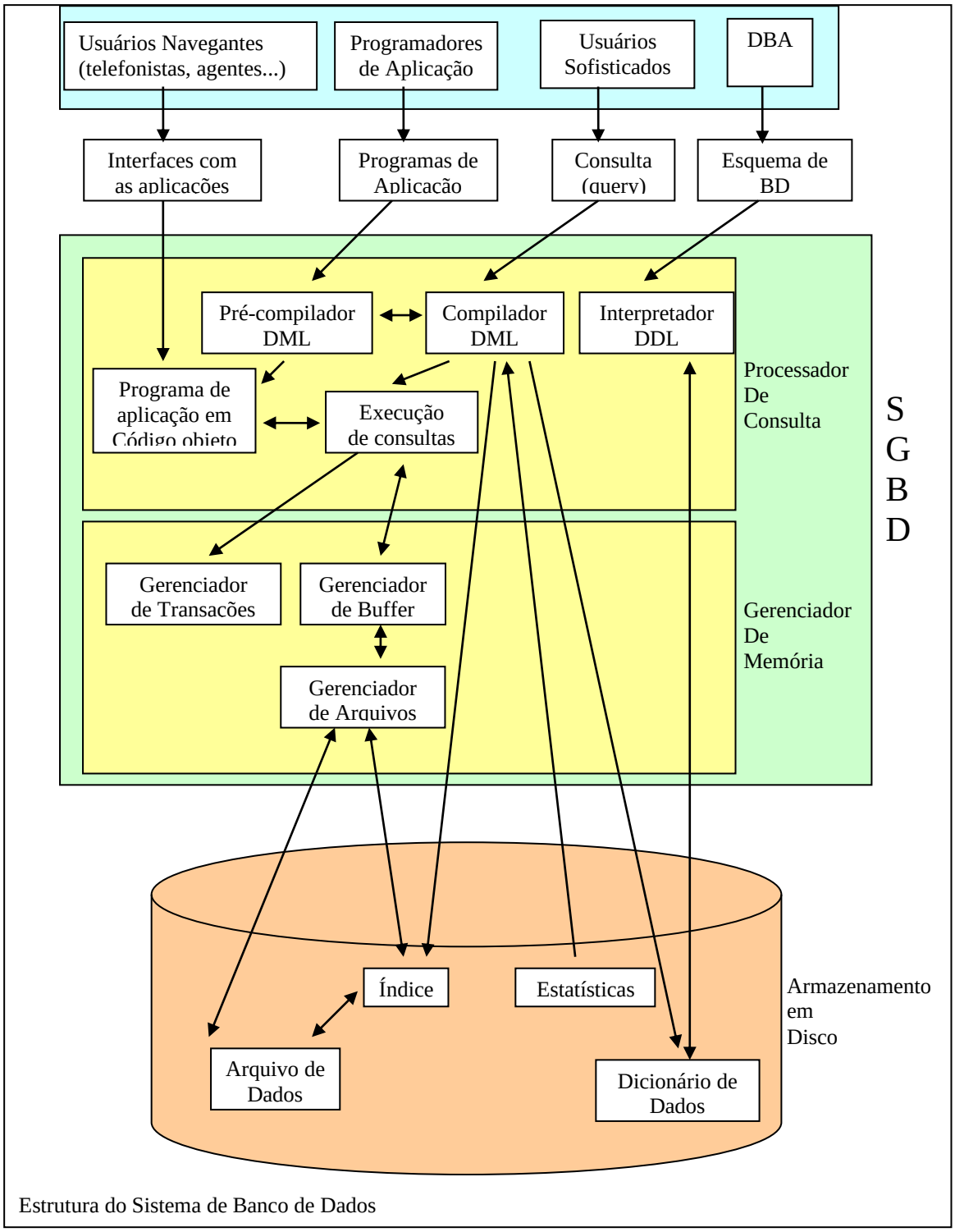



Figura1 – Estrutura do Sistema de Banco de Dados (Silberschatz, 1999)

# 4. Data Definition Language - DDL

Nesta seção abordaremos as instruções SQL que serão utilizadas para a construção, modificação ou destruição das estruturas de dados em um banco de dados. Em um banco de dados relacional é possível criarmos, mantermos ou destruirmos diversos objetos, como por exemplo: Tabelas, *Views*, Índices. Nesta seção, trabalharemos os comandos CREATE (Cria a estrutura), ALTER (altera a estrutura) e DROP (destrói a estrutura).

Para podermos exemplificar cada um dos comandos SQL, utilizaremos o modelo de dados ACADÊMICO encontrado no ANEXO1.

## 4.1 – Criando objetos - CREATE

A instrução CREATE é utilizada para criar as estruturas dentro do banco de dados. Podem ser criadas TABLES (TABELAS), *VIEWS* (Visões sobre uma ou um conjunto de tabelas ou *views*), INDEX (Índices), etc.

**4.1.1 – Criação de tabelas:** O comando de criação de uma nova tabela faz-se através do comando CREATE TABLE. A nova tabela deverá ser única dentro do banco de dados onde a estrutura está sendo criada.

Formato do comando CREATE TABLE:

**CREATE TABLE** <table_name>
(<column_name> <datatype> [not null] [not null] [,])

onde:

<table_name> Corresponde ao nome da tabela que será criada. Este nome é de livre escolha do usuário, lembrando somente que, dentro de um mesmo banco de dados, as tabelas devem ter nomes distintos. Em bancos de dados distintos, as tabelas podem ter nomes iguais.

<column_name> Corresponde ao nome da coluna dentro da tabela. Também de livre escolha do usuário, lembrando somente que dentro de uma mesma tabela, as colunas devem ter nomes distintos. Em tabelas distintas, podem haver colunas com nomes iguais. Cada coluna representa um pedaço da estrutura da tabela. Respeitando a estrutura bidimensional proposta no modelo relacional, a coluna representa a visão vertical da tabela e será responsável por armazenar uma característica do objeto modelado e armazenado pela tabela.

<datatype> Especifica o tipo de dado ao qual a coluna estará associada. De acordo com o SGBD utilizado será oferecido um elenco de tipos de dados a serem usados para formatação da estrutura. Ex.: CHAR, VARCHAR, INTEGER, NUMERIC, etc. A partir do formato estabelecido para uma coluna, determina-se o tamanho para esta coluna. Deve-se observar, neste momento, o limite máximo suportado pelo DATATYPE (tipo de dado) do SGBD utilizado.

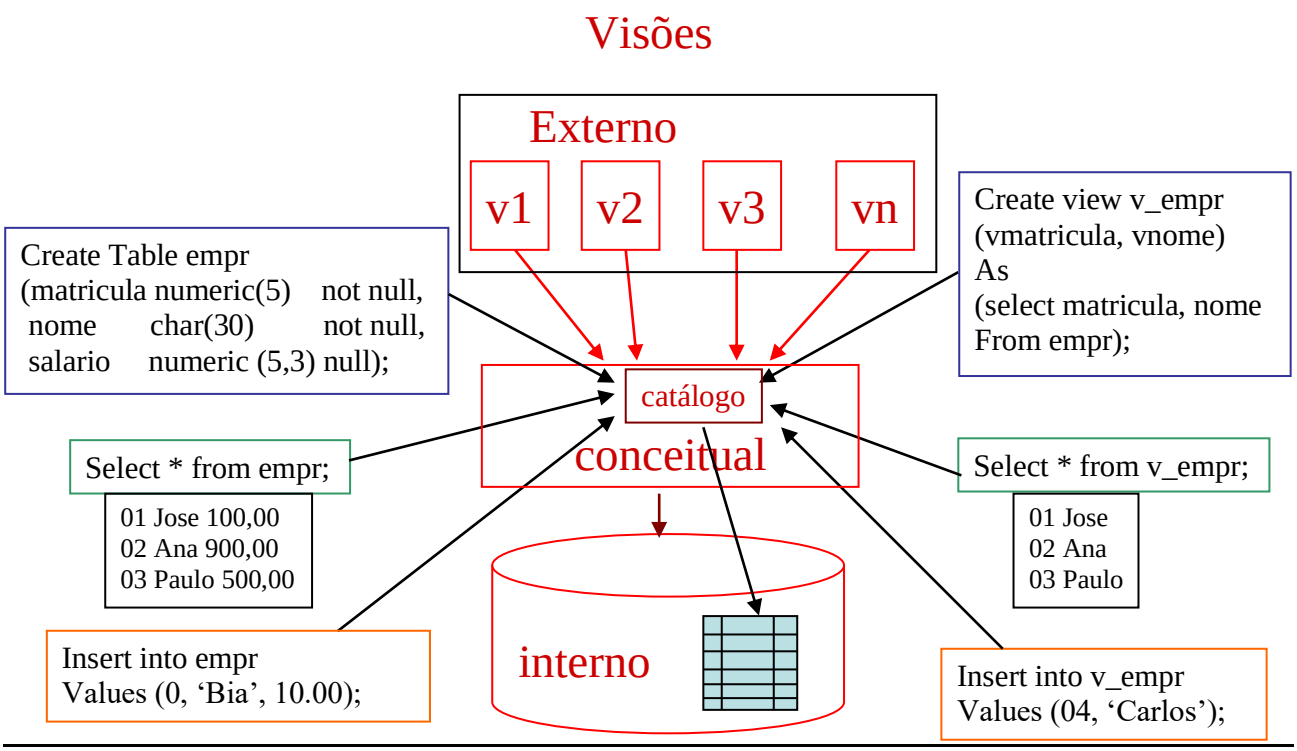
[null/not null] Especifica se o preenchimento do campo será obrigatório NOT NULL ou opcional NULL.

**Exemplo1:** Criação da tabela OFERTAS conforme modelo de dados do anexo1. No exemplo abaixo, estamos escrevendo um comando para criação da tabela referida (Ofertas). Todas as colunas presentes no modelo proposto no anexo estão sendo declaradas em um único comando CREATE. Observe que a declaração completa de uma coluna envolve o nome (disc_cod) o datatype (numeric(5)) e a obrigatoriedade (not null). Usa-se a vírgula para separar a declaração de cada coluna. A declaração completa de todas as colunas está envolvida em parêntesis.

```
CREATE TABLE ofertas
(disc_cod       numeric(5)   not null,
 prof_matr      numeric(5)   not null,
 oferta_sem     char(1)      not null,
 oferta_ano     char(4)      not null)
```

*Você poderá notar que não estamos tratando da criação da Primary Key neste momento. Não se preocupe, trataremos dessa parte, mais adiante. Acreditamos que a criação das regras (restrições de integridade) sobre uma tabela deva ser feita em momento posterior ao da criação da própria tabela. Por isso, separamos o Create Table (criação das estruturas puras) da criação das regras (restrições de integridade) sobre tal estrutura.*

**4.1.2 – Criando uma VIEW:** O comando de criação de uma *view* faz-se através do comando CREATE VIEW. Uma View é uma nova **estrutura** dentro do Dicionário de Dados do Banco de Dados que aponta para a mesma área de dados pertencente as tabelas ao qual a *view* se referencia. Geralmente as *views* são criadas para consulta (SELECT) embora a potencialidade da view permita modificações na área de dados (INSERT, UPDATE, DELETE) através dela. A *view* nada mais é do que um comando de SELECT sobre uma tabela ou um conjunto de tabelas existentes dentro de um SGBD. Para Ramalho (1999, p.147), através das visões é possível oferecer ao usuário apenas informações de que necessita, não importando se elas são oriundas de uma ou várias tabelas do banco de dados. A figura 2 demonstra o esquema de funcionamento da *view*.




Formato do comando CREATE VIEW:

Nessa figura pode-se observar que uma *view* nada mais é do que um novo esquema que aponta para um conjunto de dados pertencente a uma ou mais tabelas. Através de *views* é possível então "visualizar" os dados que estão fisicamente armazenados com estruturas diferentes daquelas representadas pelas tabelas. As *views* são construções lógicas geradas dinamicamente a medida que são invocadas. Uma abordagem alternativa para isso são as "*views* materializadas" não abordadas nesse estudo.

```
CREATE VIEW view_name
[(column_name, ...)]
as
(statment SELECT)
WITH CHECK OPTION
```

onde:

<view_name> Corresponde ao nome da view que será criada. Este nome é de livre escolha do usuário, lembrando somente que, dentro de um mesmo banco de dados, as *views* têm sempre que ter nomes distintos. Em Bancos de dados distintos as *views* podem ter nomes iguais.

<column_name> Corresponde ao nome da coluna dentro da view. Também de livre escolha do usuário, lembrando somente que dentro de uma mesma view, as colunas devem ter nomes distintos. Em *views* distintas, podem haver colunas com nomes iguais. Lembre-se que o nome da coluna na view nada mais é que um "apelido" para o conteúdo a ser obtido da(s) tabela(s) a(s) qual(is) a view se referencia. Devemos apenas especificar nome para as colunas na View caso se queira que as colunas não tenham o mesmo nome das colunas nas tabelas fonte de dados ou quando o resultado obtido não for direto de coluna mas sim proveniente de calculo ou função agregada.

<statment SELECT>Especifica a fonte de dados que irá povoar a view. Lembrando que uma view nada mais é do que um comando SELECT sobre o conteúdo dos dados controlados por uma ou varias tabelas. Este é o ponto no qual se faz o engenho de captura dos dados que irão dar "vida" a view. Toda a potencialidade que o comando SELECT possui pode ser aplicado aqui, exceto a clausula ORDER BY. Lembrando que a exceção se faz para View Materializadas onde os dados obtidos por comando SELECT são fisicamente armazenados.

[WITH CHECK OPTION] Especifica as restrições que o SGBD deverá aplicar quando se tentar utilizar a view para atualização dos dados (comandos INSERT, UPDATE e DELETE). Se esta opção for ativada na criação da view, todas as restrições descritas na clausula WHERE do comando SELECT que alimenta a view deverão ser cumpridas quando a view for utilizada para modificação de dados. Nem todos os SGBDs aceitam esta cláusula.

Figura 2 – estrutura da view

**Exemplo2:** Criação da view V_PROFESSORES baseado na tabela PROFESSORES, contendo somente as colunas matricula e nome. Além disso, exibir somente os professores que não tenham doutorado.

```
CREATE VIEW v_professores
(matricula, nome)
as
        (SELECT prof_matr, prof_nome
          FROM professores
          WHERE prof_ds_dou is null)
```

*Observe que o comando SELECT que irá fornecer dados para VIEW faz uma restrição horizontal (restrição de linhas) bem como uma restrição vertical (restrição de colunas) sobre os dados a serem exibidos. Observe também que nesse exemplo, na estrutura das View as colunas foram renomeadas para matricula e nome. Caso omitíssemos estes elementos no comando create view, os dados a serem demonstrados na VIEW teriam o nome de prof_matr e prof_nome oriundos do esquema da tabela fonte de dados obtido no dicionário de dados.*

## 4.2 – Alterando objetos - ALTER

A instrução ALTER é utilizada para alterar as estruturas dentro do banco de dados. De acordo com cada SGBD a potencialidade do ALTER pode ser bastante ampliada. Nessa apostila, nos concentraremos somente no ALTER TABLE. Podemos alterar uma tabela por diversas razões, dentre elas citamos: Acrescentar novas colunas, eliminar uma coluna, acrescentar Restrições de Integridade (RI) como por exemplo *Primary Key* (integridade da entidade) e Foreign Key (integridade referencial).

Formato do comando ALTER TABLE:

ALTER TABLE <table_name>
<operation>

onde:

<table_name> Corresponde ao nome da tabela que será modificada.

<operation> Diversas são as operações que um SGBD pode suportar quando se deseja modificar a estrutura de uma tabela. Listaremos aqui um conjunto restrito e mais comumente utilizado dessas operações. Vale lembrar que a sintaxe bem como a abrangência de cada operação pode ser levemente diferente entre os diversos SGBDs do mercado. As operações propostas são as seguintes:

<Operation ADD> Permite adicionar novas características a tabela. Essas características podem ser:
Coluna → Adicionar uma coluna à estrutura da tabela
constraint → Adicionar uma restrição de integridade a tabela. As restrições de integridade podem ser de *Primary Key* e de Foreign Key.
Index → Adiciona um índice para a tabela

<Operation DROP> Permite eliminar características da tabela. Essas características podem ser:
Coluna → eliminar uma coluna da estrutura da tabela
constraint → eliminar uma restrição de integridade a tabela. As restrições de integridade podem ser de *Primary Key* e de Foreign Key.
Index → eliminar um índice da tabela.

**Exemplo3:** Acrescentando coluna. Alteração da tabela ALUNOS, acrescentando a coluna E_MAIL como um atributo alfanumérico de 30 posicões, opcional.

**ALTER TABLE** alunos
**ADD** (e_mail char(30) null)

**Exemplo4:** Acrescentando restrição de integridade: Alteração da tabela OFERTA, acrescentando restrição de chave primaria sobre as colunas DISC_COD e PROF_MATR.

**ALTER TABLE** oferta
**ADD** primary key (disc_cod, prof_matr)

*Observe que, caso a Chave Primária seja composta, a criação da mesma se dará em um único comando, declarando-se entre parêntesis todas as colunas que comporão a chave.*

**Exemplo5:** Acrescentando restrição de integridade: Alteração da tabela OFERTA, acrescentando restrição de chave estrangeira sobre a coluna DISC_COD vinculando-a a tabela DISCIPLINAS e sobre a coluna PROF_MATR vinculando-a a tabela PROFESSORES, com integridade referencial restrita.

**ALTER TABLE** oferta
**ADD CONSTRAINT** fk_1 foreign key (disc_cod)
**REFERENCES** disciplinas (disc_cod)
**ON UPDATE** restrict
**ON DELETE** restrict

**ALTER TABLE** oferta
**ADD CONSTRAINT** fk_2 foreign key (prof_matr)
**REFERENCES** professores (prof_matr)
**ON UPDATE** restrict
**ON DELETE** restrict

*Observe que nesse caso especial, as duas colunas que formam a Chave Primaria da tabela, também são chaves estrangeiras. Independente da integridade de PK que a coluna possua, se ela também for FK deverá ser criada a integridade entre ela e a tabela por ela referenciada.*

*Alguns SGBDs não permitem a declaração de restrição de integridade ON UPDATE. Outros aceitam a declaração somente se o a ação referencial a ser estabelecida for diferente de restrict ou seja, cascade, no action, default, null.*

Conforme poderá ter sido observado pelo leitor, a construção de uma RI de Chave Estrangeira (FK) é bem mais elaborada do que a RI de Chave Primária (PK). Enquanto na RI de PK basta indicar qual(is) coluna(s) será(ão) a PK, na RI de FK é necessário indicar qual(is) coluna(s) será(ao) estrangeira(s), através da clausula ADD CONSTRAINT, Qual(is) tabela(s) será(ão) referenciada(s) pela FK, através da clausula REFERENCES bem como a ação referencial prevista no relacionamento mãe/filha, através da clausula ON UPDATE e ON DELETE.

A tabela definida na opção REFERENCES demonstra o vínculo filha/mae, ou seja, o SGBD deverá garantir que o valor colocado na coluna dita filha (FK) seja um subconjunto do conjunto dos valores da coluna dita mãe (PK). E a isso denomina-se Integridade

Referencial. Já o elemento declarado nas opções ON UPDATE e ON DELETE representam a Ação Referencial, ou seja, trata do relacionamento mãe/filha. Nesse aspecto, o SGBD deverá garantir que as ações de alteração ou exclusão realizadas na tabela dita mãe (PK) respeitem a ação prevista (cascata, restrito, nulo,...) sobre a tabela filha (FK).

**Exemplo6:** Dropando coluna: Alteração da tabela ALUNOS, eliminando a coluna e_mail.

**ALTER TABLE** alunos
**DROP** (e_mail)

**Exemplo7:** Dropando restrição de integridade da entidade: Alteração da tabela ALUNOS, eliminado a chave primária.

**ALTER TABLE** alunos
**DROP** primary key

**Exemplo8:** Dropando restrição de integridade referencial: Alteração da tabela ALUNOS, eliminado a chave estrangeira.

**ALTER TABLE** alunos
**DROP** foreign key fk_1

*Observe que quando se "dropa" uma integridade sobre a tabela, está se dropando a regra e não a coluna. Nos casos do Drop Primary e Foreign Key, as duas colunas utilizadas pela regra permanecem na tabela, sendo que não mais precisam cumprir restrição de integridade. Além disso, ao se dropar a Primary Key diz-se explicitamente Drop Primary Key, isso é possível pois existe somente uma RI para PK para cada tabela. No entanto, para FK, é necessário informar o nome da RI que será dropada já que em uma tabela pode haver mais que um RI de FK.*

## 4.3 – Destruindo objetos - DROP

A instrução DROP é utilizada para destruir as estruturas dentro do banco de dados. Podem ser destruídas TABLES e VIEWS.

**1.4.3.1 – Destruindo tabelas:** O comando de destruição de uma tabela faz-se através do comando DROP TABLE. Lembre-se que o comando DROP destrói a estrutura e conseqüentemente, os dados que esta estrutura armazenava. Caso você queira excluir somente os dados mantendo a estrutura no catálogo do SGBD, utilize o comando DELETE a ser visto na parte de DML.

Formato do comando DROP TABLE:

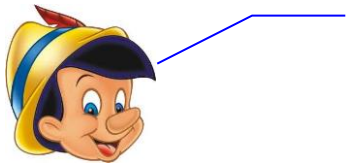
**DROP TABLE** <table_name>

onde:

<table_name> Corresponde ao nome da tabela que será destruída.

**Exemplo9:** Destruindo a tabela ALUNOS_EGRESSOS do anexo1.

**DROP TABLE** ALUNOS_EGRESSOS



*Sempre que for fazer um DROP em uma tabela, observar a integridade referencial (FK) existente com as demais tabelas do modelo.*

**1.4.3.2 – Destruindo view:** O comando de destruição de uma view faz-se através do comando DROP VIEW. Como a view não possui área de dados própria, a destruição da view causará somente a perda do objeto view no catalogo do SGBD. Os dados manipulados pela view não serão afetados por este comando, salvo nos casos de View Materializada.

Formato do comando DROP VIEW:

**DROP VIEW** <view_name>

onde:

<view_name> Corresponde ao nome da view que será destruída.

**Exemplo10:** Destruindo a view V_PROFESSORES proposta anteriormente.

**DROP VIEW** v_professores

# 5. Data Manipulation Language - DML

Nesta seção abordaremos as instruções SQL que serão utilizadas para a inclusão (INSERT), alteração (UPDATE), exclusão (DELETE) e leitura (SELECT) dos dados em um banco de dados.

Semelhante ao que foi feito na parte dos comandos DDL, para testarmos os comandos DML utilizaremos também o modelo de dados ACADÊMICO encontrado no ANEXO1.

## 5.1 – Inserindo dados - INSERT:

O comando de inserção de uma tabela faz-se através do comando INSERT. Em condições normais, o comando INSERT está preparado para inserir somente uma linha por vez. Uma variação desse comando poderá ser feita, quando se deseja inserir várias linhas em uma tabela tomando por base os dados armazenados em outra(s) tabela(s). Exemplo desse tipo de INSERT será visto abaixo.

Formato do comando INSERT:
**INSERT [INTO]** <table_name
[(column_name)]
{VALUES {expression | select statments } | SELECT}

onde:

<table_name> Corresponde ao nome da tabela que será inserida.

<column_name> Corresponde ao nome de cada uma das colunas para as quais será informado valor durante o INSERT. Embora o comando INSERT seja denominado comando de linha, é possível fazer um comando onde somente algumas colunas receberão valor. Nesse caso, somente colunas opcionais (NULL) ou colunas autoincrement (incrementadas automaticamente pelo SGBD) podem ficar sem especificação de valor, já que todas as colunas obrigatórias (NOT NULL) deverão receber valor durante o INSERT.

Os valores a serem passados para o comando insert podem vir de duas fontes: Clausula VALUES ou Comando SELECT.

<VALUES> Quando se usa a opção VALUES, significa dizer que iremos passar valores para povoar somente 1 linha na tabela a ser inserida. Cada linha será formada por um valor atômico para cada coluna, ou seja, cada coluna deverá receber um valor e este valor poderá ser um literal, uma expressão ou o resultado de um comando SELECT (que retorne também um valor atômico).

<SELECT> Quando se usa a opção SELECT, significa dizer que iremos passar valores para povoar diversas linhas na tabela em questão. Usando o SELECT como fonte de dados, as colunas declaradas na clausula SELECT do comando SELECT deverão estar de acordo (em quantidade e formato) com as colunas que irão receber os valores na tabela a ser inserida.

**Exemplo11:** Inserindo uma linha na tabela ALUNOS, informando todas as colunas. No exemplo abaixo, estão sendo declaradas todas as colunas que deverão receber valor.

```
INSERT INTO alunos
 (alu_matr, cur_cod, alu_nome, alu_dt_nasc, alu_sexo, alu_dt_matr)
VALUES (12345, 10, 'fulano de tal', '1986/07/29, 'm', '2006/07/29')
```

*Observe que a coluna CUR_COD, é uma chave estrangeira. Isso significa que sobre ela agirão as restrições de integridade referencial. Nesse caso, o valor 10 atribuido a coluna deverá existir como valor válido dentro da coluna CUR_COD na tabela CURSOS. Hipoteticamente, estamos assumindo que esse valor exista.*

**Exemplo12:** Inserindo uma linha na tabela ALUNOS, informando valor para todas as colunas. Observe que como todas as colunas receberão valor, podemos omitir o nome das colunas na escrita do comando.

```
INSERT INTO alunos
VALUES (12345, 10, 'fulano de tal', '1986/07/29, 'm', '2006/07/29')
```

**Exemplo13:** Inserindo uma linha na tabela ALUNOS, informando somente as colunas obrigatórias.

```
INSERT INTO alunos
 (alu_matr, cur_cod, alu_nome, alu_dt_nasc)
VALUES (12345, 10, 'fulano de tal', '1986/07/29)
```

**Exemplo14:** Inserindo uma linha na tabela ALUNOS_EGRESSOS, tomando por base o conteúdo da tabela ALUNOS. No comando abaixo, a tabela ALUNOS será totalmente lida e descarregada na tabela ALUNOS_EGRESSOS. Nenhuma restrição horizontal (restrição de linha) está sendo empregada no comando.

```
INSERT INTO alunos_egressos
 (ale_cod, cur_cod, ale_nome, ale_dt_nasc, ale_sexo, ale_dt_matr)
SELECT alu_matr, cur_cod, alu_nome, alu_dt_nasc, alu_sexo
 FROM alunos
```

**Exemplo14a:** Inserindo uma linha na tabela ALUNOS_EGRESSOS, tomando por base uma parcela do conteúdo da tabela ALUNOS. No comando abaixo, a tabela ALUNOS sofrerá uma restrição horizontal (restrição de linha) pois somente alunos do sexo masculino será selecionados para povoar a tabela ALUNOS_EGRESSOS.

```
INSERT INTO alunos_egressos
 (ale_cod, cur_cod, ale_nome, ale_dt_nasc, ale_sexo, ale_dt_matr)
SELECT alu_matr, cur_cod, alu_nome, alu_dt_nasc, alu_sexo
 FROM alunos
 WHERE alu_sexo = 'M'
```

**5.2 – Alterando dados - UPDATE:** O comando de alteração de uma tabela faz-se através do comando UPDATE. UPDATE é um comando de coluna que em condições normais está preparado para agir sobre os dados de uma coluna inteira e não somente de uma célula (interseção linha x coluna). Para atuar sobre um conjunto restrito de células, restrições (WHERE) deverão se aplicadas ao comando.

Formato do comando UPDATE:

**UPDATE** <table_name>
**SET** <column_name> = value
**WHERE** <condition>
onde:

| | |
|---|---|
| <table_name> | Corresponde ao nome da tabela que será alterada |
| <column_name> | Especifica a coluna que será alterada. |
| value | Para cada coluna definida na clausula SET, podemos atribuir um valor literal, ou um valor obtido de um comando SELECT. |
| WHERE | Permite que se aplique restrições sobre o conjunto de dados a serem afetados pelo comando UPDATE. |
| <condition> | conjunto de restrições que agirão sobre as linhas da tabela a ser modificada |

.

**Exemplo15:** Alterando a tabela PROFESSORES dando um aumento de 50% ao salário dos mesmos.

**UPDATE** professores
**SET** prof_sal = prof_sal * 1.5

*Este comando faz uma atualização sobre a coluna salário (prof_sal) para todas as linhas da tabela.*

**Exemplo16:** Alterando a tabela PROFESSORES dando um aumento de 50% ao salário de todos os professores cuja graduação seja em “Ciência da Computação”.

**UPDATE** professores
**SET** prof_sal = prof_sal * 1.5
**WHERE** prof_ds_grad = ‘ciencia da computacao’

*Este comando fará uma atualização sobre a coluna salário (prof_sal) somente para as linhas cuja coluna prof_ds_grad seja igual a ‘ciencia da computacao’. Trata-se de uma restrição horizontal sobre as linhas afetadas pelo comando UPDATE.*

**Exemplo17:** Alterando a tabela PROFESSORES dando um aumento de 100% ao salário de todos os professores que ministrem disciplinas de banco de dados.

```
UPDATE professores
SET prof_sal = prof_sal * 1.5
WHERE prof_matr in
        (SELECT distinct prof_matr
         FROM ofertas, disciplinas
         WHERE ofertas.disc_cod = disciplinas.disc_cod
             AND disciplinas.disc_nome = 'banco de dados')
```

*O comando demonstrado no exemplo17 irá permitir a atualização da coluna salário, somente para os professores que ministrem a disciplina 'banco de dados'. Observe que a informação sobre qual disciplina cada professor ministra não se encontra na tabela alvo da modificação (PROFESSORES). Para aplicar tal restrição, é necessário que se navegue até a tabela DISCIPLINAS, local onde o nome de cada disciplina está armazenada. No entanto, a tabela DISCIPLINAS, não fornece o identificador de qual professor a ministra e por isso, precisamos também da tabela OFERTAS. A partir da leitura dessas duas tabelas temos condições de aplicar as restrições sobre o universo de dados a ser atualizado na tabela PROFESSORES.*
*Você que está iniciando o aprendizado de SQL agora pode ficar um pouco confuso sobre o comando do exemplo 17. Não se assuste, estudaremos em detalhes as operações que manipulam mais que uma tabela na seção que trata do comando SELECT. Tal exemplo foi ilustrado aqui somente para mostrar a potencialidade do UPDATE.*

## 5.3 – Excluindo dados - DELETE:

O comando de exclusão dos dados de uma tabela se faz através do comando DELETE. DELETE é um comando de linha e naturalmente está preparado para agir sobre todas as linhas de uma tabela. Para atuar sobre um conjunto restrito de linhas, restrições (WHERE) deverão se aplicadas ao comando.

Formato do comando DELETE:

```
DELETE FROM <table_name>
WHERE <condition>
```

onde:

| | |
|---|---|
| <table_name> | Corresponde ao nome da tabela que será excluida |
| <WHERE> | Permite que se aplique restrições sobre o conjunto de dados a ser afetado pela comando DELETE. |
| <condition> | conjunto de restrições que agirão sobre as linhas da tabela a ser excluida |

.

**Exemplo18:** Excluindo toda a tabela TURMAS.

```
DELETE FROM turmas
```

**Exemplo19:** Excluindo a tabela TURMAS somente para as linhas cuja media final seja maior que 8.

```
DELETE FROM turmas
WHERE tur_med_final > 8
```

**Exemplo20:** Excluir a tabela TURMAS somente para os alunos do sexo masculino.

```
DELETE FROM turmas
WHERE alu_matr in
        (SELECT alu_matr
         FROM alunos
         WHERE alu_sexo = 'm')
```

*Semelhante ao exemplo17 do comando UPDATE, no exemplo 20 a exclusão das linhas na tabela TURMAS está condicionada a uma restrição que depende de outra tabela, no caso a tabela ALUNOS. Para aplicar tal restrição, devemos também envolver esta tabela na execução do DELETE desejado através de uma subquery (subconsulta).*

## 5.4 – Lendo dados - SELECT:

O comando de leitura de uma tabela faz-se através do comando SELECT. SELECT é o comando mais complexo e abrangente. É também, o comando mais utilizado, não só para buscar dados a serem exibidos, mas também como fonte de dados para os demais comandos (INSERT, UPDATE, DELETE, VIEW), conforme pôde ser visto em exemplos anteriormente definidos.

Formato do comando SELECT:

```
SELECT (column_name, ...)
FROM <table_name>
WHERE condition
GROUP BY (column_name)
HAVING condition
ORDER BY (column_name) [asc/desc]
```

onde:

| | |
|---|---|
| <SELECT> | Responsável pelo resultado a ser gerado |
| <FROM> | Responsável por definir a fonte de dados a ser utilizado. As fontes de dados podem ser tabelas, *views* ou outro comando select. |

<WHERE> Responsável pelas restrições a serem aplicadas sobre as tabelas (fontes de dados) especificadas na clausula FROM

<GROUP BY> Responsável pelo agrupamento de linhas para geração de cálculos produzidos por funções agregadas. Agrupar as linhas após a filtragem feita pela clausula WHERE.

<HAVING> Responsável pelas restrições a serem aplicadas sobre os dados agregados, ou seja, os dados resultantes do GROUP BY

<ORDER BY> Responsável pela ordenação do resultado final da clausula SELECT.

<column_name> Na clausula SELECT, serão enunciados o resultado que se deseja retornar a partir da execução do comando SELECT. O resultado pode ser obtido de diversas maneiras:
1) Pode ser exibido o conteúdo armazenado de uma coluna.
2) Pode ser exibido o resultado de um calculo feito sobre uma coluna: Este calculo pode ser agregado (usando Funções Agregadas), ou seja, envolvendo o uso de varias linhas como por exemplo o SUM (somatório), AVG (média), etc, ou simplesmente um calculo efetuado sobre uma única linha, como por exemplo uma multiplicação, uma adição, uma divisão.

Na Clausula GROUP BY, serão enunciadas as colunas que serão utilizadas para permitir o agrupamento de linhas. Linhas são agrupadas, geralmente, quando se desejar calcular algum resultado que necessite envolver a leitura de mais do que uma linha para que o calculo seja efetuado. O GROUP BY está diretamente vinculado ao uso de alguma das Funções Agregadas descritas no parágrafo acima. Na maioria dos SGBD, deverão ser listadas na clausula GROUP BY , todas as colunas presentes na clausula SELECT, exceto as colunas que estiverem sofrendo ação de alguma Função Agregada. Em alguns SGBDs, os aliases de coluna podem ser referenciados nas clausulas GROUP BY, HAVING e ORDER BY.

Na clausula ORDER BY serão listadas todas as colunas sobre as quais será aplicada ordenação. As colunas aqui listadas podem sofrer ordenação crescente (ASC) ou decrescente (DESC). Se mais do que uma coluna for relacionada, a ordenação final considerará a ordem da primeira coluna e sobre esta ordem, aplicará a ordem da segunda coluna e assim por diante.

<nome_da_tabela> Na clausula FROM, serão enunciadas as fontes de dados, de quais tabelas ou view serão retirados os dados que farão parte da análise e ou resultado do comando SELECT. Algumas tabelas listadas na clausula FROM não necessariamente terão seu dados exibidos na clausula SELECT, fazendo somente parte do conjunto de valores que comporão a análise. Também pode-se ter como fonte de dado o resultado de um comando SELECT.

<condition > Na clausula WHERE, as condições serão usadas para restringir o universo de dados a serem tratados pelo comando SELECT. Aqui são declaradas as restrições a serem empregadas à medida que as tabelas ou *views* são lidas, eliminando assim os dados indesejados. A clausula WHERE só tem poder de aplicar restrições sobre conteúdo

armazenado ou calculo simples. Conteúdo armazenado são exatamente os valores que estão armazenados dentro das colunas das tabelas. Cálculos simples são os cálculos básicos feitos linha a linha, sem nenhum agrupamento de linhas, como por exemplo adição, multiplicação, subtração aplicada sobre uma célula (intersecção linha x coluna) específica.

Na clausula HAVING as condições serão usadas para restringir o resultado, somente sobre os cálculos efetuados por funções agregadas (AVG (media), SUM (somatorio), MIN (mínimo), MAX (maximo), COUNT (contagem)). A clausula HAVING é utilizada em conjunto com a clausula GROUP BY. A clausula HAVING tem um poder um pouco mais abrangente do que o WHERE. O HAVING enxerga tanto os valores armazenados das linhas, os cálculos simples (que o WHERE faz) bem como os cálculos gerados por funções agregadas. Esses últimos, os cálculos de funções agregadas, não conseguem ser vistos pelo WHERE. Embora o HAVING tenha essa potencialidade, de poder restringir conteúdo armazenado ou calculo simples, não se recomenda aplicar estas restrições nesse ponto pois as linhas a serem eliminadas aqui terão "engordado" toda a classificação que foi aplicada pela clausula GROUP BY. Tais linhas, poderiam ter sido eliminadas antecipadamente, pela clausula WHERE, e dessa forma, não fariam parte do conjunto de valores a ser ordenado pelo GROUP BY.

**Exemplo21:** Selecionando dados de uma tabela, sem restrição de linha ou de coluna. No exemplo abaixo, serão selecionadas todas as colunas (clausula SELECT) e todas as linhas (clausula WHERE, que não foi declarada, pois não existe restrição de linha) da tabela alunos.

```
SELECT *
FROM alunos
```

Quando se deseja selecionar todas as colunas, o caracter especial * pode ser utilizado, ou então, pode-se declarar explicitamente todas as colunas que se deseja buscar.

```
SELECT alu_mat, cur_cod, alu_nome, alu_dt_nas, alu_sexo, alu_dt_matr
FROM alunos
```

**5.4.1 – Operadores:** O Operador é uma palavra ou um caractere reservado que é usado principalmente em uma clausula WHERE para realizar uma ou mais operações, como comparações ou cálculos aritméticos. Os operadores são usados junto com o comando SELECT para especificar **condições** em uma instrução SQL. (Stephens, Plew, 2003, p.102..) O Quadro abaixo faz um resumo dos principais operadores que utilizaremos.

| Operadores utilizados no comando SELECT | |
|---|---|
| **Comparação** | |
| Igualdade | = |
| Desigualdade | <> |
| Maior que | > |

| | |
|---|---|
| Menor que | < |
| Maior ou igual a | >= |
| Menor ou igual a | <= |
| **Lógico** | |
| AND | Verdadeiro se ambas as expressões forem verdadeiras |
| OR | Verdadeiro se qualquer das expressões for verdadeira |
| BETWEEN | Verdadeiro se o operando estiver dentro da faixa |
| LIKE | Verdadeiro se o operando encontrar um padrão |
| IN | Verdadeiro se algum item for verdadeiro |
| NOT | Inverte o valor booleano |
| **Aritmético** | |
| Multiplicação | * |
| Divisão | / |
| Subtração | - |
| Adição | + |
| **Valores nulos** | |
| Is null | Identifica que o valor é nulo |
| Is not null | Identifica que o valor não é nulo |

Tabela 3 – Relação dos operadores utilizados no comando SELECT

## Operadores de Comparação

Os *operadores de comparação* são usados para testar valores individuais em uma instrução SQL.

**Igualdade** → tem como símbolo o sinal de igual (=)

**Exemplo22:** Selecionar todos os alunos cuja matricula seja igual a 10.

```
SELECT *
FROM alunos
WHERE alu_matr = 10
```

**Desigualdade** → tem como símbolo o sinal de menor/maior junto (<>)

**Exemplo23:** Selecionar todos os alunos cuja matricula seja diferente de 10.

```
SELECT *
FROM alunos
WHERE alu_matr <> 10
```

**Maior que** → tem como símbolo o sinal de maior (>)

**Exemplo24:** Selecionar todos os professores cujo salário seja maior que 1.000

```
SELECT *
FROM professores
WHERE prof_sal > 1000
```

**Menor que** → tem como símbolo o sinal de menor (<)

**Exemplo25:** Selecionar todos os professores cujo salário seja menor que 1.000

```
SELECT *
FROM professores
WHERE prof_sal < 1000
```

**Maior ou igual a** → tem como símbolo o sinal de maior ou igual (>=)

**Exemplo26:** Selecionar todos os professores cujo salário seja maior ou igual a 1.000

```
SELECT *
FROM professores
WHERE prof_sal >= 1000
```

**Menor ou igual a** → tem como símbolo o sinal de menor ou igual (<=)

**Exemplo27:** Selecionar todos os professores cujo salário seja menor ou igual a 1.000

```
SELECT *
FROM professores
WHERE prof_sal <= 1000
```

**Combinando operadores de comparação**

**Exemplo28:** Selecionar todos os professores cujo salário seja menor ou igual a 1.000 e a descrição do mestrado seja 'mestrado em informatica'

```
SELECT *
FROM professores
WHERE prof_sal <= 1000 and prof_ds_mes = 'mestrado em informatica'
```

**Exemplo29:** Selecionar todos os professores cujo salário seja maior ou igual a 1.000 e menor ou igual a 5000

```
SELECT *
FROM professores
WHERE prof_sal >= 1000 and prof_sal <= 5000
```

## Operadores Lógicos

Os *operadores lógicos* são aqueles que usam palavras chaves em vez de símbolos para estabelecer a comparação.

**AND**→ permite a combinação de outros operadores. Será verdadeiro o resultado que satisfizer a **todas** as condições propostas.

**Exemplo30:** Selecionar os dados das turmas que tenham quantidade de falta maior que 50 e media final menor que 8

**SELECT** *
**FROM** turmas
**WHERE** tur_qt_faltas > 50 **and** tur_media_final < 8

**OR→** permite a combinação de outros operadores. Será verdadeiro o resultado que satisfizer a **uma** das condições propostas.

**Exemplo31:** Selecionar os dados das turmas que tenham media semestral ou media final menor que 8

**SELECT** *
**FROM** turmas
**WHERE** tur_media_sem > 8 **or** tur_media_final < 8

**BETWEEN→** é usado para procurar valores que estejam dentro de um conjunto de valores, especificados os valores mínimo e máximo.

**Exemplo32:** Selecionar todas as disciplinas que tenham carga horária entre 40 e 68

*Os valores declarados no operador Between serão considerados como validos para fazerem parte do resultado. No exemplo acima, tanto o valor 40 quanto o valor 68 poderão aparecer no resultado. O mesmo não ocorre quando se usa a negação, no caso o NOT Between, no qual os valores declarados não farão parte do resultado.*

**SELECT** *
**FROM** disciplinas
**WHERE** disc_choraria between 40 and 68

**LIKE→** é usado para comparar um valor a valores semelhantes, usando operadores curinga. Os operadores curinga são:

% representa zero, um ou vários caracteres.
_ representa um único caracter

**Exemplo33:** Selecionar os professores que possuam o último sobrenome ‘costa’.

**SELECT** *
**FROM** professores
**WHERE** prof_nome like ‘%costa’

*No exemplo 33 acima, o curinga % desprezou qualquer caracter que possa existir na coluna prof_nome anterior ao termo “costa” que está sendo pesquisado. No caso, o curinga poderá desprezar 0, 1 ou diversos caracteres.*

**Exemplo34:** Selecionar todos os cursos que possuam a denominação ‘bacharel’, em qualquer posição do nome

**SELECT** *
**FROM** cursos
**WHERE** cur_nome like ‘%bacharel%’

**Exemplo35:** Selecionar todos os dados dos alunos cujo nome tenha na terceira posição a letra ‘r’, independente do restante da palavra

**SELECT** *
**FROM** alunos
**WHERE** alu_nome like ‘__r%’

*No exemplo 35 vemos uma nova aplicação do operador LIKE, com o curinga underline. Semelhante ao %, o curinga _ também irá desprezar caracteres porem um para cada _ utilizado. No exemplo acima, antes da letra “r” foram colocados 2 underline, um para cada posição que se deseja desprezar de forma a atender ao critério de seleção estipulado na consulta.*

**Exemplo36:** Selecionar todos os dados dos alunos cujo nome tenha a primeira, a terceira e a quinta letra = ‘s’, independente do restante da palavra

**SELECT** *
**FROM** alunos
**WHERE** alu_nome like ‘s_s_s%’

Em algumas situações, desejamos fazer uma busca de aproximação (usando o operador LIKE), no entanto deseja-se encontrar um ou ambos os caracteres utilizados como curinga. Nesse caso, declaramos um caracter de escape. O caracter de escape anula a propriedade curinga do % e do _ quando o escape está acoplado ao mesmo.

**Exemplo37:** O modelo proposto para o exercício não possui nenhuma coluna cujo domínio possa ter um termo com o uso de algum dos curingas, nesse caso, adotaremos um exemplo hipotético para ilustrar a situação. Selecionar todos os cursos, que possuam um caracter _ em qualquer posição do nome.

**SELECT** *
**FROM** cursos
**WHERE** cur_nome like ‘%?_%’ escape ‘?’

*No exemplo 37 o caracter ‘? ‘foi definido como escape. Significa que nessa consulta ele terá o poder de anular a propriedade curinga do caracter ao qual ele estiver associado. No exemplo, ele foi colocado ao lado do underline “_”, fazendo com que nessa consulta especifica o _ assume o papel de _ e não de qualquer outro caracter, que seria o caso quando ele age como curinga. O curinga % permanece com as suas característica.* ***Alguns SGBD*** *permitem anular a propriedade curinga simplesmente envolvendo-o em um colchete [ ]. Nesse caso, não é preciso declarar o caracter de escape. Esta alternativa é visao abaixo, para a mesma simulação do exemplo36.*

**SELECT** *
**FROM** cursos
**WHERE** cur_nome like '%[_]%'

**Exemplo38:** Selecionar todos os alunos egressos cujo nome comece com Paulo e tenha somente 10 caracteres.

**SELECT** *
**FROM** alunos_egressos
**WHERE** alu_nome like 'Paulo_____'

Uma variante para esse exemplo é com o uso da função length. Length conta a quantidade de caracteres de um campo.

**SELECT** *
**FROM** alunos_egressos
**WHERE** alu_nome like 'Paulo%' and length(alu_nome) = 10

*No exemplo 38 , após a escrita do termo 'Paulo' estão colocados 5 (cinco) underline, que somados aos 5 da palavra paulo, compõem os 10 propostos na consulta. No exemplo descrito logo abaixo, a quantidade de caracteres desejada está sendo controlada pela função length. Obserque que o LIKE, com o uso do % aceitará qualquer quantidade de carateres, porém a limitação dos 10 caracteres passará para a função.*

**IN→** compara um determinado valor com uma coleção de valores. Ele funciona como um conjunto de operadores OR.

**Exemplo39:** Selecionar todas as disciplinas cuja carga horária seja igual a 34, 68, 70, 92 ou 113

**SELECT** *
**FROM** disciplinas
**WHERE** choraria in (34, 68, 70, 92, 113)

**SOME→** compara um determinado valor com uma coleção de valores. Ele funciona como um conjunto de operadores OR.

**Exemplo40:** Selecionar todas as disciplinas cuja carga horária seja igual a 34, 68, 70, 92 ou 113

**SELECT** *
**FROM** disciplinas
**WHERE** choraria some (34, 68, 70, 92, 113)

**NOT→** o NOT é usado para negar a condição do operador ao qual está acoplado

**Exemplo41:** Selecionar os professores cujo salário não esteja compreendido entre 1000 e 3000. Nesse caso, serão consideradas validas linhas cujo valor da coluna referenciada seja <1000 **ou** > 3000.

**SELECT** *
**FROM** professores
**WHERE** prof_sal not between 1000 and 3000

**Exemplo42:** selecionar os alunos cuja matricula não seja 10, 12, 14, 18, 22, 25 e 26

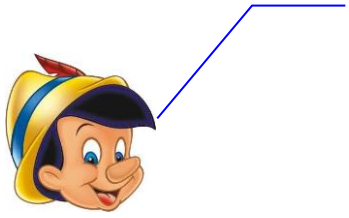
**SELECT** *
**FROM** alunos
**WHERE** alu_matr not in (10, 12, 14, 18, 22, 25, 26)

## Operadores Aritméticos

**+** → adição
**-** → subtração
***/*** → divisão
***** → multiplicacao

**Exemplo43:** Exibir o nome de cada professor, o salário atual e o salário projetado com aumento de 50%, somente se o salário projetado for menor que 10000

**SELECT** p.prof_nome as p.nome, prof_sal as “salario atual”, p.prof_sal * 1.5 as “salário projetado”
**FROM** professores p
**WHERE** (p.prof_sal * 1.5) < 10000



*No exemplo 43, voce está vendo o uso de **alias** (**nomes alternativos)** para tabela. Aliases são renomeações temporária e duram somente durante a execução da consulta. Não afetam o catalogo nem a estrutura armazenada e se aplica no contexto de uma única declaração SELECT. Os aliases de tabelas são obrigatórios somente para SELF JOIN (auto-junções). Nos demais casos, o uso de aliases de tabelas busca facilitar e escrita do código pela diminuição do nome dos objetos manipulados. (Stephens, Plew, 2003, p.183)*
*Também estão presentes nesse SELECT Aliases de colunas, que tem por objetivo melhorar a estrutura de exibição dos dados.*

**Exemplo44:** Exibir todos os dados de cada turma, cuja media final subtraída da media semestral seja maior do que 5

**SELECT** t.*
**FROM** turmas as t
**WHERE** (t.tur_media_final – t.tur_media_sem) > 5

## Valores Nulos

**is null** → o campo é nulo
**is not null** → o campo não é nulo

**Exemplo45:** selecionar todos os professores que não tenham especialização, mestrado, doutorado e pós-doutorado.

```
SELECT p.*
FROM professores as p
WHERE p.prof_ds_esp is null
    and p.prof_ds_mes is null
    and p.prof_ds_dou is null
    and p.prof_ds_pdo is null
```

**Exemplo46:** selecionar todas as disciplinas que possuam ementa.

```
SELECT d.*
FROM disciplinas as d
WHERE d.disc_ementa is not null
```

**5.4.2 – Junções:** É comum que, pelas técnicas de modelagem e principalmente de normalização, os dados fiquem dispersos em diversas tabelas relacionadas. Em algumas consultas, necessitamos obter dados de varias tabelas. É nesse momento que entram em campo as junções. Junções ou ligações são mecanismos utilizados pelos SGBDs para que se possa acessar dados de mais que uma tabela durante um comando SELECT. A forma mais universal de se criar uma junção de tabelas é através da clausula WHERE do comando SELECT (RAMALHO, 1999, p.90).

As junções podem ser separadas em dois blocos: **Junções Internas** e **Junções Externas**.

Na **Junção Interna**, participarão do resultado, somente as linhas que satisfizerem ao critério de junção especificado. Somente as linhas que encontrarem par com linhas da outra tabela serão consideradas.

A **Junção Externa** abre a possibilidade de que apareçam no resultado linhas que não formaram par. Poderemos então, ter linhas no resultado com os pares equivalentes gerados a partir da ligação com outras tabelas, assim como linhas que só existem em uma das tabelas presentes no comando de leitura. (Stephens, Plew, 2003, p.180)

Tanto nas Junções Internas quanto nas Externas, existe a possibilidade de que as tabelas envolvidas no comando sejam as mesmas, e nesse caso, denominado de **SELF JOIN**

*Se em um comando SELECT for especificado mais do que uma tabela, sem nenhuma restrição nem condições de relacionamento entre elas, o resultado será um **produto cartesiano**, ou seja, a multiplicação das linhas da primeira tabela, pelo numero de linhas da segunda e assim por diante.*

## Junções Internas:

**EQUIJOIN** → Junção de igualdade, também conhecida como INNER JOIN ou UNIÃO REGULAR, é possivelmente a junção mais utilizada (Stephens, Plew, 2003, p.180). Ela efetua a junção de duas tabelas com base em uma (ou mais que uma) coluna que **normalmente** será chave primaria em uma das tabelas e chave estrangeira na outra. Observe que usou-se o termo "normalmente" porque é possível que se faça a junção de tabelas, tomando como elemento de ligação qualquer outra coluna da tabela mesmo que estas colunas não sejam chaves. Nesse tipo de junção é obrigatório que exista uma condição de ligação. Qualquer linha que **não** satisfaça a condição de ligação especificada será retirada do resultado.

**Exemplo47:** Selecionar o nome de cada aluno e o nome de cada curso que cada aluno faz.

```
SELECT a.alu_nome, c.cur_nome
FROM aluno a, curso c
WHERE a.cur_cod = c.cur_cod
```

No exemplo acima será necessário acessar duas tabelas (aluno e curso) para se obter o resultado desejado. Para cada uma das tabelas citadas, deverão ser encontrados pares de linhas que satisfaçam a condição de junção especificada pela clausula WHERE, na qual se estipula que os pares deverão ser formados pela igualdade de duas colunas presentes uma em cada tabela, a saber, "cur_cod" da tabela aluno e "cur_cod" da tabela curso

*Se houver um nulo em uma das colunas de ligação (a.cur_cod ou c.cur_cod), então a linha que contem aquele valor nulo será descartada da tabela resultante. Isso acontece porque o null não satisfaz a nenhuma combinação. (Patrick, 2002, p.451)*

A mesma consulta acima, poderia ser escrita alternativamente pelo comando abaixo. Observe que nesse caso, a construção do JOIN se deu na clausula FROM. Este tipo de JOIN é denominado JOIN IMPLICITO e passou a ser adotado a partir do padrão SQL/92. Nos padrões, SQL/86 e SQL/89, a escrita do JOIN utilizada era conforme a escrita no exemplo 47. (DATE, DARWEN, 1997, p.15)

**Exemplo48:** Selecionar o nome de cada aluno e o nome de cada curso que cada aluno faz. (alternativa para construção do JOIN)

```
SELECT a.alu_nome, c.cur_nome
FROM aluno a natural join  curso c
```

*A opção NATURAL JOIN poderá ser utilizada se o nome das colunas utilizado como elemento de ligação for o mesmo nas duas tabelas.*

**Exemplo49:** Selecionar o nome de cada aluno e o nome de cada curso que cada aluno faz. (alternativa para construção do JOIN)

```
SELECT a.alu_nome, c.cur_nome
FROM aluno a inner join  curso c
        On (a.cur_cod = c.cur_cod)
```

*Caso o nome das colunas for diferente nas tabelas, ou porque o SGBD não suporta a escrita do Natural Join, o exemplo 49 ofereçe uma outra variante para a escrita da mesma junção, porem com a declaração explicita da condição de junção. Essa junção é denominada JUNÇÃO TETA.*

**NON-EQUIJOIN** → Junção de desigualdade efetua junção de duas ou mais tabelas com base no valor de uma coluna especificada que **não** seja igual ao valor de uma outra coluna especificada em uma outra tabela (Stephens, Plew, 2003, p.183). Junções de desigualdade geram como resultado a combinação de cada linha da primeira tabela, com cada linha da segunda cujo valor de chave de comparação for diferente. Há que se ter razões muito especificas para se usar esse tipo de junção, já que o resultado traz linhas sem nenhuma vinculação.

**Exemplo50:** Selecionar os dados dos alunos cuja data de nascimento seja diferente da data de nascimento dos professores

```
SELECT a.*
FROM aluno a, prof p
WHERE a.alu_dt_nasc <> p.prof_dt_nasc
```

## Junções Externas:

**OUTER JOIN** → Outer Join é usada para retornar todas as linhas que existem em uma tabela, embora não existam linhas correspondentes na tabela relacionada (Stephens, Plew, 2003, p.183). Ligação (junção) externa deriva de ligação interna, já que recupera linhas que foram excluídas das tabelas originais na geração do resultado. Existem 3 tipos de ligação externa: **ligação externa a esquerda, ligação externa a direita e ligação externa completa**.

*Quando varias clausulas estão na condição de ligação, o OuterJoin deve ser feito para todas as clausulas, sob pena de se transformar em um EqualJoin.*

**Ligação externa a esquerda (LEFT OUTER JOIN)** → A ligação externa a esquerda permite recuperar todas as linhas da tabela posicionada a esquerda da operação de junção. Caso o resultado (conteúdo da clausula SELECT) exiba dados da tabela da direita, esses dados ficarão com nulo nas linhas que não formarem par com a tabela da esquerda.

**Exemplo51:** listar o nome de todos os cursos e o nome de cada aluno que freqüenta cada curso. Caso não hajam alunos matriculados em um determinado curso, listar o nome do curso assim mesmo.

```
SELECT c.cur_nome, a.alu_nome
FROM cursos c, alunos a
WHERE c.cur_cod *= a.alu_cod
```

Observe o * ao lado do sinal de igual. Esta notação é utilizada por alguns SGBD e indica que a tabela posicionada ao lado do asterisco está desobrigada de encontrar o par para poder ser listada. No caso, a tabela Cursos não precisa fazer par com a tabela Alunos para que suas linhas sejam listadas. Nas linhas dos cursos que não houverem alunos relacionados, o valor para o nome do aluno (alu_nome) ficará com nulo.

O mesmo comando acima, pode também ser escrito através da clausula FROM (padrão SQL/92).

```
SELECT c.cur_nome, a.alu_nome
FROM cursos c LEFT OUTER JOIN alunos a
        on c.cur_cod = a.alu_cod
```

Uma outra variação para escrita da junção do exemplo 51 faz-se também através da clausula WHERE porém acrescentando um sinal de (+) do lado do relacionamento que deverá ser preenchido com nulo caso não forme par na junção, ou seja, a parte da linha que ficará incompleta pela falta de ligação entre as tabelas. Este é o padrão do ORACLE.

```
SELECT c.cur_nome, a.alu_nome
FROM cursos c, alunos a
WHERE c.cur_cod = a.alu_cod (+)
```

**Ligação externa a direita (RIGTH OUTER JOIN)** → A ligação externa a direita permite recuperar todas as linhas da tabela posicionada a direita da instrução JOIN. Caso o resultado (conteúdo da clausula SELECT) exiba dados da tabela da esquerda, esses dados ficarão com nulo nas linhas que não formarem par com a tabela da direita.

**Exemplo52:** listar todos os dados de todas as disciplinas. Se a disciplina foi ofertada, listar também o semestre e o ano de oferta de cada disciplina.

```
SELECT d.*, o.oferta_sem, o.oferta_ano
FROM ofertas o, disciplinas d
WHERE o.disc_cod =* d.disc_cod
```

O mesmo comando acima, pode também ser escrito das seguintes formas:

```
SELECT d.*, o.oferta_sem, o.oferta_ano
```

**FROM** ofertas o RIGTH OUTER JOIN disciplinas d
**on** o.disc_cod =* d.disc_cod

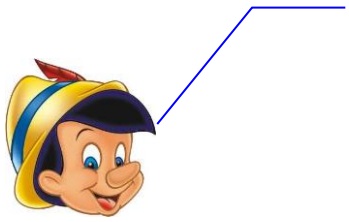
**SELECT** d.*, o.oferta_sem, o.oferta_ano
**FROM** ofertas o, disciplinas d
**WHERE** o.disc_cod = d.disc_cod (+)

**Ligação externa completa (FULL OUTER JOIN) →** A ligação externa completa permite recuperar todas as linhas de todas as tabelas participantes do relacionamento, independente da linha formar par ou não. Alguns SGBD já possuem suporte para ligação externa completa através do comando FULL OUTER JOIN escrito na clausula FROM. Para os bancos que não suportam o comando acima, pode-se obter o mesmo resultado através da seguinte construção: (Patrick, 2002, p.483)

- ligação externa esquerda
- union
- ligação externa direita.

Ou

- ligação externa direita
- union
- ligação externa esquerda.



*Na sintaxe proposta acima, você está tomando conhecimento de um novo comando SQL, o Union. O Union permite juntar dados de duas ou mais tabelas em uma única tabela resultante. Para que isso seja possível, as linhas das tabelas usadas na junção devem ser idênticas, ou seja, devem ter o mesmo numero de colunas e o tipo de dados em cada uma das colunas deve estar na mesma ordem e serem do mesmo tipo. Observe que foi dito que as linhas usadas na Union devam ser idênticas, mas não que a estrutura das tabelas fonte devam ser idênticas, basta somente que as colunas selecionadas de cada tabela formando a linha da mesma, seja idêntica com a possível linha das demais tabelas participantes do SELECT. Por padrão, as linhas duplicadas são eliminadas do resultado.*

Devido a limitações do modelo proposto, faremos a exemplificação do Full Outer Join usando um modelo hipotético.

**Exemplo53:** Mostrar todos os dados das tabelas tabelaa e tabelab, independente de formarem par na ligação. No exemplo abaixo, no primeiro SELECT, anterior ao union, será demonstrado os dados de toda a tabelaa, e quando tabelaa formar par com tabelab, mostrará também os dados da tabelab, caso não forme par, as colunas referentes a tabelab ficarão com nulo. No segundo SELECT, posterior ao union, será demonstrado os dados de toda a tabelab, e quando tabelab formar par com tabelaa mostrará também os dados da tabelaa, caso não forme par, as colunas referentes à tabelaa ficarão com nulo. Para demonstrar a sintaxe da junção externa completa, adotaremos hipoteticamente duas tabelas fictícias (“tabelaa” e “tabelab”).

```
SELECT a.*, b.*
FROM tabelaa a, tabelab b
WHERE a.coluna_ligacao *= b.coluna_ligacao
Union
SELECT a.*, b.*
FROM tabelaa a, tabelab b
WHERE a.coluna_ligacao =* b.coluna_ligacao
```

ou

```
SELECT a.*, b.*
FROM tabelaa a, tabelab b
WHERE a.coluna_ligacao = b.coluna_ligacao (+)
Union
SELECT a.*, b.*
FROM tabelaa a, tabelab b
WHERE a.coluna_ligacao (+) = b.coluna_ligacao
```

ou

```
SELECT a.*, b.*
FROM tabelaa a FULL OUTER JOIN tabelab b
        on a.coluna_ligacao = b.coluna_ligacao
```

ou

```
SELECT a.*, b.*
FROM tabelaa a LEFT OUTER JOIN tabelab b
on a.coluna_ligacao = b.coluna_ligacao
Union
SELECT a.*, b.*
FROM tabelaa a RIGTH OUTER JOIN tabelab b
on a.coluna_ligacao = b.coluna_ligacao
```

**Ligação externa usando duas ou mais colunas para ligação** → Em alguns casos, a ligação entre as tabelas poderá envolver mais do que uma coluna.

**Exemplo54:** Listar os dados das turmas e o código das disciplinas ofertadas em cada turma.

```
SELECT t.*, o.*
FROM turmas t, ofertas o
WHERE t.disc_cod        =*      o.disc_cod
    and  t.prof_matr    =*      o.prof_matr
```

ou

```
SELECT t.*, o.*
FROM turmas t, ofertas o
WHERE t.disc_cod (+)    =       o.disc_cod
    and  t.prof_matr (+)    =       o.prof_matr
```

ou

```
SELECT t.*, o.*
FROM turmas t FULL OUTER JOIN ofertas o
        on t.disc_cod              =       o.disc_cod
            and  t.prof_matr       =       o.prof_matr
```

**5.4.3 – Funções de Grupo ou Agregadas:** Funções são palavras-chave na SQL usadas para manipular valores dentro de colunas. Uma função é um comando sempre usado junto com um nome de coluna ou expressão. (Stephens, Plew, 2003, p.123). As funções podem ser de linha simples ou escalares (*single row*) ou de grupo ou agregadas (*group).* Nessa apostila abordaremos somente as funções de grupo ou agregadas. Uma função agregada é usada para fornecer informações de resumo de uma instrução SQL.

As funções agregadas são: COUNT, AVG, SUM, MIN E MAX.

COUNT → é usada para contar filas ou valores de uma coluna que não contenham um valor nulo.

**Exemplo55:** exibir a quantidade de alunos do curso 23

```
SELECT count(*)
FROM alunos
WHERE cur_cod = 23
```

AVG → usada para encontrar média de grupo de linhas

**Exemplo56:** exibir o salário médio dos professores

```
SELECT avg(prof_sal)
FROM professores
```

*Observe que a função média baseia-se na soma dos valores dividido pelo conjunto de valores somados. Ficar atento para situações onde a coluna sobre a qual a função esteja sendo aplicada tenha algum valor nulo, já que o valor não afetará a soma, porém a linha também não será computada no conjunto de valores afetando incorretamente o resultado.*

SUM → usada para retornar um total sobre uma coluna de valores

**Exemplo57:** exibir o somatório dos salários dos professores

```
SELECT sum(prof_sal)
FROM professores
```

MIN → usada para retornar o menor valor de uma coluna

**Exemplo58:** exibir o menor salário de professor

**SELECT** min(prof_sal)
**FROM** professores

MAX → usada para retornar o maior valor de uma coluna

**Exemplo59:** exibir o maior salário de professor

**SELECT** max(prof_sal)
**FROM** professores

**5.4.4 – Agrupamento dos dados:** As funções agregadas estudadas logo acima possibilitam resumir um universo de dados, delimitado ou não, em um único resultado. Porém, algumas vezes desejamos geral o resumo por blocos de linhas. A partir do agrupamento dos dados, efetuado pela clausula GROUP BY, temos condições de gerar resumos por grupos distintos de linhas. (Patrick, 2002, p.399). O agrupamento de dados é o processo de combinação de colunas com valores repetidos em uma ordem lógica.

As colunas relacionadas na clausula SELECT são as que podem ser mencionadas na clausula GROUP BY. Se uma coluna não for encontrada na clausula SELECT ela também não poderá ser usada na clausula GROUP BY (Stephens, Plew, 2003, p.133). No entanto, todas as colunas presentes na clausula select, deverão compor a clausula GROUP BY.

O uso do agrupamento se dá em conjunto com as funções agregadas descritas no item 5.4.3. para gerar o resultado do agrupamento.

**Exemplo60:** Exibir a menor média final por código de disciplina

**SELECT** disc_cod, min(tur_media_final)
**FROM** turmas
**GROUP BY** disc_cod

**Exemplo61:** Exibir o código do curso, o nome do curso e a quantidade de alunos matriculados em cada curso

**SELECT** a.cur_cod, c.cur_nome, count(a.alu_matr)
**FROM** alunos as a, cursos as c
**WHERE** a.cur_cod = c.cur_cod
**GROUP BY** a.cur_cod, c.cur_nome

**HAVING**

A clausula HAVING possibilita que se faça restrição sobre o resultado sumarizado, ou seja, sobre o resultado produzido pelas funções agregadas. Age sobre as funções agregadas e acompanham a clausula GROUP BY.

**Exemplo62:** Exibir o código do curso, o nome do curso e a quantidade de alunos matriculados em cada curso, somente se o curso tiver menos que 10 alunos

```
SELECT a.cur_cod, c.cur_nome, count(a.alu_matr)
FROM alunos as a, cursos as c
WHERE a.cur_cod = c.cur_cod
HAVING count(a.alu_matr) < 10
```

**Exemplo63:** Exibir o código do curso, o nome do curso e a quantidade de alunos matriculados em cada curso, somente se o curso tiver menos que a quantidade de alunos do curso de biologia.

```
SELECT a.cur_cod, c.cur_nome, count(a.alu_matr)
FROM alunos as a, cursos as c
WHERE a.cur_cod = c.cur_cod
HAVING count(a.alu_matr) < (select count(*) from alunos
                            Where cur_cod in (select cur_cod
                                              from cursos
                                              where cur_nome = 'biologia'));
```

**5.4.5 – Subconsultas ou Subqueries:** Uma subconsulta é uma consulta embutida dentro de uma outra consulta. (Stephens, Plew, 2003, p.196) Ou seja, é a presença de uma declaração SELECT dentro de outra declaração SELECT. (Patrick, 2002, p.607) É também denominada *consulta aninhada.* Uma subconsulta é uma consulta cujos resultados são passados como argumentos de outra consulta. (Ramalho, 1999, p.103).

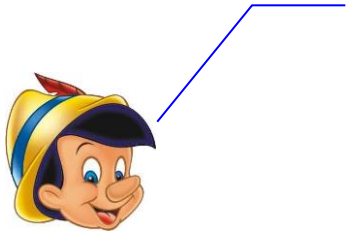
**Exemplo64:** Exibir todos os dados de todos os alunos matriculados no curso de ciência da computação

```
SELECT a.*
FROM alunos as a
WHERE a.cur_cod in
      (SELECT c.cur_cod
       FROM cursos as c
       WHERE c.cur_nome = 'ciencia da computacao')
```

*O exemplo64 demonstra uma situação comum entre os desenvolvedores, onde optou-se por utilizar uma subconsulta ao invés de uma junção. Deve-se atentar para isso, pois atualmente, as junções possuem desempenho igual e em alguns casos melhor do que as subconsultas* (PATRICK, 2002, p. 607).

**Exemplo65:** Exibir todos os cursos que não tenham nenhum aluno matriculado

```
SELECT c.*
FROM cursos as c
WHERE c.cur_cod not in
      (SELECT a.cur_cod
       FROM alunos as a)
```



*Tomar cuidado se o conjunto de valores resultante da subconsulta incluir um valor nulo e a condição utilizada na clausula WHERE da consulta externa for NOT IN. A consulta externa poderá ficar "perdida" devido a presença do nulo na lista de valores a ser analisado por ela* (PATRICK, 2002, p. 613). *Nesse caso, é melhor garantir que nenhum valor nulo retornará da subconsulta.*

As duas consultas descritas nos exemplos 64 e 65 recebem da subconsulta uma lista de valores. Por esta razão a condição de comparação a ser utilizada na clausula WHERE da consulta externa utilizou o **IN** e o **NOT IN**.

No entanto, na consulta descrita no exemplo66 abaixo, o resultado obtido da subconsulta é representado por um único valor, por isso, foi possível utilizar uma condição de comparação com o operador <. Quando a subconsulta retornar um único valor, pode-se utilizar quaisquer das seguintes condições: =, <>, >, <, >=, <=, IN, NOT IN ou BETWEEN (PATRICK, 2002, p. 609- 613).

**Exemplo66:** Exibir a quantidade de professores por tipo de Mestrado, para os tipos que tenham menos professores que o total de professores de doutorado.

```
SELECT p.prof_ds_mes, count(prof_ds_mes)
FROM professores as p
GROUP BY p.prof_ds_mes
HAVING count(prof_ds_mes) <
        (SELECT count(prof_ds_dou)
          FROM professores)
```

**Subconsultas Correlacionadas**

Geralmente as subconsultas são ditas não **correlacionadas** (ou não correlatas), quando não faz referencia a nenhuma tabela presente na consulta externa (Stephens, Plew, 2003, p.204). E são ditas correlacionadas (ou correlatas) quando fazem referência a uma tabela presente na consulta externa. Usamos subconsultas correlacionadas quando necessitamos de alguma informação da consulta externa para a realização da consulta interna (subconsulta).

**Exemplo67:** Exibir nome dos professores e a quantidade de ofertas, dos professores que tiveram em 2008 menos oferta de disciplinas que em 2007.

```
SELECT p.prof_nome, count(o.prof_matr)
FROM professores as p, ofertas as o
WHERE p.prof_matr = o.prof_matr
    And o.oferta_ano = 2008
GROUP BY p.prof_nome
HAVING count(o.prof_matr) <
        (SELECT count(o2.prof_matr)
          FROM ofertas as o2
          WHERE o2.prof_matr = p.prof_matr
               And  o2.oferta_ano = 2007)
```

*Observe que em uma subconsulta correlacionada, a junção com a tabela existente na consulta externa deverá ser feita na clausula WHERE, ou seja, não é possível realizar a junção da subconsulta correlacionada escrevendo a junção pela clausula FROM.*

*Já a subconsulta correlacionada não requer a clausula GROUP BY já que o conjunto de linhas a ser analisado por ela para realizar a função de contagem refere-se a um único professor, já filtrado pela junção com a tabela presente na consulta externa.*

## Bibliografia Utilizada:

COSTA.R. L. **SQL Guia Prático.** Rio de Janeiro: Brasport, 2004.

DATE, C.J. **Introdução a Sistemas de Banco de Dados.** 7 ed. Rio de Janeiro: Campus, 1999.

GARCIA-MOLINA, H., ULLMAN, J.D., WIDON, J. **Implementação de Sistemas de Bancos de Dados.** Rio de Janeiro: Campus, 2001.

GRAY, J., REUTER, A. **Transaction Processing** Concepts and Techniques**.** San Francisco: Morgan Kaufmann Publishers, 1993.

MANZANO, J. A. N. G. **Estudo dirigido de SQL.** São Paulo: Érica, 2002.

OLIVEIRA, C. H. P. de. **SQL Curso Prático.** São Paulo: Nocatec, 2002.

PATRICK, J. J. **SQL fundamentos.** 2ed. São Paulo: Berkeley Brasil, 2002.

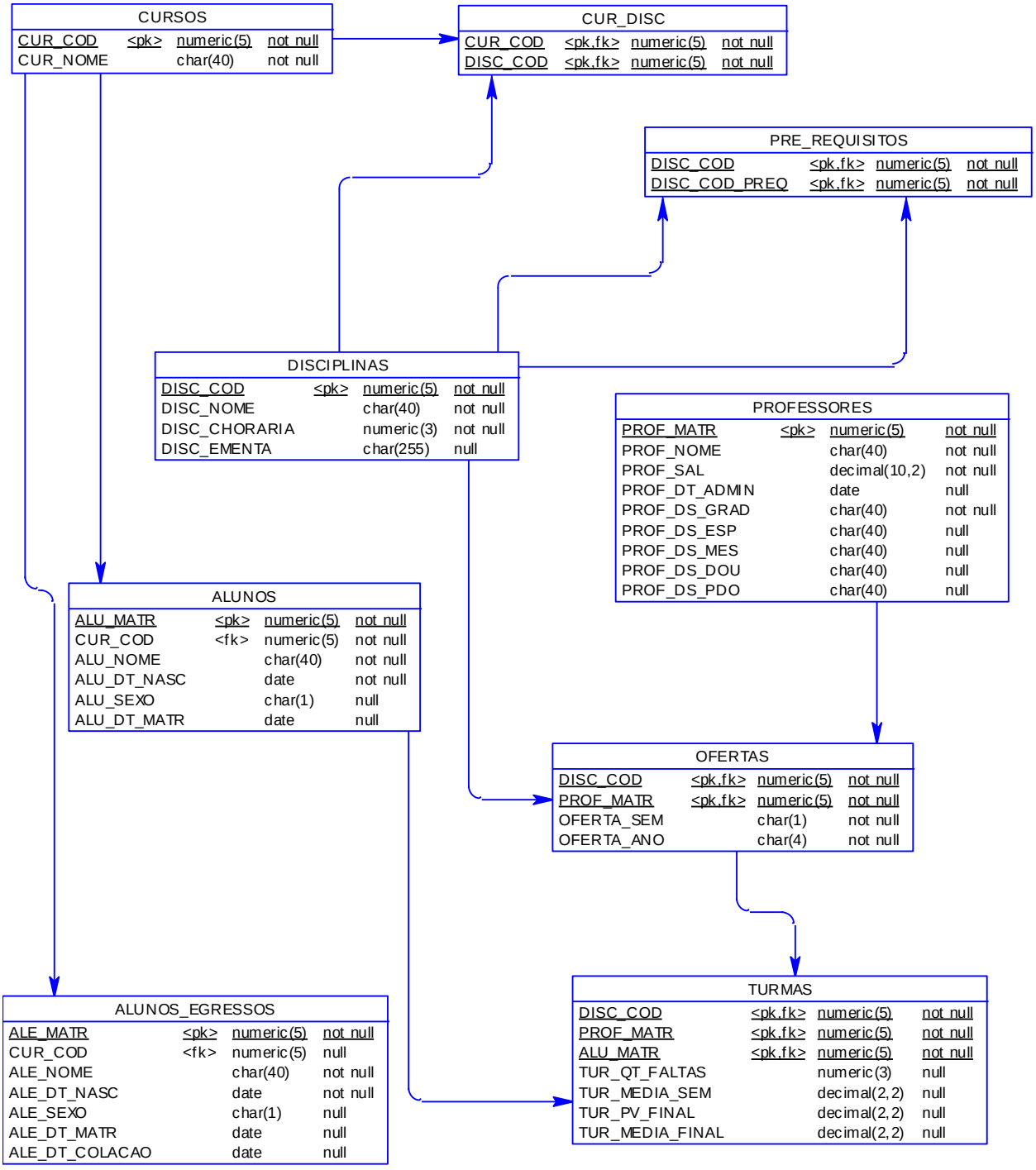
RAMALHO, J. A. A. **SQL A linguagem dos Bancos de Dados.** São Paulo: Berkeley Brasil, 1999.

STEPHENS, R. R., PLEW, R. **Aprenda em 24 horas SQL3.** Rio de Janeiro: Campus, 2003.

Todas as imagens dos personagens Pinóquio, Gepeto e Grilo Falante presentes na apostila e na lista de exercícios foram obtidas no Google Imagens.

# ANEXOS

# ANEXO1




Modelo de Dados Academico
Utilizado na Apostila de SQL