Linguagem SQL

Índice

Linguagem *SQL*

SQL (*Structured Query Language*) trata-se de um conjunto completo de comandos que permite ao usuário acessar um banco de dados relacional. É uma linguagem não procedural que permite dizer apenas o que você quer e não como fazer isto.

A linguagem *SQL* surgiu no início do anos 70 como parte do projeto System R, no laboratório de pesquisa da IBM em San Jose. Inicialmente foi chamada de Sequel. Ela surgiu com o intuito de fornecer ao "usuário" (ou desenvolvedor) uma interface mais amigável com os bancos de dados. Obteve grande sucesso, sendo que a maioria dos gerenciadores de bancos de dados atuais a utilizam.

Embora existam inúmeras versões do *SQL*, e cada produto utiliza uma, pode-se dizer que as diferenças chegam a ser secundárias, pois o núcleo, ou seja, os principais comandos mantém o mesmo padrão. Em 1986, o American National Standart Intitute (ANSI) publicou um primeiro padrão *SQL*, e grandes esforços vem sendo tomados para que o padrão ANSI seja cada vez mais obedecido. A primeira grande revisão do padrão SQL foi completa em 1992, recebendo a denominação de *SQL2*.Contudo, uma nova reformulação, já chamada de *SQL3*, que incluirá também o suporte a objetos, está em vias de ser completada.

A *SQL* estabeleceu-se claramente como a linguagem padrão de bancos de dados. Alguns dizem que *SQL* é uma linguagem de consulta, entretanto, o seu propósito inicial foi ampliado e ela permite uma interação completa com o banco, desde a atualização de dados, criação de tabelas até a administração da base de dados.

Podemos dividir a linguagem em alguns módulos. São eles:

- ***Data definition language (DDL)*** (Linguagem de definição de dados): A *SQL* fornece comandos para definição, modificação e remoção de tabelas, criação de índices, definição de visões. Inclui também comandos para especificação de direitos de acessos a tabelas e visões.
- ***Interactive data manipulation language (DML)*** (Linguagem de manipulação de dados interativa): A *SQL DML* inclui uma linguagem de consulta baseada na álgebra relacional e no cálculo relacional de *tuplas* (linhas). Possui também comandos para inserir, remover e modificar linhas num banco de dados.
- ***Integrity*** (Integridade): A *SQL* possui comandos para se especificar restrições de integridade, como chaves primárias e chaves estrangeiras.
- ***Transaction control*** (Controle de Transações): A *SQL* inclui comandos para especificação de início e fim de transações, visando a integridade dos dados no momento em que ocorrem atualizações.

A linguagem *SQL* pode ser utilizada de duas maneiras diferentes:

- Interativamente através de um programa que faz a interface com o banco de dados. Por exemplo, utilizando o *SQLPlus* do *Oracle*. Ele se encarrega de manter a conexão com o banco e o usuário entra com as instruções *SQL*, recebendo as respostas imediatamente; ou
- Embutida numa aplicação utilizando uma linguagem de programação comum, como Pascal, C, *Centura Builder*, *Delphi* ou *Visual Basic*, etc.

Nesta apostila, particularmente, daremos ênfase à *DML*.

Notação para Sintaxes

Nesta apostila utilizamos uma convenção para descrever as sintaxes de alguns comandos e cláusulas. Utilizaremos o comando **insert** para explicar a notação:

```
insert into <nome da tabela> | (<C1, C2 ... Cn>) | values (< V1, V2 ... Vn >)
```

Os símbolos "<" (menor) e ">" (maior) indicam que devemos substituir a frase de indicação por uma *string* válida para o comando *SQL*. Utilizando o exemplo, poderíamos substituir <nome da tabela> por FUNCIONARIO.

Quando encontrarmos uma lista do tipo: C1,C2 ... Cn, isto indica que deve-se trocar esta seqüência simbólica por uma seqüência de colunas ou valores, dependendo do caso. C1, C2 ... Cn, seguindo o exemplo, poderia ser trocada pelas colunas da tabela FUNCIONARIO, ou seja, COD_FUNCIONARIO, NOME, RG, etc.

Blocos de comandos limitados pelo símbolo "|" (pipe) são opcionais, ou seja, podem ser omitidos sem invalidar a expressão. Neste caso, a seqüência C1, C2 ... Cn pode ser omitida por estar entre pipes: | (<C1, C2 ... Cn>) | .

As outras palavras que não se encontram entre "< >" ou entre "| |" são obrigatórias (palavras chaves). As palavras **insert**, **into** e **values**, neste caso, são palavras-chaves.

Tabelas, Colunas e Linhas

Nos bancos de dados relacionais, os dados são armazenados em relações ou tabelas. Um exemplo de tabela poderia conter os dados básicos dos setores de uma empresa, como seu Código e seu Nome.

COD_SETOR	NOME
100	Vendas
200	PCP
300	Produção
400	Expedição

Tabela de Setores

Código e Nome, que no Modelo Entidade-Relacionamento (MER) nós chamamos de atributos, aqui são denominados colunas.

Uma linha é a denominação dada a cada instância armazenada na tabela de Setores, ou seja, (100, Vendas) constitui uma linha ou registro.

Da mesma forma que os dados são armazenados, o resultado de um comando de consulta SQL, também é obtido sob a forma de tabelas, ou seja, um conjunto de linhas e colunas.

O banco de dados utilizado para a construção dos exemplos citados nesta apostila foi desenvolvido a partir do exemplo "Controle de Estoque" mostrado a seguir através do seu Projeto Conceitual (Diagrama de Entidades e Relacionamentos) e correspondente diagrama de tabelas.

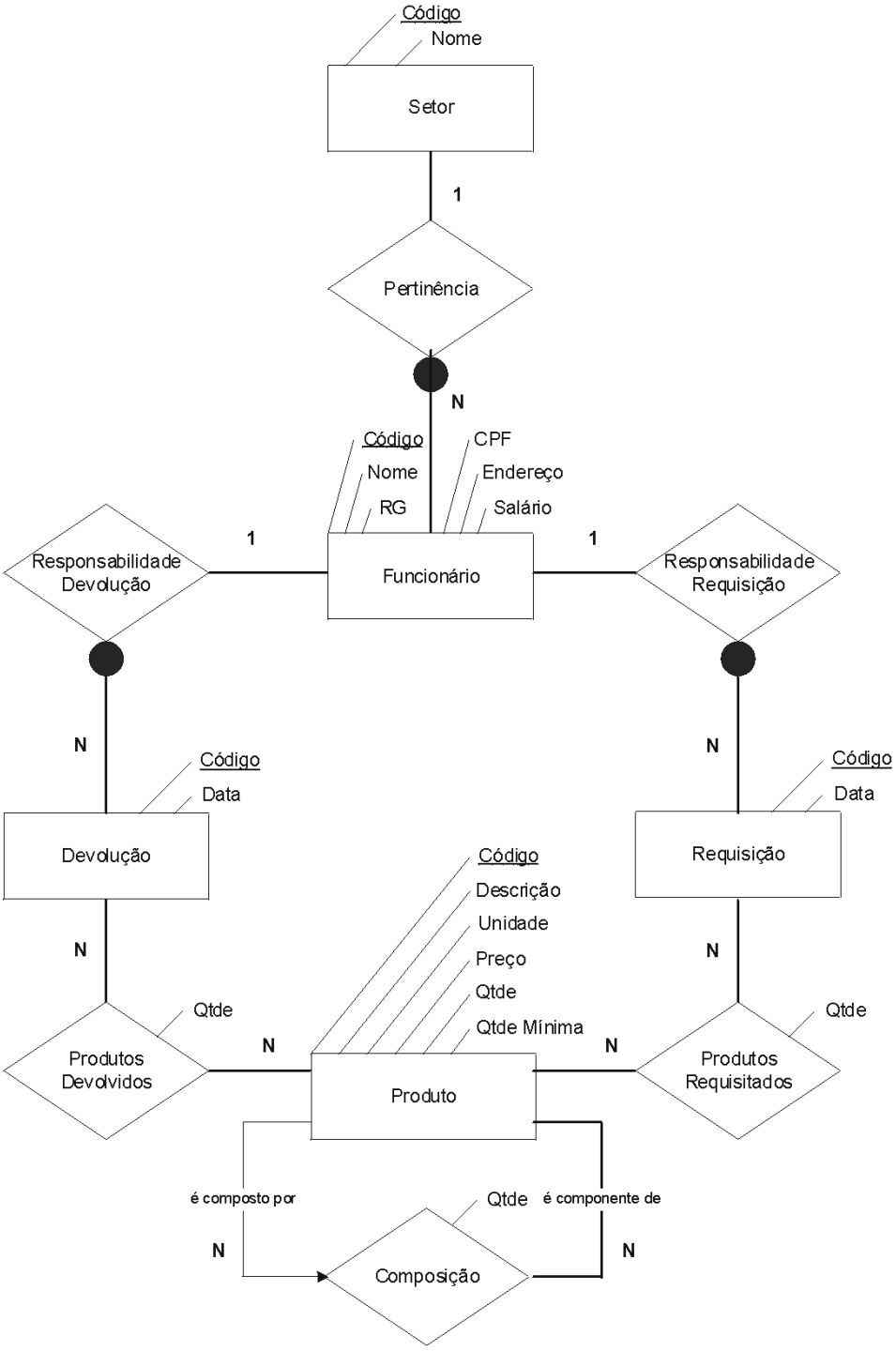
O Banco de Dados Exemplo

Todos os exemplos desta apostila estão baseados num bancos de dados exemplo, que se propõe a armazenar, simplificadamente, os dados de um controle de estoque.

A seguir, temos o seu Modelo de Entidade-Relacionamento (MER), seguido pelo seu Diagrama de Tabelas (ou Diagrama Relacional).

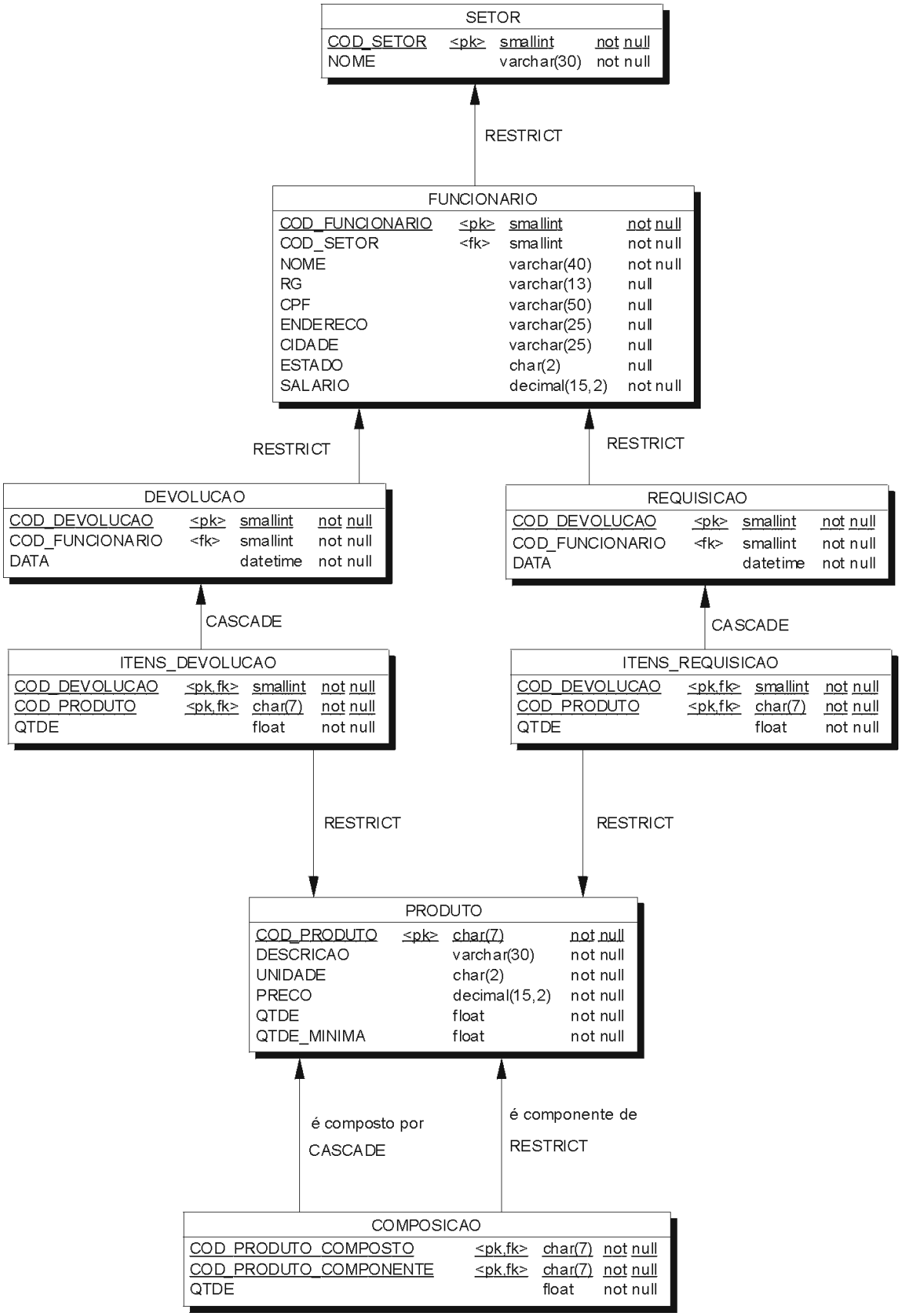
Controle de Estoque

Diagrama de Entidade-Relacionamento



Controle de Estoque

Diagrama de Tabelas



Linguagem de Manipulação de Dados Interativa (DML)

Como citado anteriormente, esta porção da linguagem *SQL* cuida de fornecer aos usuários comandos e funções para a recuperação de informações, bem como a atualização destas no banco de dados.

A instrução utilizada para consultas na base de dados é a **select**, enquanto para atualizações são utilizadas as instruções **insert**, para inserção de linhas, **delete**, para remoção de linhas, e **update**, para alterações nas linhas.

SELECT

A estrutura básica da instrução de consulta **select** consiste em três cláusulas:

- **select:** É usada para listar os atributos desejados no resultado da consulta, ou seja, as colunas requisitadas como respostas.Esta lista, além de nomes de colunas, pode conter:
  - um asterisco (*), indicando que todas as colunas devem ser retornadas;
  - expressões;
  - constantes;
  - funções;
  - qualquer combinação destes, conectadas por operadores aritméticos e parênteses.
- **from:** Lista as tabelas a serem examinadas na avaliação da expressão.As colunas indicadas na cláusula select devem constar nas tabelas listadas nesta cláusula.
- **where:** Consiste de uma condição de procura envolvendo atributos das tabelas que aparecem na cláusula **from**. É o lugar onde se restringe ou seleciona quais linhas deverão ser retornadas.

Uma consulta típica *SQL* tem a forma:

```
select    < A1, A2, ..., An >
from      < R1, R2, ..., Rn >
| where   P |
```

onde:

- Ai : representa uma coluna (atributo) de uma tabela Ri, uma expressão, constante ou função;
- Ri : representa uma tabela;
- P : uma condição de procura que utiliza os atributos de R1, ..., Rn.

A cláusula **where** pode ser omitida; neste caso, o predicado é tido como sempre verdadeiro e todas as linhas serão retornadas.

A lista de colunas na cláusula **select** (A1, .., An) pode ser substituída por um asterisco (*) para selecionar todos os atributos de todas as tabelas presentes na cláusula **from.**

Considerando uma simples consulta ao banco de dados exemplo: “Quais os códigos de todos os funcionários que fizeram alguma requisição ao estoque ?”; a instrução correspondente será:

```
select  COD_FUNCIONARIO
from    REQUISICAO
```

Eliminação de linhas duplicadas

No exemplo anterior, caso um funcionário tivesse feito mais de uma requisição, o código dele teria aparecido mais de uma vez na resposta dada pelo gerenciador de banco de dados. Entretanto, a eliminação de linhas repetidas ou duplicadas é possível utilizando a palavra-chave **distinct** depois de **select**. Então podemos reescrever a consulta anterior da seguinte maneira:

```
select distinct   COD_FUNCIONARIO
from              REQUISICAO
```

Nota-se que é permitido o uso da palavra-chave **all** para especificar explicitamente que as duplicações não serão removidas. Uma vez que a detenção das duplicações é o padrão, o uso palavra-chave **all** pode ser omitido. Abaixo segue um exemplo utilizando **all:**

```
select all   COD_FUNCIONARIO
from         REQUISICAO
```

Constantes e Expressões

Como citado anteriormente, a lista da cláusula **select** também pode conter constantes e expressões.

Exemplo utilizando constante:

```
select   COD_REQUISICAO, 'datada de', DATA
from     REQUISICAO
```

As constantes utilizadas na cláusula **select**, além de *strings*, podem ser de quaisquer outros tipos, tais como números, datas, etc.

Exemplo utilizando expressão:

```
select   COD_REQUISICAO, ( QTDE / QTDE_MINIMA ) / 100
from     PRODUTO
```

Uma expressão pode conter uma combinação de parênteses e das quatro operações aritméticas básicas:

- adição (+)
- subtração (-)
- multiplicação (*)
- divisão (/)

Condições de Procura

Uma condição de procura na cláusula **where** qualifica o escopo de um consulta, pois especifica as condições particulares que devem ser encontradas. A cláusula **where** pode ser usada nas instruções **select**, **delete**, **update**.

Cada condição de procura contém um ou mais predicados conectados por operadores lógicos **or**, **and**, e **not**, conforme o exemplo:

```
where (not P1 and P2) and P3 or P4
```

onde: Pi são predicados da cláusula **where**.

A seguinte ordem de precedência é utilizada ao avaliar uma expressão numa condição:

1. Primeiro são analisados as expressões dentro de parênteses;
2. O operador **not** é analisado antes do **and**;
3. **And** antes do **or**;
4. Operadores do mesmo nível de precedência são analisados da esquerda para a direita.

Utilizando a regra de precedência, a ordem de avaliação dos predicados, do exemplo anterior, será:

1. R1 = **not** P1

2. R2 = R1 **and** P2
3. R3 = R2 **and** P3
4. R4 = R3 **or** P4

Onde:

Ri trata-se um resultado booleano intermediário.

Predicados usam operadores, expressões, e constantes para especificar a condição a ser avaliada.

Os seguintes tipos de predicados serão discutidos:

- Relacionais
- *BETWEEN*
- *NULL*
- *LIKE*
- *EXISTS*
- *IN*

Predicados Relacionais

Os predicados relacionais são aqueles que relacionam duas expressões segundo um operador. Os seguintes operadores são suportados:

Operador	Significado
=	Igual
!=	Diferente
<>	Diferente
>	Maior
!>	Não maior (menor ou igual)
<	Menor
!<	Não menor (maior ou igual)
>=	Maior ou igual
<=	Menor ou igual

Um predicado relacional segue a seguinte sintaxe: **<expressão> <operador> <expressão>**.

Logo abaixo encontra-se um exemplo utilizando operadores relacionais. Neste exemplo, queremos obter todos os dados dos funcionários cujos salários são maiores ou iguais a R$ 1.000,000.

```
select  *
from    FUNCIONARIO
where   SALARIO >= 1000
```

Lembre que o uso do asterisco (*) na cláusula **select** indica o desejo de que todas as colunas sejam retornadas.

Predicado Between

O predicado **between** compara um valor com uma faixa de valores. Os limites da faixa estão inclusos. Ele possui a seguinte sintaxe:

<expressão> | not | between <expressão> and <expressão>

O exemplo seguinte mostra uma instrução usando **between**:

```
select   DESCRICAO
from     PRODUTO
where    PRECO between 50 and 100
```

Neste exemplo, recuperamos todas as descrições de produtos, cujos seus preços estão entre R$ 50,00 e R$ 100,00.

Predicado NULL

O predicado **NULL** testa valores nulos. Verifica, por exemplo, se colunas não contém nenhum valor armazenado. Ele possui a seguinte sintaxe:

<nome da coluna> is | not | NULL

Caso seja necessário conhecer os funcionários cujo cadastro do endereço esteja faltando, utilizaríamos a seguinte instrução:

```
select   COD_FUNCIONARIO,
         NOME
from     FUNCIONARIO
where    ENDERECO is NULL
```

É possível utilizar o operador **not** juntamente com a palavra-chave **is** (**is not NULL**) para, por exemplo, montar condições de colunas que não possuem valores nulos.

Predicado LIKE

O predicado LIKE procura por strings que se encontram dentro de um determinado padrão. O predicado LIKE só pode ser usado com tipos de dados CHAR e VARCHAR.

A sintaxe para o predicado é seguinte:

<nome da coluna> | not | like < padrão de pesquisa >

O padrão de pesquisa trata-se de uma string a ser comparada com a coluna especificada. Este padrão pode ser formatado com a utilização de coringas, permitindo a procura de substrings. Os dois caracteres coringas existentes são o símbolo porcentagem ‘%’ e o undescore ‘_’.

%	Equivale a zero ou mais caracteres
_	Equivale a um caracter qualquer

Os exemplos abaixos mostram a utilização do predicado LIKE:

- Verificar todos os funcionários que possuem “Silva” como último sobrenome.

```
select   *
from     FUNCIONARIO
where    NOME like ‘%Silva’
```

- Retornar todos os códigos das requisições dos produtos que os códigos comecem com ‘01’.

```
select   COD_REQUISICAO
from     ITENS_REQUISICAO
where    COD_PRODUTO like ‘01%’
```

- Verificar todos os produtos onde na descrição aparece a palavra ‘couro’.

```
select  *
from    PRODUTO
where   DESCRICAO like '%couro%'
```

- Retornar todos os funcionários cujo o nome possui uma das preposições: de, do ou da.

```
select  *
from    FUNCIONARIO
where   NOME like '% d_ %'
```

Predicado Exists

É utilizado para testar se o conjunto de linhas resultantes de uma consulta é vazio. Sintaxe:

| not | exists < subselect >

O exemplo seguinte mostra a utilização do predicado **exists**. Neste caso desejamos todos os nomes de funcionários que tenham feito pelo menos uma requisição. Para fazermos isto, utilizamos a seguinte lógica: para cada funcionário em FUNCIONARIO, pesquisamos em REQUISICAO se ele requisitou algo.

```
select  NOME
from    FUNCIONARIO
where   exists
        (  select  *
           from    REQUISICAO
           where   FUNCIONARIO.COD_FUNCIONARIO =
                   REQUISICAO.COD_FUNCIONARIO      )
```

Predicado IN

O predicado **in** compara um valor com um conjunto de valores. Este conjunto de valores pode ser uma lista ou o resultado de um subselect. Sintaxe:

<expressão> | not | in < subselect >

ou

<expressão> | not | in < lista de valores >

Os seguintes exemplos demonstram o uso do predicado **in**:

- Selecionar todos os dados dos funcionários que não possuem os códigos: 11, 20 e 21.

```
select  *
from    FUNCIONARIO
where   COD_FUNCIONARIO not in (11,20,21)
```

- Selecionar todas as requisições realizadas por funcionários que recebem salários superiores a R$ 1000,00.

```
select  *
from    REQUISICAO
where   COD_FUNCIONARIO in
        (  select  COD_FUNCIONARIO
           from    FUNCIONARIO
           where   SALARIO > 1000)
```

Utilizando mais de uma tabela na cláusula from

Nos exemplos que discutimos até o presente momento, só utilizamos uma tabela na cláusula **from**, ou seja, pesquisamos apenas em uma tabela individualmente. Entretanto, na maioria dos casos, precisaremos combinar e recuperar dados de diversas tabelas simultaneamente. Para conseguirmos isto, devemos relacionar na cláusula **from** todas as tabelas necessárias.

Caso fosse necessário selecionar, por exemplo, todos os funcionários (código e nome) e seus respectivos setores (código e nome) seria preciso extrair dados de duas tabelas: FUNCIONARIOS e SETORES, pois o código, nome e código do setor do funcionário poderia ser extraído da tabela FUNCIONÄRIOS, mas o nome do setor apenas da tabela setor. Desta maneira a instrução seria a seguinte:

```
select  COD_FUNCIONARIO,
        FUNCIONARIO.NOME,
        FUNCIONARIO.COD_SETOR,
        SETOR.NOME
from    FUNCIONARIO,
        SETOR
where   FUNCIONARIO.COD_SETOR = SETOR.COD_SETOR
```

A linguagem utiliza-se de um método chamado produto cartesiano para resolver esta questão. O produto cartesiano de tabelas, gera uma nova tabela auxiliar contendo todas as colunas de todas as tabelas listadas na cláusula **from**, relacionando cada linha de uma tabela com todas as outras da segunda tabela, e assim por diante, gerando uma grande tabela que possui todas as combinações de linhas posssíveis.

Para discutirmos melhor este método imaginemos que as nossas tabelas FUNCIONARIO e SETOR tivessem os seguintes dados:

FUNCIONARIO:

COD_FUNCIONARIO	NOME	COD_SETOR	Outras Colunas
10	André Pires	100	nononononononononon
11	Roberto Azevedo	100	nononononononononon
20	Carla Campos	200	nononononononononon
30	Cleber Barros	300	nononononononononon

A coluna "Outras Colunas" está representando todas as demais colunas (RG, CPF, ENDERECO, CIDADE, ESTADO, SALARIO) da tabela, apenas para simplificar a demonstração.

SETOR:

COD_SETOR	NOME
100	Vendas
200	PCP
300	Produção

Aplicando o produto cartesiano entre FUNCIONARIO e SETOR a tabela auxiliar resultante FUNCIONARIO-SETOR seria a seguinte:

FUNCIONARIO-SETOR:

COD_FUNCIONARIO	FUNCIONARIO.NOME	FUNCIONARIO.COD_SETOR	Outras Colunas	SETOR.COD_SETOR	SETOR. NOME
10	André Pires	100	nononon	100	Vendas
10	André Pires	100	nononon	200	PCP
10	André Pires	100	nononon	300	Produção
11	Roberto Azevedo	100	nononon	100	Vendas
11	Roberto Azevedo	100	nononon	200	PCP
11	Roberto Azevedo	100	nononon	300	Produção

20	Carla Campos	200	nononon	100	Vendas
20	Carla Campos	200	nononon	200	PCP
20	Carla Campos	200	nononon	300	Produção
30	Cleber Barros	300	nononon	100	Vendas
30	Cleber Barros	300	nononon	200	PCP
30	Cleber Barros	300	nononon	300	Produção

Verificamos, então, que a tabela gerada possui todas as colunas de FUNCIONARIO adicionadas às colunas de SETOR. Note que para o atributos COD_SETOR e NOME foram necessários qualificadores, representados pelo nome da tabela, para não causar ambigüidades. Sempre que colunas com nomes idênticos existirem será necessário inserir os nomes das tabelas à frente dos nomes das colunas, para fazer a distinção.

A tabela resultante ficou com 12 linhas, resultado do produto entre o número de linhas da tabela FUNCIONARIO (4 linhas) e a tabela SETOR (3 linhas). No entanto, dentre estas linhas, várias não têm sentido prático, pois FUNCIONARIO.COD_SETOR é diferente de SETOR.COD_SETOR. Deste modo, sempre que utilizarmos várias tabelas numa cláusula **from**, ou seja, utilizarmos produto cartesiano, devemos fazer o que a álgebra relacional chama de ligação natural, eliminando essas linhas. No caso a restrição foi feita através do predicado "where FUNCIONARIO.COD_SETOR = SETOR.COD_SETOR".

Chaves

Antes de discutirmos o que vem a ser a ligação natural, devemos discutir o conceito de chaves (*keys*). Num banco de dados relacional, existem dois tipos de chaves:

Chave Primária (*primary key*)

Uma chave primária é uma coluna ou um conjunto de colunas que identificam unicamente uma linha numa tabela. As colunas que fazem parte da chave primária são os atributos determinantes do MER.Por exemplo, na tabela SETOR, o código (COD_SETOR) identifica unicamente cada linha. Deste modo, uma chave primária é um valor que não pode se repetir dentro de uma mesma tabela.

Chave Estrangeira (*foreign key*)

Uma chave estrangeira é uma ou mais colunas numa tabela que são chaves primárias em outra tabela, o que significa que qualquer valor para uma coluna que é chave estrangeira deve corresponder a um valor existente na outra tabela onde a coluna é chave primária.Na linguagem dos sistemas gerenciadores de banco de dados, isto é conhecido como **integridade referencial**. Por exemplo, temos o código do setor (COD_SETOR) na tabela FUNCIONARIO que é uma chave estrangeira para o campo de mesmo nome na tabela SETOR onde este é chave primária.

Decorrente do conceito de chave estrangeira, também surge a definição de regras de deleção. Esta regra determina qual processamento deve ser praticado quando da remoção de linhas de uma tabela e que são referenciadas como chaves estrangeiras em outras tabelas. Os dois tipos de regra de deleção mais comuns são:

- **Restrita (*restrict*)**: este regra determina que uma linha somente pode ser excluída de uma tabela se os seus valores de chave primária não forem referenciados por nenhuma linha na tabela onde ele pode aparecer como chave estrangeira. No nosso exemplo, é o que acontece nas tabelas SETOR e FUNCIONARIO, onde um setor somente pode ser removido caso não exista nenhum funcionário pertencente àquele setor;
- **Em Cascata (*cascade*)**: este regra determina que a exclusão de uma linha numa tabela dispara, automaticamente (em cascata), a remoção de todas as linhas que referenciavam o seu valor como chave estrangeira na outra tabela. No exemplo, é o que ocorre entre as tabelas REQUISIÇÃO e ITENS_REQUISIÇÃO, onde a remoção de uma requisição ocasiona a remoção automática de todos os seus itens, ou seja, a remoção de todas as linhas da tabela ITENS_REQUISIÇÃO que referenciavam a requisição excluída.

Contudo, deve-se ressaltar que a regra de deleção é definida para cada relação entre duas tabelas separadamente. Deste modo, é possível a ocorrência de casos onde uma chave primária é

referenciada como chave estrangeira de regra *restrict* por uma tabela, e como regra *cascade* por outra. É o que ocorre com a tabela PRODUTO do nosso exemplo. Um produto somente poderá ser removido se não existirem nenhuma linhas nas tabelas ITENS_DEVOLUÇÃO e ITENS_REQUISIÇÃO que o referenciem e também na tabela COMPOSIÇÃO caso ele aparece como componente de outro produto. Entretanto, caso o produto apareça somente como composto, ele poderá ser normalmente excluído e todas as suas composições (linhas que o referenciam como composto na tabela COMPOSIÇÃO) também serão removidas automaticamente.

Variáveis Tuplas

SQL empresta a noção de varáveis tuplas do Cálculo Relacional de tuplas. Uma variável Tupla em SQL deve ser associada a uma relação particular, definida na cláusula **from**. Para ilustrar, vamos reescrever uma das consultas anteriores utilizando esse novo conceito. A consulta "selecionar todos os funcionários (código e nome) e seus respectivos setores (código e nome)", utilizando variáveis tuplas ficaria assim:

```
select   T.COD_FUNCIONARIO, T.FUNCIONARIO_NOME,
         S.COD_SETOR, S.NOME
from     FUNCIONARIO T, SETOR S
where    T.COD_SETOR = S.COD_SETOR
```

Varáveis Tuplas são particularmente importantes quando queremos comparar duas tuplas numa mesma relação, o que não seria possível simplesmente com a aplicação do produto cartesiano. Por exemplo, a consulta "selecionar todos os funcionários que trabalham no mesmo setor que o funcionário de código 20", resultaria na seguinte sentença SQL:

```
select   S.COD_FUNCIONARIO
from     FUNCIONARIO T,S
where    T.COD_FUNCIONARIO = 20 and
         T.COD_SETOR = S.COD_SETOR
```

Note que a variável T foi utilizada para localizar a tupla relativa ao funcionário de código 20 e seu respectivo setor, ao passo que a variável S foi utilizada para localizar todos os funcionários que trabalham neste setor. A variável S selecionará inclusive o próprio funcionário de código 20.

Operações de Conjuntos

A álgebra relacional, na qual a *SQL* se baseia, inclui as operações **union, intersect e minus,** para junções de conjuntos. Entretanto, apenas a instrução **union** foi incluída no padrão ANSI da linguagem *SQL*. Isto não chega a ser um problema, pois a instrução **intersect** pode ser realizada com a construção **in,** e a **minus** com a **not in**.

Para demonstrarmos a utilidade da instrução **union**, vamos considerar que a seguinte consulta seja requisitada: forneça todos os códigos dos funcionários que provocaram qualquer movimentação no estoque, ou seja, uma requisição ou uma devolução.

Para recuperarmos os funcionários que requisitaram produtos, utilizaríamos a seguinte instrução:

```
select   COD_FUNCIONARIO
from     REQUISICAO
```

Para aqueles que devolveram algum produto:

```
select   COD_FUNCIONARIO
from     DEVOLUCAO
```

Agora para fazermos a união destes dois conjuntos de valores, utilizamos o operador **union**. Assim a instrução resultante é:

```
select   COD_FUNCIONARIO
from     REQUISICAO

union

select   COD_FUNCIONARIO
from     DEVOLUCAO
```

Se a consulta fosse os códigos dos funcionários que participaram de uma requisição e de uma devolução, teríamos:

```
select   COD_FUNCIONARIO
from     REQUISICAO

intersect

select   COD_FUNCIONARIO
from     DEVOLUCAO
```

O mesmo resultado obteríamos com a seguinte sentença:

```
select   COD_FUNCIONARIO
from     REQUISICAO
where    COD_FUNCIONARIO in
         (select     COD_FUNCIONARIO
          from       DEVOLUCAO)
```

Se a consulta fosse para obter os códigos dos funcionários que participaram de alguma requisição, mas não participaram de nenhuma devolução, teríamos:

```
select   COD_FUNCIONARIO
from     REQUISICAO

minus

select   COD_FUNCIONARIO
from     DEVOLUCAO
```

O mesmo resultado obteríamos com a seguinte sentença:

```
select   COD_FUNCIONARIO
from     REQUISICAO
where    COD_FUNCIONARIO not in
         (select     COD_FUNCIONARIO
          from       DEVOLUCAO)
```

Portanto pudemos comprovar que tanto a operação **intersect** como a operação **minus** podem ser realizadas em construções análogas utilizando **in** e **not in**, respectivamente.

É importante ressaltar que o operador **union** trata as tabelas como conjuntos, e pela regra de união de conjuntos, valores duplicados são descartados. Assim, se um funcionário requisitou e devolveu algum produto, ele aparecerá apenas uma vez na lista resultante. Caso seja necessário manter as duplicações, deve-se utilizar o operador **union all** ao invés de **union.**

Exibindo Resultados Ordenados

SQL oferece controle sobre a ordem na qual as linhas do resultado de uma consulta estão dispostas. A cláusula **order by** ocasiona o aparecimento das linhas de resultado de uma consulta em uma ordem determinada. Por exemplo para listar em ordem alfabética todos os funcionários do setor 100, escrevemos:

```
select     F.NOME, COD_FUNCIONARIO
from       FUNCIONARIO F,
           SETOR S
where      F.COD_SETOR = S.COD_SETOR and
           S.COD_SETOR = 100
order by   F.NOME
```

Como padrão, *SQL* lista itens na ordem ascendente. Para especificar a ordem de classificação, podemos especificar **desc** para descendente ou **asc** para ascendente. Além do mais, a ordenação pode ser feita por múltiplos atributos, bastando, para isto listá-los na cláusula **order by**. Suponha que fosse necessário gerar uma lista contendo as requisições realizadas em ordem descendente, ou seja, primeiro pelos mais recentes. Para as requisições realizadas no mesmo momento, ordenamos em ordem ascendente pelo número da requisição. Neste caso o comando *SQL* seria:

```
select     *
from       REQUISICAO
order by   data desc, COD_REQUISICAO asc
```

Para atender a cláusula **order by**, *SQL* precisa executar uma classificação. Uma vez que uma classificação de um grande número de linhas pode ser custoso, deve-se utilizá-la apenas quando necessário.

Funções de agregação

SQL oferece a possibilidade de computar funções em grupos de linhas usando a cláusula **group by**. Os atributos, dados na cláusula **group by,** são usados para formar grupos. Linhas com o mesmo valor em todos os atributos na cláusula **group by** são colocados em um grupo.

SQL inclui funções de agregação para computar:

- média: **avg**
- mínimo: **min**
- máximo: **max**
- total: **sum**
- contar: **count**

As operações, como a **avg**, são chamadas funções de agregação, porque operam em agregações de linhas. O resultado de uma função de agregação é um valor único. Para ilustrar, considere a consulta “Selecione o salário médio em cada setor da empresa”. A instrução ficaria assim:

```
select     COD_SETOR,
           avg (SALARIO)
from       FUNCIONARIO
group by   COD_SETOR
```

Função de Agregação	Resultado
SUM (| DISTINCT | expressão)	A somatória de valores (distintos) na expressão numérica
AVG (| DISTINCT | expressão)	A média de valores (distintos) na expressão numérica
COUNT (| DISTINCT | expressão)	O número de valores (distintos e) não nulos na expressão
COUNT (*)	O número de linhas selecionadas
MAX (expressão)	O maior valor computado para a expressão
MIN (expressão)	O menor valor computado para a expressão

Funções de Agregação e seus resultados

Deve-se notar que as funções SUM e AVG somente aceitam expressões numéricas como argumentos, uma vez que elas calculam a soma e a média de valores, respectivamente. As demais (COUNT, MAX e MIN) aceitam qualquer tipo de dado na expressão como parâmetro.

A principal particularidade a ser ressaltada é com relação à função COUNT. COUNT(expressão) e COUNT(*) são sutilmente diferentes, uma vez que COUNT(expressão) não conta as linhas nulas e COUNT(*) realiza a contagem de todas as linhas, sejam elas nulas ou não.

Outra questão a ser ressaltada com relação à utilização da cláusula **group by** é que a lista de valores da cláusula **select** somente aceita funções de agregação, constantes e expressões combinando-as, além dos campos listados na cláusula **group by**. Tal restrição é evidente, uma vez que a inclusão de outros campos na lista do **select** seria totalmente sem sentido e impossível de ser retornada.

Porém, podemos também usar as funções de agregação sem a cláusula **group by**. Neste caso a tabela inteira é tratada como um único grupo. Uma consulta, para contar o número de funcionários da sua empresa, pode ser expressa da seguinte maneira:

```
select   count (* )
from FUNCIONARIO
```

A Cláusula HAVING

Em certas situações, é necessário definir uma condição que se aplique a grupos, ao invés de linhas. Imagine se estivéssemos interessados em saber o salário médio apenas daqueles setores, cujo salário médio seja maior que R$ 1100,00. Esta condição não se aplica a linhas simples, mas sim a cada grupo construído com a cláusula **group by**. Deste modo, não podemos utilizar funções de agregação na cláusula **where** e para expressar tal condição utilizamos a cláusula **having**. Os predicados na cláusula **having** são aplicados depois da formação dos grupos, para que funções de agregação possam ser usadas. Expressamos esta consulta *SQL* assim:

```
select      COD_SETOR,
            avg (SALARIO)
from        FUNCIONARIO
group by    COD_SETOR
having      avg (SALARIO) > 1100
```

Se uma cláusula **where** e uma cláusula **having** aparecerem numa mesma consulta, o predicado na cláusula **where** é aplicado primeiro. As linhas que satisfazem a condição da cláusula **where** são então agrupadas pela cláusula **group by**. Em seguida, o predicado da cláusula **having** é aplicado a cada grupo. Ilustrando a utilização de **where** e **having** na mesma instrução, faremos a seguinte consulta: código do produto e quantidade total requisitada, para os produtos que possuem quantidade total requisitada maior que 10.

```
select      COD_PRODUTO,
            sum (QTDE )
```

```
from      REQUISICAO R,
          ITENS_REQUISICAO I
where     R.COD_REQUISICAO = I.COD_REQUISICAO
group by  COD_PRODUTO
having    sum (QTDE) > 100
```

Consultas Complexas

SQL permite que alguns consultas mais elaboradas possam ser feitas, utilizando apenas variáveis tuplas sem as funções de agregação. Por exemplo, uma sentença em SQL para devolver todos os setores que têm mais que 15 funcionários alocados, seria:

```
select    distinct  S.NOME
from      SETOR S
where     ( select    count(*)
            from      FUNCIONARIO T
            where     T.COD_SETOR = S.COD_SETOR)  > 15
```

Funções Built-in

Além dos comandos, cláusulas e funções agregadas da linguagem *SQL*, os gerenciadores de bancos de dados disponibilizam uma série de outras funções para ajudar o usuário na manipulação dos dados. Estas funções podem ser utilizadas normalmente com as instruções *SQL* comuns.

Contudo, essas extensões acarretam uma grande desvantagem quando se pretende desenvolver sistemas multiplataformas, pois estas funções são específicas para cada servidor de banco de dados. Por exemplo, uma função de conversão de inteiro para string, por exemplo, é diferente entre o *Oracle* e o *Sybase*.

O *Oracle* fornece um grande conjunto de funções para manipular strings, datas e números. Dentre elas, podemos destacar:

Funções para manipulação de strings

Função	Propósito
ASCII(x)	Retorna o valor numérico ASCII do caracter x
CHR(x)	Retorna o caracter ASCII correspondente ao inteiro x
CONCAT(x,y)	Retorna x concatenado com y
INITCAP(x)	Retorna a string x com a primeira letra de cada palavra em maiúscula e as restantes em minúscula (nome próprio)
LENGTH(x)	Retrona o tamanho da string x
LENGTHB(x)	Retorna o tamanho em bytes da string x
LOWER(x)	Retorna a string x com todas as letras em minúscula
LTRIM(x|,y|)	Elimina caracteres à esquerda da string x e retorna a string resultante. Se y não for especificado, caracteres em branco serão removidos. O valor da string y (opcional) é o conjunto de caracteres a ser eliminado
REPLACE(x,y|,z|)	Retorna a string x com todas as ocorrências de y substituídas por z. Se z for omitido, todas as ocorrências de y serão simplesmente removidas
RTRIM(x|,y|)	Elimina caracteres à direita da string x e retorna a string resultante. Se y não for especificado, caracteres em branco serão removidos. O valor da string y (opcional) é o conjunto de caracteres a ser eliminado
SUBSTR(x,y|,z|)	Retorna a parte da string x, iniciando com o caracter y. Se z for especificado, a função retorna z caracteres a partir de y (inclusive); caso contrário, a parte retornada vai até o final da string x
UPPER(x)	Retorna a string x com todas as letras em maiúscula

Um exemplo, de utilização das funções de manipulação de *strings*, seria a eliminação de todos os espaços em branco que tenham sido digitados à direita dos nomes dos funcionários:

```
update  FUNCIONARIO
set     NOME = RTRIM(NOME)
```

Funções para manipulação de números

Função	Propósito
ASCII(x)	Retorna o valor absoluto de x
CEIL(x)	Retorna o menor inteiro superior ou igual a x
COS(x)	Retorna o co-seno de x
COSH(x)	Retorna o co-seno hiperbólico de x
EXP(x)	Retorna *e* elevado à potência de x
FLOOR(x)	Retorna o maior inteiro inferior ou igual a x
LN(x)	Retorna o logaritmo natural de x
LOG(x,y)	Retorna o logaritmo de base x de y
MOD(x,y)	Retorna o resto de x dividido por y
POWER(x,y)	Retorna o x elevado à potência y
ROUND(x|,y|)	Retorna o x arredondado y casas à direita da casa decimal. O valor padrão de y, caso seja omitido, é 0 (zero)
SIGN(x)	Retorna o sinal de x como -1 se x for negativo; 0 (zero) se x for 0 (zero); e 1 se x for positivo
SIN(x)	Retorna o seno de x
SINH(x)	Retorna o seno hiperbólico de x
SQRT(x)	Retorna a raiz quadrada de x
TAN(x)	Retorna a tangente de x
TANH(x)	Retorna a tangente hiperbólica de x
TRUNC(x|,y|)	Retorna x truncado para y casas decimais. O valor padrão de y, caso omitido, é 0 (zero)

Para exemplificar o uso de funções de manipulação de números, desejamos eliminar os centavos dos preços dos produtos, arredondando todos os preços para cima.

```
update  PRODUTO
set     PRECO = CEIL(PRECO)
```

Funções para manipulação de datas

Função	Propósito
ADD_MONTHS(x,y)	Retorna a data x acrescida de y meses
LAST_DAY(x)	Retorna o último dia do mesmo da data x
MONTHS_BETWEEN(x)	Retorna o número de meses entre as datas x e y
SYSDATE	Retorna a data e hora atuais do sistema

Para exemplificar o uso de funções de manipulação de datas, desejamos saber quantos meses decorreram desde a primeira requisição do produto ‘0101001’ até a data atual.

```
select  MONTHS_BETWEEN(MIN(R.DATA),SYSDATE)
from    REQUISICAO R,
        ITENS_REQUISICAO IR
```

```
where   R.COD_REQUISICAO = IR.COD_REQUISICAO and
        IR.COD_PRODUTO='0101001'
```

Funções de conversão

Função	Propósito
TO_CHAR(x|,y|)	Converte uma data ou um número x em uma string de caracteres. A string y define o formato de conversão da data ou número. Exemplos de formato são 'dd-MM-yyy' e '999.99'
TO_DATE(x|,y|)	Converte a string x em uma data, usando o modelo de formato y
TO_NUMBER(x|,y|)	Converte a string x em um número, usando o modelo de formato y

Um exemplo do uso de funções de conversão, seria um comando select que retornasse o valor numérico dos anos em que foram realizadas requisições ao estoque.

```
select distinct   TO_NUMBER(TO_CHAR(DATA,'yyyy'))
from              REQUISICAO
```

Modificando o Banco de Dados

Até o presente momento, restringimos nossa atenção à consulta ao banco de dados. Agora discutiremos as instruções da *SQL* para atualização do banco, através dos comandos **insert, delete** e **update**.

Inserindo Linhas: INSERT

Para inserir um dado numa tabela, podemos fazê-lo de duas formas:

- especificando uma linha a ser inserida, ou seja, os campos e seus valores;
- escrevendo uma consulta cujo o resultado seja um conjunto de linhas a serem inseridas.

A primeira maneira é mais simples, pois inserimos uma linha por vez, atribuindo valores as colunas da tabela. Sintaxes:

insert into <nome da tabela> | (<C1, C2 ... Cn>) | values (< V1, V2 ... Vn >)

insert into <nome da tabela> <select ... from ... where ...>

onde:

- Ci : nome da coluna
- Vi : valor a ser inserido. Pode ser uma constante ou uma expressão.

Os nomes das colunas podem ser omitidos, se a ordem dos valores a serem inseridos corresponder com a ordem de criação dos mesmos.

Para incluirmos o setor de Compras no nosso banco de dados exemplo utilizaríamos a seguinte instrução:

```
insert into   SETOR
              (COD_SETOR, NOME )
values        (500, 'Compras')
```

Deve-se notar que ao fazer uma inserção todas as restrições (*constraints*) de integridade referencial declaradas serão processadas, de modo a preservar a consistência dos dados. Na tabela SETOR há apenas a restrição de chave-primária (COD_SETOR), que não permitirá a inserção de um setor com o mesmo código. Entretanto, em outras tabelas, como a FUNCIONARIO, existem restrições de chave estrangeira. Esta restrição não permitirá que um funcionário seja inserido num setor inexistente, ou seja, que o seu COD_SETOR não exista na tabela SETOR.

Pode-se também fazer uma inserção através do resultado de uma consulta. Conseguimos assim fazer uma inserção em lote. Imaginemos um caso, onde a requisição de código 1015 realizada pelo funcionário 10, deve ser extornada, ou seja, todos os itens que constam como requisitados deverão ser lançados na devolução de código 5005 com a mesma data da requisição. Deveríamos fazer os seguintes **inserts**:

```
insert into   DEVOLUCAO
              select   5005,
                       COD_FUNCIONARIO,
                       DATA
              from     REQUISICAO
              where    COD_REQUISICAO = 1015

insert into   ITENS_DEVOLUCAO
              select   5005,
                       COD_PRODUTO,
                       QUANTIDADE
              from     ITENS_REQUISICAO
              where    COD_REQUISICAO = 1015
```

No exemplo dado sobre **insert** considerou-se que todos os valores foram dados. Porém isto não é uma regra, pois há caso de colunas que aceitam valores nulos. Deste modo pode-se numa instrução **insert** utilizar a palavra reservada **null** para indicar a inserção de um valor nulo. Exemplo:

```
insert into   FUNCIONARIOS
              (COD_FUNCIONÁRIO, COD_SETOR, NOME, RG, CPF,
              ENDERECO, CIDADE, ESTADO, SALARIO)
values        (51,500,'José Martins',null,null,null,null,null,1300)
```

Com esta instrução, inserimos o funcionário José Martins. Todas as colunas que poderiam conter valores nulos (RG, CPF, ENDERECO, CIDADE, ESTADO), segundo suas definições no banco de dados, receberam **null** como conteúdo. Outra forma de fazer esta inserção seria explicitando apenas os nomes das colunas que receberiam valores nulos. Desta outra forma, o exemplo acima ficaria do seguinte modo:

```
insert into   FUNCIONARIOS
              (COD_FUNCIONÁRIO, COD_SETOR, NOME, SALARIO)
values        (51,500,'José Martins',1300)
```

Removendo Linhas: DELETE

Uma requisição de remoção é expressa da mesma forma que uma consulta. Podemos remover apenas linhas inteiras, e não apenas alguns atributos. A instrução **delete** possui a seguinte sintaxe:

delete from <nome da tabela> | where < condição de procura > |

Se desejarmos remover linhas de diversas tabelas, precisaremos utilizar um comando **delete** para cada uma. A condição de procura na cláusula **where** pode ser tão complexa quanto a do comando **select**. Podemos também omitir a cláusula **where**, contudo devemos ter muito cuidado, pois, neste caso, todas as linhas da tabelas serão excluídas.

Em seguida é mostrado um comando **delete** que exclui um item de uma requisição:

```
delete from   ITENS_REQUISICAO
where         COD_REQUISICAO = 432 and
              COD_PRODUTO = '0101001'
```

Assim como na inserção, a remoção de linhas também ocasiona o processamento de todas as restrições relativas a integridade referencial tangentes à tabela que está sofrendo a remoção.

Nota-se que neste **delete** não haverá nenhum problema quanto as restrições declaradas, pois as chaves primárias da tabela ITENS_REQUISICAO não são chave estrangeira em nenhuma outra tabela.

Porém, se tentarmos excluir um funcionário, poderíamos ter problemas, se no relacionamento entre FUNCIONARIO e DEVOLUCAO ou REQUISICAO, a restrição relativa a chave estrangeira COD_FUNCIONARIO estiver configurada como ***restrict*** (que é o caso do nosso exemplo). Neste caso, a deleção só iria se proceder, caso o funcionário a ser excluído não tivesse feito nenhuma requisição ou devolução. Se a regra de deleção estiver setada como ***cascade***, ao excluir o funcionário todas as requisições e devoluções seriam eliminadas juntamente com o funcionário em questão.

Modificando Linhas: UPDATE

Utilizamos o comando **update,** quando desejamos mudar alguns valores em algumas colunas de uma tabela. Ressalta-se que a atualização pode ser feita em apenas uma tabela por vez. Como no **select** e **delete,** podemos utilizar de uma condição de procura, para selecionar as linhas que desejamos que sofram modificações. Sintaxe do comando **update**:

update <nome da tabela> set (C1 = V1, C2=V2 ... Cn = Vn) | where <condição de procura> |

onde:

- Ci : coluna da tabela;

- Vi : novo valor, podendo ser uma constante, ou uma expressão.

Suponha que queiramos atualizar os sálarios dos funcionários do setor 200 em 10%. Utilizaríamos a seguinte instrução:

```
update    FUNCIONARIO
set       SALARIO = SALARIO * 1.1
where     COD_SETOR = 200
```

Assim como no comando **delete**, a clásula **where** também pode ser omitida no **update**. Deste modo, todas as linhas da tabela sofreriam a atualização.

Supondo que desejemos atualizar o salário de todos os funcionários em 10%, independentemente do setor, a instrução *SQL* se resumiria a:

```
update    FUNCIONARIO
set       SALARIO = SALARIO * 1.1
```

Introdução a DDL - Linguagem de Definição de Dados

*Tabelas (*Tables*) - Criando, Alterando e Excluindo*

Como já citado, as tabelas são os blocos básicos de construção de qualquer banco de dados e, com os comandos de definição de dados da linguagem SQL, podemos criar, excluir e alterar tabelas.

Quando se cria uma tabela com SQL, na verdade, são definidas as colunas, com seus tipos e outras características, que cada linha criada na tabela armazenará.Para a definição de uma coluna são necessários: um nome, um tipo e se a coluna permitirá o armazenamento de valores nulos.

O comando para criação de tabelas, é o **CREATE TABLE**. Eis, sua sintaxe:

create table <nome da tabela> (<nome da coluna 1> <tipo de dado> | NULL ou NOT NULL |,

..., <nome da coluna n> <tipo de dado> | NULL ou NOT NULL |)

Como exemplo do comando CREATE_TABLE, citamos o comando para criação da tabela de setores:

```
create table  SETOR
          (   COD_SETOR     smallint       not null,
              NOME          varchar(30)    not null )
```

O padrão *SQL-92* (também conhecido como *SQL2*) acrescentou ao comando CREATE TABLE algumas cláusulas para declaração de restrições para os campos de tabelas. Estas cláusulas são:

- **PRIMARY KEY**
- **DEFAULT**: quando um campo for declarado, é possível especificar o seu valor default, ou seja, se na inserção de uma linha este campo não tiver um valor fornecido, o valor default é armazenado. Isto é útil quando se tem uma grande quantidade (quase maioria) de linhas onde uma coluna assume sempre um mesmo valor. Deste modo, somente seria necessário especificar um valor para esta coluna, numa inserção, quando este fosse diferente do default, ou seja, em poucos casos.
- **CHECK**: restringe a faixa de valores que a coluna pode assumir. Restrições de *check* são também chamadas de regras (*rules*) ou regras de validação (*validation rules*), uma vez que permitem que o próprio gerenciador de bancos de dados verifique se o valor sendo atribuído para a coluna está dentre os valores válidos.
- **REFERENCES**: caso uma coluna seja definida com REFERENCES, ou seja, a coluna é uma chave estrangeira, quando um valor está para ser atribuído para esta coluna, primeiro é verificado se o mesmo valor existe na tabela e coluna referenciadas.
- **UNIQUE**: garante que valores para uma coluna não se repitam dentro da tabela. Esta restrição também impõe que a coluna não possa aceitar valores nulos.

Como exemplo da definição de restrições, temos um comando para criação da tabela REQUISIÇÃO:

```
create table  REQUISICAO
          (   COD_REQUISICAO    smallint     primary key,
              COD_FUNCIONARIO   smallint     not null  references FUNCIONARIO,
                 DATA                        date         default SYSDATE
                                  check (DATA <= TO_DATE('01-01-1980','dd-MM-yyyy') ) )
```

Deve-se notar que na definição da tabela REQUISICAO foram utilizadas algumas funções *built-in* do Oracle, as quais devem ser substituídas caso a tabela seja gerada em outro gerenciador de bancos de dados.

Já com relação às regras de deleção, a sua definição varia de acordo com cada gerenciador. No *Oracle*, ela pode ser declarada após a restrição *references*.

Outra questão a ser ressaltada é que quando a chave primária é composta por mais de um campo, esta deve ser declarada após todos os campos.

Vejamos como ficaria um exemplo, acrescentando uma regra de deleção e uma chave primária com mais de um campo. Deve-se considerar que, no Oracle, a regra de deleção default é a restrita (*restrict*):

```
create table   ITENS_REQUISICAO
          (  COD_REQUISICAO    smallint      not null  references REQUISICAO
                                                       on delete cascade,
             COD_PRODUTO     char(7)       not null  references PRODUTO,
                DATA                      float          not null )
```

A linguagem *SQL* também permite a alteração de componentes de uma tabela. O comando utilizado para este fim é o ALTER TABLE. Sua sintaxe inclui palavras-chave para adicionar e remover colunas, bem como alterar seus atributos de nome, tipo de dado, tamanho, etc. Sintaxe:

```
alter table <nome da tabela>  ADD ( <nome da nova coluna> <tipo de dado> | NULL ou NOT NULL | )

alter table <nome da tabela>  DROP ( <nome da coluna>

alter table <nome da tabela>  MODIFY ( <nome da coluna> <tipo de dado> | NULL ou NOT NULL | )
```

Contudo, deve-se considerar que as alterações de tabelas gozam de algumas restrições em certos gerenciadores de bancos de dados, vetando, por exemplo, a adição de uma coluna que não permita valores nulos, numa tabela que já contém linhas armazenadas.

A DDL também dispõe de um comando para excluir tabelas: o DROP TABLE. Sintaxe:

```
drop table <nome da tabela>
```

*Visões (*Views*)*

Uma visão *(view)* permite que você associe os resultados de uma consulta a um nome, que você pode utilizar em outras consultas, como se fosse uma tabela normal. Quando você acessa uma *view*, a consulta que está definida para esta é executada e os resultados desta trabalham com uma nova tabela base para a nova consulta que está sendo praticada. Sintaxe:

```
create view <nome da visão> as <select ... from ... where ...>
```

Como exemplo, poderíamos criar uma visão de nome FUNCIONARIOS_VENDAS, a qual conteria apenas os funcionários do setor de Vendas. Eis o comando:

```
create view   FUNCIONARIOS_VENDAS as
                 select   *
                             from     FUNCIONARIO
              where   COD_SETOR = 100
```

Após criada a *view* FUNCIONARIOS_VENDAS, caso realizemos um select * na mesma (select * from FUNCIONARIOS_VENDAS), apenas as linhas dos funcionários cujo código do setor é igual a 100 (Vendas) seriam exibidos como resultado.

Para excluir *views*, a DDL dispõe do comando DROP VIEW. Sintaxe:

```
drop view <nome da visão>
```

*Índices (*Indexes*)*

Os índices são mecanismos que permitem que o gerenciador de bancos de dados acesse um dado mais rápidos; todavia, o mecanismo de índice não é padrão e nem todos os gerenciadores o implementam. Os sistemas criam uma estrutura interna de dados (o índice), os quais são envolvidos, quando um comando de consulta *SQL* (um *select*) é baseado na coluna do índice, para que a consulta seja processada mais rapidamente.

O comando para criação de índices é o CREATE INDEX. Sintaxe:

create index <nome do índice> on <nome da tabela> (<nome da coluna 1>, ..., <nome da coluna n>)

A título de exemplo, poderíamos imaginar que várias consultas envolvendo a tabela FUNCIONARIO baseiam-se na coluna NOME. Deste modo, deveríamos criar um índice para esta coluna, de modo que as consultas se processem de forma mais rápida:

```
create index IDX_NOME_FUNC on FUNCIONARIO (NOME)
```

Para a exclusão de um índice, o comando é o DROP INDEX. Sintaxe:

drop index <nome da tabela>.<nome do índice>