

# **Apostila de Banco de Dados II**

**V1.0 2014B**

## Sumário

<b>1</b>	<b>Introdução .....</b>	<b>8</b>
1.1	<i>História dos Sistemas de Banco de Dados .....</i>	9
1.2	<i>Linguagem de Banco de Dados .....</i>	9
1.2.1	Linguagem de Definição de Dados .....	9
1.2.2	Linguagem de Manipulação de Dados .....	10
1.2.3	Outras Linguagens .....	10
1.2.3.1	Query-by-Example(QBE) .....	10
1.2.3.2	Quel.....	10
1.2.3.3	DataLog .....	10
1.3	<i>Os Módulos Componentes de um SGBD .....</i>	11
1.3.1	Gerenciador de Armazenamento.....	11
1.3.2	Processador de Consultas .....	12
1.3.3	Gerenciamento de Transação .....	12
<b>2</b>	<b>Álgebra Relacional.....</b>	<b>14</b>
2.1	<i>A Operação Select .....</i>	14
2.2	<i>A Operação Project.....</i>	15
2.3	<i>Seqüencialidade de Operações.....</i>	15
2.4	<i>Operações Matemáticas .....</i>	16
2.5	<i>Produto Cartesiano .....</i>	19
2.6	<i>Operação Rename .....</i>	21
2.7	<i>Junção Natural .....</i>	22
2.8	<i>Junção Externa.....</i>	23
2.9	<i>Otimização de Consultas.....</i>	24
2.10	<i>Grafo de Consulta .....</i>	24
<b>3</b>	<b>SQL .....</b>	<b>26</b>
3.1	<i>Histórico.....</i>	26
3.2	<i>Conceitos .....</i>	26
3.3	<i>Partes.....</i>	26
<b>4</b>	<b>SQL DDL .....</b>	<b>27</b>
4.1	<i>Linguagem de Definição de Dados.....</i>	27
4.2	<i>Esquema Base.....</i>	27
4.3	<i>Tipos de Domínios em SQL .....</i>	27
4.3.1	Definição de Domínios .....	28
4.4	<i>Definição de Esquema em SQL .....</i>	28
4.4.1	Extensão Create Table .....	30
4.4.2	Esvaziar Tabelas .....	30
4.4.3	Remover Tabelas .....	31
4.4.4	Adicionar Atributos .....	31
4.4.5	Alterar Atributos .....	31
4.4.6	Renomear Atributos .....	31
4.4.7	Excluir Atributos.....	31
4.4.8	Renomear Tabelas.....	31
4.4.9	Metadados .....	31
4.5	<i>Integridade .....</i>	32

4.5.1	Definição de Constraints.....	32
4.5.1.1	CONSTRAINTS IN-LINE .....	32
4.5.1.2	CONSTRAINTS OUT-OF-LINE .....	32
4.5.2	Constraints .....	32
4.5.2.1	NOT NULL CONSTRAINT .....	32
4.5.2.2	UNIQUE CONSTRAINT .....	33
4.5.2.3	PRIMARY KEY CONSTRAINT .....	33
4.5.2.4	FOREIGN KEY CONSTRAINT .....	33
4.5.2.5	CHECK CONSTRAINT .....	34
4.5.2.6	DEFAULT SPECIFICATION .....	34
4.5.3	Padronização Nomes de Constraints .....	34
4.5.4	Deleção em Cascata .....	35
4.5.5	Atualização em Cascata .....	35
4.5.6	Adicionando uma Foreign Key para a própria tabela.....	35
4.5.7	Adicionar Constraints .....	35
4.5.8	Excluir Constraint .....	36
4.5.9	Renomear Constraint .....	36
4.5.10	Ativar Constraints .....	36
4.5.11	Desativar Constraints .....	36
4.5.12	Metadados .....	37
4.6	<i>Dicionário de Dados</i> .....	37
4.6.1	Comentando Tabelas.....	37
4.6.2	Visualizando o comentário de Tabelas.....	37
4.6.3	Comentando Atributos .....	37
4.6.4	Visualizando o comentário de Atributos .....	38
<b>5</b>	<b>SQL DML .....</b>	<b>39</b>
5.1	<i>Linguagem de Manipulação de Dados</i> .....	39
5.2	<i>Esquema Base</i> .....	39
5.3	<i>Estruturas Básicas</i> .....	39
5.3.1	Cláusula Select.....	39
5.3.2	A cláusula Where .....	40
5.3.3	A cláusula From.....	40
5.3.4	A operação Rename .....	41
5.3.5	Variáveis Tuplas .....	41
5.3.6	Operações em Strings.....	41
5.3.7	Expressão Regular .....	42
5.3.8	Precedência dos Operadores .....	45
5.3.9	Ordenação e Apresentação de Tuplas .....	46
5.4	<i>Composição de Relações</i> .....	46
5.4.1	Tipos de Junções e Condições .....	46
5.4.2	Junção Interna .....	48
5.4.3	Junção de Mais de duas tabelas.....	48
5.4.4	Junção Externa .....	48
5.4.5	Auto Junção .....	49
5.5	<i>Operações de Conjuntos</i> .....	49
5.5.1	A operação de União.....	49
5.5.2	A operação Interseção .....	50
5.5.3	A operação Exceto(Subtração).....	50
5.6	<i>Funções Agregadas</i> .....	50
5.6.1	Outras Funções.....	52
5.6.1.1	RollUp.....	52
5.6.1.2	Cube.....	52
5.6.1.3	ListAgg .....	52
5.6.2	Mais funções .....	53
5.7	<i>Valores Nulos</i> .....	53

5.8	<i>Subconsultas Aninhadas</i> .....	53
5.8.1	Membros de Conjuntos .....	53
5.8.2	Comparação de Conjuntos .....	54
5.8.3	Verificação de Relações Vazias .....	55
5.8.4	Teste para a Ausência de Tuplas Repetidas .....	55
5.8.5	Subconsulta Escalar .....	55
5.9	<i>Modificações no Banco de Dados</i> .....	55
5.9.1	Remoção .....	55
5.9.2	Inserção .....	56
5.9.2.1	Inserção mais complexa .....	56
5.9.2.2	Inserção com null .....	56
5.9.2.3	Inserção de Data .....	57
5.9.3	Atualização .....	57
5.10	<i>Transação</i> .....	57
5.10.1	Commit .....	58
5.10.2	RollBack .....	58
5.10.3	Pontos de Transação .....	58
5.10.4	Locks .....	59
5.10.5	Estado dos Dados .....	60
5.10.5.1	Antes do Commit e Rollback .....	60
5.10.5.2	Após o Commit e Rollback .....	60
5.10.6	Rollforward .....	60
5.10.7	Syncpoint .....	60
5.11	<i>Índice</i> .....	60
5.11.1	Criando Índice .....	61
5.11.2	Apagando Índice .....	61
5.12	<i>Visões</i> .....	61
5.12.1	Visões Normais .....	61
5.12.2	Visões Materializadas .....	62
5.12.3	Atualizações de uma Visão .....	64
5.12.4	Metadados .....	64
5.13	<i>Seqüências</i> .....	64
5.13.1	Usando Seqüências .....	65
5.13.2	Criando uma Seqüência .....	65
5.13.3	Apagando uma Seqüência .....	65
5.13.4	Alterando uma Seqüência .....	65
5.13.5	Usando uma Seqüência .....	65
5.13.6	Metadados .....	66
5.14	<i>Triggers</i> .....	66
5.14.1	Tipos de Trigger .....	66
5.14.2	Componentes de uma Trigger .....	66
5.14.3	Momento de disparo de uma trigger .....	66
5.14.4	Criação de uma Trigger .....	66
5.14.5	Ativação/Desativação de uma Trigger .....	67
5.14.6	Alteração de uma Trigger .....	67
5.14.7	Remoção de uma Trigger .....	67
5.14.8	Limitações de Uso de Trigger .....	67
5.14.9	Referência a Colunas dentro de uma Trigger .....	67
5.14.10	Predicados Condicionais .....	68
5.14.11	Metadados .....	68
5.15	<i>Stored Procedures</i> .....	68
5.15.1	Sintaxe de Stored Procedure .....	68
5.15.2	Criando uma Stored Procedure .....	69
5.15.3	Mostrando Erros .....	69
5.15.4	Executando uma Stored Procedure .....	69
5.15.5	Apagando uma Stored Procedure .....	69
5.15.6	Passando Argumentos .....	70

5.15.7	Metadados .....	70
5.16	<i>Funções</i> .....	70
5.16.1	Sintaxe de Função .....	70
5.16.2	Criando uma Função .....	71
5.16.3	Executando uma Função .....	71
5.16.4	Apagando uma Função.....	71
5.16.5	Metadados .....	71
5.17	<i>Cursores</i> .....	71
5.17.1	Criando um Cursor.....	72
5.17.2	Declaração de um Cursor.....	72
5.17.3	Abertura de um Cursor.....	72
5.17.4	Atributos de um Cursor.....	72
5.17.5	Acessando as Linhas de um Cursor .....	73
5.17.6	Fechando um Cursor .....	73
5.17.7	Exemplo sem argumento.....	73
5.17.8	Exemplo com argumento .....	74
5.17.9	Usando o FOR...LOOP .....	74
5.18	<i>Tabelas Temporárias</i> .....	75
5.18.1	Criando uma Tabela Temporária .....	75
5.19	<i>Exceções</i> .....	75
5.19.1	Exceções Definidas Internamente .....	76
5.19.1.1	Algumas Exceções Internas .....	76
5.19.1.2	Retornando Erros .....	76
5.19.1.3	Sintaxe .....	76
5.19.1.4	Exemplo de Exceção Interna.....	76
5.19.2	Exceções Definidas pelo Usuário.....	77
5.19.2.1	Sintaxe .....	77
5.19.3	Função Interna .....	77
5.19.3.1	Sintaxe .....	77
5.19.3.2	Exemplo .....	77
5.20	<i>Packages</i> .....	77
5.20.1	Estrutura de um Package.....	77
5.20.1.1	Seção de especificação.....	78
5.20.1.2	Package Body.....	78
5.20.2	Recompilando um Package .....	78
5.20.3	Apagando um Package.....	78
5.20.4	Exemplo .....	78
5.20.5	Referenciando um Subprograma do Package.....	79
5.21	<i>Gerenciamento de Usuários</i> .....	79
5.21.1	Privilégio.....	79
5.21.1.1	Privilégio de Sistema .....	79
5.21.1.2	Privilégio de Objeto .....	81
5.21.2	Papel.....	82
5.21.3	Criando Usuários .....	82
5.21.4	Alterando Usuários .....	83
5.21.5	Criando Papéis .....	83
5.21.6	Alterando Papéis .....	83
5.21.7	Concedendo Privilégios e Papéis a um Papel.....	83
5.21.8	Concedendo um Papel a um Usuário .....	84
5.21.9	Concedendo um Privilégio de Objeto para um Usuário.....	84
5.21.10	Visualizando os Papéis e Privilégios.....	85
5.21.10.1	Papéis e Privilégios de um Usuário .....	85
5.21.10.2	Papéis definidos no banco de dados .....	85
5.21.10.3	Privilégios de um Papel.....	85
5.21.10.4	Privilégios de tabela atribuídos a um Papel.....	86
5.21.11	Apagando um Usuário .....	86
5.21.12	Revogando um Privilégio de Sistema/Papel Concedido .....	86

5.21.13	Revogando um Privilégio de Objeto de um Usuário.....	87
5.21.14	Apagando um Papel.....	88
5.22	<i>Sinônimos</i> .....	88
5.22.1	Sintaxe de sinônimo.....	88
5.22.2	Criando um sinônimo.....	88
5.22.3	Renomeando um sinônimo.....	88
5.22.4	Apagando um sinônimo.....	88
5.22.5	Metadados.....	88
5.23	<i>Database Links</i> .....	89
5.23.1	Sintaxe básica database link.....	89
5.23.2	Criando um database link.....	89
5.23.3	Consultado os database link criados.....	89
5.23.4	Consultando tabelas via database link.....	90
5.23.5	Two Phase Commit.....	90
5.23.6	As 12 regras para sistemas distribuídos.....	90
5.24	<i>Agendamento de Tarefas</i> .....	91
5.24.1	Criando um Job.....	91
5.24.2	Listando os Jobs agendados.....	92
5.24.3	Listando os Jobs em execução.....	92
5.24.4	Removendo um Job.....	92
<b>6</b>	<b>Banco de Dados de Orientado a Objeto.....</b>	<b>93</b>
6.1	<i>Vantagens</i> .....	93
6.2	<i>Características</i> .....	93
6.3	<i>Banco de Dados Oracle</i> .....	94
6.3.1	Tipos de Objetos.....	94
6.3.2	Operações.....	95
6.3.3	Herança de Tipos.....	95
6.3.4	Tabelas.....	96
6.3.5	Tipo REF.....	96
6.3.6	Alterações de Tipos.....	96
6.3.7	Construtor.....	97
6.3.8	Atualização e Exclusão.....	98
6.3.9	Consulta.....	98
6.3.10	Constraints.....	99
<b>7</b>	<b>Bibliografia.....</b>	<b>100</b>
<b>8</b>	<b>Apêndice 1 – Oracle.....</b>	<b>101</b>
8.1	<i>Características</i> .....	101
8.1.1	Limites do Oracle.....	101
8.2	<i>Tabelas do Dicionário de Dados</i> .....	101
8.3	<i>Visões do Dicionário de Dados</i> .....	101
8.3.1	USER_.....	101
8.3.2	ALL_.....	101
8.3.3	DBA_.....	101
8.4	<i>Tabelas</i> .....	102
8.4.1	Metadados.....	102
8.4.2	Listando as tabelas que o usuário é dono.....	102
8.4.3	Listando as tabelas que o usuário tem acesso.....	102
8.4.4	Mostrando a Estrutura de uma Tabela.....	102
8.4.5	Mostrando todos os Dados de uma Tabela.....	102
8.4.6	Mostrando a Tabelas e o Atributo por tabela.....	102
8.4.7	Apagando Tabelas.....	102
8.4.8	Apagando Tabelas com Restrições.....	102

8.5	<i>Constraint (Integridade)</i> .....	102
8.5.1	Metadados.....	102
8.5.2	Listando os Nomes as Restrições.....	102
8.5.3	Listando os Nomes as Restrições e suas Tabelas.....	102
8.6	<i>Índices</i> .....	102
8.6.1	Metadados.....	102
8.6.2	Listando os índices criados.....	103
8.6.3	Listando as tabelas e seus índices.....	103
8.7	<i>Visões</i> .....	103
8.7.1	Visões Normais.....	103
8.7.1.1	Metadados.....	103
8.7.1.2	Listando as visões que o usuário é dono e tem acesso.....	103
8.7.2	Visões Materializadas.....	103
8.7.2.1	Metadados.....	103
8.7.2.2	Listando as visões materializadas que o usuário é dono e tem acesso.....	103
8.8	<i>Sequências</i> .....	103
8.8.1	Metadados.....	103
8.8.2	Listando as sequências criadas pelo usuário.....	103
8.9	<i>Triggers</i> .....	103
8.9.1	Metadados.....	103
8.9.2	Listando as Triggers que o usuário é dono.....	103
8.10	<i>Procedimentos, Funções e Packages</i> .....	103
8.10.1	Metadados.....	103
8.10.2	Mostrando os Argumentos de uma Function ou Procedure.....	104
8.10.3	Listando os Procedimentos criados(Procedures, Functions e Packages).....	104
8.10.4	Mostrando os erros no Procedimentos (Procedures, Functions e Packages).....	104
8.11	<i>Synônimos</i> .....	104
8.11.1	Metadados.....	104
8.11.2	Listando os Sinônimos que o usuário é dono.....	104
8.12	<i>Usuário</i> .....	104
8.12.1	Metadados.....	104
8.12.2	Mostrando o Usuário Conectado.....	104
8.12.3	Alterando a Senha do Usuário.....	104
8.12.4	Senhas e definidas no Oracle.....	104
8.12.5	Papéis Definidos no Oracle.....	104
8.12.6	Sessões de Usuário no Banco.....	105
8.13	<i>Linha de Comando SQL</i> .....	105
8.13.1	Conectando ao Banco de Dados.....	105
8.13.2	Desconectando do Banco de Dados.....	105
8.13.3	Saindo da Linha de Comando SQL.....	105
8.13.4	Limpando a tela.....	105
8.13.5	Modificando a linha de Exibição.....	105
8.13.6	Exibindo o tamanho da linha de Exibição.....	105
8.13.7	Modificando o tamanho de exibição de coluna.....	106
8.13.8	Habilitando a saída.....	106
8.13.9	Salvando Comandos em Arquivo.....	106
8.13.10	Carregando Comandos em Arquivo.....	106
8.13.11	Usando & para substituir variável.....	106
8.14	<i>TableSpace</i> .....	106
8.14.1	O que é o tablespace USERS?.....	109
8.14.2	O que é o tablespace SYSTEM?.....	109
8.14.3	O que é o tablespace TEMP?.....	109
8.14.4	O que é o tablespace UNDO?.....	109
8.14.5	O que é o tablespace SYSAUX?.....	110
8.15	<i>Gerenciamento de Espaço em Tablespaces</i> .....	110

8.16	<i>Propriedades do SGBD</i> .....	111
8.17	<i>Versão do SGBD</i> .....	112
<b>9</b>	<b>Apêndice 2 – Oracle PL/SQL</b> .....	<b>113</b>
9.1	<i>Conceitos</i> .....	113
9.2	<i>Vantagens</i> .....	113
9.2.1	Portabilidade .....	113
9.2.2	Integração com RDBMS .....	113
9.2.3	Capacidade Procedural .....	113
9.2.4	Produtividade .....	113
9.3	<i>Blocos</i> .....	113
9.3.1	Estrutura de Básica .....	113
9.3.2	Tipos de Blocos .....	114
9.4	<i>DataTypes</i> .....	114
9.4.1	Datatypes mais Utilizados .....	114
9.4.2	Declaração .....	114
9.4.3	O atributo %TYPE .....	114
9.4.4	O atributo %ROWTYPE .....	115
9.5	<i>Funções</i> .....	115
9.5.1	Funções Matemáticas .....	116
9.5.2	Funções de Data .....	117
9.5.2.1	Recuperando o nome dos clientes que nascem em um determinado dia .....	117
9.5.2.2	Recuperando o nome dos clientes que nascem em um determinado mês. ....	117
9.5.2.3	Recuperando o nome dos clientes que nascem em um determinado ano. ....	117
9.5.3	Funções de conversão .....	117
9.5.3.1	Conversão Implícita .....	118
9.5.3.2	Conversão Explícita .....	118
9.5.4	Funções de Caracteres .....	119
9.5.4.1	Consultar um literal que contenha espaços e somente em maiúsculo. ....	120
9.5.5	Funções Diversas .....	120
9.5.6	Funções de Erro do SQL Oracle .....	121
9.6	<i>Estruturas de Controle</i> .....	121
9.6.1	Comando IF...THEN .....	121
9.6.2	Comando CASE..WHEN .....	122
9.6.3	Comando LOOP .....	123
9.6.4	Comando FOR...LOOP .....	123
9.6.5	Comando WHILE .....	124
9.7	<i>Operadores</i> .....	124
9.7.1	Operador CASE .....	124
9.7.2	Operador CAST .....	125
<b>10</b>	<b>Apêndice 3 - Exemplo de um Banco de Dados</b> .....	<b>126</b>



# 1 Introdução

A tecnologia aplicada aos métodos de armazenamento de informações vem crescendo e gerando um impacto cada vez maior no uso de computadores, em qualquer área em que os mesmos podem ser aplicados.

Um “banco de dados” pode ser definido como um conjunto de “dados” devidamente relacionados. Por “dados” podemos compreender como “fatos conhecidos” que podem ser armazenados e que possuem um significado implícito. Porém, o significado do termo “banco de dados” é mais restrito que simplesmente a definição dada acima. Um banco de dados possui as seguintes propriedades:

- um banco de dados é uma coleção lógica coerente de dados com um significado inerente; uma disposição desordenada dos dados não pode ser referenciada como um banco de dados;
- um banco de dados é projetado, construído e populado com dados para um propósito específico; um banco de dados possui um conjunto pré definido de usuários e aplicações;
- um banco de dados representa algum aspecto do mundo real, o qual é chamado de “mini-mundo” ; qualquer alteração efetuada no mini-mundo é automaticamente refletida no banco de dados.

Um banco de dados pode ser criado e mantido por um conjunto de aplicações desenvolvidas especialmente para esta tarefa ou por um “Sistema Gerenciador de Banco de Dados” (SGBD). Um SGBD permite aos usuários criarem e manipularem bancos de dados de propósito geral. O objetivo principal do SGBD é proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação e armazenamento das informações no banco de dados.

O conjunto formado por um banco de dados mais as aplicações que manipulam o mesmo é chamado de “Sistema de Banco de Dados”.

Os banco de dados são amplamente usados:

- Banco: para informações de clientes, contas, empréstimos e todas as transações bancárias.
- Linhas aéreas: para reservas, e informações de horários. As linhas aéreas foram umas das primeiras a usar bancos de dados de maneira geograficamente distribuída.
- Universidades: para informações de alunos, registros de cursos e notas.
- Transações de cartão de crédito: para compras com cartões de crédito e geração de faturas mensais.
- Telecomunicação: para manter registros de chamadas realizadas, gerar cobranças mensais, manter saldos de cartões de chamada pré-pagos e armazenar informações sobre as redes de comunicações.
- Finanças: para armazenar informações sobre valores mobiliários, vendas e compras de instrumentos financeiros como ações e títulos; também para armazenar dados de mercado em tempo real a fim de permitir negócios on-line por clientes e transações automatizadas pela empresa.
- Vendas: para informações de clientes, produtos, compra.
- Revendedores on-line: para dados de vendas descritos aqui, além de acompanhamento de pedidos, geração de lista de recomendações personalizadas e manutenção de avaliações de produto-online.
- Indústria: para gerenciamento da cadeia de suprimento e para controlar a produção de itens nas fábricas, estoques de itens em armazéns e lojas, além de itens.
- Recursos humanos: para informações sobre funcionários, salários, descontos em folha de pagamento, benefícios e para geração de contra-cheques.

## 1.1 *História dos Sistemas de Banco de Dados*

O processamento de dados tem impulsionado o crescimento dos computadores desde os primeiros dias dos computadores comerciais. Na verdade, a automação das tarefas de processamento de dados já existia antes mesmo dos computadores. Cartões perfurados, inventados por Herman Hollerith, foram usados no início do século XX para registrar dados do censo dos Estados Unidos, e sistemas mecânicos foram usados para processar os cartões perfurados e tabular os resultados. Mais tarde, os cartões perfurados passaram a ser amplamente usados como um meio de inserir dados em computadores.

As técnicas de armazenamento e processamento de dados evoluíram ao longo dos anos.

- **Década de 1950 e início da década de 1960:**
  - Processamento de dados usando fitas magnéticas para armazenamento
    - Fitas fornecem apenas acesso sequencial
  - Cartões perfurados para entrada
- **Final da década de 1960 e década de 1970:**
  - Discos rígidos permitem acesso direto aos dados
  - Modelos de dados de rede e hierárquico em largo uso
  - Ted Codd define o modelo de dados relacional
    - Ganharia o ACM Turing Award por este trabalho
    - IBM Research inicia o protótipo do System
    - UC Berkeley inicia o protótipo do Ingres
  - Processamento de transação de alto desempenho (para a época)
- **Década de 1980:**
  - Protótipos relacionais de pesquisa evoluem para sistemas comerciais
    - SQL se torna o padrão do setor
  - Sistemas de banco de dados paralelos e distribuídos
  - Sistemas de banco de dados orientados a objeto
- **Década de 1990:**
  - Grandes aplicações de suporte a decisão e exploração de dados
  - Grandes data warehouses de vários terabytes
  - Surgimento do comércio Web
- **Década de 2000:**
  - Padrões XML e XQuery
  - Administração de banco de dados automatizada

## 1.2 *Linguagem de Banco de Dados*

Um sistema de banco de dados proporciona dois tipos de linguagens: uma específica para os esquemas do banco de dados e outra para expressar consultas e atualizações

### 1.2.1 Linguagem de Definição de Dados

Para a definição dos esquemas lógico ou físico pode-se utilizar uma linguagem chamada DDL (Data Definition Language - Linguagem de Definição de Dados). O SGBD possui um compilador DDL que permite a execução das declarações para identificar as descrições dos esquemas e para armazená-las em tabelas que constituem um arquivo especial chamado dicionário de dados ou diretório de dados.

Um dicionário de dados é um arquivo de metadados – isto é, dados a respeito de dados. Em sistema de banco de dados, esse arquivo ou diretório é consultado antes que os dados reais seja modificados.

Em um SGBD em que a separação entre os níveis lógico e físico são bem claras, é utilizado uma outra linguagem, a SDL (Storage Definition Language - Linguagem de Definição de Armazenamento) para a especificação do nível físico. A especificação do esquema conceitual fica por conta da DDL.

Em um SGBD que utiliza a arquitetura três esquemas, são necessários a utilização de mais uma linguagem para a definição de visões, a VDL (Vision Definition Language - Linguagem de Definição de Visões).

### 1.2.2 Linguagem de Manipulação de Dados

Uma vez que o esquema esteja compilado e o banco de dados esteja populado, usa-se uma linguagem para fazer a manipulação dos dados, a DML (Data Manipulation Language - Linguagem de Manipulação de Dados).

Por manipulação entendemos:

A recuperação das informações armazenadas no banco de dados;

Inserção de novas informações no banco de dados;

A remoção das informações no banco de dados;

A modificação das informações no banco de dados.

No nível físico, precisamos definir algoritmos que permitam os acessos eficientes aos dados. Nos níveis mais altos de abstração, enfatizamos a facilidade de uso. O Objeto é proporcionar uma interação eficiente entre homens e sistema.

### 1.2.3 Outras Linguagens

#### 1.2.3.1 Query-by-Example(QBE)

Query-by-Example é o nome tanto da linguagem de manipulação como do sistema de banco de dados que a contém. O sistema de banco de dados QBE foi desenvolvido pela IBM, no Centro de Pesquisa T.J. Watson, no início de 1970. A linguagem de manipulação de dados QBE foi usada mais tarde na Query Management Facility(QMF, da IBM).

Consulta são expressa “*por exemplo*”. Em vez de determinar um procedimento para obtenção da resposta desejada, o usuário dá um exemplo do que é desejado. O sistema generaliza o exemplo para o processamento da resposta da consulta.

#### 1.2.3.2 Quel

A Quel foi lançada como linguagem de consulta para o sistema de banco de dados Ingres desenvolvido na Universidade da Califórnia, Berkeley. A estrutura básica está restritamente ligada ao cálculo relacional das tuplas. A maioria das consultas é expressa através de três tipos de cláusulas: range of, retrieve e where.

Exemplo de quel:

```
RANGE OF (d,e) IS (Departamento, Empregado)
RETRIEVE INTO nome(d.nome)
WHERE (d.cod_dep=e.cod_dep)
```

#### 1.2.3.3 DataLog

A DataLog é linguagem de consulta não procedural baseada na lógica de programação da linguagem Prolog. Como ela faz no cálculo relacional o usuário descreve a informação desejada sem fornecer um procedimento específico para obtenção dessa informação. A sintaxe da Datalog remonta à da Prolog. Entretanto, o sentido de um programa Datalog é definido de uma maneira puramente declarative, sem a semântica mais procedural da Prolog, assim a Datalog é mais simples, permitindo a escrita de consultas mais fáceis e tornando mais simples sua otimização.

### 1.3 Os Módulos Componentes de um SGBD

Um sistema de banco de dados é particionado em módulos que lidam com cada uma das responsabilidades do sistema geral. Os componentes funcionais de um sistema de banco de dados podem ser divididos, de modo amplo, nos componentes do gerenciador de armazenamento e processador de consultas.

O gerenciador de armazenamento é importante porque os bancos de dados normalmente exigem uma grande quantidade de espaço de armazenamento. Os bancos de dados corporativos variam de centenas de gigabytes a – para os maiores banco de dados – terabytes de dados. Um gigabyte possui 1000 megabytes (ou 1 bilhão de bytes) e um terabyte possui 1 milhão de megabytes (ou 1 trilhão de bytes). Como a memória principal dos computadores não pode armazenar tanta quantidade de dados, as informações são armazenadas em discos. Os dados são movidos entre o armazenamento em disco e a memória principal conforme o necessário. Uma vez que o movimento de dados para o disco é lento, em comparação à velocidade da CPU, é fundamental que o sistema de banco de dados estruture os dados de modo a minimizar a necessidade de mover dados entre disco e memória principal.

O processador de consulta é importante porque ajuda o sistema de banco de dados a simplificar o acesso aos dados. As visões de alto nível ajudam a alcançar esses objetivos; com ela, os usuários do sistema não são desnecessariamente afligidos com os detalhes físicos da implementação do sistema. Entretanto, o rápido processamento das atualizações e consultas é importante. É função do sistema de banco de dados é traduzir atualizações e consultas escritas em uma linguagem não procedural, no nível lógico, em uma seqüência eficiente de operações no nível físico.

#### 1.3.1 Gerenciador de Armazenamento

Um gerenciador de Armazenamento é um módulo de programa que fornece a interface entre os dados de baixo nível armazenados no banco de dados e os programas de aplicação e consultas submetidos ao sistema. O gerenciador de armazenamento é responsável pela interação com o gerenciador de arquivos. Os dados brutos são armazenados no disco usando o sistema de arquivos, que normalmente é fornecido por um sistema operacional convencional. O gerenciador de armazenamento traduz as várias instruções DML em comandos de sistema de arquivo de baixo nível. Portanto, o gerenciador de armazenamento é responsável por armazenar, recuperar e atualizar dados no banco de dados.

Os componentes do gerenciador de armazenamento incluem:

- **Gerenciador de autorização e integridade**, que testa a satisfação das restrições de integridade e verifica a autoridade dos usuários para acessar dados.
- **Gerenciador de transações**, que garante que o banco de dados permaneça em estado consistente (correto) a despeito de falhas no sistema e que transações concorrentes serão executadas sem conflitos em seus procedimentos.
- **Gerenciador de arquivos**, que controla a alocação de espaço no armazenamento de disco e as estruturas de dados usadas para representar informações armazenadas no disco.
- **Gerenciador de buffer**, responsável por buscar dados do armazenamento de disco para a memória principal e decidir que dados colocar em **cachê** na memória principal. O gerenciador de buffer é uma parte crítica do sistema de banco de dados, já que permite que o banco de dados manipule tamanhos de dados que sejam muito maiores do que o tamanho da memória.

O gerenciador de armazenamento implementa estruturas de dados como parte da implementação do sistema físico:

- **Arquivo de Dados**, que armazena o próprio banco de dados.
- **Dicionário de Dados**, que armazena os **metadados** relativos à estrutura do banco de dados, em especial o esquema de banco de dados.

- **Índices**, que proporcionam acesso rápido aos itens de dados que são associados a valores determinados.
- **Estatísticas de dados** armazenam informações estatísticas relativas aos dados contidos no banco de dados.

### 1.3.2 Processador de Consultas

Os componentes do processador de consultas incluem:

- **Interpretador DML**, que interpreta os comandos DDL e registra as definições no dicionário de dados.
- **Compilador de DML**, que traduz instruções DML em uma linguagem de consulta para um plano de avaliação consistindo em instruções de baixo nível que o mecanismo de avaliação de consulta entende. Uma consulta normalmente pode ser traduzida em qualquer um de vários planos de avaliação que produzem todos os mesmo resultados. O compilador de DML também realiza otimização de consulta; ou seja, ele seleciona o plano de avaliação de menor custo dentre as alternativas.
- **Mecanismo de avaliação de consulta**, que executa instruções de baixo nível geradas pelo compilador de DML.

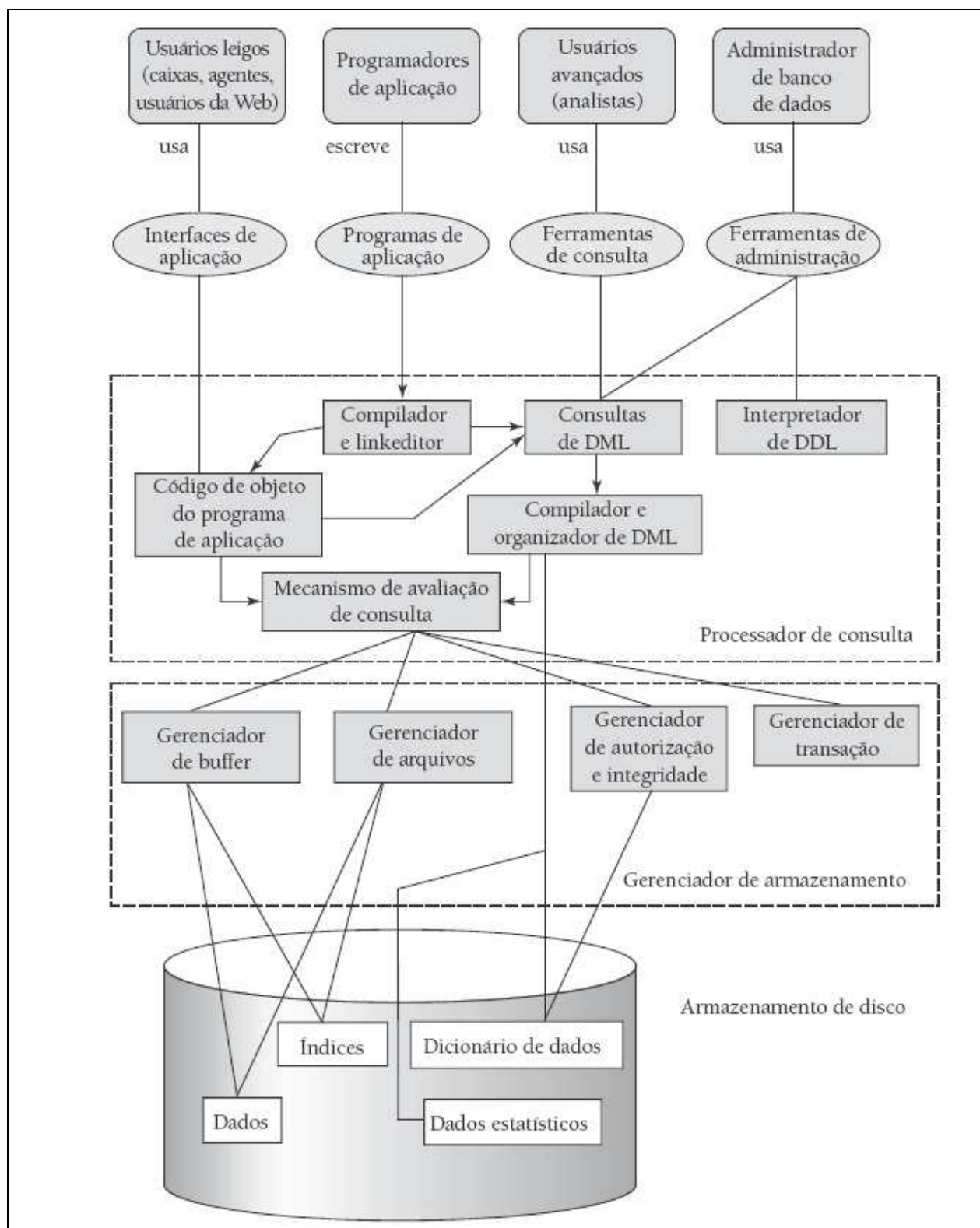
### 1.3.3 Gerenciamento de Transação

Muitas vezes, várias operações no banco de dados forma uma única unidade lógica de trabalho. Um exemplo é uma transferência de fundos, em que uma conta (digamos A) é debitada e outra conta (digamos B) é creditada. Obviamente, é fundamental que tanto o débito quanto o crédito ocorram, ou então que nenhuma ocorra. Ou seja, a transferência de fundos precisa acontecer em sua totalidade ou não acontecer. Essa propriedade “tudo ou nada” é chamada **atomicidade**. Além disso, é essencial que a execução da transferência de fundos preserve a consistência do banco de dados. Isto é, o valor da soma de A+B precisa ser preservado. Essa propriedade de exatidão é chamada consistência. Finalmente, após a execução bem-sucedida de uma transferência de fundos, os novos valores das contas A e B precisa persistir, apesar da possibilidade de falha do sistema. Essa propriedade de persistência é chamada de **durabilidade**.

Uma transação é um conjunto de operações que realiza uma única função lógica em uma aplicação de banco de dados. Cada transação é uma unidade da atomicidade e da consistência.

Garantir as propriedades da atomicidade e da durabilidade é função do próprio sistema de banco de dados – especificamente, do componente de gerenciamento de transação.

**Figura 1 - Estrutura de um Sistema Gerenciador de Banco de Dados**



## 2 Álgebra Relacional

A **álgebra relacional** é uma coleção de operações canônicas<sup>1</sup> que são utilizadas para manipular as relações. Estas operações são utilizadas para selecionar tuplas de relações individuais e para combinar tuplas relacionadas de relações diferentes para especificar uma consulta em um determinado banco de dados. O resultado de cada operação é uma nova relação, a qual também pode ser manipulada pela álgebra relacional.

Todos os exemplos envolvendo álgebra relacional implicam na utilização do banco de dados descrito no Apêndice 3.

### 2.1 A Operação Select

A operação **select** é utilizada para selecionar um subconjunto de tuplas de uma relação, sendo que estas tuplas devem satisfazer uma **condição de seleção**. A forma geral de uma operação **select** é:

$\sigma_{\langle \text{condição de seleção} \rangle}(\langle \text{nome da relação} \rangle)$

A letra grega  $\sigma$  (sigma minúscula) é utilizada para representar a operação de seleção; **<condição de seleção>** é uma expressão *booleana* aplicada sobre os atributos da relação e **<nome da relação>** é o nome da relação sobre a qual será aplicada a operação **select**.

Levando em consideração a consulta a seguir:

Encontre os dados dos empregados que ganham menos que \$2.500,00.

**consulta1**  $\leftarrow \sigma_{\text{Salario} < 2.500,00}(\text{EMPREGADO})$

gera a seguinte relação como resultado:

Relação <b>consulta1</b>					
<u>EmpregadoId</u>	Nome	CPF	DeptoId	SupervisorId	Salario
30303030	Ricardo	33333333	2	10101010	2.300,00
50505050	Renato	55555555	3	20202020	1.300,00

Levando em consideração a consulta a seguir:

Encontre os dados dos dependentes que a relação com o empregado seja filho e do sexo feminino.

**consulta2**  $\leftarrow \sigma_{(\text{Relacao} = \text{"Filho"}) \text{ .and. } (\text{sexo} = \text{"Feminino"})}(\text{DEPENDENTE})$

gera a seguinte relação como resultado:

Relação <b>consulta2</b>				
<u>EmpregadoId</u>	<u>Nome</u>	<u>DtNascimento</u>	Relacao	Sexo
30303030	Andreia	01/05/90	Filho	Feminino

As operações relacionais que podem ser aplicadas na operação **select** são:

$<, >, \leq, \geq, =, \neq$  ou  $<, >, <=, >=, =, <>$

além dos operadores booleanos:

and, or, not ou .e., .ou., .não. ou ainda  $\wedge, \vee$  e  $\neg$ .

<sup>1</sup> Canônico, substantivo masculino, relativo a canônes; conforme aos canônes; direito; código da igreja romana, horas canônicas; coleção de orações para diferentes horas do dia.

A operação **select** é unária, ou seja, só pode ser aplicada a uma única relação. Não é possível aplicar a operação sobre tuplas de relações distintas.

## 2.2 A Operação Project

A operação **project** seleciona um conjunto determinado de colunas de uma relação. A forma geral de uma operação **project** é:

$$\pi_{\langle \text{lista de atributos} \rangle} (\langle \text{nome da relação} \rangle)$$

A letra grega  $\pi$  (pi minúscula) representa a operação **project**,  $\langle \text{lista de atributos} \rangle$  representa a lista de atributos que o usuário deseja selecionar e  $\langle \text{nome da relação} \rangle$  representa a relação sobre a qual a operação **project** será aplicada.

Levando em consideração a consulta a seguir:

Mostre o nome e a data de nascimento de todos os dependentes.

**consulta3**  $\leftarrow \pi_{\text{Nome, DtNascimento}} (\text{DEPENDENTE})$

gera a seguinte relação como resultado:

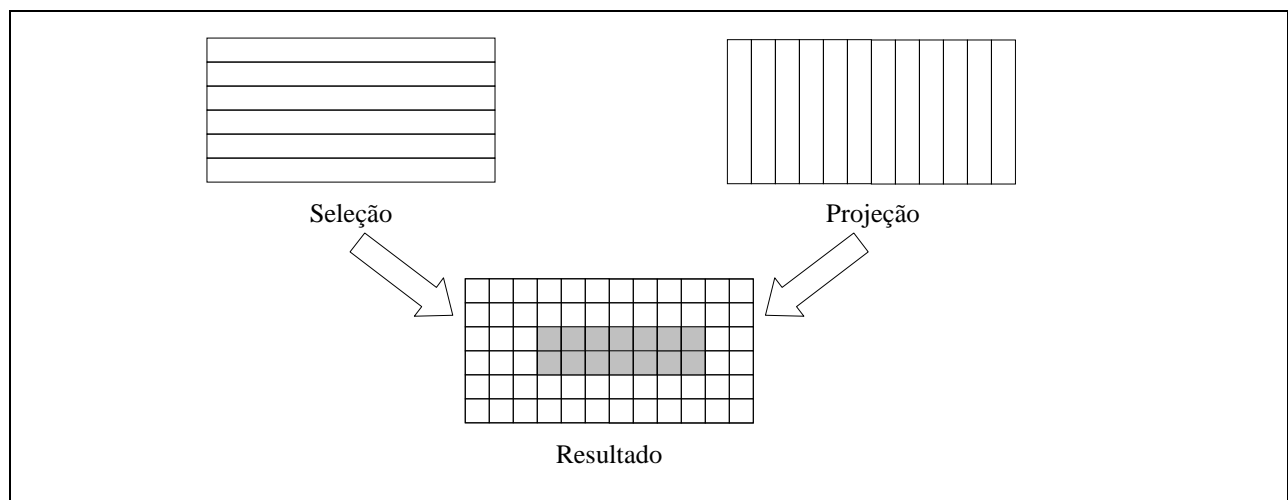
Relação <b>consulta3</b>	
Nome	DtNascimento
Jorge	27/12/86
Luiz	18/11/79
Fernanda	14/02/69
Ângelo	10/02/95
Adreia	01/05/90

Outra característica da projeção é que ela elimina tuplas **repetidas**.

## 2.3 Seqüencialidade de Operações

As operações **project** e **select** podem ser utilizadas de forma combinada, permitindo que apenas determinadas colunas de determinadas tuplas possam ser selecionadas.

**Figura 2 - Resultado da Seqüencialidade**





A forma geral de uma operação sequencializada é:

$$\pi_{\langle \text{lista de atributos} \rangle} (\sigma_{\langle \text{condição de seleção} \rangle} (\langle \text{nome da relação} \rangle))$$

Levando em consideração a consulta a seguir:

Encontre o nome, deptoId e salário dos empregados que ganham menos que \$2.500,00.

$$\text{consulta4} \leftarrow \pi_{\text{Nome, DeptoId, Salario}} (\sigma_{\text{Salario} < 2.500,00} (\text{EMPREGADO}))$$

produz a relação a seguir como resultado:

Relação <b>consulta4</b>		
Nome	DeptoId	Salario
Ricardo	2	2.300,00
Renato	3	1.300,00

A **consulta4** pode ser reescrita da seguinte forma:

$$\text{consulta5} \leftarrow \sigma_{\text{Salario} < 2.500,00} (\text{EMPREGADO})$$

Relação <b>consulta5</b>					
<u>EmpregadoId</u>	Nome	CPF	DeptoId	SupervisorId	Salario
30303030	Ricardo	33333333	2	10101010	2.300,00
50505050	Renato	55555555	3	20202020	1.300,00

$$\text{consulta6} \leftarrow \pi_{\text{Nome, DeptoId, Salario}} (\text{CONSULTA5})$$

Relação <b>consulta6</b>		
Nome	DeptoId	Salario
Ricardo	2	2.300,00
Renato	3	1.300,00

porém é mais elegante utilizar a forma descrita na **consulta4**.

## 2.4 Operações Matemáticas

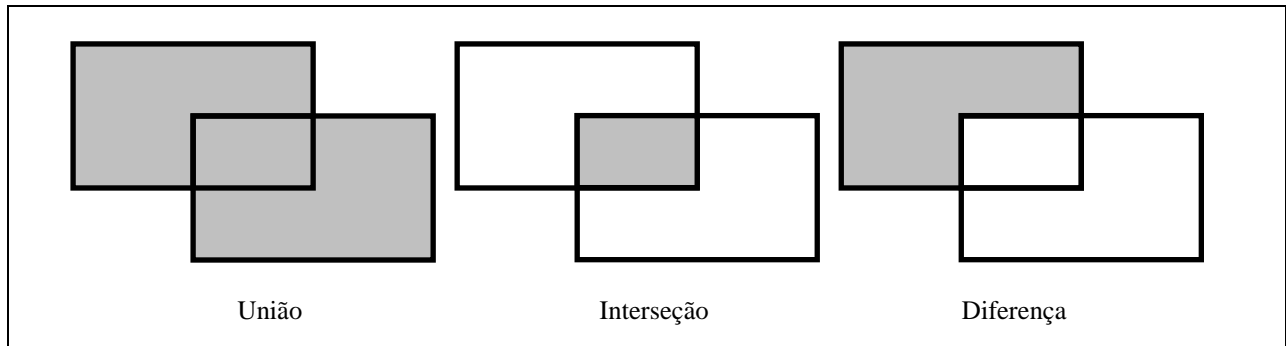
Levando em consideração que as relações podem ser tratadas como conjuntos, podemos então aplicar um conjunto de operações matemáticas sobre as mesmas. Estas operações são: **união** ( $\cup$ ), **intersecção** ( $\cap$ ) e **diferença** ( $-$ ). Este conjunto de operações não é unário, ou seja, podem ser aplicadas sobre mais de uma tabela, porém, existe a necessidade das tabelas possuírem tuplas exatamente do mesmo tipo.

Estas operações podem ser definidas da seguinte forma:

- **união** - o resultado desta operação representada por  $\mathbf{R} \cup \mathbf{S}$  é uma relação **T** que inclui todas as tuplas que se encontram em **R** e todas as tuplas que se encontram em **S**;
- **intersecção** - o resultado desta operação representada por  $\mathbf{R} \cap \mathbf{S}$  é uma relação **T** que inclui as tuplas que se encontram em **R** e em **S** ao mesmo tempo;
- **diferença** - o resultado desta operação representada por  $\mathbf{R} - \mathbf{S}$  é uma relação **T** que inclui todas as tuplas que estão em **R** mas não estão em **S**.

A Figura 3 demonstra as operações segundo os diagramas de Venn-Euler<sup>2</sup>.

**Figura 3 – Diagramas de Venn Euler**



Leve em consideração a seguinte consulta:

*Selecione o id dos empregados que trabalham no departamento número 2 ou que supervisionam empregados que trabalham no departamento número 2.*

Vamos primeiro selecionar todos os funcionários que trabalham no departamento número 2.

**consulta7**  $\leftarrow \sigma_{\text{DeptId} = 2}(\text{EMPREGADO})$

Relação <b>consulta7</b>					
<u>EmpregadoId</u>	Nome	CPF	DeptId	SupervisorId	Salário
20202020	Fernando	22222222	2	10101010	2.500,00
30303030	Ricardo	33333333	2	10101010	2.300,00
40404040	Jorge	44444444	2	20202020	4.200,00

Vamos agora selecionar os id dos supervisores dos empregados que trabalham no departamento número 2.

**consulta8**  $\leftarrow \pi_{\text{SupervisorId}}(\text{CONSULTA7})$

Relação <b>consulta8</b>
SupervisorId
10101010
20202020

Vamos projetar apenas o id dos empregados selecionados:

**consulta9**  $\leftarrow \pi_{\text{EmpregadoId}}(\text{CONSULTA7})$

Relação <b>consulta9</b>
<u>EmpregadoId</u>
20202020
30303030
40404040

<sup>2</sup> Criados em 1881 pelo filósofo e matemático britânico John Venn. Pronuncia-se 'oiler'.

E por fim, vamos unir as duas relações, obtendo o resultado final.

**consulta10**  $\leftarrow$  CONSULTA8  $\cup$  CONSULTA9

Relação <b>consulta10</b>
<u>EmpregadoId</u>
20202020
30303030
40404040
10101010

Leve em consideração a próxima consulta:

*Selecione o id dos empregados que desenvolvem algum projeto e que trabalham no departamento número 2;*

Vamos primeiro selecionar todos os empregados que trabalham em um projeto.

**consulta11**  $\leftarrow \pi_{\text{EmpregadoId}}(\text{EMP\_PROJ})$

Relação <b>consulta11</b>
<u>EmpregadoId</u>
20202020
30303030
40404040
50505050

Vamos agora selecionar todos os empregados que trabalham no departamento 2.

**consulta12**  $\leftarrow \pi_{\text{EmpregadoId}}(\sigma_{\text{DepptoId} = 2}(\text{EMPREGADO}))$

Relação <b>consulta12</b>
<u>EmpregadoId</u>
20202020
30303030
40404040

Obtemos então todo o id dos empregados que trabalham no departamento 2 e que desenvolvem algum projeto.

**consulta13**  $\leftarrow$  CONSULTA11  $\cap$  CONSULTA12

Relação <b>consulta13</b>
<u>EmpregadoId</u>
20202020
30303030
40404040

Leve em consideração a seguinte consulta:

*Selecione o id dos empregados que não desenvolvem projetos;*

**consulta14**  $\leftarrow \pi_{\text{EmpregadoId}}(\text{EMP\_PROJ})$

Relação <b>consulta14</b>
<u>EmpregadoId</u>
20202020
30303030
40404040
50505050

**consulta15**  $\leftarrow \pi_{\text{EmpregadoId}}(\text{EMPREGADO})$

Relação <b>consulta15</b>
<u>EmpregadoId</u>
10101010
20202020
30303030
40404040
50505050

**consulta16**  $\leftarrow \text{CONSULTA15} - \text{CONSULTA14}$

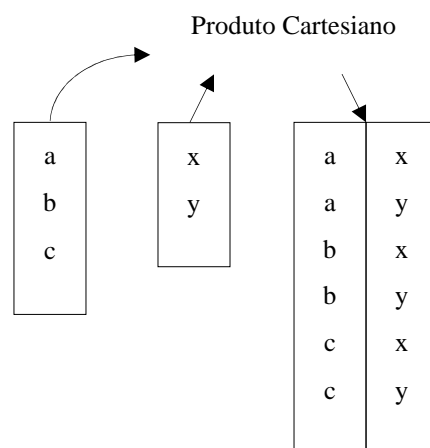
Relação <b>consulta16</b>
<u>EmpregadoId</u>
10101010

## 2.5 Produto Cartesiano

O **produto cartesiano** é uma operação binária que combina todas as tuplas de duas relações. Diferente da operação **união**, o **produto cartesiano** não exige que as tuplas das relações possuam exatamente o mesmo tipo. O **produto cartesiano** permite então a consulta entre relações relacionadas utilizando uma condição de seleção apropriada. O resultado de um **produto cartesiano** é uma nova relação formada pela combinação das tuplas das relações sobre as quais se aplicou a operação.

O formato geral do **produto cartesiano** entre duas relações **R** e **S** é:

**R X S**



Leve em consideração a seguinte consulta:

Encontre os nomes dos empregados e o nome do departamento onde trabalha.

**consulta17**  $\leftarrow$  EMPREGADO X DEPTO

Relação consulta17								
EmpregadoId	Nome	CPF	Empregado. DeptId	SupervisorId	Salario	Depto. DeptId	Depto. Nome	GerenteId
10101010	João Luiz	11111111	1	NULO	3.000,00	1	Contabil	10101010
10101010	João Luiz	11111111	1	NULO	3.000,00	2	Civil	30303030
10101010	João Luiz	11111111	1	NULO	3.000,00	3	Mecânica	20202020
20202020	Fernando	22222222	2	10101010	2.500,00	1	Contabil	10101010
20202020	Fernando	22222222	2	10101010	2.500,00	2	Civil	30303030
20202020	Fernando	22222222	2	10101010	2.500,00	3	Mecânica	20202020
30303030	Ricardo	33333333	2	10101010	2.300,00	1	Contabil	10101010
30303030	Ricardo	33333333	2	10101010	2.300,00	2	Civil	30303030
30303030	Ricardo	33333333	2	10101010	2.300,00	3	Mecânica	20202020
40404040	Jorge	44444444	2	20202020	4.200,00	1	Contabil	10101010
40404040	Jorge	44444444	2	20202020	4.200,00	2	Civil	30303030
40404040	Jorge	44444444	2	20202020	4.200,00	3	Mecânica	20202020
50505050	Renato	55555555	3	20202020	1.300,00	1	Contabil	10101010
50505050	Renato	55555555	3	20202020	1.300,00	2	Civil	30303030
50505050	Renato	55555555	3	20202020	1.300,00	3	Mecânica	20202020

Vamos agora selecionar as tuplas resultantes que estão devidamente relacionadas que são as que possuem o mesmo valor em Empregado.DeptoId e Depto.DeptoId e projetar o nome do empregado e o nome do departamento.

**consulta18**  $\leftarrow \pi_{\text{Empregado.nome, Depto.Nome}} \left( \sigma_{(\text{Empregado.DeptoId} = \text{Depto.DeptoId})} (\text{CONSULTA17}) \right)$

Relação consulta18	
Nome	Depto.Nome
João Luiz	Contabil
Fernando	Civil
Ricardo	Civil
Jorge	Civil
Renato	Mecânica

Outro exemplo:

*Encontre os id dos empregados que desenvolvem projetos em Campina.;*

**consulta19**  $\leftarrow$  EMP\_PROJ X PROJETO

Relação <b>consulta19</b>					
EmpregadoId	Emp_Proj.ProjetoId	Hora	Projeto.ProjetoId	Nome	Localizacao
20202020	5	10	5	Financeiro 1	São Paulo
20202020	5	10	10	Motor 3	Rio Claro
20202020	5	10	20	Prédio Central	Campinas
20202020	10	25	5	Financeiro 1	São Paulo
20202020	10	25	10	Motor 3	Rio Claro
20202020	10	25	20	Prédio Central	Campinas
30303030	5	35	5	Financeiro 1	São Paulo
30303030	5	35	10	Motor 3	Rio Claro
30303030	5	35	20	Prédio Central	Campinas
40404040	20	50	5	Financeiro 1	São Paulo
40404040	20	50	10	Motor 3	Rio Claro
40404040	20	50	20	Prédio Central	Campinas
50505050	20	35	5	Financeiro 1	São Paulo
50505050	20	35	10	Motor 3	Rio Claro
50505050	20	35	20	Prédio Central	Campinas

Vamos agora selecionar as tuplas resultantes que estão devidamente relacionadas que são as que possuem o mesmo valor em Projeto.ProjetoId e Emp\_Proj.ProjetoId e cuja localização seja 'Campinas'.

**consulta20**  $\leftarrow \pi_{\text{EmpregadoId}} (\sigma_{(\text{Projeto.ProjetoId} = \text{Emp\_Proj.ProjetoId}) \text{ and } (\text{localizacao} = \text{'Campinas'})} (\text{CONSULTA19}))$

Relação <b>consulta20</b>
<u>EmpregadoId</u>
40404040
50505050

## 2.6 Operação Rename

A operação rename é uma operação unária e permite dar um novo nome a uma relação, aos atributos ou ambos. É representado pela letra grega  $\rho$  (rô minúscula).

O formato geral da **operação rename** para renomear uma relação é:

$\rho_{\langle \text{novo nome da relação} \rangle (\text{A1}, \text{A2}, \dots, \text{An})} (\langle \text{nome da relação} \rangle)$

ou

$\rho_{\langle \text{novo nome da relação} \rangle} (\langle \text{nome da relação} \rangle)$

ou

$\rho_{(\text{B1}, \text{B2}, \dots, \text{Bn})} (\langle \text{nome da relação} \rangle)$

A primeira expressão renomeia tanto a relação quanto seus atributos, a segunda renomeia somente a relação e a terceira renomeia somente os atributos. Se os atributos da relação forem (A1, A2, ... Na) nesta ordem, então cada Ai é renomeado como Bi.

Leve em consideração a consulta a seguir:

*Encontre o nome dos empregados e o nome do seu respectivo supervisor;*

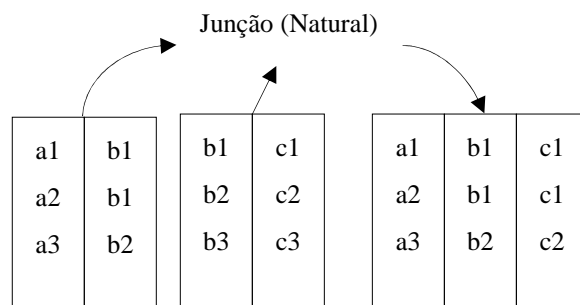
**consulta21**  $\leftarrow \pi_{\text{emp.Nome, sup.Nome}} (\sigma_{(\text{emp.EmpregadoId}=\text{sup.SupervisorId})} (\rho_{\text{emp}}(\text{Empregado}) \times \rho_{\text{sup}}(\text{Empregado}))$

Relação <b>Consulta21</b>	
emp.Nome	sup.Nome
Fernando	João Luiz
Ricardo	João Luiz
Jorge	Fernando
Renato	Fernando

## 2.7 Junção Natural

Freqüentemente é desejável simplificar certas consultas que exijam um produto cartesiano. Tipicamente, uma consulta em um produto cartesiano envolve uma seleção de operações sobre o resultado do produto cartesiano. A junção natural (*Natural Join*) é uma operação binária que nos permite combinar certas relações em um produto cartesiano dentro de uma operação. Isto é representado pelo símbolo de “Join”  $\bowtie$ . As operações de junção natural formam um produto cartesiano de seus dois argumentos, promovem uma seleção obedecendo à equivalência dos atributos que aparecem em ambas as relações e, finalmente removem os atributos em duplicidade. Devido ao critério de igualdade é também chamada de **junção interna** (*Inner Join*). A forma geral da operação **junção natural** entre duas relações **R** e **S** é a seguinte:

**R**  $\bowtie$  **S**



Que equivale a

$\pi_{R \cup S} (\sigma_{(R.A1 = S.A1 \text{ e. } R.A2 = S.A2 \dots R.AN = S.AN)} (R \times S))$

Se os atributos das relações não forem equivalentes é necessário especificar que atributos devem ser comparados através de uma condição. Neste caso dizemos que é apenas **junção**. A forma geral da operação **junção** onde os atributos não são equivalentes entre duas tabelas **R** e **S** é a seguinte:

**R**  $\bowtie_{\langle \text{condição de junção} \rangle}$  **S**

Leve em consideração a consulta a seguir:

*Encontre todos os empregados que desenvolvem projetos em Campinas;*

**consulta22**  $\leftarrow \text{EMP\_PROJ} \bowtie_{\text{Emp\_Proj.ProjetoId} = \text{Projeto.ProjetoId}} \text{PROJETO}$

Relação consulta22				
<u>EmpregadoId</u>	<u>ProjetoId</u>	Nome	Número	Localizacao
20202020	5	Financeiro 1	5	São Paulo
20202020	10	Motor 3	10	Rio Claro
30303030	5	Financeiro 1	5	São Paulo
40404040	20	Prédio Central	20	Campinas
50505050	20	Prédio Central	20	Campinas




**Consulta23**  $\leftarrow \sigma_{\text{localização} = \text{'Campinas'}}$  (CONSULTA22)

Relação consulta23				
<u>EmpregadoId</u>	<u>ProjetoId</u>	Nome	Número	Localizacao
40404040	20	Prédio Central	20	Campinas
50505050	20	Prédio Central	20	Campinas

## 2.8 Junção Externa

A operação de junção externa é uma extensão da operação de junção para tratar informações omitidas.

Além de trazer as tuplas que possuem um correspondente a junção externa também inclui no resultado tuplas que não possuem uma correspondente na outra relação. Estas tuplas que não possuem relação podem estar tanto na relação da direta, esquerdas ou ambas por isto são de três tipos:

- Junção Externa a Esquerda (*Left Outer Join*) 
- Junção Externa a Direita (*Right Outer Join*) 
- Junção Externa Total (*Full Outer Join*) 

Leve em consideração a consulta a seguir:

*Mostre os nomes de todos os empregados e as horas que tem em cada projeto. Mesmo que o empregado não esteja em nenhum projeto mostre o seu nome.*

**Consulta24**  $\leftarrow \text{EMPREGADO} \bowtie_{\text{Empregado.EmpregadoId} = \text{Emp_Proj.EmpregadoId}} \text{EMP\_PROJ}$

Relação Consulta24								
<u>Empregado.</u> <u>EmpregadoId</u>	Nome	CPF	DeptoId	SupervisorId	Salario	<u>Emp_Proj.</u> <u>EmpregadoId</u>	<u>ProjetoId</u>	Horas
10101010	João Luiz	11111111	1	NULO	3.000,00	NULO	NULO	NULO
20202020	Fernando	22222222	2	10101010	2.500,00	20202020	5	10
20202020	Fernando	22222222	2	10101010	2.500,00	20202020	10	25
30303030	Ricardo	33333333	2	10101010	2.300,00	30303030	5	35
40404040	Jorge	44444444	2	20202020	4.200,00	40404040	20	50
50505050	Renato	55555555	3	20202020	1.300,00	50505050	20	35

Obs.: Repare o Empregado João Luiz de Id 10101010 não esta em nenhum projeto, mas faz parte da consulta e os atributos que não possuem relação com ele em EMP\_PROJ recebem nulo.



Finalizando a consulta temos:

**Consulta25**  $\leftarrow \pi_{\text{Nome,horas}} (\text{Consulta24})$

Relação <b>Consulta25</b>	
Nome	Horas
João Luiz	<i>NULO</i>
Fernando	10
Fernando	25
Ricardo	35
Jorge	50
Renato	35

## 2.9 Otimização de Consultas

Em algumas linguagens de consulta, a estratégia de execução é definida pela maneira como o usuário (ou programador) expressa a consulta. Em SQL, que é uma linguagem declarativa, apenas os resultados desejados são especificados. Portanto, a otimização de consultas é necessária em SGBDs relacionais baseados em SQL.

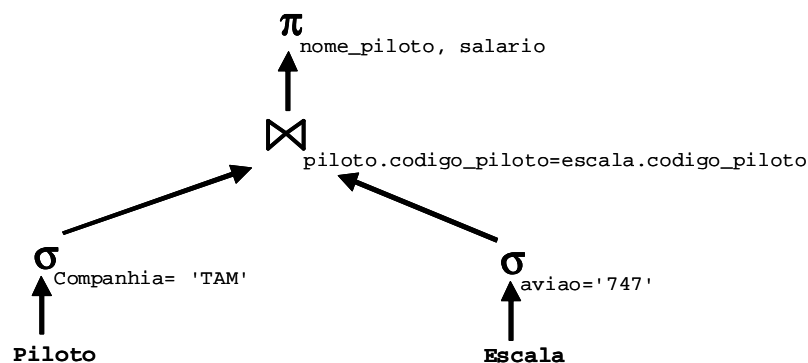
Passos principais para a otimização da consulta:

- 1 - Tradução da consulta SQL para a álgebra relacional
- 2 - Otimização do resultado

## 2.10 Grafo de Consulta

Grafo ou Árvore de consulta descreve um fluxo de dados com base na execução dos operadores. Os nós internos representam os operadores da consulta. Os nós folhas representam as tabelas sobre as quais a consulta é executada. As arestas indicam a direção do fluxo de dados. Representa uma estratégia de execução para a construção da relação resultado. Baseado nos operadores da álgebra relacional. É implementado nos SGBDs existentes.

Considere a consulta O nome e salário dos pilotos da TAM e que voam com avião do tipo 747.



**Figura 4 - Exemplo grafo de consulta**

Para localizar tuplas que satisfazem a condição de seleção o SGBD utiliza:

- Table Scan
  - Busca linear
    - A tabela é lida seqüencialmente, todas tuplas são testadas. Não pressupõe sort ou existência de índices
  - Busca binária
    - Tabela ordenada pelo atributo da condição (igualdade), busca binária para primeira página, depois busca linear.
    -
- Index Scan
  - Ler todos os nós folhas da árvore B+

- Index Seek (com operador de igualdade)
  - Índice Primário, definido sobre uma chave de busca que determina a ordem física dos registros
    - Em atributos do tipo primary key
    - Em atributos não primary key
  - Índice Secundário, chave de busca não determina a ordem física das tuplas Índices determinam uma ordem (lógica) das tuplas
    - Em atributos do tipo primary key
    - Em atributos não primary key
  - Índice Hash, baseado no cálculo de endereço.

## 3 SQL

### 3.1 Histórico

Certamente o SQL tem representado o padrão para linguagens de banco de dados relacionais. Existem diversas versões de SQL. A versão original foi desenvolvida pela IBM no Laboratório de Pesquisa de San José. Essa linguagem, originalmente chamada de Sequel, foi implementada como parte do projeto do Sistema R no início dos anos 70. Desde então, a linguagem Sequel foi evoluindo e seu nome foi mudado para SQL (Structured Query Language – Linguagem de Consulta Estruturada). Inúmeros produtos dão suporte atualmente para a linguagem SQL. O SQL se estabeleceu claramente como a linguagem padrão de banco de dados relacional.

Em 1986 a American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) publicaram um padrão SQL, chamado SQL-86. Em 1989, o ANSI publicou um padrão estendido para a linguagem: SQL-89. A próxima versão do padrão foi a SQL-92 (SQL 2), seguida das versões SQL:1999 (SQL 3), SQL:2003, SQL:2006 e SQL:2008. A versão mais recente é a SQL:2011.

### 3.2 Conceitos

SQL é um conjunto de declarações que é utilizado para acessar os dados utilizando gerenciadores de banco de dados.

SQL pode ser utilizada para todas as atividades relativas a um banco de dados podendo ser utilizada pelo administrador de sistemas, pelo DBA, por programadores, sistemas de suporte à tomada de decisões e outros usuários finais.

SQL é uma combinação de construtores em Álgebra Relacional.

### 3.3 Partes

**Linguagem de Definição de Dados** (Data-Definition Language – DDL). A DDL do SQL fornece comandos para definir esquemas de relações, excluir relações e modificar esquemas.

**Linguagem de Manipulação de Dados Interativa** (Data-Manipulation Language – DML). A DML da SQL inclui uma linguagem consulta baseada em álgebra relacional e no cálculo relacional de tupla. Ela também inclui comandos para inserir, excluir e modificar tuplas no banco de dados.

**Integridade.** A DDL SQL inclui comandos para especificar restrições de integridade às quais os dados armazenados no banco de dados precisam satisfazer. As atualizações que violam as restrições de integridade são proibidas.

**Definições de visões.** A DDL SQL inclui comandos para definir visões.

**Controle de transações.** A SQL inclui comandos para especificar o início e o fim de transações.

**SQL embutida** (Embedded DML) e **SQL Dinâmica.** A SQL embutida e a dinâmica definem como as instruções SQL podem ser incorporados dentro das linguagens de programação de finalidade geral como C, C++, Java, Cobol, Pascal e Fortran.

**Autorização** A DDL SQL inclui comandos para especificação de direitos de acesso para relações e visões.

## 4 SQL DDL

### 4.1 Linguagem de Definição de Dados

O conjunto de Relações (Tabelas) em um banco de dados deve ser especificado para o sistema por meio de uma linguagem de definição de dados (DDL).

A SQL DDL permite não só a especificação de um conjunto de relações, como também informações acerca de cada uma das relações, incluindo:

- Esquema de cada relação.
- Domínio dos valores associados a cada atributo.
- As regras de integridade.
- Conjunto de índices para manutenção de cada relação.
- Informações sobre segurança e autoridade sobre cada relação.
- A estrutura de armazenamento físico de cada relação no disco.

### 4.2 Esquema Base

Para DDL iremos considerar uma empresa na área bancária que possui as seguintes relações:

```
Cidade = (nome_cidade, estado_cidade)
Agencia = (nome_agencia, nome_cidade, fundos)
Departamento = (codigo_departamento, nome_departamento, total_salario)
Funcionario = (rg_funcionario, nome_funcionario, cpf_funcionario,
salario_funcionario, rg_supervisor, nome_cidade, codigo_departamento)
Cliente = (nome_cliente, rua_cliente, nome_cidade, salario_cliente,
data_nascimento, cpf_cliente)
Emprestimo = (numero_emprestimo, nome_agencia, total)
Devedor = (nome_cliente, numero_emprestimo)
Conta = (numero_conta, nome_agencia, saldo, rg_funcionario)
Depositante = (nome_cliente, numero_conta)
```

### 4.3 Tipos de Domínios em SQL

O padrão SQL aceita diversos tipos de domínios internos, incluindo:

**char(n)** – é uma cadeia de caracteres de tamanho fixo, com o tamanho n definido pelo usuário. Pode ser usada a sua forma completa **character** no lugar de **char**.

**varchar(n)** – é uma cadeia de caracteres de tamanho variável, com o tamanho n máximo definido pelo usuário. A forma completa, **character varying**, é equivalente.

**int** – é um inteiro (um subconjunto finito de inteiros que depende do equipamento). A forma completa, **integer** é equivalente.

**smallint** é um inteiro pequeno (um subconjunto do domínio dos tipos inteiros que depende do equipamento).

**numeric(p,d)** – é um número de ponto fixo cuja precisão é definida pelo usuário. O número consiste de p dígitos (mais o sinal), e d dos p dígitos estão a direita da vírgula decimal. Assim, **numeric(3,1)** permite que 44,5 seja armazenado exatamente, mas nem 444,4 nem 0,32 podem ser armazenados exatamente em um campo deste tipo.

**real, double precision** - são números de ponto flutuante e ponto flutuante de precisão dupla cuja precisão depende do equipamento.

**float(n)** - é um número de ponto flutuante com precisão definida pelo usuário em pelo menos n dígitos.

**date** – é uma data de calendário contendo um ano (com quatro dígitos), mês e dia do mês.

**time** – a hora do dia, em horas, minutos e segundos.

**timestamp** – uma combinação de **date** e **time**.

**clob** – objetos grandes para dados de caractere.

**blob** – objetos grandes para dados binários. As letras lob significam “**Large Object**”.

A SQL permite que a declaração de domínio de um atributo inclua a especificação de **not null**, proibindo, assim, a inserção de valores nulos para este tipo de atributo.

#### 4.3.1 Definição de Domínios

A SQL permite definir domínios usando a cláusula **create domain**, como mostra o exemplo:

```
CREATE DOMAIN nome_pessoa VARCHAR(20);
```

Podemos então usar o domínio de nome em nome\_pessoa para definir o tipo de um atributo com um domínio exato embutido.

A cláusula check da SQL permite modos poderosos de restrições de domínios que a maioria dos sistemas de tipos de linguagens de programação não permite. Especificamente a cláusula check permite ao projeto do esquema determinar um predicado que deva ser satisfeito por qualquer valor designado a uma variável cujo tipo seja o domínio. Por exemplo uma cláusula check pode garantir que o domínio relativo ao salário de um funcionário contenha somente valores maiores que 0.

```
CREATE DOMAIN salario_funcionario NUMERIC(10,2)
    CONSTRAINT ck_teste_salario_funcionario CHECK (VALUE > 0);
```

A cláusula check pode ser usada para restringir os valores nulos em um domínio:

```
CREATE DOMAIN salario_cliente NUMERIC(10,2)
    CONSTRAINT teste_nulo_salario_cliente CHECK (VALUE NOT NULL);
```

Como outro exemplo, o domínio pode ser restrito a determinado conjunto de valores por meio do uso da cláusula in:

```
CREATE DOMAIN estado_civil VARCHAR(20) CONSTRAINT ck_teste_estado_civil
    CHECK (VALUE IN ('Casado', 'Solteiro', 'Separado'));
```

A cláusula **constraint** é opcional e é usada para dar um nome a restrição. O nome é usado para indicar quais restrições foram violadas em determinada atualização.

A cláusula check pode ser mais complexa, dado que são permitidas subconsultas que se refiram a outras relações na codição check.

Por exemplo, a seguinte codição check poderia ser especificada na relação deposito:

```
CHECK (nome_agencia IN (SELECT nome_agencia FROM agencia));
```

### 4.4 Definição de Esquema em SQL

Definimos uma relação SQL usando o comando **create table**:

```
CREATE TABLE r (A1 D1,
                A2 D2,
                ...,
                An Dn,
                <regras de integridade 1>,
                <regras de integridade 2>,
                ...,
                <regras de integridade n>);
```

em que *r* é o nome da relação, cada *Ai* é o nome de um atributo no esquema da relação *r* e *Di* é o tipo de domínio dos valores no domínio dos atributos *Ai*. É possível definir valores default para os atributos, esta especificação é feita depois da especificação do domínio do atributo pela para **default** e o valor desejado.

As principais regras de integridade são:

```
PRIMARY KEY(Aj1, Aj2, ..., Ajm)
```

```
CHECK(P)
```

```
FOREIGN KEY(Aj1, Aj2, ..., Ajm) REFERENCES r
```

A especificação **PRIMARY KEY** diz que os atributos Aj1, Aj2, ..., Ajm formam a chave primária da relação. É necessário que os atributos de chave primária sejam não nulos e únicos; isto é, nenhuma tupla pode ter um valor nulo para a chave primária, e nenhum par de tuplas na relação pode ser igual em todos os atributos de chave primária.

A cláusula **CHECK** especifica um predicado P que precisa ser satisfeito por todas as tuplas em uma relação. No exemplo abaixo a cláusula check foi usada para simular um tipo enumerado, especificando que estado civil pode ser atribuído.

A cláusula **FOREIGN KEY** inclui a relação dos atributos que constituem a chave estrangeira (Aj1, Aj2, ..., Ajm) quanto o nome da relação à qual a chave estrangeira faz referência(r).

Todos os atributos de uma chave primária são declarados implicitamente como **NOT NULL**.

Definição do banco de dados de exemplo:

```
CREATE TABLE cidade(
    nome_cidade VARCHAR(15),
    estado_cidade VARCHAR(2) DEFAULT 'SC',
    PRIMARY KEY(nome_cidade));

CREATE TABLE Agencia (
    nome_agencia VARCHAR(15),
    nome_cidade VARCHAR(15),
    fundos NUMERIC(9,2) DEFAULT 0,
    PRIMARY KEY(nome_agencia),
    FOREIGN KEY (nome_cidade) REFERENCES cidade,
    CHECK (fundos >=20)
);

CREATE TABLE departamento(
    codigo_departamento INT,
    nome_departamento VARCHAR(30),
    total_salario NUMERIC(9,2) DEFAULT 0,
    PRIMARY KEY(codigo_departamento));

CREATE TABLE funcionario(
    rg_funcionario VARCHAR(7),
    nome_funcionario VARCHAR(20) NOT NULL,
    cpf_funcionario VARCHAR(11),
    salario_funcionario NUMERIC(9,2) DEFAULT 0 NOT NULL,
    rg_supervisor VARCHAR(7),
    nome_cidade VARCHAR(15),
    codigo_departamento INT,
    PRIMARY KEY(rg_funcionario),
    FOREIGN KEY(nome_cidade) REFERENCES cidade,
    FOREIGN KEY(codigo_departamento) REFERENCES departamento);

ALTER TABLE funcionario ADD FOREIGN KEY(rg_supervisor)
    REFERENCES funcionario;

CREATE TABLE Cliente (
    nome_cliente VARCHAR(15),
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15) NOT NULL,
```

```

        estado_civil VARCHAR(20),
        cpf_cliente VARCHAR(11),
        PRIMARY KEY (nome_cliente),
        FOREIGN KEY (nome_cidade) REFERENCES cidade,
        CHECK (estado_civil IN ('Casado','Solteiro','Viúvo'))
    );

CREATE TABLE Emprestimo(
    numero_emprestimo INT NOT NULL,
    nome_agencia VARCHAR(15) NOT NULL,
    total NUMERIC(9,2),
    PRIMARY KEY(numero_emprestimo),
    FOREIGN KEY (nome_agencia) REFERENCES agencia,
    CHECK (total > 0)
);

CREATE TABLE Devedor(
    nome_cliente VARCHAR(15) NOT NULL,
    numero_emprestimo INT NOT NULL,
    PRIMARY KEY (cpf_cliente, numero_emprestimo),
    FOREIGN KEY (cpf_cliente) REFERENCES cliente,
    FOREIGN KEY (numero_emprestimo) REFERENCES emprestimo
);

CREATE TABLE Conta(
    numero_conta INT,
    nome_agencia VARCHAR(15),
    saldo NUMERIC(9,2) DEFAULT 0,
    PRIMARY KEY (numero_conta),
    FOREIGN KEY (nome_agencia) REFERENCES agencia
);

CREATE TABLE Depositante(
    nome_cliente VARCHAR(15),
    numero_conta INT,
    PRIMARY KEY (nome_cliente, numero_conta),
    FOREIGN KEY (nome_cliente) REFERENCES cliente,
    FOREIGN KEY (numero_conta) REFERENCES conta
);

```

#### 4.4.1 Extensão Create Table

As aplicações exigem a criação de tabelas que possuem o mesmo esquema de uma tabela existente. A SQL fornece uma extensão para manipular esta tarefa.

```
CREATE TABLE <Nome_Tabela> AS SELECT * FROM <Tabela_Origem>
```

Onde <Nome\_Tabela> é o nome da tabela a ser criada e logo após a cláusula AS a consulta que irá dar origem da tabela. Esta consulta pode vir com ou sem dados. Para que isto ocorra especifique uma condição vazia, ou seja, que não traga tuplas.

#### 4.4.2 Esvaziar Tabelas

Para apagar o conteúdo de uma tabela sem remover o esquema utiliza um dos comandos abaixo.

```

TRUNCATE TABLE <Nome_Tabela>;
ou
DELTE FROM <Nome_Tabela>;
COMMIT;

```

Onde <Nome\_Tabela> é a tabela que se deseja esvaziar o conteúdo sem perder o esquema no banco de dados.

#### 4.4.3 Remover Tabelas

O comando drop table remove todas as informações de uma relação do banco de dados. Tanto os dados como o esquema é excluído.

```
DROP TABLE <Nome_Tabela>;
```

Onde <Nome\_Tabela> é a tabela que se deseja excluir do banco de dados.

#### 4.4.4 Adicionar Atributos

Usamos o comando alter table em SQL para adicionar atributos a uma relação existente. Todas as tuplas da relação recebem valores nulos para seu novo atributo.

```
ALTER TABLE <Nome_Tabela> ADD <Nome_Atributo> <Tipo_Atributo>;
```

Onde <Nome\_Tabela> é o nome da relação existente, <Nome\_Atributo> é o nome do novo atributo que será adicionado e <Tipo\_Atributo> é seu domínio.

#### 4.4.5 Alterar Atributos

Usamos o comando alter table em SQL também para alterar atributos a uma relação existente.

```
ALTER TABLE <Nome_Tabela> MODIFY <Nome_Atributo> <Tipo_Atributo>;
```

Onde <Nome\_Tabela> é o nome da relação existente, <Nome\_Atributo> é o nome do atributo que será alterado e <Tipo\_Atributo> é seu novo domínio. O atributo não deve ter valores a ser que seja literal e se deseja alterar o tamanho.

#### 4.4.6 Renomear Atributos

Usamos o comando alter table em SQL também para renomear atributos de uma relação existente.

```
ALTER TABLE <Nome_Tabela> RENAME COLUMN <Nome_Origem> TO <Nome_Destino>;
```

Onde <Nome\_Tabela> é o nome da relação existente, <Nome\_Origem> é o nome do atributo que será alterado e <Nome\_Destino> é seu novo nome.

#### 4.4.7 Excluir Atributos

Podemos excluir atributos de uma relação usando o comando

```
ALTER TABLE <Nome_Tabela> DROP COLUMN <Nome_Atributo>;
```

Onde <Nome\_Tabela> é a tabela de onde se deseja apagar o atributo <Nome\_Atributo>.

#### 4.4.8 Renomear Tabelas

Permite da um novo nome a uma tabela.

```
RENAME <Nome_Origem> TO <Nome_Destino>;
```

Onde <Nome\_Origem> é a tabela que deseja renomear e <Nome\_Destino> o novo nome para a tabela. O comando rename também funciona para visões, seqüências e sinônimos.

#### 4.4.9 Metadados

Olhar anexo Apêndice 1 item 8.4.



## 4.5 Integridade

No SQL todas as regras de integridade de dados e entidade são definidas por objetos chamados CONSTRAINT. Que podem ser definidos quando da criação da tabela ou posteriori via comando ALTER TABLE.

Os constraints suportados são:

```
NOT NULL
UNIQUE
PRIMARY KEY
FOREIGN KEY
CHECK
```

### 4.5.1 Definição de Constraints

Pode ser definidas de duas formas In-Line ou Out-Of-Line. Nem todas as constraints aceitam os dois tipos.

#### 4.5.1.1 CONSTRAINTS IN-LINE

Exemplo:

```
CREATE TABLE Cliente (
    nome_cliente VARCHAR(15) CONSTRAINT PK_cliente PRIMARY KEY,
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20),
    cpf_cliente VARCHAR(11),
    idade_cliente INT,
    data_nascimento DATE);
```

#### 4.5.1.2 CONSTRAINTS OUT-OF-LINE

Exemplo:

```
CREATE TABLE Cliente (
    nome_cliente VARCHAR(15) NOT NULL,
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20),
    cpf_cliente VARCHAR(11),
    idade_cliente INT,
    data_nascimento DATE,
    CONSTRAINT PK_cliente PRIMARY KEY (nome_cliente));
```

**Nota:** Quando o constraint for definido sem nome, o oracle define um nome para o mesmo - sys\_c00n - onde n é um número sequencial crescente.

### 4.5.2 Constraints

#### 4.5.2.1 NOT NULL CONSTRAINT

A especificação not null proíbe a inserção de um valor nulo para esse atributo. Qualquer modificação de banco de dados que causaria a inserção de um nulo em um atributo declarado para ser not null gera um erro.

Exemplo de uma tabela que precisa que todos os campos sejam preenchidos:

```
CREATE TABLE cliente (
    nome_cliente VARCHAR(15) CONSTRAINT NN_cliente_nome NOT NULL,
    rua_cliente VARCHAR(20) CONSTRAINT NN_cliente_rua NOT NULL,
```

```

nome_cidade VARCHAR(15) CONSTRAINT NN_cliente_cidade NOT NULL,
estado_civil VARCHAR(20) CONSTRAINT NN_cliente_estado_civil NOT NULL,
cpf_cliente VARCHAR(11) CONSTRAINT NN_cliente_cpf NOT NULL,
data_nascimento DATE CONSTRAINT NN_cliente_data_nascimento NOT NULL);

```

#### 4.5.2.2 UNIQUE CONSTRAINT

Atributos de uma declaração que seja **unique** (isto é, atributos de uma chave candidata) têm a garantia de unicidade deste valor para todas as tuplas. Mesmo a tabela já possuindo chave primária pode-se garantir a unicidade do atributo.

Exemplo de uma tabela que possui um atributo como chave primária e outro com uma chave candidata ativada como única.

```

CREATE TABLE cliente (
    nome_cliente VARCHAR(15) CONSTRAINT NN_cliente_nome NOT NULL,
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20),
    cpf_cliente VARCHAR(11),
    data_nascimento DATE,
    CONSTRAINT UK_cliente_cpf UNIQUE (cpf_cliente));

```

#### 4.5.2.3 PRIMARY KEY CONSTRAINT

Valor único que identifica cada linha da tabela.

Exemplo:

```

CREATE TABLE cliente (
    nome_cliente VARCHAR(15) CONSTRAINT NN_cliente_nome NOT NULL,
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20),
    cpf_cliente VARCHAR(11),
    data_nascimento DATE,
    CONSTRAINT PK_cliente PRIMARY KEY(nome_cliente),
    CONSTRAINT UK_cliente_cpf UNIQUE (cpf_cliente));

```

#### 4.5.2.4 FOREIGN KEY CONSTRAINT

Deve estar associada a uma **primary key** ou **unique** definida anteriormente.

Pode assumir valor nulo ou igual ao da chave referenciada.

Não existe limite para um número de **foreign keys**.

Garante a consistência com a primary key referenciada.

Pode fazer referência a própria tabela.

Não pode ser criada para views, synonyms e remote table.

Exemplo out-of-line:

```

CONSTRAINT FK_conta_cliente
    FOREIGN KEY (nome_cliente) REFERENCES cliente(nome_cliente)

```

ou

```

CONSTRAINT FK_conta_cliente
    FOREIGN KEY (nome_cliente) REFERENCES cliente

```

Exemplo in-line:

```

CREATE TABLE Conta (

```

```
...
cpf_cliente VARCHAR(11) CONSTRAINT FK_conta_cliente
REFERENCES cliente,
...);
```

#### 4.5.2.5 CHECK CONSTRAINT

As validações de colunas são feitas utilizando o CHECK CONSTRAINT.

Exemplo:

```
CREATE TABLE cliente(
    nome_cliente VARCHAR(15),
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20),
    cpf_cliente VARCHAR(11),
    data_nascimento DATE,
    idade_cliente INT,
    CONSTRAINT PK_cliente PRIMARY KEY(nome_cliente),
    CONSTRAINT UK_cliente_cpf UNIQUE (cpf_cliente),
    CONSTRAINT CK_cliente_estado_civil
        CHECK (estado_civil IN ('solteiro','casado','viúvo')),
    CONSTRAINT CK_cliente_idade
        CHECK (idade_cliente BETWEEN 1 AND 250));
```

#### 4.5.2.6 DEFAULT SPECIFICATION

Podemos atribuir valores default para colunas, visando facilitar a inserção de dados

Exemplo:

```
CREATE TABLE cliente (
    nome_cliente VARCHAR(15),
    rua_cliente VARCHAR(20),
    nome_cidade VARCHAR(15),
    estado_civil VARCHAR(20) DEFAULT 'solteiro',
    cpf_cliente VARCHAR(11),
    data_nascimento DATE DEFAULT SYSDATE,
    idade_cliente INT,
    CONSTRAINT PK_cliente PRIMARY KEY(nome_cliente),
    CONSTRAINT UK_cliente_cpf UNIQUE (cpf_cliente),
    CONSTRAINT CK_cliente_estado_civil
        CHECK (estado_civil IN ('solteiro','casado','viúvo')),
    CONSTRAINT CK_cliente_idade CHECK (idade_cliente BETWEEN 1 AND 250));
```

### 4.5.3 Padronização Nomes de Constraints

Para facilitar a identificação da origem de uma constraint é interessante padronizar seus nomes. Abaixo uma sugestão para os mesmos:

#### Primary Key

PK\_<Nome da Tabela>  
Ex.: PK\_CLIENTE, PK\_PRODUTO, PK\_PEDIDO

#### Foreign Key

FK\_<Tabela\_Origem>\_<Tabela\_Destino>  
Ex.: FK\_PEDIDO\_CLIENTE, PK\_ITEM\_PRODUTO

#### Check

CK\_<Nome\_Tabela>\_<Nome\_Atributo>  
Ex.: CK\_CLIENTE\_IDADE, CK\_CLIENTE\_SALARIO

**Not Null**

NN\_<Nome\_Tabela>\_<Nome\_Atributo>  
 Ex.: NN\_CLIENTE\_NOME, CK\_CLIENTE\_CPF

**Unique**

UK\_<Nome\_Tabela>\_<Nome\_Atributo>  
 Ex.: UK\_CLIENTE\_CPF, UK\_CLIENTE\_RG, UK\_USUARIO\_LOGIN

**4.5.4 Deleção em Cascata**

Opção a ser utilizada quando da definição do constraint foreign key, para que quando deletamos registros da tabela pai os registros da tabela filho sejam automaticamente deletados.

Exemplo:

```
CREATE TABLE depositante(
.....
nome_cliente VARCHAR(15) CONSTRAINT fk_depositante_cliente
REFERENCES cliente ON DELETE CASCADE
.....);
```

**4.5.5 Atualização em Cascata<sup>3</sup>**

Opção a ser utilizada quando da definição do constraint foreign key, para que quando atualizados registros da tabela pai os registros da tabela filho sejam automaticamente atualizados. Pode ser utilizado junto com deleção em cascata.

Exemplo:

```
CREATE TABLE depositante(
.....
nome_cliente VARCHAR(15) CONSTRAINT fk_depositante_cliente
REFERENCES cliente ON DELETE CASCADE ON UPDATE CASCADE
.....);
```

**4.5.6 Adicionando uma Foreign Key para a própria tabela**

Se você tiver um relacionamento recursivo onde existe um atributo que é uma chave estrangeira relacionado com a própria tabela geralmente você não tem como criar a constraint da chave estrangeira ao mesmo tempo em que cria a tabela. É necessário primeiro criar a tabela com a chave primaria e depois adicionar a chave estrangeira. Alguns bancos como o Oracle permite que você especifique no próprio script de criação da tabela.

Exemplo de uma tabela com relacionamento recursivo.

```
CREATE TABLE Funcionario(
  rg_funcionario VARCHAR(7),
  nome_funcionario VARCHAR(20),
  cpf_funcionario VARCHAR(11),
  salario_funcionario NUMERIC(9,2),
  rg_supervisor VARCHAR(7),
  nome_cidade VARCHAR(15),
  CONSTRAINT pk_funcionario PRIMARY KEY(rg_funcionario),
  CONSTRAINT fk_funcionario_cidade references Cidade);

ALTER TABLE Funcionario add CONSTRAINT Fk_funcionario_funcionario
  FOREIGN KEY (rg_supervisor) REFERENCES Funcionario;
```

**4.5.7 Adicionar Constraints**

Permite adicionar uma constraint a uma tabela.

---

<sup>3</sup> Não está disponível no oracle.

```
ALTER TABLE <Nome_Tabela> ADD CONSTRAINT <Nome_Constraint> <Especificacao>;
```

Onde <Nome\_Tabela> é a tabela na qual onde se deseja adicionar a constraint, <Nome\_Constraint> o nome da constraint e <Especificacao> a definição da constraint.

Exemplo:

```
ALTER TABLE Funcionario
        ADD CONSTRAINT uk_funcionario_cpf UNIQUE(cpf_funcionario);
```

#### 4.5.8 Excluir Constraint

Permite excluir uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> DROP CONSTRAINT <Nome_Constraint>;
```

Onde <Nome\_Tabela> é a tabela que possui a constraint e <Nome\_Constraint> o nome da constraint se deseja excluir.

Exemplo:

```
ALTER TABLE Funcionario DROP CONSTRAINT uk_funcionario_cpf;
```

#### 4.5.9 Renomear Constraint

Permite renomear uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> RENAME CONSTRAINT <Nome_Antigo> TO <Nome_Novo>;
```

Onde <Nome\_Tabela> é a tabela que possui a constraint que deseja renomear e <Nome\_Antigo> o nome da constraint se deseja renomear e <Nome\_Novo> o novo nome da constraint.

Exemplo:

```
ALTER TABLE Funcionario RENAME CONSTRAINT SYS_C000000 TO PK_Funcionario;
```

#### 4.5.10 Ativar Constraints

Permite ativar uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> ENABLE CONSTRAINT <Nome_Constraint>;
```

Onde <Nome\_Tabela> é a tabela que possui a constraint e <Nome\_Constraint> o nome da constraint se deseja ativar.

Exemplo:

```
ALTER TABLE Funcionario ENABLE CONSTRAINT uk_funcionario_cpf;
```

#### 4.5.11 Desativar Constraints

Permite desativar uma constraint de uma tabela.

```
ALTER TABLE <Nome_Tabela> DISABLE CONSTRAINT <Nome_Constraint>;
```

Onde <Nome\_Tabela> é a tabela que possui a constraint e <Nome\_Constraint> o nome da constraint se deseja desativar.

Exemplo:

```
ALTER TABLE Funcionario DISABLE CONSTRAINT uk_funcionario_cpf;
```

#### 4.5.12 Metadados

Olhar anexo Apêndice 1 item 8.5.

### 4.6 *Dicionário de Dados*

O Dicionário de dados não é refletido somente pelos domínios dos atributos em um SGBD. Os comentários realizados no dicionário de dados para as tabelas e atributos podem ser armazenados no banco de dados nas suas respectivas tabelas e atributos.

Os comentários criados de tabelas e colunas são armazenados no dicionário de dados do Oracle em:

```
ALL_COL_COMMENTS
USER_COL_COMMENTS
ALL_TAB_COMMENTS
USER_TAB_COMMENTS
```

Utilize ALL para ver os comentários de todas as tabelas e USER para os comentários criados somente pelo usuário em suas tabelas e atributos.

#### 4.6.1 Comentando Tabelas

Para adicionar um comentário a uma tabela.

```
COMMENT ON TABLE <Nome_Tabela> IS '<Comentario>';
```

Onde <Nome\_Tabela> é a tabela que irá receber o comentário <Comentario>.

Exemplo:

```
COMMENT ON TABLE Funcionario IS 'Tabela que armazena os dados do funcionario';
```

#### 4.6.2 Visualizando o comentário de Tabelas

Para visualizar o comentário de uma tabela.

```
SELECT COMMENTS FROM USER_TAB_COMMENTS WHERE TABLE_NAME = '<Nome_Tabela>';
```

Onde <Nome\_Tabela> é a tabela que será consultada.

Exemplo:

```
SELECT COMMENTS
FROM USER_TAB_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário e sua tabela:

```
SELECT TABLE_NAME, COMMENTS
FROM USER_TAB_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário de todas as tabelas:

```
SELECT TABLE_NAME, COMMENTS
FROM USER_TAB_COMMENTS;
```

#### 4.6.3 Comentando Atributos

Para adicionar um comentário a um atributo de uma tabela.

```
COMMENT ON COLUMN <Nome_Tabela>.<Nome_Atributo> IS '<Comentario>';
```

Onde <Nome\_Tabela> é a tabela que possui o atributo <Nome\_Atributo> que irá receber o comentário <Comentario>.

Exemplo:

```
COMMENT ON COLUMN Funcionario.rg_funcionario IS 'Campo chave primaria da
tabela funcionario';
```

#### 4.6.4 Visualizando o comentário de Atributos

Para visualizar os comentários de atributos de uma tabela.

```
SELECT COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = '<Nome_Tabela>';
```

Onde <Nome\_Tabela> é a tabela que será consultada:

Exemplo:

```
SELECT COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

Exibindo o comentário e o atributo e sua tabela:

```
SELECT TABLE_NAME, COLUMN_NAME, COMMENTS
FROM USER_COL_COMMENTS
WHERE TABLE_NAME = 'FUNCIONARIO';
```

## 5 SQL DML

### 5.1 Linguagem de Manipulação de Dados

Uma vez que o esquema esteja compilado e o banco de dados esteja populado, usa-se uma linguagem para fazer a manipulação dos dados, a DML (Data Manipulation Language - Linguagem de Manipulação de Dados).

Por manipulação entendemos:

- A recuperação das informações armazenadas no banco de dados;
- Inserção de novas informações no banco de dados;
- A remoção das informações no banco de dados;
- A modificação das informações no banco de dados

### 5.2 Esquema Base

Para DML iremos considerar uma empresa na área bancária que possui as seguintes relações:

```
Cidade = (nome_cidade, estado_cidade)
Agencia = (nome_agencia, nome_cidade, fundos)
Departamento = (codigo_departamento, nome_departamento, total_salario)
Funcionario = (rg_funcionario, nome_funcionario, cpf_funcionario,
salario_funcionario, rg_supervisor, nome_cidade, codigo_departamento)
Cliente = (nome_cliente, rua_cliente, nome_cidade, salario_cliente,
data_nascimento, cpf_cliente)
Emprestimo = (numero_emprestimo, nome_agencia, total)
Devedor = (nome_cliente, numero_emprestimo)
Conta = (numero_conta, nome_agencia, saldo, rg_funcionario)
Depositante = (nome_cliente, numero_conta)
```

### 5.3 Estruturas Básicas

A estrutura básica consiste de três cláusulas: select, from e where.

**Select** – corresponde a operação de projeção da Álgebra relacional. Ela é usada para relacionar os atributos desejados no resultado de uma consulta.

**From** – Corresponde a operação de produto cartesiano da Álgebra Relacional. Ela associa as relações que serão pesquisadas durante a evolução de uma expressão.

**Where** – Corresponde a seleção do predicado da Álgebra Relacional. Ela consiste em um predicado envolvendo atributos da relação que aparece na cláusula from.

Uma consulta típica em SQL tem a seguinte forma:

```
SELECT A1, A2, ..., An
FROM R1, R2, ..., Rn
WHERE P;
```

Cada  $A_i$  representa um atributo em  $R_i$  uma relação.  $P$  é um predicado. A consulta é equivalente à seguinte expressão em álgebra relacional:

$$\pi_{A_1, A_2, \dots, A_n} (\sigma_P (R_1 \times R_2 \times \dots \times R_n))$$

#### 5.3.1 Cláusula Select

Exemplo: Encontre os nomes de todas as agências da relação empréstimo.

```
SELECT nome_agencia
FROM emprestimo;
```



No casos em que desejamos forçar a eliminação da duplicidade, podemos inserir a palavra chave `distinct` depois de `select`.

```
SELECT distinct nome_agencia
FROM emprestimo;
```

No SQL permite usar a palavra chave `all` para especificar explicitamente que as duplicidades não serão eliminadas:

O asterisco “\*” pode ser usado para denotar ‘todos os atributos’, assim para exibir todos os atributos de empréstimo:

```
SELECT *
FROM emprestimo;
```

A cláusula `SELECT` pode conter expressões aritméticas envolvendo os operadores `+`, `-`, `/` e `*` e operandos constantes ou atributos das tuplas:

```
SELECT nome_agencia, numero_emprestimo, total * 100
FROM emprestimo;
```

### 5.3.2 A cláusula Where

Considere a consulta ‘encontre todos os números de empréstimos feitos na agência ‘Central’ com totais emprestados acima de 1200 dólares’. Esta consulta pode ser escrita em SQL como:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE nome_agencia = 'Central' AND total > 1200;
```

A SQL usa conectores lógicos **and**, **or** e **not** na cláusula `where`. Os operandos dos conectivos lógicos podem ser expressões envolvendo operadores de comparação `<`, `<=`, `>`, `>=`, `=` e `!=`. Para a diferença pode ser utilizando ainda o operador `<>`.

A SQL possui o operador de comparação **between** para simplificar a cláusula `where` que especifica que um valor pode ser menor ou igual a algum valor e maior ou igual a algum outro valor. Se desejarmos encontrar os números de empréstimos cujos montantes estejam entre 90 mil dólares e 100 mil dólares, podemos usar a comparação **between** escrevendo:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total BETWEEN 90000 AND 100000;
```

em vez de

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total >=90000 AND total <= 100000;
```

Similarmente podemos usar o operador de comparação **not between**.

### 5.3.3 A cláusula From

A cláusula `from` por si só define um produto cartesiano das relações da cláusula. Uma vez que a junção natural é definida em termos de produto cartesiano, uma seleção é uma projeção é um meio relativamente simples de escrever a expressão SQL para uma junção natural.

Escrevemos a expressão em álgebra relacional:

$$\pi_{\text{nome\_cliente, numero\_emprestimo}}(\text{devedor} \bowtie \text{emprestimo})$$

ou com produto cartesiano

$$(\pi_{\text{nome\_cliente, numero\_emprestimo}}(\sigma_{\text{devedor.numero\_emprestimo=emprestimo.numero\_emprestimo}}(\text{devedor} \bowtie \text{emprestimo})))$$

para a consulta ‘para todos os clientes que tenham um empréstimo em um banco, encontre seus nomes e números de empréstimos’. Em SQL, essa consulta pode ser escrita como:

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

Note que a SQL usa a notação `nome_relação.nome_atributo`, como na álgebra relacional para evitar ambiguidades nos casos em que um atributo aparecer no esquema mais de uma relação.

O SQL incorpora extensões para realizar junções naturais e junções externas na cláusula `from`.

#### 5.3.4 A operação Rename

A SQL proporciona um mecanismo para rebatizar tanto relações quanto atributos, usando a cláusula **as**, da seguinte forma:

```
nome_antigo AS nome_novo
```

Considere novamente a consulta usada anteriormente:

```
SELECT distinct nome_cliente, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

Por exemplo se desejarmos que o nome do atributo `nome_cliente` seja substituído pelo nome `nome_devedor`, podemos reescrever a consulta como:

```
SELECT distinct nome_cliente AS nome_devedor, devedor.numero_emprestimo
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo;
```

A cláusula **as** pode aparecer tanto na cláusula **select** quanto na cláusula **from**.

#### 5.3.5 Variáveis Tuplas

Variáveis tuplas são definidas na cláusula **from** por meio do uso da cláusula **as**. Para ilustrar, reescrevemos a consulta ‘para todos os funcionários encontre o nome do seu supervisor’, assim:

```
SELECT distinct F.nome_funcionario, S.nome_funcionario
FROM funcionario AS F, funcionario AS S
WHERE F.rg_funcionario = S.rg_supervisor;
```

Variáveis tuplas são úteis para comparação de duas tuplas de mesma relação.

#### 5.3.6 Operações em Strings

As operações em strings mais usadas são as checagens para verificação de coincidências de pares, usando o operador **like**. Indicaremos esses pares por meio do uso de dois caracteres especiais:

Porcentagem (%) : o caracter % compara qualquer substring.  
 Sublinhado ( \_ ) : o caracter \_ compara qualquer caracter.

Comparações desse tipo são sensíveis ao tamanho das letras; isto é, minúsculas não são iguais a maiúsculas, e vice-versa. Para ilustrar considere os seguintes exemplos;

'**Pedro%**' corresponde a qualquer string que comece com 'Pedro'  
 '%**inh%**' corresponde a qualquer string que possua uma substring 'inh', por exemplo 'huguinho', 'zezinho' e 'luizinho'  
 ' \_ \_ \_' corresponde a qualquer string com exatamente três caracteres  
 ' \_ \_ \_%' corresponde a qualquer string com pelo menos três caracteres

Pares são expressões em SQL usando o operador de comparação **like**. Considere a consulta 'encontre os nomes de todos os clientes possuam a substring 'Silva' '. Esta consulta pode ser escrita assim:

```
SELECT nome_cliente
FROM cliente
WHERE nome_cliente LIKE '%Silva%';
```

A SQL permite-nos pesquisar diferenças em vez de coincidências, por meio do uso do operador de comparação **not like**.

### 5.3.7 Expressão Regular

As expressões regulares são uma maneira de descrever um conjunto de string com base em características comuns compartilhados por cada string no conjunto. Elas podem ser usadas para pesquisar, editar ou manipular texto e dados.

No Oracle existe a função REGEXP\_LIKE semelhante ao like, mas que segue as regras das expressões regulares.

```
SELECT *
FROM cliente
WHERE nome_cliente Like 'Joao%';

SELECT *
FROM cliente
WHERE REGEXP_LIKE(nome_cliente, '^Joao');
```

Para negar a condição de procura use o operador NOT;

```
SELECT *
FROM cliente
WHERE nome_cliente NOT LIKE 'Joao%';

SELECT *
FROM cliente
WHERE NOT REGEXP_LIKE(nome_cliente, '^Joao');
```

O operador LIKE realiza a pesquisa sempre exata. Para entrar com parte de uma palavra é necessário utilizar o curinga % ou \_. Com REGEX\_LIKE, a procura parcial é válida.

```
SELECT *
FROM cliente
WHERE nome_cliente LIKE 'Joao';

SELECT *
FROM cliente
WHERE REGEXP_LIKE(nome_cliente, '^Joao');
```

O operador LIKE diferencia maiúsculas de minúsculas. Para ignorar esta diferença ou você digita tudo em maiúsculo ou minúsculo ou utiliza alguma função de transformação como UPPER ou LOWER. Com a função REGEXP\_LIKE você pode passar um terceiro argumento para definir se você deseja ignorar ou não a diferença entre maiúsculas e minúsculas. O parâmetro a ser passado é “i” (minúsculo) para ignorar e “c” considerar a diferença.

```
SELECT *
FROM cliente
WHERE REGEXP_LIKE(nome_cliente,'joao'); -- Depende da digitação correta
```

```
SELECT *
FROM cliente
WHERE REGEXP_LIKE(nome_cliente,'JoAo','i'); -- Igual
```

```
SELECT *
FROM cliente
WHERE REGEXP_LIKE(nome_cliente,'Joao','c'); -- Diferente
```

Para utilização de Expressões Regulares, se faz necessário conhecer alguns metacaracteres que são encontrados na utilização de expressões regulares. A seguir a lista destes caracteres:

**Quadro 1 - Lista de Metacaracteres**

Operador	Descrição
( )	Trata a expressão ou o conjunto de literais como uma subexpressão.
[...]	O par de colchetes delimita uma lista de uma ou mais expressões: combinações de elementos, símbolos, classes equivalentes, classes de caractere ou expressões de dimensão.
[^...]	Uma expressão de não igualdade. Indica que a lista de expressões dentro dos colchetes não deve ser encontrada.
[. ...]	O uso do ponto especifica uma combinação de elementos de acordo com o local. Muito útil em situações onde dois ou mais caracteres são necessários para especificar um elemento, como por exemplo, na especificação de um limite entre “a” e “ch” utilizaremos [a...[.ch.]].
[:::]	Especifica uma classe de caracteres.
[==]	Especifica uma classe de equivalência, por exemplo, [=e=] representa “e”, “é”, “è”, “ë”.
.	O ponto combina qualquer caractere.
?	Combina zero ou uma ocorrência da subexpressão que o precede.
*	Combina zero ou mais ocorrências da subexpressão que o precede.
+	Combina zero ou mais ocorrências da subexpressão que o precede.
{n1}	Combina precisamente n1 ocorrências da subexpressão que o precede.
{n1,}	Combina n1 ou mais ocorrências da subexpressão que o precede.
{n1, n2}	Combina as ocorrências entre n1 e n2, inclusive os limites, da subexpressão que o precede.
\	Dependendo do contexto a contra barra é apenas uma contra barra, se estiver precedendo outro operador este é transformado em literal, por exemplo, \+ é o valor literal do mais.
\n1	Referencia anterior, repetição de “n1 vezes” da subexpressão dentro da expressão anterior.
	Operador lógico “OU”. Utilizado para separar duas expressões, onde uma delas é combinada. Ex.: ('joão' 'maria')
^	Início de linha. Ex.: ^A strings que se iniciem com A.
\$	Final de linha. Ex.: \$B strings que se terminem com B.

O padrão POSIX possui classes pré-definidas que podem ser utilizados com ([ ]) – colchetes. São utilizadas para simplificar as expressões regulares. A seguir a lista de algumas classes:

**Quadro 2 - Classes Pré-definidas**

Operador	Descrição
[ :alnum: ]	Caracteres alfanuméricos. Inclui letras e números, omitindo pontuação. [A-Za-z0-9]
[ :alpha: ]	Caracteres do alfabeto. Apenas letras. [A-Za-z]
[ :blank: ]	Caracteres que formam espaços.
[ :cntrl: ]	Caracteres de controle (que não são impressos).
[ :graph: ]	Todas as classes de caracter combinadas, [:punct:], [:upper:], [:lower:], [:digit:].
[ :lower: ]	Caracteres Minúsculos [a-z]
[ :print: ]	Caracteres que podem ser impressos.
[ :punct: ]	Caracteres de pontuação. [.,!?:;]
[ :space: ]	Caracteres de espaço que não podem ser impressos. . [\r\n\r\f\v]
[ :upper: ]	Caracteres maiúsculos. [A-Z]
[ :xdigit: ]	Caracteres hexadecimais.

As faixas de valores servem para facilitar a especificação de intervalos nas expressões regulares. A seguir uma lista de algumas faixas:

**Quadro 3 - Faixa de valores**

Operador	Descrição
[A-Z]	Todos os caracteres do alfabeto maiúsculos.
[a-z]	Todos os caracteres do alfabeto minúsculos.
[0-9]	Todos os dígitos numéricos.
[1-9]	Todos os dígitos numéricos, menos zero.
[A-Za-z]	Todos caracteres do alfabeto maiúsculos e minúsculos.

Alguns exemplos de expressões:

Validar o formato da Hora (hh:mm):

```
...:..
[0-9]{2}:[0-9]{2}
[012][0-9]:[0-9]{2}
[012][0-9]:[0-5][0-9]
([01][0-9]|2[0-3]):[0-5][0-9]
```

Validar o formato da Data (dd/mm/aaaa):

```
../.../....
[0-9]{2}/[0-9]{2}/[0-9]{4}
[0123][0-9]/[0-9]{2}/[0-9]{4}
[0123][0-9]/[01][0-9]/[0-9]{4}
[0123][0-9]/[01][0-9]/[12][0-9]{3}
([012][0-9]|3[01])/[01][0-9]/[12][0-9]{3}
([012][0-9]|3[01])/(0[1-9]|1[012])/[12][0-9]{3}
(0[1-9]|1[2][0-9]|3[01])/(0[1-9]|1[012])/[12][0-9]{3}
```

Validar o formato do Telefone (9999-9999):

```
....-....
[0-9]{4}-[0-9]{4}
```

```

\(...\)[0-9]{4}-[0-9]{4}
\(...\)?[0-9]{4}-[0-9]{4}
\(...\)?[0-9]{4}-[0-9]{4}
\(...\)?[0-9]{4}-[0-9]{4}
\(...\)?[0-9]{4}-[0-9]{4}
\(...\)?[0-9]{4}-[0-9]{4}

```

Existe outros operadores:

REGEXP\_SUBSTR - Para obter trechos de texto, para informar a posição de início da busca passe um terceiro parâmetro.

REGEXP\_INSTR - Para saber a posição de início.

REGEXP\_COUNT - Para saber o número de vezes de ocorrência do texto.

REGEXP\_REPLACE - Para substituir texto.

Outra consulta, encontrar o nome e cpf dos funcionários que onde foram digitados corretamente ou seja somente 11 números.

```

SELECT nome_funcionario, cpf_funcionario
FROM funcionario
WHERE REGEXP_LIKE(CPF_FUNCIONARIO, '^([[:digit:]]{11})$');

```

Para encontrar os que foram digitados errado utilize NOT REGEXP\_LIKE.

Encontrar o nome e data de nascimento dos clientes que nasceram entre 2000 e 2005.

```

SELECT nome_cliente, data_nascimento
FROM cliente
WHERE REGEXP_LIKE(TO_CHAR(data_nascimento, 'YYYY'), '^200[0-5]$');

```

Você pode utilizar expressão regular também para validação de dados em constraints, por exemplo validar o formato do cpf:

```

ALTER TABLE funcionario ADD ck_funcionario_cpf
CHECK (REGEXP_LIKE(cpf_funcionario, '^d{3}.^d{3}.^d{3}-d{2}$'));

```

Uma constraint para validar o formato do email do cliente:

```

ALTER TABLE cliente ADD (CONSTRAINT ck_cliente_email
CHECK (REGEXP_LIKE(email,
'^([[:alnum:]]+)([[:alnum:]]+).(com|net|org|edu|gov|mil)$')));

```

Uma constraint para validar o numero do telefone de cliente no formato (XX) XXXX-XXXX.

```

ALTER TABLE cliente ADD (CONSTRAINT ck_cliente_telefone
CHECK (REGEXP_LIKE(numero_telefone,
'^\([[:digit:]]{2}\) [[:digit:]]{4}-[[:digit:]]{4}$')));

```

### 5.3.8 Precedência dos Operadores

Como qualquer linguagem existe uma ordem na precedência dos operadores:

**Quadro 4 - Precedência dos operadores**

Ordem de Avaliação	Operadores
1	Operadores Aritméticos
2	Operador de Concatenação
3	Condições de Comparação
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Condição lógica NOT

7	Condição lógica AND
8	Condição lógica OR

Para modificar a ordem de avaliação utilize parênteses.

### 5.3.9 Ordenação e Apresentação de Tuplas

A SQL oferece ao usuário algum controle sobre a ordenação por meio da qual as tuplas de uma relação serão apresentadas. A cláusula `order by` faz com que as tuplas do resultado de uma consulta apareçam em uma determinada ordem.

Para listar em ordem alfabética todos os clientes que tenham um empréstimo na agência Centro, escrevemos:

```
SELECT distinct nome_cliente
FROM devedor, emprestimo
WHERE devedor.numero_emprestimo = emprestimo.numero_emprestimo
AND nome_agencia = 'Centro'
ORDER BY nome_cliente;
```

Por default, a cláusula **order by** relaciona os itens em ordem ascendente. Para a especificação da forma de ordenação, devemos indicar **desc** para ordem descendente e **asc** para ordem ascendente. A ordenação pode ser realizada por diversos atributos.

Suponha que desejamos listar a relação cliente inteira por ordem ascendente de nome de cidade. Se diversos cliente residirem na mesma cidade, queremos que seja realizada uma segunda ordenação descendente pelo nome do cliente. Expressamos essa consulta da seguinte forma:

```
SELECT nome_cidade, nome_cliente
FROM cliente
ORDER BY nome_cidade ASC, nome_cliente DESC;
```

Para completar uma solicitação de `order by`, a SQL precisa realizar uma *sort* (intercalação). Uma vez que a intercalação de um grande número de tuplas pode ser custosa, é desejável usá-la somente quando realmente necessário, ou utilizar em um campo indexado.

## 5.4 Composição de Relações

Além de fornecer o mecanismo básico do produto cartesiano para a composição das tuplas de duas relações o SQL oferece diversos outros mecanismos para composição de relações como as junções condicionais e as junções naturais, assim como várias formas de junções externas. Essas operações adicionais são usadas tipicamente como expressões de subconsultas na cláusula `from`.

Um produto cartesiano é formado nas seguintes condições:

Uma condição de junção(`join`) é omitida

Uma condição de junção(`join`) é inválida

Todas as linhas da primeira tabela são multiplicadas com todas as linhas da segunda tabela.

Para impedir a formação de um produto cartesiano, sempre inclua uma condição de restrição válida na cláusula `WHERE`.

### 5.4.1 Tipos de Junções e Condições

Operações de junção tomam duas relações e têm como resultado outra relação. Embora as expressões para junção externa sejam normalmente usadas na cláusula `from`, elas podem ser usadas em qualquer lugar onde se usa uma relação.

As junções podem ser de quatro tipos:

EquiJoin = Junção Interna

Non-EquiJoin = Não Junção Interna  
 OuterJoin = Junção Externa  
 SelfJoin = Auto Junção(interna)

Cada uma das variantes das operações de junção em SQL consiste em um tipo de junção e em uma condição de junção. As condições de junção definem quais tuplas das duas relações apresentam correspondência e quais atributos são apresentados no resultado de uma junção. O tipo de junção define como as tuplas em cada relação que não possuam nenhuma correspondência (baseado na condição de junção) com as tuplas da outra relação devem ser tratadas. Abaixo estão alguns dos tipos de junções permitidos e as condições de junção. O primeiro tipo é a junção interna (INNER JOIN) e os outros tipos são de junções externas(OUTER JOIN). As três condições são NATURAL, ON e USING.

**Quadro 5 - Cláusulas e condições de junção**

<b>Claúsulas de Junção</b>	<b>Condições de junção</b>
Inner join => join	Natural
Left outer join => left join	On <predicado>
Right outer join => right join	Using (A1, A2, ..., An)
Full outer join => full join	

O uso de uma condição de junção é obrigatório para junções externas, mas opcional para junções internas (se for omitido, o resultado será um produto cartesiano). Sintaticamente a palavra-chave NATURAL aparece antes do tipo da junção, e as condições ON e USING apareçam no final de uma expressão de junção. As palavras chaves INNER e OUTER são opcionais, uma vez que os nomes dos demais tipos de junções nos permitem deduzir se se trata de uma junção interna ou externa.

O significado da condição NATURAL, em termos de correspondência de tuplas das duas relações é bastante rígido. A ordem dos atributos no resultado de uma junção natural é a seguinte. Os atributos da junção (isto é, os atributos comuns a ambas as relações) aparecem primeiro, conforme a ordem em que aparecem na relação do lado esquerdo. Depois vêm todos os atributos para os quais não há correspondência aos do lado esquerdo e, finalmente, todos os atributos sem correspondência aos da relação do lado direito.

O tipo de junção RIGHT OUTER JOIN é simétrico a LEFT OUTER JOIN. As tuplas da relação do lado direito que não correspondem a nenhuma tupla da relação do lado esquerdo são preenchidos com nulos e adicionados ao resultado da junção externa direita.

A expressão seguinte é um exemplo da combinação dos tipos de junção NATURAL e do RIGHT OUTER JOIN.

```
Emprestimo NATURAL RIGHT OUTER JOIN devedor
```

Os atributos do resultado são definidos por um tipo de junção, que é uma junção natural; então numero\_emprestimo aparece somente uma vez.

A condição de junção USING(A1, A2, ..., An) é similar à condição de junção natural exceto pelo fato de que seus atributos de junção são os atributos A1, A2, ..., An, em vez de todos os atributos comuns a ambas as relações. Os atributos A1, A2, ..., An devem ser somente os atributos comuns a ambas as relações e eles aparecem apenas uma vez no resultado da junção.

O tipo FULL OUTER JOIN é uma combinação dos tipos de junções externas à esquerda e à direita. Depois que o resultado de uma junção interna é processado, as tuplas da relação do lado esquerda que não correspondem a nenhuma das tuplas do lado direito são preenchidas com valores nulos, e depois adicionados ao resultado. Similarmente, as tuplas da relação do lado direito que não coincidem com



nenhuma das tuplas da relação do lado esquerdo são também preenchidas com nulos e adicionados ao resultado.

Por exemplo, o resultado da expressão:

```
Emprestimo FULL OUTER JOIN devedor USING(numero_emprestimo)
```

é mostrado abaixo.

nome_agencia	numero_emprestimo	total	nome_cliente
...	...	...	...

#### 5.4.2 Junção Interna

Junção Interna (Inner Join) é quando tuplas são incluídas no resultado de uma consulta somente se existir uma correspondente na outra relação.

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 INNER JOIN tabela2 ON tabela1.atributo1 = tabela2.atributo1;
```

Se o nome dos atributos for igual a cláusula ON e desnecessária, para isto usa-se o a Junção Natural com cláusula NATURAL.

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 NATURAL INNER JOIN tabela2;
```

A cláusula INNER é opcional, uma vez que os nomes dos demais tipos de junções nos permite deduzir quando se trata de outro tipo junção. Sendo assim consulta anterior pode ser reescrita desta forma:

```
SELECT tabela1.atributo1, tabela2.atributo2
FROM tabela1 NATURAL JOIN tabela2;
```

#### 5.4.3 Junção de Mais de duas tabelas

É possível realizar a junção de mais de duas tabelas utilize parentes para separar cada par de tabelas unidas.

```
SELECT tabela1.atributo1, tabela2.atributo2, tabela3.atributo3
FROM (tabela1 INNER JOIN tabela2
      ON tabela1.atributo1 = tabela2.atributo1) INNER JOIN tabela3
      ON tabela2.atributo3 = tabela3.atributo3;
```

Ou com produto cartesiano:

```
SELECT tabela1.atributo1, tabela2.atributo2, tabela3.atributo3
FROM tabela1, tabela2, tabela3
WHERE tabela1.atributo1 = tabela2.atributo1
      AND tabela2.atributo3 = tabela3.atributo3;
```

#### 5.4.4 Junção Externa

Junção externa é quando tuplas são incluídas no resultado sem que exista uma tupla correspondente na outra relação. Utiliza-se junção externa para listar tuplas que não usualmente se reúnem numa condição join.

Podem ser de três tipos:

RIGHT OUTER JOIN = Junção Externa a Direita

LEFT OUTER JOIN = Junção Externa a Esquerda

FULL OUTER JOIN = Junção Externa Total (Junção Externa a Esquerda + Junção Externa a Direita)

O operador de junção externa é o sinal de adição (+)

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE R1.A1(+) = R2.A1;
```

ou

```
SELECT R1.A1, R2.A2
FROM R1, R2
WHERE R1.A1 = R2.A2(+);
```

ou OUTER JOIN tanto para a esquerda(LEFT) como a direita(RIGHT)

```
SELECT R1.A1, R2.A2
FROM R1 RIGHT OUTER JOIN R2 ON R1.A1= R2.A2;
```

ou

```
SELECT R1.A1, R2.A2
FROM R1 LEFT OUTER JOIN R2 ON R1.A1 = R2.A2;
```

#### 5.4.5 Auto Junção

Se for necessário realizar a junção da tabela com ela mesma, obriga-se renomear uma das tabelas ou as duas para não ocorrer ambigüidade de nomes.

```
SELECT tabela1.atributo1, tabela1.atributo2
FROM tabela1 t1 INNER JOIN tabela1 t2 ON t1.atributo1 = t2.atributo2;
```

ou com produto cartesiano:

```
SELECT tabela1.atributo1, tabela1.atributo2
FROM tabela1 t1, tabela1 t2
WHERE t1.atributo1 = t2.atributo2
```

### 5.5 Operações de Conjuntos

Os operadores union, intersect e except operam relações e correspondem às operações  $\cup$ ,  $\cap$  e  $-$  da álgebra relacional. Como a união, interseção e diferença de conjuntos da álgebra relacional, as relações participantes das operações precisam ser compatíveis, isto é, elas precisam ter o mesmo conjunto de atributos.

#### 5.5.1 A operação de União

Para encontrar todos os clientes do banco que possuem empréstimo, uma conta ou ambos, escrevemos:

```
(SELECT nome_cliente
FROM depositante)
UNION
(SELECT nome_cliente
FROM devedor);
```

A operação de union, ao contrário da cláusula select, automaticamente elimina as repetições. Se desejarmos obter todas as repetições, teremos de escrever **union all** no lugar de **union**:

```
(SELECT nome_cliente
FROM depositante)
UNION ALL
(SELECT nome_cliente
FROM devedor);
```

### 5.5.2 A operação Interseção

Para encontrar todos os clientes que tenham tanto empréstimo quanto contas, escrevemos:

```
(SELECT distinct nome_cliente
FROM depositante)
INTERSECT
(SELECT distinct nome_cliente
FROM devedor);
```

A operação intersect automaticamente elimina todas as repetições.

Se desejarmos obter todas as repetições, teremos de escrever **intersect all**<sup>4</sup> no lugar de **intersect**.

```
(SELECT nome_cliente
FROM depositante)
INTERSECT ALL
(SELECT nome_cliente
FROM devedor);
```

### 5.5.3 A operação Exceto(Subtração)

Para encontrar todos os clientes que tenham uma conta e nenhum empréstimo no banco, escrevemos:

```
(SELECT distinct nome_cliente
FROM depositante)
EXCEPT
(SELECT distinct nome_cliente
FROM devedor);
```

A operação **except**<sup>5</sup> automaticamente elimina todas as repetições.

Se desejarmos obter todas as repetições, teremos de escrever **except all** no lugar de **except**.

No oracle no lugar de except é utilizado a palavra minus.

## 5.6 Funções Agregadas

Funções agregadas ou de grupo são funções que tomam uma coleção (um conjunto ou subconjunto) e valores como entrada, retornando um valor simples. A SQL oferece cinco funções agregadas pré-programadas:

Média (average): **avg**.

Mínimo (minimum): **min**.

Máximo (maximum): **max**.

Total (total): **sum**.

Contagem (count): **count**.

A entrada para **sum** e **avg** precisa ser um conjunto de números, mas as outras podem operar com conjunto de tipos de dados não-numéricos, como strings e semelhantes.

Para ‘encontrar a média dos saldos em contas na agência Perryridge’, escrevemos:

```
SELECT avg(saldo)
```

<sup>4</sup> Não existe este comando no oracle.

<sup>5</sup> No oracle é o comando **minus**.

```
FROM conta
WHERE nome_agencia = 'Perryridge';
```

O resultado dessa consulta é uma relação com um atributo único, contendo uma única linha com um valor numérico correspondente à média dos saldos na agência Perryridge. Podemos, opcionalmente, dar um nome para esse atributo da relação resultante usando a cláusula **as**.

Existem circunstância em que gostaríamos de aplicar uma função agregada não somente a um conjunto de tuplas, mas também a um grupo de conjunto de tuplas o que é possível usando a cláusula SQL **group by**. O atributo ou atributos fornecidos em uma cláusula **group by** são usados para forma grupos. Tuplas com os mesmos valores em todos os atributos da cláusula **group by** são colocados em um grupo.

Encontrar a média dos saldos nas contas de cada uma das agências do banco', escrevemos:

```
SELECT nome_agencia, avg(saldo)
FROM conta
GROUP BY nome_agencia;
```

Se desejarmos eliminar repetições, usamos a palavra-chave **distinct** na expressão agregada.

Para encontrar 'o números de depositantes de cada agência', neste caso, um depositante é contado somente uma vez, independente do número de contas que o correntista possua.

```
SELECT nome_agencia, count(distinct nome_cliente)
FROM depositante, conta
WHERE depositante.numero_conta = conta.numero_conta
GROUP BY nome_agencia;
```

Às vezes, é mais interessante definir condições e aplicá-las a grupos do que aplicá-las a tuplas. Por exemplo, poderíamos estar interessados em quais agências possuem média dos saldos aplicados em conta maior que 1200 dólares. Essa condição não se aplica a uma única tupla, mas em cada grupo determinado pela cláusula **group by**. Para exprimir tal consulta, usamos a cláusula **having** da SQL. Os predicados da cláusula **having** são aplicados depois da formação dos grupos, assim poderão ser usadas funções agregadas. Escrevemos essa consulta em SQL como segue:

```
SELECT nome_agencia, avg(saldo)
FROM conta
GROUP BY nome_agencia
HAVING avg(saldo)>1200;
```

Às vezes, desejamos tratar a relação como um todo, como um grupo simples. Nesses casos, não usamos a cláusula **group by**. Considere a consulta 'encontre a média dos saldos de todas as contas'. Escrevemos essa consulta assim:

```
SELECT avg(saldo)
FROM conta;
```

Usamos a função agregada **count** com muita frequência para contar o número de tuplas em uma relação. A notação para essa função em SQL é **count(\*)**. Assim, para encontrar o número de tuplas da relação cliente escrevemos:

```
SELECT count(*)
FROM cliente;
```

A SQL não permite o uso do **distinct** com **count(\*)**. É válido usar **distinct** com **max** e **min**, mesmo não alterando o resultado. Podemos usar a palavra-chave **all** no lugar de **distinct** quando desejamos a manutenção das repetições.

Se uma cláusula **where** é uma cláusula **having** aparecem na mesma consulta, o predicado que aparece primeiro é aplicado primeiro.

## 5.6.1 Outras Funções

### 5.6.1.1 RollUp

Esta expressão em uma consulta de agrupamento (GROUP BY) permite obter os totais da função utilizada para calcular a consulta depois de exibir os dados do agrupamento.

Ex.: Encontre a quantidade de clientes por cidade e o total geral.

```
SELECT nome_cidade, COUNT (*)
FROM cliente
GROUP BY ROLLUP(nome_cidade);
```

Você pode escolher mais de uma coluna para ver os totais e subtotais.

Ex.: Encontre a quantidade de clientes por cidade e rua os subtotais e o total geral.

```
SELECT nome_cidade, rua_cliente, COUNT(*)
FROM cliente
GROUP BY ROLLUP(nome_cidade, rua_cliente);
```

### 5.6.1.2 Cube

É similar ao anterior, só que calcula todos os subtotais relativos à consulta antes de exibir os dados dos agrupamentos.

Ex.: Encontre a quantidade de clientes por cidade e o total geral.

```
SELECT nome_cidade, COUNT(*)
FROM cliente
GROUP BY CUBE(nome_cidade);
```

Você pode escolher mais de uma coluna para ver os totais e subtotais.

Ex.: Encontre a quantidade de clientes por cidade e rua os subtotais e o total geral.

```
SELECT nome_cidade, rua_cliente, COUNT(*)
FROM cliente
GROUP BY CUBE(nome_cidade, rua_cliente);
```

### 5.6.1.3 ListAgg

A função LISTAGG (list aggregate | lista agregada), é uma função SQL de grupo específica do Oracle. Foi incluída na versão 11g release2. É uma função analítica, que agrupa os dados na mesma linha dentro de uma instrução SELECT.

Pode ser utilizada sozinha para produzir conjuntos de dados agrupados por linhas simples, ou em conjunto com a função GROUP BY.

Ex.: Agrupe o nome dos funcionários, criando uma lista separada por ”;”.

```
SELECT LISTAGG(nome_funcionario, ';' )
WITHIN GROUP (ORDER BY nome_funcionario) NOME_FUNCIONARIOS
FROM funcionario;
```

Você pode utilizar a função LISTAGG com a função GROUP BY.

Ex.: Agrupe o nome dos funcionários por departamento, separando cada nome com ”;”.

```
SELECT nome_departamento,
LISTAGG(nome_funcionario, ';' ) WITHIN GROUP (
```

```
ORDER BY nome_funcionario) NOME_FUNCIONARIOS
FROM departamento, funcionario
WHERE departamento.codigo_departamento=funcionario.codigo_departamento
GROUP BY nome_departamento;
```

### 5.6.2 Mais funções

Olhar anexo Apêndice 2 item 9.5.

## 5.7 Valores Nulos

Podemos usar a palavra-chave `null` como predicado para testa a existência de valores nulos. Assim, para encontrar os números de empréstimos que aparecem na relação empréstimo com valores nulos para total, escrevemos:

```
SELECT numero_emprestimo
FROM emprestimo
WHERE total is null;
```

O predicado **is not null** testa a ausência de valores nulos.

Valores nulos foram criados a fim de indicar dados vazios. Imagine a seguinte situação, onde você deseja limpar um campo texto, para este campo você iria atribuir espaços em branco. Mas para um campo numérico você não pode atribuir espaços em branco pois não é número. Isto significa que não existe diferença entre um `null` em uma coluna numérica e um `null` de uma coluna de texto.

## 5.8 Subconsultas Aninhadas

A SQL proporciona um mecanismo para o aninhamento de subconsultas. Um subconsulta é uma expressão `select-from-where` aninhada dentro de uma outra consulta. As aplicações mais comuns para as subconsultas são testes para membros de conjuntos, comparações de conjuntos e cardinalidade de conjuntos.

### 5.8.1 Membros de Conjuntos

A SQL permite verificar se uma tupla é membro ou não de uma relação. O conectivo **in** testa se os membros de um conjunto, no qual o conjunto é a coleção de valores produzidos pela cláusula `select`. O conectivo `not in` verifica a ausência de membros de um conjunto.

Como exemplo, considere a consulta ‘encontre todos os clientes que tenham tanto conta quanto empréstimo no banco’. Podemos escrever tal consulta por meio de interseção de dois conjuntos: o conjunto dos correntistas do banco e o conjunto daqueles que contraíram empréstimo no banco.

Podemos optar por um enfoque alternativo para encontrar todos os correntistas que também contraíram empréstimos. Logicamente, essa formulação gera o mesmo resultado obtido anteriormente, mas leva-nos a escrever a consulta por meio do conectivo **in** do SQL. Começaremos por encontrar todos os correntistas, escrevendo esta subconsulta:

```
SELECT nome_cliente
FROM depositante;
```

Precisamos, agora, encontrar todos os clientes que contraíram empréstimos no banco e que aparecem na lista dos correntistas obtidos pela subconsulta. Faremos isso por meio do aninhamento da subconsulta em uma consulta externa. O resulta dessa forma é:

```
SELECT distinct nome_cliente
FROM devedor
WHERE nome_cliente IN (SELECT nome_cliente
                        FROM depositante);
```

Para ilustrar o uso do **not in** ‘encontre todos os clientes que tenham um empréstimo no banco, mas que não tenham uma conta no banco’:

```
SELECT distinct nome_cliente
FROM devedor
WHERE nome_cliente NOT IN (SELECT nome_cliente
                           FROM depositante);
```

Os operadores **in** e **not in** podem também ser usados em conjuntos enumerados. As consultas a seguir selecionam os nomes dos clientes que tenham um empréstimo no banco e cujos os nomes não sejam nem 'Smith' nem 'Jones'.

```
SELECT distinct nome_cliente
FROM devedor
WHERE nome_cliente NOT IN ('Smith', 'Jones');
```

### 5.8.2 Comparação de Conjuntos

Considere a consulta 'encontre os nomes de todas as agências que tenham fundos maiores que ao menos uma agência localizada no Brooklyn'. Anteriormente escreveríamos esta consulta da seguinte forma:

```
SELECT distinct T.nome_agencia
FROM agencia AS T, agencia AS S
WHERE T.fundos > S.fundos AND S.nome_cidade = 'Brooklyn';
```

A SQL, de fato, oferece alternativas de estilos para escrever a consulta precedente. A frase 'maior que ao menos uma' é representada em SQL por **> some**. Esse construtor permite-nos reescrever a consulta em uma forma que remonta intimamente sua formulação em português.

```
SELECT nome_agencia
FROM agencia
WHERE fundos > SOME(SELECT fundos
                    FROM agencia
                    WHERE nome_cidade = 'Brooklyn');
```

A SQL permite, além disso, comparações **< some**, **<= some**, **>= some**, **= some** e **<> some**. Verifique que **= some** é idêntico a **in**, enquanto, **<> some** não é a mesma coisa que **not in**.

Agora modificaremos ligeiramente nossa consulta. Encontraremos os nomes de todas as agências que tenham fundos maiores que cada uma das agências do Brooklyn. O construtor **> all** corresponde à frase 'maior que todos'. Usando esse construtor, escrevemos a consulta como segue:

```
SELECT nome_agencia
FROM agencia
WHERE fundos > ALL(SELECT fundos
                  FROM agencia
                  WHERE nome_cidade = 'Brooklyn');
```

Como ocorre para **some**, a SQL permite, além disso, comparações **< all**, **<= all**, **>= all**, **= all** e **<> all**. Como exercício, verifique que **= some** é idêntico a **in**, enquanto, **<> some** não é a mesma coisa que **not in**. Verifique que **<> all** é idêntica a **not in**.

Como outro exemplo de comparações de conjunto 'encontre a agência que tem o maior saldo médio'. Funções agregadas não podem ser agregadas em SQL. Então, o uso de **max(avg(...))** não é permitido. Em vez disso, nossa estratégia é a seguinte. Começamos escrevendo uma consulta para encontrar todos os saldos médios e depois aninhamos isso com uma subconsulta de uma consulta maior que descubra as agências para cada saldo médio maior ou igual a todos os saldos médios:

```
SELECT nome_agencia
FROM conta
GROUP BY nome_agencia
HAVING avg(saldo) >= ALL(SELECT avg(saldo)
                        FROM conta)
```

```
GROUP BY nome_agencia);
```

### 5.8.3 Verificação de Relações Vazias

A SQL possui meios de testar se o resultado de uma subconsulta possui alguma tupla. O construtor **exists** retorna o valor **true**(verdadeiro) se o argumento de uma subconsulta é não-vazio. Por meio do construtor **exists** podemos escrever a consulta ‘encontre todos os clientes que tenham tanto conta quanto empréstimo no banco’ de outro modo:

```
SELECT nome_cliente
FROM devedor
WHERE EXISTS (SELECT *
               FROM depositante
               WHERE depositante.nome_cliente = devedor.nome_cliente);
```

Podemos testar a não existência de tuplas na subconsulta por meio do construtor **not exists**.

### 5.8.4 Teste para a Ausência de Tuplas Repetidas

A SQL contém recursos para testar se uma subconsulta possui alguma tupla repetida em seu resultado. O construtor **unique** retorna o valor **true**(verdadeiro) caso o argumento da subconsulta não possua nenhuma tupla repetida. Usando o construtor **unique** podemos escrever a consulta ‘encontre todos os clientes que tenham somente uma conta na agência Perryridge’, como segue:

```
SELECT T.nome_cliente
FROM depositante AS T
WHERE UNIQUE (SELECT R.nome_cliente
               FROM conta, depositante AS R
               WHERE T.nome_cliente = R.nome_cliente AND
                     R.numero_conta = conta.numero_conta AND
                     Conta.nome_agencia = 'Perryridge');
```

Podemos testar a existência de tuplas repetidas em uma subconsulta por meio do construtor **not unique**.

### 5.8.5 Subconsulta Escalar

Uma expressão de subconsulta escalar é uma subconsulta contida na lista select para retornar um único valor.

```
SELECT fundos, (select max(fundos) from conta) maior
FROM conta;
```

## 5.9 Modificações no Banco de Dados

### 5.9.1 Remoção

Um pedido para remoção de dados é expresso muitas vezes do mesmo modo que uma consulta. Podemos remover somente tuplas inteiras; não podemos excluir valores de um atributo em particular. Em SQL, a remoção é expressa por:

```
DELETE FROM r
WHERE P;
```

em que **P** representa um predicado e **r**, uma relação. O comando **delete** encontra primeiro todas as tuplas **t** em **r** para as quais **P(t)** é verdadeira e então remove-las de **r**. A cláusula **where** pode ser omitida nos casos de remoção de todas as de **r**.

O comando **delete** opera somente uma relação.

Exemplos:



Remova todos os registros de contas do cliente Smith.

```
DELETE FROM depositante
WHERE nome_cliente = 'JOAO';
```

Remova todos os empréstimos com total entre 1.300 e 1.500 dólares.

```
DELETE FROM emprestimo
WHERE total BETWEEN 1300 AND 15000;
```

### 5.9.2 Inserção

Para inserir dados em relação podemos especificar uma tupla a ser inserida ou escrever uma consulta cujo resultado é um conjunto de tuplas a inserir. A inserção é expressa por:

Remova todos os registros de contas do cliente Smith.

```
INSERT INTO r(A1, A2, ..., AN)
VALUES (V1, V2, ..., VN);
```

Em que **r** é a relação **Ai** representa os atributos a serem inseridos e **Vi** os valores contidos nos atributos **Ai**.

Para inserir a informação que existe uma conta 25 na agência de Agencia13 e que ela tem um saldo de 1.99 dólares. Escrevemos:

```
INSERT INTO conta
VALUES(25, 'AGENCIA13', 1.99);
```

O seguinte comando é idêntico ao apresentado anteriormente:

```
INSERT into conta(NUMERO_CONTA,NOME_AGENCIA,SALDO)
VALUES(25, 'AGENCIA13', 1.99);
```

#### 5.9.2.1 Inserção mais complexa

A instrução insert pode conter subconsultas que definem os dados a serem inseridos:

```
INSERT INTO r_temp(A1, A2, ..., AN)
SELECT (A1, A2, ..., AN)
FROM r;
```

Com isto a instrução insert insere mais de uma tupla.

#### 5.9.2.2 Inserção com null

Na inserção também podemos utilizar a palavra-chave null para que se construir uma única expressão de insert.

No exemplo a seguir somente o nome do cliente é inserido.

```
INSERT INTO
cliente(nome_cliente, rua_cliente, nome_cidade, salario_cliente, cpf_cliente)
VALUES ('João da Silva', null, null, null, null);
```

Para preencher os outros atributos de cliente basta trocar null pelo valor desejado.

### 5.9.2.3 Inserção de Data

Para inserir datas no banco de dados é necessário especificar o formato utilizado pelo sistema gerenciador de banco de dados. As datas geralmente são consideradas como texto na inserção e convertidas para um tipo específico depois de armazenadas (Date). Por serem consideradas texto no SQL são inseridas entre aspas. No entanto, precisam ser formatadas para sua inserção no campo.

Portanto para inserir um cliente com sua respectiva data de nascimento, escrevemos:

```
INSERT INTO cliente(nome_cliente,
                    rua_cliente,
                    nome_cidade,
                    salario_cliente,
                    data_nascimento)
VALUES('FELIX', 'rua_cliente 1', 'FLORIANOPOLIS', 10000,
      to_date('01/02/1999', 'DD/MM/YYYY'));
```

Onde `to_date` é uma função do Oracle que converte o texto `'01/02/1999'` para uma data. É necessário consultar a documentação de cada SGBD para verificar a função e o formato da data. Na função `to_date` são especificados dois parâmetros. O primeiro parâmetro passado é a data a ser convertida e o segundo o formato da data. O segundo parâmetro `'DD/MM/YYYY'` especifica que DD refere-se ao dia com dois dígitos, MM ao mês com dois dígitos e YYYY ao ano com quatro dígitos. Para hora o formato pode ser `'HH:MM:SS'`, e data e hora juntos `'DD/MM/YYYY HH:MM:SS'`. No apêndice 2 item 9.5.3.2 existe outros formatos para as datas.

### 5.9.3 Atualização

Em determinadas situações, podemos querer modificar valores das tuplas sem, no entanto alterar todos os valores. Para esse fim, o comando **update** pode ser usado.

```
UPDATE r
SET A1 = V1, A2 = V2, ... NA = VN
WHERE p;
```

em `r` é a relação, `Ai` representa o atributo a ser atualizado e `Vi` o valor a ser atualizado no atributo `Ai`, e `P` um predicado.

Suponha que pagamento da taxa de juros anual esteja sendo efetuado e todos os saldos deverão ser atualizados de 5 por cento. Escrevemos:

```
UPDATE conta
SET saldo = saldo * 1.05;
```

O comando anterior é aplicado uma vez para cada tupla de conta.

Suponha agora que constas com saldo superior a 10 mil dólares recebam 6 por cento e juros, embora todas as outras recebam 5 por cento. Escrevemos

```
UPDATE conta
SET saldo = saldo * 1.06
WHERE saldo > 10000;
```

Para atualizar campos de data é necessário realizar as conversões da mesma forma que a inserção.

## 5.10 Transação

Transação é uma unidade atômica de trabalho que atua sobre um banco de dados. Uma transação pode ser constituída por uma ou mais operações de acesso à base de dados. Todas as operações devem ser bem-sucedidas, caso contrário os efeitos da transação devem ser revertidos.

Uma transação inicia com o primeiro comando SQL emitido para a base de dados e finaliza com os seguintes eventos:

COMMIT ou ROLLBACK

Comandos DDL executam commit automático  
Saída do Usuário  
Queda do Sistema

### 5.10.1 Commit

Uma transação bem-sucedida termina quando um comando **COMMIT** é executado. O comando **COMMIT** finaliza e efetiva todas as alterações feitas na base de dados durante a transação.

Grava uma transação no banco de dados.

Sintaxe:

```
COMMIT;
```

Exemplo:

Alteração de dados

```
SQL> UPDATE funcionario
2 SET      salario_funcionario = 3
3 WHERE    rg_funcionario = '1112';
1 linha atualizada.
```

Commit nos dados

```
SQL> COMMIT;
Commit completado.
```

### 5.10.2 RollBack

Se uma transação aborta antes de o comando **COMMIT** ser executado, a transação deve ser desfeita, isto é, todas as mudanças feitas durante a transação devem ser desconsideradas. O processo de recuperação automática que permite desfazer as alterações feitas contra a base é chamado **ROLLBACK**. O **ROLLBACK** retorna a situação dos objetos da base alterados na transação à mesma situação em que se encontravam no início da transação.

O **ROLLBACK** reverte os efeitos de uma transação como se ela nunca tivesse existido.

Restaura uma transação. Recupera o banco de dados para a posição que esta após o último comando **COMMIT** ser executado.

Sintaxe:

```
ROLLBACK;
```

Exemplo:

```
SQL> DELETE FROM funcionario;
14 linhas apagadas.
```

```
SQL> ROLLBACK;
Rollback completado.
```

### 5.10.3 Pontos de Transação

Marca um ponto de início de transação.

Sintaxe:

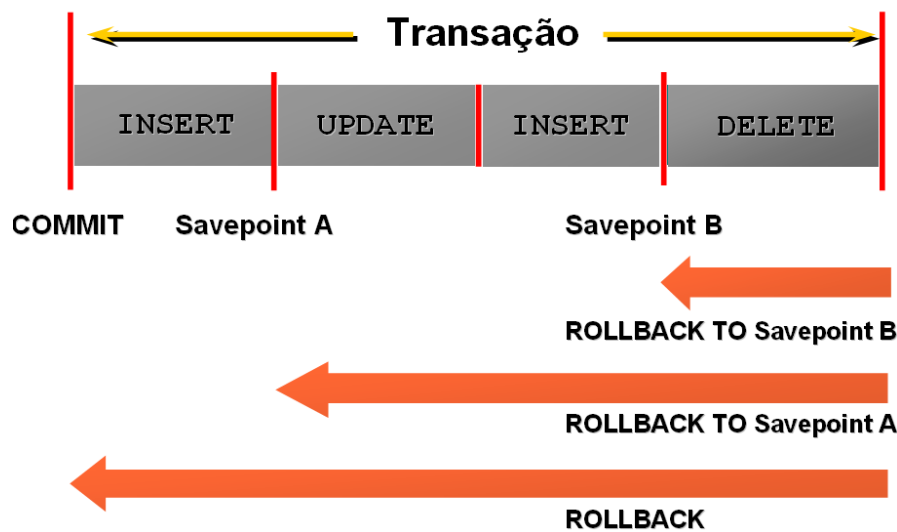
```
SAVEPOINT <nome-do-ponto>;
```

Executa um **ROLLBACK** até um ponto de transação salva. A Figura 5 mostra uma seqüência de utilização de savepoints.

Sintaxe:

```
ROLLBACK [TO SAVEPOINT <nome-do-ponto>;
```

Figura 5 – Usando SavePoint



#### 5.10.4 Locks

Quando um dado é acessado por uma transação, o perigo de interferência de outras transações acessarem o mesmo dado é constante. O gerenciamento de LOCKS ajuda a prevenir isto.

Locks são recursos de compartilhamento de dados, que permitem que o dado seja atualizado e pesquisados dentro de um ambiente multi-usuário de maneira segura e que lhes garante confiabilidade e integridade.

Através dos locks é possível garantir que somente um usuário esteja atualizando o dado em um determinado momento, que vários usuários possam pesquisar o mesmo dado ao mesmo tempo, etc.

Há, normalmente, independente da nomenclatura dada por fornecedores de SGBDs, dois níveis de locks importantes:

Lock Exclusivos: ou **XLOCKS (eXclusive Locks)**, usados para garantir o uso de um determinado dado por um único usuário. É especialmente utilizado em casos de atualizações.

Locks Compartilhados: ou **SLOCKS (Shared Locks)**, usados para permitir que mais de um usuário acesse o mesmo dado ao mesmo tempo. É especialmente utilizado em casos de consulta, por exemplo. Os SLOCKS garantem que um usuário possa consultar (e somente consultar) um dado se houver outros usuários acessando o mesmo dado para consulta.

Quando um usuário adquire um nível de lock sobre um dado, ele pode requisitar outro nível ao SGBD. A este conceito damos o nome de promoção de níveis de LOCK.

Imagine um usuário em nível SLOCK sobre um registro. Para alterar o registro, será necessário que adquira XLOCK, para depois poder alterá-lo. Isto só será possível se mais nenhum usuário estiver atuando sobre o mesmo registro, seja em SLOCK ou XLOCK. Isto se explica da seguinte forma, se fosse possível a qualquer usuário alterar valores de registros em uso por parte de outros usuários, a informação consultada nunca seria garantidamente segura, uma vez que a qualquer momento poderia ser alterada, inclusive no exato momento de um processamento, quando iniciaria com um valor e terminaria com outro valor, alterado por outro usuário.

A tabela abaixo mostra as transições de níveis de LOCK possíveis:

Quadro 6 - Transições de níveis de Lock

Corrente	Requisitado	
	<i>SLOCK</i>	<i>XLOCK</i>
<i>SLOCK</i>	OK	X
<i>XLOCK</i>	OK	X

A Tabela a seguir mostra a compatibilidade de Locks entre transações, sendo a transação T1 requisita o lock em concorrência a outra transação T2.

**Quadro 7 - Compatibilidade de Locks**

T1	T2	
	<i>SLOCK</i>	<i>XLOCK</i>
-	OK	OK
<i>SLOCK</i>	OK	X
<i>XLOCK</i>	X	X

### 5.10.5 Estado dos Dados

São os momentos antes de executar o commit ou rollback ao final de uma transação.

#### 5.10.5.1 Antes do Commit e Rollback

O estado anterior do dado pode ser recuperado.

Outros usuários não podem ver as alterações efetuadas.

As linhas afetadas pela transação são bloqueadas (locked) até se completar a transação.

#### 5.10.5.2 Após o Commit e Rollback

Os dados são alterados definitivamente na base de dados.

valor anterior dos dados não são recuperados.

Todos os usuários vêem o mesmo resultado.

Os bloqueios (locks) são desfeitos, liberando os dados para os outros usuários.

Todos os Savepoints são eliminados.

### 5.10.6 Rollforward

Quando da reinicialização de um banco de dados após uma falha, algumas transações podem ter sido perdidas na memória, embora um comando Commit já tenha sido emitido. Isto quer dizer que no log de transações a transação é considerada completa. No entanto, os efeitos não foram registrados em definitivo na base de dados.

Isto quer dizer que esta transação deve ser completada novamente. Usando o log de transações o SGDB "sabe" que partes da transação ainda não foram gravados em definitivo. As etapas que ainda faltam ser gravadas são então executadas até ser encontrado o comando COMMIT gravado no log de transações. A este processo chamamos refazer a transação ou rollforward.

Diferente do rollback, que pode ser executado por uma aplicação ou no processo de reinicialização do banco de dados, o rollforward pode ser apenas executado quando do processo de reinicialização do banco de dados.

### 5.10.7 Syncpoint

Embora alguns SGDBs permitam que cada transação completada seja imediatamente gravada na base, este é o melhor meio de se efetivar transações, devido à sobrecarga de gravação contra o banco, o que ocasiona uma queda de performance.

Cada operação que constitui uma transação pode ser mantida em memória (gerenciada por páginas, gerenciadores de cache, etc) e registrada contra o log de transações, sendo gravada contra a base em intervalos de tempo pré-determinados. Quando terminado o intervalo, todos os efeitos da alterações mantidos em memória são gravados efetivamente contra a base, sem prejuízo do gerenciamento de lock em curso. A este "alarme" que permite disparar a efetivações contra a base chamamos syncpoint.

Os syncpoints sincronizam log de transações, base de dados e memória.

## 5.11 Índice

A SQL-padrão não provê nenhuma maneira de o usuário ou administrador de banco de dados controlarem quais índices são criados e mantidos no sistema de banco de dados. Os índices não são necessários por questões de precisões, pois são estruturas de dados redundantes. Entretanto, os índices são

importantes para o processamento eficiente de transações, incluindo transações de atualizações e consultas. Os índices também são importantes para a implementação eficiente das restrições de integridade. Por exemplo, implementações típicas obrigam uma declaração de chave por meio da criação de índice com a chave declarada, como chave de procura do índice.

Em princípio, o sistema de banco de dados pode decidir automaticamente quais índices criar. Porém, devido ao custo de espaço dos índices, assim como o efeito dos índices no processamento de atualizações, não é fácil fazer automaticamente a escolha correta de quais índices manter. Então, a maioria das implementações de SQL permite que o programador controle a criação e a remoção de índices por meio de comandos da linguagem para definição de dados.

### 5.11.1 Criando Índice

Um índice é criado pelo comando `create index`, que tem a seguinte forma:

```
CREATE [UNIQUE] INDEX
<nome_do_indice> ON <nome_da_tabela> (lista_de_atributos);
```

A `lista_de_atributos` é a lista de atributos das relações que forma a chave de procura para o índice.

Para definir um índice chamado **índice\_b** na relação **agencia** com **nome\_agencia** como chave de procura escrevemos:

```
CREATE INDEX indice_b on agencia (nome_agencia);
```

Se desejamos declarar que a chave de procura é uma chave candidata, adicionamos o atributo **unique** à definição do índice. Assim o comando:

```
CREATE unique INDEX indice_b on agencia (nome_agencia);
```

declara `nome_agencia` para ser uma chave candidata para `agencia`. Se, na ocasião em que o comando `create unique index` for solicitado, `nome_agencia` não for uma chave candidata, uma mensagem de erro será exibida, e a tentativa de criar o índice falhará. Se a tentativa de criação de índice for bem-sucedida, qualquer tentativa subsequente de inserir uma tupla que viole a declaração da chave falhará. Observe que a característica `unique` é redundante se o atributo for chave primária.

### 5.11.2 Apagando Índice

O índice pode ser removido usando o comando `DROP INDEX`.

```
DROP INDEX <nome_do_indice>;
```

## 5.12 Visões

Visões ou *View* é uma tabela virtual ou lógica que permite a visualização e manipulação do conteúdo de uma ou mais tabelas. Uma visão tem a aparência e o funcionamento parecido com a de uma tabela normal, só que as suas colunas podem ser oriundas de diversas tabelas diferentes. As tabelas que dão origem às colunas são chamadas de tabelas-base.

Um dos objetivos do uso de *views* é restringir o acesso a certas porções dos dados por questões de segurança, além de pré-definir (armazenar) certas consultas através de tabelas virtuais que poderão ser utilizadas por outras consultas.

Alguns bancos de dados trabalham com dois tipos de *views*. O Oracle trabalha com *views* “normais” e *views* “materializadas”.

### 5.12.1 Visões Normais

A View Normal é “materializada” no momento da consulta. Com isto não é necessário fazer cópias dos dados. A consulta à *view* é resolvida sobre as próprias tabelas originais, o que minimiza qualquer perda de desempenho ou espaço

Definimos uma visão em SQL usando o comando **create view**. Para definir a visão, precisamos dar-lhe um nome e definir a consulta a consulta que processará essa visão. A forma do comando **create view** é:

```
CREATE [OR REPLACE] VIEW <nome_da_visão>
AS <expressão_da_consulta>;
```

em que <expressão\_da\_consulta> é qualquer expressão de consulta válida e o nome da visão é representado por <nome\_da\_visão>. A opção **or replace** recria uma visualização já existente.

Como exemplo, considere uma visão composta dos nomes de agências e nomes de clientes que tenham uma conta ou um empréstimo na agência. Suponha que desejamos que essa visão seja denominada **VW\_Todos\_Clientes**. Definimos essa visão como segue:

```
CREATE VIEW VW_Todos_Clientes AS
( SELECT nome_agencia, nome_cliente
  FROM depositante, conta
  WHERE depositante.numero_conta = conta.numero_conta)
UNION
( SELECT nome_agencia, nome_cliente
  FROM devedor, emprestimo
  WHERE emprestimo.numero_emprestimo = devedor.numero_emprestimo)
```

Os nomes dos atributos de visão devem ser especificados explicitamente, conforme segue:

```
CREATE VIEW VW_Emprestimo_total_agencia (nome_agencia, emprestimo_total) AS
  SELECT nome_agencia, sum(saldo)
  FROM emprestimo
  GROUP BY nome_agencia
```

A visão anterior cede para cada agência a soma dos totais de todos os empréstimos da agência. Uma vez que a expressão soma (total) não possui nome, o nome do atributo é especificado explicitamente na definição da visão.

Os nomes de visão podem aparecer em qualquer lugar onde o nome de uma relação aparece. Usando a visão **VW\_Todos\_clientes**, podemos encontrar todos os clientes da agência Perryridge, escrevendo:

```
SELECT nome_cliente
FROM VW_Todos_clientes
WHERE nome_agencia = 'Perryridge'
```

Consultas complexas são mais fáceis de escrever e entender se quebrarmos em visões menores para depois combiná-las, assim como estruturamos programas por meio da divisão de suas tarefas em procedimentos. Entretanto, ao contrário da definição de um procedimento, a cláusula **create view** cria uma definição de visão em um banco de dados e essa definição fica no banco até que um comando **drop view nome\_visão** seja executado.

### 5.12.2 Visões Materializadas

As Views Materializadas, são views que a cada requisição ou chamada acessa dados em tabelas virtuais gerenciadas pelo banco de dados, aos quais são previamente otimizadas para que o retorno dos dados seja feita de forma mais otimizada, para dados que precisam sofrer algum processamento (cálculo) no SGBD. Esses dados são atualizados sob demanda, ou seja, quando solicitado pelo usuário ou quando o mesmo é programado para que seja de forma automática. Indicado para ambientes que trabalham com Data Warehouse pois são utilizados principalmente para consultas de grandes volumes de dados. No Oracle são conhecidos também como SNAPSHOTS pois são fotos de dados de tabelas em um determinado instante do tempo.

A sintaxe para criar *views* materializadas é:

```
CREATE MATERIALIZED VIEW <nome_da_visão>
  BUILD [IMMEDIATE | DEFERRED]
  REFRESH [FAST | COMPLETE | FORCE ]
  ON [COMMIT | DEMAND ]
  [[ENABLE | DISABLE] QUERY REWRITE]
AS <expressão_da_consulta>;
```

em que <expressão\_da\_consulta> é qualquer expressão de consulta válida e o nome da visão é representado por <nome\_da\_visão>. Visões materializadas precisam ser apagadas para serem modificadas, portanto a opção OR REPLACE não funciona.

A cláusula BUILD possui as opções: IMMEDIATE onde a *view* é preenchida imediatamente após a sua criação ou a opção DEFERRED onde ela é preenchida na primeira atualização solicitada. O padrão é IMMEDIATE.

A cláusula REFRESH possui as opções: FAST, onde somente alterações realizadas no intervalo de tempo são atualizadas na *view*, COMPLETE, recria toda a estrutura da *view* materializada mesmo que não seja necessário e FORCE, que faz o FAST ser for possível, caso contrário faz o processo COMPLETE. O padrão é FORCE.

A atualização pode ser acionada de duas maneiras, ON COMMIT onde a atualização é desencadeada por uma alteração de dados em uma das tabelas base. Com isto você não precisa lembrar de atualizar as *views* materializadas, ON DEMAND onde a atualização é iniciada por um pedido manual(DBMS\_MVIEW) ou uma tarefa agendada ou START WITH usando uma data e hora para calcular a próxima atualização.

A última opção é a cláusula QUERY REWRITE esta opção indica que o SELECT presente dentro da *view* Materializada será rescrito e atualizado para os novos valores passados pela *view*. Por default é DISABLE.

Reescrevendo a visão normal escrita anteriormente para uma visão materializada temos:

```
CREATE MATERIALIZED VIEW VWM_emprestimo_total_agencia AS
  SELECT nome_agencia, sum(total) emprestimo_total
  FROM emprestimo
  GROUP BY nome_agencia;
```

Este comando equivale:

```
CREATE MATERIALIZED VIEW VWM_EMPRESTIMO_TOTAL_AGENCIA
  BUILD IMMEDIATE
  REFRESH FORCE
  ON DEMAND
  DISABLE QUERY REWRITE AS
  SELECT nome_agencia, sum(total) emprestimo_total
  FROM emprestimo
  GROUP BY nome_agencia;
```

Para listar as visões materializadas criadas e o seu estado use o comando:

```
SELECT mview_name, staleness
FROM User_Mviews;
```

O campo Staleness indica se a visão foi atualizada ou não, podendo ser NEEDS\_COMPILE que precisa de compilação ou FRESH atualizada.

A atualização de visões materializadas e realizada através do comando:

```
BEGIN
  DBMS_MVIEW.REFRESH('VWM_emprestimo_total_agencia');
END;
```



Para remover uma visão materializada é necessário utilizar o comando **drop materialized view nome\_visão**.

### 5.12.3 Atualizações de uma Visão

A atualização de uma visão é mais complicada quando uma visão é definida em termos de diversas relações. Como resultado, muitos dos sistemas de banco de dados impõem algumas regras para alterações por meio de visões:

Uma alteração por meio de visão somente é permitida se a visão em questão é definida em termos de uma relação real do banco de dados isto é, no nível lógico do banco de dados.

### 5.12.4 Metadados

Olhar anexo Apêndice 1 item 8.7.

## 5.13 Seqüências

As seqüências são usadas para facilitar o processo de criação de identificadores únicos de um registro em um banco de dados. Uma seqüência nada mais é do que um contador automático incrementado toda vez que é acessado. O número gerado pelo contador pode ser usado para atualizar um campo do tipo código do produto ou código do cliente em uma tabela, garantido que não haja duas linhas com o mesmo código.

Quando uma seqüência é criada, ela adota alguns valores-padrão que são adequados para a maioria dos casos. Uma seqüência padrão tem as seguintes características:

- Começa sempre a partir do número 1
- Tem ordem ascendente
- É aumentada em 1

O comando SQL usado para a criação de uma seqüência é o comando **create sequence**.

Sintaxe básica:

```
CREATE SEQUENCE <nome_da_sequencia>
[start with <numero_inicio>]
[increment by <numero_incremento>]
[minvalue <numero_minimo>]
[nominvalue]
[maxvalue <numero_maximo>]
[nomaxvalue]
[cycle]
[nocycle]
[cache]
[nocache];
```

**start with** – Indica o valor inicial que a seqüência deve ter, ou seja a primeira vez que for utilizada, ela retornará o valor especificado por essa cláusula.

**increment by** – Indica o valor pelo qual a seqüência será aumentada toda vez que for acessada.

**minvalue** – Indica o valor mínimo que a seqüência poderá ter.

**nominvalue** – Indica que a seqüência não tem valor mínimo predefinido.

**maxvalue** – Indica o valor máximo que a seqüência poderá ter.

**nomaxvalue** – Indica que a seqüência não tem valor máximo predefinido.

**cycle/nocycle** – Cycle indica que, ao atingir o valor máximo, a seqüência deve retornar ao valor inicial. Por sua vez nocycle impede que a seqüência volte ao seu ciclo.

**cache/nocache** – Utilize para obter acesso mais rápido a partir das suas seqüências, se não puder dar-se ao luxo de perder um número de seqüência de vez em quando. Esse parâmetro faz com que o Oracle armazene as seqüências em cachê na memória para acesso mais rápido. O parâmetro padrão, nocache, desativa este recurso.

### 5.13.1 Usando Seqüências

Uma vez que um número de seqüência é utilizado, ele fica disponível apenas para a sessão em que foi criado.

Para obter o valor atual de uma seqüência, sem incrementar o seu valor uma pseudocoluna chamada **CURRVAL** deve ser usada. Para obter e incrementar o valor da seqüência, uma pseudocoluna chamada **NEXTVAL** deve ser usada.

Essas duas pseudo-colunas podem ser usadas nas seguintes ocasiões:

Em um comando INSERT, como valor da cláusula VALUES.

Na cláusula SET do comando update.

Na lista de colunas do comando SELECT. No caso do comando SELECT, as pseudocolunas não podem ser usadas se o comando possuir as cláusulas ORDER BY, GROUP BY, DISTINCT ou uma subquery.

### 5.13.2 Criando uma Seqüência

Uma seqüência somente está disponível para o esquema que a criou. Todas as seqüências criadas nesses exemplos são feitas sob um usuário. O próximo exemplo ilustra a criação da seqüência `sq_departamento` usando os valores padrão.

```
CREATE SEQUENCE sq_departamento;
```

Esta seqüência começa com 1 e é incrementada em 1. Não tem valor mínimo e máximo definido e do tipo **nocycle**.

```
CREATE SEQUENCE sq_teste start with 100 increment by 25;
```

Esta seqüência começa com 100 e é incrementada em 25. Não tem valor mínimo e máximo definido e do tipo **nocycle**.

### 5.13.3 Apagando uma Seqüência

O comando DROP SEQUENCE remove a seqüência do esquema no qual foi criado.

```
DROP SEQUENCE sq_departamento;
```

### 5.13.4 Alterando uma Seqüência

Através do comando ALTER SEQUENCE, o usuário pode mudar alguns dos parâmetros da seqüência. Contudo, deve observar as restrições que existem para que uma alteração seja feita.

Não é permitido alterar o valor inicial da seqüência.

O valor mínimo da tabela não pode ser maior do que o valor atual.

O valor de incremento pode ser alterado sem problemas.

### 5.13.5 Usando uma Seqüência

Pela primeira vez, a seqüência é acionada e traz o seu valor inicial.

```
SELECT sq_departamento.nextval FROM dual;
```

Para ver o valor corrente na seqüência sem incrementar o seu valor utilize a pseudo-coluna `currval`.

```
SELECT sq_departamento.currval FROM dual;
```

Para incluir um registro considerando a relação departamento(codigo\_departamento, nome) usa-se da seguinte forma:

```
INSERT into DEPARTAMENTO VALUES(sq_departamento.nextval, 'ESTOQUE');
INSERT into DEPARTAMENTO VALUES(sq_departamento.nextval, 'FINANÇAS');
```

Duas linhas foram incluídas em departamento e não foi preciso se preocupar com o código do departamento.

Obs. Uma sequência não está associada a uma tabela. Ela pode ser usada por outras tabelas.

#### 5.13.6 Metadados

Olhar anexo Apêndice 1 item 8.8.

### 5.14 Triggers

Um gatilho é um comando que é executado pelo sistema automaticamente, em consequência de uma modificação no banco de dados. Duas exigências devem ser satisfeitas para a projeção de um mecanismo de gatilho:

Especificar as condições sob as quais o gatilho deve ser executado.

Especificar as ações que serão tomadas quando um gatilho for disparado.

Os gatilhos são mecanismos úteis para avisos a usuários ou para executar automaticamente determinadas tarefas quando as condições para isso são criadas.

#### 5.14.1 Tipos de Trigger

Existem dois tipos distintos de trigger que podem ser usados em uma tabela.

**Statement-level- Trigger** Essa trigger é disparado apenas uma vez. Por exemplo, se o comando update atualizar 15 linhas, os comandos contidos na trigger serão executados uma única vez. Também chamado de trigger em nível de instrução.

**Row-level- Trigger** Essa trigger tem os seus comandos executados para todas as linhas que sejam afetadas pelo comando que gerou o acionamento do trigger. Também chamado de trigger em nível da linha.

#### 5.14.2 Componentes de uma Trigger

Uma trigger é composta por três partes.

**Comando SQL que aciona a trigger.** Uma trigger pode ser ativada pelos comandos INSERT, DELETE e UPDATE. Uma mesma trigger pode ser invocada quando mais de uma ação ocorrer, ou seja, uma trigger pode ser invocada somente quando um comando INSERT for executado, ou então quando um comando UPDATE ou DELETE for executado.

**Limitador de ação da trigger.** Representado pela cláusula WHEN, especifica qual a condição deve ser verdadeira para que a trigger seja disparada.

**Ação executada pelo trigger.** É o bloco de sql que é executado pela trigger.

#### 5.14.3 Momento de disparo de uma trigger

Uma trigger pode ser disparado antes(BEFORE), ou depois(AFTER) que um dos comandos de ativação (INSERT, UPDATE, DELETE) for executado. Uma tabela pode conter até 12 trigger associados aos comandos de ativação e momento de disparo. São as seis trigger do tipo **row-level** e seis do tipo **statement-level**.

```
BEFORE INSERT
AFTER INSERT
BEFORE DELETE
AFTER DELETE
BEFORE UPDATE
AFTER UPDATE
```

#### 5.14.4 Criação de uma Trigger

Uma trigger pode ser criada através do comando SQL CREATE TRIGGER. Vejamos a sintaxe do comando SQL responsável pela criação de triggers.

```
CREATE [OR REPLACE] TRIGGER <nome_da_trigger>
```

```
{BEFORE|AFTER} Comando_de_disparo [OF <nome_da_coluna>] ON <nome_da_tabela>
[FOR EACH ROW]
[WHEN (condição)]
[Declare
    [Declarações]]
BEGIN
    Comandos;
END;
```

**OR REPLACE.** Permite alterar a trigger sem que seja necessário apagá-lo previamente.

**Nome\_da\_trigger.** Nome de identificação da trigger.

**Comando\_de\_disparo.** É o nome do comando que ativa a trigger (INSERT, DELETE, UPDATE)

**OF nome\_da\_coluna.** É a lista de nomes dos campos na tabela que podem disparar a trigger.

**ON nome\_da\_tabela.** Indica a tabela ou esquema para qual a trigger está sendo criada.

**FOR EACH ROW.** Indica que a trigger é do tipo **row-level** e deve ser executado para todas as linhas selecionadas.

**WHEN(condição).** Especifica a restrição da trigger. Uma condição SQL que precisa ser atendida para disparar a trigger.

**Declarações.** Nessa seção são declaradas constantes, variáveis com a utilização de **declare**.

**Comandos.** Nessa seção são colocados os comandos que serão executados pela trigger.

#### 5.14.5 Ativação/Desativação de uma Trigger

Quando uma trigger é criada, ela fica automaticamente ativa, sendo disparado toda vez que o comando de disparo e condição de execução for verdadeiro. Para ativar ou desativar a execução de uma trigger, deve ser usado o comando ALTER TRIGGER com a cláusula DISABLE. Para reativá-lo, deve ser usada a cláusula ENABLE.

```
ALTER TRIGGER <nome_da_trigger> DISABLE;
```

#### 5.14.6 Alteração de uma Trigger

Uma trigger não pode ser alterada diretamente. Na verdade a única opção de alterar uma trigger é sua recriação usando a opção OR REPLACE do comando CREATE. Se uma trigger teve privilégios cedidos para outros usuários, eles permanecem válidos enquanto o trigger existir.

#### 5.14.7 Remoção de uma Trigger

Para apagar uma trigger, o seguinte comando deve ser usado:

```
DROP TRIGGER <nome_da_trigger>;
```

#### 5.14.8 Limitações de Uso de Trigger

Uma trigger não pode executar os comando COMMIT ou ROLLBACK nem tão pouco chamar procedures ou funções que executem essas tarefas.

Uma trigger do tipo row-level não pode ler ou modificar o conteúdo de uma tabela em mutação. Uma tabela mutante é aquela na qual seu conteúdo está sendo alterado por um comando INSERT, DELETE e UPDATE, e o comando não foi terminado, ou seja, ainda não foram gravados com COMMIT.

#### 5.14.9 Referência a Colunas dentro de uma Trigger

Dentro de uma trigger do tipo row-level é possível acessar o valor de um campo de uma linha. Dependendo da operação que está sendo executada é necessário preceder o nome da coluna com o sufixo :new ou :old.

Para um comando INSERT, os valores dos campos que serão gravados devem ser precedidos pelo sufixo :new.

Para um comando DELETE, os valores dos campos da linha que está sendo processada devem ser precedidos do sufixo :old.

Para um comando UPDATE, o valor original que está sendo gravado é acessado com o sufixo :old. Os novos valores que serão gravados devem ser precedidos do sufixo :new.

#### 5.14.10 Predicados Condicionais

INSERTING  
UPDATING  
DELETING

Exemplo:

```
CREATE OR REPLACE TRIGGER TOTAL_SALARIO_POR_DEPARTAMENTO
AFTER DELETE OR INSERT OR UPDATE OF codigo_departamento, salario_funcionario
ON funcionario
FOR EACH ROW
BEGIN
    IF DELETING THEN
        UPDATE departamento
        SET total_salario = total_salario - :OLD.salario_funcionario
        WHERE codigo_departamento = :OLD.codigo_departamento;
    ELSIF INSERTING THEN
        UPDATE departamento
        SET total_salario = total_salario + :NEW.salario_funcionario
        WHERE codigo_departamento = :NEW.codigo_departamento;
    ELSIF UPDATING THEN
        UPDATE departamento
        SET total_salario = total_salario +
            ( :NEW.salario_funcionario - :OLD.salario_funcionario )
        WHERE codigo_departamento = :OLD.codigo_departamento;
    END IF;
END;
```

#### 5.14.11 Metadados

Olhar anexo Apêndice 1 item 8.9.

### 5.15 Stored Procedures

Uma stored procedure é um grupo de comandos SQL, que executa uma determinada tarefa. Diferente de uma trigger, que é executado automaticamente, uma procedure precisa ser chamada a partir de um programa ou manualmente pelo usuário. O uso de stored procedure traz uma série de benefícios.

**Reusabilidade de código** – Uma procedure pode ser usada por diversos usuários em diversas ocasiões, como um script SQL, dentro de uma trigger ou aplicação.

**Portabilidade** – Uma procedure é totalmente portátil dentro de plataformas nas quais o oracle roda.

**Aumento de Performance** – Guardar a lógica de uma aplicação no próprio banco de dados diminui o tráfego na rede em um ambiente cliente/servidor.

**Manutenção centralizada** – Mantendo o código no servidor, uma alteração feita é imediatamente disponibilizada para todos os usuários. Isso evita o controle de versões de programas que são distribuídos pela empresa.

#### 5.15.1 Sintaxe de Stored Procedure

Uma stored procedure possui duas partes. Uma seção de especificação e o corpo da procedure. Vejamos a sintaxe do comando SQL responsável pela criação de procedures.

```
CREATE [OR REPLACE] PROCEDURE <nome_da_procedure>
[ ( lista de parâmetros ) ] IS
[ declarações ]
BEGIN
    Comandos;
```

```
END [nome_da_procedure];
```

**OR REPLACE** – Essa opção recria a função mantendo os privilégios previamente concedidos

**Lista de parâmetros** – Se mais de um parâmetro for usado pela procedure devem ser separados por vírgula. Um parâmetro deve ser definido com a cláusula IN e inicializado como uma variável.

**Declarações** – Nessa seção são declaradas constantes, variáveis e até mesmo outras procedures e funções locais.

**Comandos** – Nessa seção são colocados os comandos que serão executados pela procedure.

Uma stored procedure pode executar determinadas tarefas sem que nenhuma informação seja passada para ela ou utilizar parâmetros (valores) que serão usados pela procedure na execução de sua tarefa. Por exemplo, pode-se criar uma stored procedure que atualize a tabela de empregados aumentando o salário de todos em 10% ou criar uma procedure que use dois parâmetros. Um seria o código do departamento e o outro o percentual de aumento. Nesse caso, apenas os funcionários do departamento especificado receberão o aumento indicado. Assim como triggers uma procedure pode ser criada através de um editor de textos e Ter seu conteúdo inserido no buffer do SQL \*Plus e também através do Navigator.

### 5.15.2 Criando uma Stored Procedure

No próximo exemplo, vamos criar uma stored procedure que aumenta o salário de todos os funcionários por um valor fixo de 10%. Essa stored procedure não exige nenhum parâmetro. Usando o SQL\*Plus, digite o seguinte texto:

```
CREATE OR REPLACE PROCEDURE Aumenta_Salario
IS
BEGIN
    UPDATE Funcionario
    SET salario_funcionario = salario_funcionario *1.1;
END;
```

Como resposta de sucesso na criação tem-se:

```
Procedimento criado.
```

### 5.15.3 Mostrando Erros

Se ocorrer algum erro você pode usar o comando SHOW ERRORS para saber o que aconteceu.

```
SHOW ERRORS;
```

### 5.15.4 Executando uma Stored Procedure

A execução de uma procedure, é feita através de uma chamada ao seu nome.

Dentro de um trigger ou outra stored procedure basta especificar o nome da procedure e seus parâmetros.

```
EXECUTE Aumenta_Salario;
```

Como resposta tem-se:

```
Procedimento PL/SQL concluído com sucesso.
```

### 5.15.5 Apagando uma Stored Procedure

Para apagar uma stored procedure deve ser usado o comando Drop Procedure.

Sintaxe Básica:

```
DROP PROCEDURE <nome_da_procedure>;
```

Exemplo:

```
DROP PROCEDURE Aumenta_Salario;
```

### 5.15.6 Passando Argumentos

Criar uma stored procedure para atualizar o salário de um determinado funcionário, o argumento podem ser do tipo básico de dados ou %TYPE. Se existir mais argumentos estes deve ser separados por vírgula.

```
CREATE OR REPLACE PROCEDURE Aumenta_Salario
    (Argumento Funcionario.RG_funcionario%Type)
IS
BEGIN
    UPDATE Funcionario
    SET salario_funcionario = salario_funcionario * 1.1
    WHERE rg_funcionario = Argumento;
END;
```

Como resposta de sucesso na criação tem-se:

Procedimento criado.

Para executa a procedure:

```
EXECUTE Aumenta_Salario('1112');
```

Agora é necessário especificar que funcionário você quer aumentar salário.

### 5.15.7 Metadados

Olhar anexo Apêndice 1 item 8.10.

## 5.16 Funções

Uma função é muito parecida com uma stored procedure. A principal diferença está no fato que uma função retorna um valor e a stored procedure não. Outra diferença é a forma como uma função pode ser chamada. Por retorna um valor, ela pode ser chamada através de um comando SELECT e também usada em cálculos como outra função qualquer do Oracle.

### 5.16.1 Sintaxe de Função

```
CREATE [OR REPLACE] FUNCTION <nome_da_function>
    [ ( lista de parâmetros ) ]
    RETURN Tipo_de_Dado IS
[ declarações ]
BEGIN
    Comandos
    RETURN Valor_da_Função;
END [nome_da_function];
```

**OR REPLACE** – Essa opção recria a função mantendo os privilégios previamente concedidos

**Lista de parâmetros** – Se mais de um parâmetro for usado pela função devem ser separados por vírgula. Um parâmetro deve ser definido com a cláusula IN e inicializado como uma variável.

**Declarações** – Nessa seção são declaradas constantes variáveis e até mesmo outras procedures e funções locais.

**Comandos** – Nessa seção são colocados os comandos que serão executados pela procedure.

**Valor\_da\_Função** – É uma constante ou variável que contém o valor retornado pela função.

### 5.16.2 Criando uma Função

Vamos criar uma função que retorna o número(quantidade) de funcionários de um determinado departamento. Para isso, ela usará um argumento que recebe o código do departamento.

```
CREATE OR REPLACE FUNCTION QtdeFuncionario(argdepartamento int)
RETURN number IS
    totalFuncionario int;
BEGIN
    SELECT COUNT(*) into totalFuncionario
    FROM Funcionario
    WHERE codigo_departamento = argdepartamento;
    RETURN totalFuncionario;
END;
```

Como resposta de sucesso na criação tem-se:

Função criada.

### 5.16.3 Executando uma Função

Para exibir o resultado de uma função, deve ser usado o comando SELECT especificando o nome da função e a tabela DUAL deve ser pesquisada. Veja o exemplo.

```
SELECT QtdeFuncionario(3)
FROM DUAL;
```

Vamos selecionar a quantidade de funcionários por departamento que seja maior que a quantidade do departamento 2;

```
SELECT count(*), codigo_departamento
FROM funcionario
GROUP BY codigo_departamento
HAVING count(*) > QtdeFuncionario (3);
```

### 5.16.4 Apagando uma Função

Para apagar uma função deve ser usado o comando Drop Function.

Sintaxe Básica:

```
DROP FUNCTION <nome_da_function>;
```

Exemplo:

```
DROP FUNCTION QtdeFuncionario;
```

### 5.16.5 Metadados

Olhar anexo Apêndice 1 item 8.10.

## 5.17 Cursores

Se um bloco PL/SQL, trigger ou stored procedure usa um comando SELECT que retorna mais de uma linha, o Oracle exibe uma mensagem de erro que invoca a exceção `TOO_MANY_ROWS`. Para contornar esse problema o Oracle usa um mecanismo chamado cursor. Desassocie imediatamente o nome Cursor com o ponteiro do mouse que aparece na tela.

Um cursor pode ser visto como um arquivo temporário que armazena e controla as linhas retornas por um comando SELECT. O SQL-Plus gera automaticamente cursores para as queries(consultas) executadas. Já no PL/SQL é necessário que o usuário crie cursores específicos.



### 5.17.1 Criando um Cursor

A criação de um cursor envolve quatro etapas descritas a seguir.

1. Declaração do cursor. Cria um nome para o cursor e atribui um comando SELECT a ele.
2. Abertura do cursor. Executa a query associada ao cursor e determina quantas linhas serão retornadas.
3. Fetching. As linhas (conteúdo) encontradas são enviadas para o programa PL/SQL.
4. Fechamento do Cursor. Libera os recursos alocados para o cursor.

### 5.17.2 Declaração de um Cursor

Na declaração do cursor, você deve especificar o nome e a query que será executada. Essa tarefa é feita na seção de declaração de um bloco. No próximo exemplo, um cursor chamado CRS2 é criado com uma query que seleciona os funcionários do departamento 2.

Exemplo:

```
DECLARE
    CURSOR CRS2 IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = 2
        ORDER BY nome_funcionario;
```

Não existem limitações para a quantidade de cursores criados, a não ser pela memória alocada para os cursores. Assim como uma procedure ou função, um cursor pode receber parâmetros.

Exemplo:

```
DECLARE
    CURSOR CRS2(temp_depto INT) IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = temp_depto
        ORDER BY nome_funcionario;
```

### 5.17.3 Abertura de um Cursor

A abertura do cursor é a operação que executa a query e cria o active set(conjunto ativo), que é o grupo de linhas que satisfaz a condição da query. Um cursor é aberto pelo comando Open.

Exemplo:

```
OPEN CRS2;
```

O cursor deixa a primeira linha como linha atual.

### 5.17.4 Atributos de um Cursor

Um cursor possui quatro atributos que podem ser usados para manipulá-lo.

**%ISOPEN** – Esse atributo retornará True se o cursor já estiver aberto.

**%NOTFOUND** – Retorna True quando a última linha do cursor é processada e nenhuma outra está disponível. É equivalente a encontrar o fim de arquivo.

**%FOUND** – Funciona de maneira oposta a %NOTFOUND. Se o comando Fetch (Responsável por ler as linhas do cursor) encontrar uma linha, esse parâmetro retornará True.

**%ROWCOUNT** – Retorna o número total de linhas retornadas pelo comando FETCH. Cada vez que o comando FETCH for executado, %ROWCOUNT aumenta em 1.

Estes mesmos atributos podem ser usados de forma implícita. Com isto você pode obter informações sobre o comando SQL executado mais recentemente. A forma de referenciar é através de SQL%.

Exemplo:

```

DELETE * FROM funcionario;
IF SQL%ROWCOUNT > 0 THEN
    dbms_output.put_line('Excluiu ' || to_char(sql%rowcount)
                        || ' registros ');
END IF;

```

#### 5.17.5 Acessando as Linhas de um Cursor

Após a abertura do cursor pelo comando OPEN, as linhas selecionadas ficam à disposição. Para ler o conteúdo de uma linha, o PL/SQL usa o comando FETCH, que lê o valor de cada coluna da linha especificada no comando SELECT e o atribui a uma variável. Ele pode ser visto como um equivalente ao comando SELECT INTO.

Exemplo;

```

FETCH CRS2 INTO temp_nome, temp_codigo_depto, temp_salario;
LOOP
    EXIT WHEN CRS2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' || TO_CHAR(temp_codigo_depto)
                        || ' ' || TO_CHAR(temp_salario));
END LOOP;

```

Se for executado apenas uma vez, somente o conteúdo de uma linha será acessado. Para ler sequencialmente as linhas do cursor, é conveniente usar o comando FOR...LOOP. O próximo exemplo lê o cursor CRS2, atribuindo o conteúdo das colunas selecionadas a três variáveis que foram previamente declaradas. A cláusula EXIT testa o atributo %NOTFOUND. Quando ele retornar True, o loop será encerrado. Enquanto isto não ocorrer, as variáveis serão exibidas.

Exemplo:

```

LOOP
    FETCH CRS2 INTO temp_nome, temp_codigo_depto, temp_salario;
    EXIT WHEN CRS2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(temp_nome || ' ' || TO_CHAR(temp_codigo_depto)
                        || ' ' || TO_CHAR(temp_salario));
END LOOP;

```

#### 5.17.6 Fechando um Cursor

Quando um cursor não estiver sendo utilizado, ele deve ser fechado para que libere os recursos que estavam sendo alocados para ele. O comando responsável por essa tarefa é o comando Close.

Exemplo:

```

CLOSE CRS2;

```

#### 5.17.7 Exemplo sem argumento

```

SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE LOOPCRS2 IS
    temp_nome funcionario.nome_funcionario%TYPE;
    temp_codigo_depto funcionario.codigo_departamento%TYPE;
    temp_salario funcionario.salario_funcionario%TYPE;

    CURSOR CRS2 IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = 2
        ORDER BY nome_funcionario;
BEGIN
    DBMS_OUTPUT.ENABLE;

```

```

OPEN CRS2;
LOOP
    FETCH CRS2 INTO temp_nome, temp_codigo_depto, temp_salario;
    EXIT WHEN CRS2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(temp_nome || ' '
                          || TO_CHAR(temp_codigo_depto)
                          || ' ' || TO_CHAR(temp_salario));
END LOOP;
CLOSE CRS2;
END;

```

### 5.17.8 Exemplo com argumento

```

SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE
    PROCEDURE LOOPCRS2(DEPTO funcionario.codigo_departamento%TYPE) IS
    temp_nome funcionario.nome_funcionario%TYPE;
    temp_codigo_depto funcionario.codigo_departamento%TYPE;
    temp_salario funcionario.salario_funcionario%TYPE;

    CURSOR CRS2(CODDEPTO funcionario.codigo_departamento%TYPE) IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = CODDEPTO
        ORDER BY nome_funcionario;
BEGIN
    DBMS_OUTPUT.ENABLE;
    OPEN CRS2;
    LOOP
        FETCH CRS2(DEPTO) INTO temp_nome, temp_codigo_depto, temp_salario;
        EXIT WHEN CRS2%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(temp_nome || ' '
                              || TO_CHAR(temp_codigo_depto)
                              || ' ' || TO_CHAR(temp_salario));
    END LOOP;
    CLOSE CRS2;
END;

```

### 5.17.9 Usando o FOR...LOOP

Uma alternativa mais simples para exibir as linhas retornadas por um cursor e evitar o trabalho manual de abrir, atribuir e fechar um cursor é o comando FOR...LOOP. Ao usar esse cursor, o Oracle automaticamente declara uma variável como o mesmo nome da variável usada como contador do comando FOR, que é do mesmo tipo de variável criada pelo cursor. Basta preceder o nome do campo selecionado com o nome dessa variável para ter acesso ao seu conteúdo.

Exemplo

```

SQL> SET SERVEROUTPUT ON;
SQL> CREATE OR REPLACE PROCEDURE FORCRS2 IS
    CURSOR CRS2 IS
        SELECT nome_funcionario, codigo_departamento, salario_funcionario
        FROM funcionario
        WHERE codigo_departamento = 2
        ORDER BY nome_funcionario;
BEGIN
    DBMS_OUTPUT.ENABLE;
    FOR X IN CRS2 LOOP
        DBMS_OUTPUT.PUT_LINE(X.nome_funcionario
                              || ' ' || TO_CHAR(X.codigo_departamento)
                              || ' ' || TO_CHAR(X.salario_funcionario));
    END LOOP;

```

END ;

### 5.18 Tabelas Temporárias

Existe algumas situações em que é necessário criar relatórios ou exibir informações, e para isto resgata-se os dados realiza-se alguma manipulação e apresenta-se o resultado ao usuário. Após apresentar o resultado e descarta-se a "manipulação" realizada uma vez que esta não precisa ser persistida e deve ser refeita a cada solicitação. É muito comum nestes casos utilizar uma tabela física para realizar a manipulação. Os desenvolvedores criam tabelas, carregam os dados, fazem as manipulações necessárias, utilizam as informações e depois apagam os dados e ou as tabelas criadas. Não há a necessidade de trabalhar com tabelas físicas para trabalhar com dados temporários. Neste caso deve-se criar tabelas temporárias. Com isto você evita que o banco de dados registre informações das operações realizadas nestas tabelas.

Uma tabela temporária é uma tabela com vida útil de uma sessão ou transação. Ela está vazia quando a sessão ou transação começa e descarta os dados ao fim da sessão ou transação. Uma tabela temporária é associada à transação. Isto significa que ao término da transação os dados da tabela são perdidos, porém sua descrição permanece gravada no banco de dados mesmo após a mudança de sessão. As tabelas temporárias podem ser de dois tipos, sessão ou transação. Nas tabelas temporárias sessão os dados são visíveis durante a sessão que os criou. A tabela temporária de transação preserva os dados somente durante a transação do usuário, após isto os dados são apagados.

#### 5.18.1 Criando uma Tabela Temporária

Para criar uma tabela temporária de sessão utilize o comando abaixo. A constraint FOREIGN KEY não pode ser criada para tabelas temporárias.

```
CREATE GLOBAL TEMPORARY TABLE r (A1 D1,
    A2 D2,
    ...,
    An Dn,
    <regras de integridade 1>,
    <regras de integridade 2>,
    ...,
    <regras de integridade n>) ON COMMIT PRESERVE ROWS;
```

Criando uma tabela temporária de sessão para cliente. Os dados são apagados após a finalização da sessão do usuário.

```
CREATE GLOBAL TEMPORARY TABLE TempCliente(
    cpf_cliente VARCHAR(11),
    saldo numeric(9,2),
    PRIMARY KEY (cpf_cliente)) ON COMMIT PRESERVE ROWS;
```

Para criar uma tabela temporária para transação e necessário alterar somente para ON DELETE PRESERVE ROWS. Com isto após o commit os dados da tabela são apagados.

```
CREATE GLOBAL TEMPORARY TABLE TempCliente(
    cpf_cliente VARCHAR(11),
    saldo numeric(9,2),
    PRIMARY KEY (cpf_cliente)) ON COMMIT DELETE ROWS;
```

### 5.19 Exceções

Exceções são usadas no pl/sql para lidar com quaisquer erros que ocorram durante a execução de um bloco. Existe dois tipos de exceções, as definidas internamente pela pl/sql e as definidas pelo usuário.

Quando ocorre um erro no bloco pl/SQL e existe uma exception definida, o controle de execução do bloco passa para a exception que executará os comandos nela contidos, retornando para o fim do bloco onde ocorreu o erro.

### 5.19.1 Exceções Definidas Internamente

#### 5.19.1.1 Algumas Exceções Internas

**Quadro 8 - Exceções internas Oracle**

Exceção	Código	Significado
DUP_VAL_ON_INDEX	ORA-00001 - SQLCODE -1	Tentativa de gravação de chave duplicada para índice único
INVALID_CURSOR	ORA-01001 - SQLCODE-1001	Operação com cursor ilegal. Ex. Fechar cursor não aberto.
INVALID_NUMBER	ORA-01722 - SQLCODE-1722	Conversão de caracter para numérico falha.
LOGIN_DENIED	ORA-01017 - SQLCODE-1017	Nome do usuário ou senha inválida
NO_DATA_FOUND	ORA-01403 - SQLCODE-+100	Select não retorna nenhuma linha
NOT_LOGGED_ON	ORA-01012 - SQLCODE-1012	PL/SQL emite uma chamada ao oracle sem estar conectado
OTHERS		Qualquer tipo de erro
PROGRAM_ERROR	ORA-01001 - SQLCODE-1001	PL/SQL tem um problema interno
STORAGE_ERROR	ORA-06501 - SQLCODE-6501	PL/SQL não tem memória suficiente para rodar ou memória esta danificada
TIMEOUT_ON_RESOURCE	ORA-00051 - SQLCODE-0051	Decurso de tempo enquanto o oracle espera por um recurso
TOO_MANY_ROWS	ORA-02112 - SQLCODE-2112	Select retorna mais de uma linha
TRANSACTION_BACKED_OUT	ORA-01061 - SQLCODE-0061	Oracle volta atrás uma transação por causa de erro de processamento.
VALUE_ERROR	ORA-06502 - SQLCODE-6502	Ocorrência de erro em expressões aritméticas, conversões e truncamentos
ZERO_DIVIDE	ORA-01476 - SQLCODE-1476	Tentativa de divisão por zero

#### 5.19.1.2 Retornando Erros

SQLERR - Retorna o número do erro

SQLERRM – Retorna o número e a descrição do erro

#### 5.19.1.3 Sintaxe

```

DECLARE
    <declarações>
BEGIN
    <Comandos>
    EXCEPTION
        WHEN <nome_exception> THEN
            <Comandos><INSERT,UPDATE,DELETE,SELECT,rollback,commit,null>
            <funções><sqlcode,sqlerrm>
        WHEN OTHERS THEN
            <Comandos>
            <funções>
END;
```

#### 5.19.1.4 Exemplo de Exceção Interna

```

DECLARE
    rg_temp funcionario.rg_funcionario%TYPE;
BEGIN
    SELECT rg_funcionario INTO rg_temp
    FROM funcionario
    WHERE UPPER(nome_funcionario) = UPPER('Joao');
    DBMS_OUTPUT.PUT_LINE('Um João encontrado');
    EXCEPTION
```

```

        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('João não encontrado');
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE('Mais de um João');
        WHEN OTHERS THEN
            ROLLBACK;
    END;

```

## 5.19.2 Exceções Definidas pelo Usuário

### 5.19.2.1 Sintaxe

```

DECLARE
    <declarações>
    <nome_da_exceção> EXCEPTION;
BEGIN
    <Comandos>
    if <condição> then
        RAISE <nome_da_exceção>;
    <Comandos>
    EXCEPTION
        WHEN <nome_da_exceção> THEN
            <Comandos>
END;

```

## 5.19.3 Função Interna

Para mostrar suas mensagens de erro você pode usar `RAISE_APPLICATION_ERROR`. Ela é uma função interna e mostra uma mensagem pelo mesmo caminho dos erros do Oracle. Você deve usar um número negativo entre -20000 até -20999 para `error_number` e uma a mensagem de erro não pode exceder 512 caracteres para `error_message`.

### 5.19.3.1 Sintaxe

```

RAISE_APPLICATION_ERROR (error_number, error_message);

```

### 5.19.3.2 Exemplo

```

DECLARE
    nome VARCHAR(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Digite o seu nome');
    nome := '&nome';
    IF length(nome) > 5 THEN
        RAISE_APPLICATION_ERROR(-20000, 'Nome Longo');
    ELSE
        RAISE_APPLICATION_ERROR(-20001, 'Nome Curto');
    END IF;
END;

```

## 5.20 Packages

Uma package (pacote) é uma coleção de objetos de banco de dados como stored procedures, funções, exceptions, variáveis, constantes e cursores. Um package pode conter subprogramas que podem ser chamados a partir de uma trigger, procedure ou função. Um package é um grande aliado do desenvolvedor, pois permite organizar melhor os componentes de um sistema em módulos. Além disso, a administração de privilégios é facilitada.

### 5.20.1 Estrutura de um Package

Um package possui duas seções, a seção de especificação e o corpo (body) que precisam ser criadas separadamente.

### 5.20.1.1 Seção de especificação

A seção de especificação funciona como uma espécie de sumário do conteúdo do corpo do package. Fisicamente, eles são objetos distintos. Nessa seção são declarados os nomes de funções e stored procedure juntamente com nomes de variáveis e constantes, incluindo aí sua inicialização. A seção de especificação é criada com o comando CREATE PACKAGE, cuja sintaxe é exibida a seguir:

Sintaxe:

```
CREATE [OR REPLACE] PACKAGE <nome_da_package> IS
    [seção de declaração]
END;
```

### 5.20.1.2 Package Body

O corpo do package contém a definição formal de todos os objetos referenciados na seção de declaração. A criação dessa seção é feita através do comando CREATE PACKAGE BODY.

Sintaxe:

```
CREATE [OR REPLACE] PACKAGE BODY <nome_da_package> IS
    [seção de declaração]
    [definição de procedures]
    [definição de funções]
    [seção de inicialização]
END;
```

A seção de inicialização é opcional e é executada apenas a primeira vez que o package é referenciado. As seções de definição de procedures e functions são as áreas onde o código PL/SQL que criam esses objetos devem ser colocados.

### 5.20.2 Recompilando um Package

Sintaxe:

```
ALTER PACKAGE <nome_package> COMPILE BODY;
```

### 5.20.3 Apagando um Package

Sintaxe:

```
DROP PACKAGE [BODY] <nome_package>;
```

### 5.20.4 Exemplo

```
CREATE OR REPLACE PACKAGE APPS_RH IS
    PROCEDURE AUMENTA_SALARIO(depto funcionario.codigo_departamento%TYPE,
                               percentual NUMBER);
    FUNCTION QTDE_FUNCIONARIO(depto funcionario.codigo_departamento%TYPE)
                               RETURN NUMBER;
END;

CREATE OR REPLACE PACKAGE BODY APPS_RH IS
    PROCEDURE AUMENTA_SALARIO(depto funcionario.codigo_departamento%TYPE,
                               percentual NUMBER) IS
    BEGIN
        UPDATE funcionario
        SET salario_funcionario = salario_funcionario * (1+percentual/100)
        WHERE codigo_departamento = depto;
    END;

    FUNCTION QTDE_FUNCIONARIO(depto funcionario.codigo_departamento%TYPE)
    RETURN NUMBER IS
```

```

        total NUMBER;
BEGIN
    SELECT COUNT(*) INTO total
    FROM funcionario
    WHERE codigo_departamento = depto;
    RETURN TOTAL;
END;
END;

```

### 5.20.5 Referenciando um Subprograma do Package

Para acessar um objeto especificado dentro de um package, basta, precedê-lo com o nome do package.

```

SQL> execute APPS_RH.AUMENTA_SALARIO(2,10);
      PL/SQL Procedure successfully completed;

```

## 5.21 Gerenciamento de Usuários

Um importante conceito que deve ser bem compreendido pelo administrador de banco de dados é aquele que diz respeito à criação de usuários. Qualquer pessoa que quiser acessar um banco de dados precisa ser previamente cadastrada como um usuário de banco de dados e ter estabelecido para ela privilégios com relação às tarefas que poderá executar no banco de dados. Quando você trabalha com um banco de dados em um microcomputador Standalone, somente um usuário estará acessando o banco de dados em um determinado momento, provavelmente você. Contudo, mesmo nesse ambiente, outras pessoas podem acessar o banco de dados em outros momentos.

Controlar o acesso ao banco de dados é uma das principais tarefas de um administrador de banco de dados. Para realizar esse controle, o banco de dados conta com um mecanismo que permite cadastrar uma pessoa, chamada a partir de agora, de usuário. Cada usuário cadastrado recebe uma senha de acesso que precisa ser fornecida em diversas situações. Para cada usuário são atribuídos privilégios individuais ou um papel(role) que consiste de um grupo de privilégios que podem ser atribuídos de uma vez ao usuário que recebe aquele papel.

### 5.21.1 Privilégio

É uma autorização para que o usuário acesse e manipule um objeto de banco de dados de uma certa forma. Por exemplo, um usuário final pode ter o privilégio de selecionar tabelas, porém, não modificá-las. Outro usuário pode tanto ler como atualizar e até mesmo alterar a estrutura de tabelas e outros objetos. Como toda a manipulação de um banco de dados é feita através de comandos SQL, um privilégio garante a um usuário o direito de usar ou não determinados comandos SQL.

Existem dois tipos de privilégios, os privilégios de sistema(system privileges) e privilégios de objeto(object privileges).

#### 5.21.1.1 Privilégio de Sistema

Um privilégio de sistema é o direito ou permissão de executar uma ação no banco de dados em um tipo específico de objeto de banco de dados. Existem mais de 70 tipos de privilégios associados a ações de banco de dados. O nome do privilégio é praticamente o nome da ação que ele executa. Por exemplo, o privilégio de nome ALTER TABLE garante o privilégio de alterar uma tabela do próprio usuário.

Abaixo a lista de privilégios do sistema.

**Quadro 9 - Privilégios do sistema Oracle**

Privilégio	Operação Permitida
CREATE SESSION	Permite o usuário conectar-se ao banco de dados e abrir uma sessão.
ALTER ANY TABLE	Permite o usuário alterar tabelas em qualquer schema.
CREATE ANY TABLE	Permite o usuário criar tabelas em qualquer schema.
CREATE TABLE	Permite o usuário criar tabelas.
DELETE ANY TABLE	Permite ao usuário apagar dados em tabelas de qualquer schema.



DROP ANY TABLE	Permite o usuário apagar tabelas em qualquer schema.
DROP TABLE	Permite o usuário apagar tabelas.
INSERT ANY TABLE	Permite ao usuário inserir dados em tabelas de qualquer schema.
SELECT ANY TABLE	Permite ao usuário fazer queries em tabelas de qualquer schema.
UPDATE ANY TABLE	Permite ao usuário atualizar em tabelas de qualquer schema.
CREATE ANY SYNONYM	Permite ao usuário criar sinônimos em qualquer schema.
CREATE PUBLIC SYNONYM	Permite ao usuário criar sinônimos públicos.
CREATE SYNONYM	Permite ao usuário criar sinônimo.
DROP ANY SYNONYM	Permite ao usuário apagar sinônimos em qualquer schema.
DROP PUBLIC SYNONYM	Permite ao usuário apagar sinônimos públicos.
ALTER ANY ROLE	Permite ao usuário alterar roles em qualquer schema.
CREATE ROLE	Permite ao usuário criar roles.
DROP ANY ROLE	Permite ao usuário apagar roles em qualquer schema.
ALTER TRIGGER	Permite ao usuário alterar triggers.
CREATE ANY TRIGGER	Permite ao usuário criar triggers em qualquer schema.
CREATE TRIGGER	Permite ao usuário criar triggers.
DROP TRIGGER	Permite ao usuário apagar triggers.
ALTER ANY PROCEDURE	Permite ao usuário alterar procedures em qualquer schema.
CREATE ANY PROCEDURE	Permite ao usuário criar procedures em qualquer schema.
CREATE PROCEDURE	Permite ao usuário criar procedures.
DROP ANY PROCEDURE	Permite ao usuário apagar procedures em qualquer schema.
EXECUTE ANY PROCEDURE	Permite ao usuário executar qualquer procedure de qualquer schema.
ALTER USER	Permite ao usuário alterar usuários.
CREATE USER	Permite ao usuário criar novos usuários.
DROP USER	Permite ao usuário apagar usuários.
ALTER ANY SEQUENCE	Permite ao usuário alterar seqüências em qualquer schema.
CREATE ANY SEQUENCE	Permite ao usuário criar seqüências em qualquer schema.
CREATE SEQUENCE	Permite ao usuário criar seqüências.
DROP ANY SEQUENCE	Permite ao usuário apagar seqüências em qualquer schema.
SELECT ANY SEQUENCE	Permite ao usuário selecionar seqüências em qualquer schema.
CREATE ANY VIEW	Permite ao usuário criar visões em qualquer schema.
CREATE VIEW	Permite ao usuário criar visões.
DROP ANY VIEW	Permite ao usuário apagar visões em qualquer schema.
ALTER ANY INDEX	Permite ao usuário alterar índices em qualquer schema.
CREATE ANY INDEX	Permite ao usuário criar índices em qualquer schema.
DROP ANY INDEX	Permite ao usuário apagar índices em qualquer schema.

Exemplo:

```
GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW, CREATE SYNONYM, EXECUTE ANY
PROCEDURE, CREATE PROCEDURE TO ADMINISTRADOR;
```

### 5.21.1.2 Privilégio de Objeto

É autorização que permitem um usuário manipular dados de uma tabela ou view, sequence – alterar suas estruturas, executar triggers ou procedures armazenadas – podendo repassar essa autorização a outros usuários.

O privilégio de objeto é o direito de executar uma determinada ação em um objeto específico, como, por exemplo, o direito de incluir uma linha em uma tabela determinada. Os privilégios de objeto não se aplicam a todos os objetos de banco de dados. Triggers, procedures e indexes não possuem privilégios de objeto.

Abaixo a lista de privilégios de objeto.

**Quadro 10 - Privilégios de objeto Oracle**

Privilégio	Operação Permitida
ALTER TABLE	Permite o usuário alterar a estrutura de tabelas.
DELETE TABLE	Permite o usuário apagar dados em tabelas.
INSERT TABLE	Permite o usuário inserir dados em tabelas.
SELECT TABLE	Permite o usuário fazer query em tabelas.
UPDATE TABLE	Permite ao usuário atualizar dados em tabelas.
CREATE VIEW	Permite o usuário criar visões.
DELETE VIEW	Permite o usuário apagar dados em visões.
INSERIR VIEW	Permite ao usuário inserir dados em visões.
SELECT VIEW	Permite o usuário fazer query em visões.
UPDATE VIEW	Permite ao usuário atualizar dados em visões.
REFERENCES	Permite o usuário criar ou alterar tabelas para definir foreign key.
INDEX	Permite o usuário criar índices para tabelas.
ALTER SEQUENCE	Permite ao usuário alterar seqüências.
SELECT SEQUENCE	Permite o usuário selecionar seqüências.
EXECUTE PROCEDURE	Permite ao usuário executar qualquer procedures.
ALL	Todos os privilégios acima.

Exemplo:

```
GRANT SELECT, UPDATE(NOME_CLIENTE), REFERENCES(NOME_CIDADE) ON CLIENTE TO ADMINISTRADOR;
```

```
GRANT EXECUTE ON AUMENTAR_SALARIO TO ADMINISTRADOR;
```

Quando um usuário cria um objeto como uma tabela, ela só pode ser visualizada pelo próprio usuário que a criou. Para que outro usuário tenha acesso a ela, é necessário que o usuário proprietário da tabela conceda privilégios para o usuário ou papel que irá acessar a tabela.

Um privilégio pode ser concedido através do comando SQL chamado GRANT.

### 5.21.2 Papel

Um papel é um grupo de privilégios (Sistema e/ou Objetos), e outros papéis que são associados a um nome que os identifica e podem ser atribuídos a um usuário ou a outro papel. Dessa forma, em vez de conceder oito privilégios para um usuário, você pode criar um papel que recebeu oito privilégios e, em seguida, atribuir o papel ao usuário. Usar papéis simplifica a administração dos usuários.

Para criar um papel o usuário tem que ter o privilégio CREATE ROLE. Quando um papel é criado pelo usuário, ele recebe a opção ADM que permite que o usuário conceda o papel para outros papéis ou usuários e para retirar o papel concedido a outro usuário/papel, alterar o acesso ao papel e remover o papel.

Características:

- Não tem proprietário;
- Pode ser dado Grant para qualquer usuário ou para outro papel, exceto para ele mesmo;
- Pode ser ativado ou desativado por usuário.

Utilização:

- Dar segurança a base de dados;
- Controlar usuários por grupo de afinidades.

### 5.21.3 Criando Usuários

O comando CREATE USER é o responsável pela criação de novos usuários.

Sintaxe:

```
CREATE USER <nome_do_usuario> IDENTIFIED {BY <senha> | EXTERNALLY}
  DEFAULT TABLESPACE <nome_do_tablespace>
  TEMPORARY TABLESPACE <nome_do_tablespace>
  QUOTA { numero[K|M] | UNLIMITED} ON TABLESPACE <nome_do_tablespace>
  PROFILE <nome_do_profile>;
```

**IDENTIFIED BY** – Deve ser seguido da senha do usuário ou da palavra EXTERNALLY para indicar que banco de dados deve procurar a senha no sistema operacional.

**DEFAULT TABLESPACE** – Identifica a tablespace usada para os objetos do usuário. Se omitido, o banco de dados assume a tablespace system.

**TEMPORARY TABLESPACE** – Identifica a tablespace usada para os objetos temporários do usuário.

**QUOTA** – Especifica a quantidade máxima de espaço na tablespace em Kbytes ou Mbytes que o usuário terá.

**PROFILE** – Atribui os valores armazenados no profile especificado para o usuário. Se omitido, assume o profile chamado Default. Um profile é um arquivo que contém limites de uso do banco de dados para um usuário. Assim como um papel, um profile pode ser utilizado por vários usuários.

Para criar um usuário, você tem que possuir os privilégios adequados. Quando é criado um usuário no Oracle, ele simultaneamente, cria um Schema(de mesmo nome) para este usuário. O Schema é um espaço do usuário para armazenar os seus objetos. O exemplo mostra o usuário System criando um usuário na forma mais simples do comando.

```
SQL> connect
Entre nome do usuário: system
Entre senha: *****
Conectado
SQL> CREATE USER joao IDENTIFIED BY silva;
```

Usuário criado;

#### 5.21.4 Alterando Usuários

O comando ALTER USER é o responsável pela alteração das características de um usuário.

Sintaxe:

```
ALTER USER <nome_do_usuario> IDENTIFIED {BY <senha> | EXTERNALLY}
      DEFAULT TABLESPACE <nome_do_tablespace>
      TEMPORARY TABLESPACE <nome_do_tablespace>
      QUOTA { numero[K|M] | UNLIMITED} ON TABLESPACE <nome_do_tablespace>
      PROFILE <nome_do_profile>;
```

Para alterar um usuário, você tem que possuir os privilégios adequados. O exemplo mostra o usuário System alterando a senha do usuário joao para joao.

```
SQL> connect
Entre nome do usuário: system
Entre senha: *****
Conectado.
SQL> ALTER USER joao IDENTIFIED BY joao;
```

Usuário alterado;

#### 5.21.5 Criando Papéis

O comando SQL usado para criar um papel é o CREATE ROLE.

Sintaxe básica:

```
CREATE ROLE <nome_do_papel>
      {NOT IDENTIFIED
      | IDENTIFIED {BY <senha> | EXTERNALLY}}}
```

**IDENTIFIED BY** – Solicita uma senha de verificação para o usuário. Pode ser fornecida a senha do banco de dados ou usar a opção EXTERNALLY para obter a senha no sistema operacional.

```
SQL> CREATE ROLE GRUPO;
```

Role created;

#### 5.21.6 Alterando Papéis

O comando SQL usado para alterar um papel é o ALTER ROLE.

Sintaxe básica:

```
ALTER ROLE <nome_do_papel>
      {NOT IDENTIFIED
      | IDENTIFIED {BY <senha> | EXTERNALLY}}}
```

#### 5.21.7 Concedendo Privilégios e Papéis a um Papel

Sintaxe básica:

```
GRANT <nome_papel>/<nome_do_privilegio> TO <nome_do_usuario>/<nome_papel>
PUBLIC WITH ADMIN OPTION;
```

O próximo exemplo concede os papéis CONNECT e RESOURCE para o papel GRUPO.

```
SQL> GRANT CONNECT, RESOURCE TO grupo;
```

```
Grant succeeded;
```

#### 5.21.8 Concedendo um Papel a um Usuário

O usuário João foi criado, mas não possui nenhum privilégio ou papel atribuído a ele. Usa-se o comando GRANT para atribuir o papel GRUPO a ele.

```
SQL> GRANT grupo TO joao;
```

```
Grant succeeded;
```

```
SQL> CONNECT
Enter user-name: joão
Enter password: *****
Connect
SQL>
```

#### 5.21.9 Concedendo um Privilégio de Objeto para um Usuário

Uma variação do comando GRANT permite atribuir privilégios de objeto para um usuário ou papel. Os privilégios concedidos podem ser: ALTER, DELETE, EXECUTE, INSERT, INDEX, REFERENCES e UPDATE. Os objetos que podem conceder privilégios são tabelas, visões, seqüências e sinônimos.

Sintaxe básica:

```
GRANT {<nome_do_privilegio>|ALL [PRIVILEGES]} [(coluna[,coluna]...)]
[, {<nome_do_privilegio>|ALL [PRIVILEGES]} [(coluna[,coluna]...)]...]
ON [schema.]objeto
TO {<nome_do_usuario>|<nome_do_papel>|PUBLIC}
[, {<nome_do_usuario>|<nome_do_papel>|PUBLIC}]...
[WITH GRANT OPTION];
```

**ALL PRIVILEGES** – Atribui todos os privilégios do objeto.

**(COLUNA)** – Especifica as colunas para as quais o privilégio(somente INSERT, UPDATE E REFERENCE) está sendo concedido.

**PUBLIC** – Concede o privilégio para todos os usuários do banco de dados.

**GRANT OPTION** – Permite que o usuário/papel que recebe o privilégio possa concedê-lo a outros usuários.

O exemplo abaixo mostra o usuário Scott concedendo o privilégio SELECT para o usuário João acessar a tabela dept.

```
SQL> CONNECT
Enter user-name: scott
Enter password: *****
Connect
SQL> GRANT SELECT ON scott.dept TO joao;
```

```
Grant succeeded
```

```

SQL> CONNECT
Enter user-name: jo o
Enter password: *****
Connect
SQL> SELECT * FROM scott.dept;
  DEPTO DNAME          LOC
-----
    10 ACCOUNTING      NEW YORK
    20 RESERCH          DALLAS
    30 SALES             CHICAGO
    40 OPERATIONS       BOSTON
SQL>

```

#### 5.21.10 Visualizando os Pap is e Privil gios

O banco de dados possui algumas tabelas especiais para controlar os privil gios e papeis:

USER\_ROLE\_PRIVS, Privil gios do usu rios atual.  
 DBA\_ROLE, Exibe todos os pap is definidos no banco de dados.  
 DBA\_ROLE\_PRIVS, pap is atribu dos para usu rios e pap is.  
 ROLE\_SYS\_PRIVS, privil gios de sistema dos pap is  
 ROLE\_TAB\_PRIVS, privil gios de tabela dos pap is.  
 ROLE\_ROLE\_PRIVS, pap is que s o atribu dos a pap is.  
 SESSION\_ROLES, pap is ativados para o usu rio atual.

##### 5.21.10.1 Pap is e Privil gios de um Usu rio

A tabela USER\_ROLE\_PRIVS exibe os pap is atribu dos ao usu rio atual.

```

SQL> SELECT * FROM USER_ROLE_PRIVS;

```

USERNAME	GRANTED_ROLE	ADM	DEF	OS_
JOAO	CONNECT	NO	YES	NO
JOAO	RESOURCE	NO	YES	NO

##### 5.21.10.2 Pap is definidos no banco de dados

A tabela DBA\_ROLE exibe todos os pap is definidos no banco de dados.

```

SQL> SELECT * FROM DBA_ROLE;

```

ROLE	PASSWORD_REQUIRED
CONNECT	NO
RESOURCE	NO
DBA	NO
GRUPO	NO
...	

##### 5.21.10.3 Privil gios de um Papel

A tabela ROLE\_SYS\_PRIVS exibe os privil gios de sistema atribu dos a um papel.

```

SQL> SELECT * FROM ROLE_SYS_PRIVS;

```

ROLE	PRIVILEGE	ADMIN_OPTION
CONNECT	ALTER SESSION	NO
CONNECT	CREATE CLUSTER	NO
CONNECT	CREATE DATABASE LINK	NO

```

CONNECT                                CREATE SEQUENCE                        NO
CONNECT                                CREATE SESSION                        NO
...

```

#### 5.21.10.4 Privilégios de tabela atribuídos a um Papel

A tabela `ROLE_TAB_PRIVS` exibe os privilégios de tabela atribuídos a um papel.

```
SQL> SELECT * FROM ROLE_TAB_PRIVS;
```

ROLE	OWNER	TABLE_NAME	COLUMN_NAME	PRIVILEGE	GRANTABLE
DBA	SYS	DBMS_DEFER_QUERY		EXECUTE	NO
DBA	SYS	DBMS_DEFER_SYS		EXECUTE	NO
DBA	SYS	DBMS_FLASHBACK		EXECUTE	NO
DBA	SYS	DBMS_RESUMABLE		EXECUTE	NO
DBA	SYS	JIS_EXP		EXECUTE	NO
DBA	SYS	JIS_IMP_AUX		EXECUTE	NO

...

#### 5.21.11 Apagando um Usuário

A remoção de um usuário do banco de dados é feita pelo comando `DROP USER`. Ele remove tanto o usuário como todos os objetos contidos no esquema do usuário. Nesse caso, é necessário especificar a cláusula `CASCADE` do comando. O banco de dados também remove toda a integridade referencial associado aos objetos do usuário removido.

Sintaxe básica:

```
DROP USER <nome_do_usuario> [CASCADE];
```

Exemplo:

```
SQL> DROP USER joao;
```

User dropped.

Para remover os objetos criados pelo usuário joão utilize a cláusula `CASCADE`, por exemplo:

```
SQL> DROP USER joao CASCADE;
```

User dropped.

Este comando apaga o usuário joao e todos os objetos que tenha criado.

#### 5.21.12 Revogando um Privilégio de Sistema/Papel Concedido

Assim como concedeu um privilégio, você pode revogá-lo. O comando SQL responsável por essa tarefa é o comando `REVOKE`. Para que um privilégio de sistema possa ser revogado, ele precisa ter recebido a opção `ADMIN OPTION`.

Sintaxe básica:

```

REVOKE {<nome_papel>|<nome_do_privilegio>|ALL [PRIVILEGES]}
      FROM {<nome_do_usuario>|<nome_do_papel>|PUBLIC}
      [WITH ADMIN OPTION];

```

Vamos tentar revogar o privilégio dado ao usuário joao e que não recebeu essa opção.

```
SQL> REVOKE grupo FROM joao;
```

```

revoke grupo from joao
*
ERROR at line 1:
ORA-01932: ADMIN option not grant for role 'GRUPO'

```

Ocorreu um erro, pois o papel não recebeu a opção ADMIN ao ser concedido. Nós conectamos como o usuário que atribuiu o privilégio ao papel e atribuímos novamente os papéis CONNECT e RESOURCE para o papel GRUPO, agora com a opção ADMIN.

```

SQL> CONNECT
Enter user-name: system
Enter password: *****
Connected.
SQL> GRANT CONNECT, RESOURCE to grupo WITH ADMIN OPTION;

```

Grant succeeded

Agora vamos usar o comando REVOKE para retirar o papel grupo do usuário joao.

```
SQL> REVOKE grupo FROM joao;
```

Revoke succeeded.

Para verificar a funcionalidade do comando, tente conectar com o usuário joao. O banco de dados verificou que joao não mais o papel CONNECT e não permitiu a conexão com o banco de dados.

```

SQL> CONNECT
Enter user-name: joao
Enter password: *****
Connected.
ERROR:ORA-01045: user JOAO lacks CREATE SESSION privilege; logon denied

```

Atribua novamente o papel GRUPO para joao.

### 5.21.13 Revogando um Privilégio de Objeto de um Usuário

Para revogar um privilégio, você deve se conectar como um usuário que tenha autoria da concessão. Um usuário não pode conceder/revogar privilégios para si mesmo.

```

SQL> CONNECT
Enter user-name: Scott
Enter password: *****
Connect
SQL> REVOKE SELECT ON scott.dept FROM joao

```

Revoke succeeded.

Para ver se a revogação funcionou, conecte com o usuário joao e tente acessar a tabela Scott.dept. A falta do privilégio impediu o acesso a ela.

```

SQL> Connect
Enter user-name: joao
Enter password: *****
Connect
SQL> SELECT * FROM scott.dept;
SELECT * FROM Scott.dept
*
ERROR at line 1:
ORA-00942: table or view does not exist

```



#### 5.21.14 Apagando um Papel

O comando responsável por apagar um papel é DROP ROLE. Para que o usuário possa remover um papel, ele precisa ter o privilégio de sistema DROP ANY ROLE ou o papel que tenha sido criado com a opção ADMIN OPTION.

Sintaxe básica:

```
DROP ROLE <nome_da_role>;
```

### 5.22 Sinônimos

Um sinônimo é um apelido dado para um objeto de banco de dados. Quando um sinônimo é criado, é feita uma referência ao objeto original.

Os sinônimos trazem muitas vantagens.:

A maior delas talvez seja o fato de esconder a identidade do objeto que está sendo referenciado. Se o objeto for mudado ou movido, basta atualizar o sinônimo,

Uma tabela com o nome extenso e que pertença a um esquema diferente daquele do usuário pode ser abreviado com um sinônimo e, com isso, facilitar a execução de comandos.

Um sinônimo pode ser criado para uma tabela, visualização, sequência, procedure em função de package ou até mesmo de outro sinônimo.

Os sinônimos podem ser públicos e visíveis para todos os usuários ou privados e disponíveis apenas para o usuário que o criou.

#### 5.22.1 Sintaxe de sinônimo

```
CREATE [PUBLIC] SYNONYM <esquema.nome> FOR <esquema.objeto>;
```

**esquema.nome** – Novo nome para o objeto do banco de dados.

**esquema.objeto** – Objeto a receber um sinônimo.

#### 5.22.2 Criando um sinônimo

Exemplo:

```
CREATE SYNONYM departamento FOR scott.dept;
```

#### 5.22.3 Renomeando um sinônimo

Exemplo:

```
RENAME departamento TO dept;
```

#### 5.22.4 Apagando um sinônimo

Sintaxe:

```
DROP [PUBLIC] SYNONYM <nome_sinomimo>;
```

Exemplo:

```
RENAME departamento TO depto;
```

#### 5.22.5 Metadados

Olhar anexo Apêndice 1 item 8.11.

### 5.23 Database Links

Um database link(dblink) é um objeto criado em um esquema de um banco de dados que possibilita o acesso a objetos de outro banco de dados, seja ele Oracle ou não. Esse tipo de sistema é conhecido como Sistema de Banco de Dados Distribuídos e pode ser **Homogêneo** – quando acessa outros bancos de dados Oracle - e **Heterogêneo** – quando acessam outros tipos de bancos de dados.

Para acessar bancos de dados que não sejam Oracle é necessário utilizar o Oracle Heterogeneous Services em conjunto com um agente.

Oracle server a partir de sua versão 8i que habilita a tecnologia dos produtos Oracle Transparent Gateway, sendo que Heterogeneous Services provê uma arquitetura comum e mecanismos de administração para os produtos Oracle gateway, além de outras facilidades de acesso a bancos heterogêneos.

Para acessar um banco de dados não Oracle utilizando um Oracle Transparent Gateway (agente) deve-se selecionar uma aplicação específica do sistema, ou seja, cada banco de dados específico requer um agente diferente.

Por exemplo, para criar um database link com um banco de dados Sybase é necessário obter um gateway transparente específico para Sybase para que então o banco de dados Oracle possa comunicar-se com ele. O agente executa comandos SQL e requisições de transações a bancos de dados não Oracle em nome do usuário da base de dados Oracle.

Pode-se, também, utilizar uma conectividade genérica para acessar bancos de dados não Oracle, como os protocolos ODBC ou OLE DB, através do uso dos Heterogeneous Services ODBC e OLE-DB, sem a necessidade de adquirir e configurar um agente específico para o banco de dados que se deseja acessar.

Ao criar um database link é possível utilizar e referenciar tabelas e visões do outro banco de dados, acrescentando ao final do nome destes objetos @nome\_do\_dblink.

Com o dblink e os privilégios necessários é possível utilizar comandos SELECT, INSERT, UPDATE, DELETE ou LOCK TABLE sobre os objetos desejados do banco de dados remoto, além de tornar as operações de COMMIT e ROLLBACK transparentes.

Nem todo o banco possui este recurso consulta a documentação do seu banco.

#### 5.23.1 Sintaxe básica database link

```
CREATE [PUBLIC] DATABASE LINK <nome_link> CONNECT TO <usuario> IDENTIFIED BY
<senha> USING '//'<endereco>:<porta>/<instancia>';
```

**nome\_link** – Nome do link para ser utilizado no banco local.

**usuario** – Nome de um usuário autorizado a se conectar no banco de dados remoto.

**senha** – Senha do usuário autorizado a se conectar no banco de dados remoto.

**endereco** – Endereço ip do servidor de banco de dados remoto.

**porta** – Porta do servidor de banco de dados remoto.

**instancia** – Nome da instancia do servidor de banco de dados remoto.

#### 5.23.2 Criando um database link

Exemplo:

```
CREATE DATABASE LINK xelink CONNECT TO scott IDENTIFIED BY tiger USING
'//192.168.0.1:1521/xe';
```

Não esqueça de verificar se o firewall esta com a porta TCP e UDP liberada para acessar o banco.

#### 5.23.3 Consultado os database link criados

Exemplo:

```
SELECT *
FROM DBA_DB_LINKS;
```

#### 5.23.4 Consultando tabelas via database link

Exemplo:

```
SELECT * FROM cliente@xelink;
```

Para manter-se a transparência no acesso a objetos de outros bancos de dados pode-se criar sinônimos públicos para os objetos acessados através do dblink.

#### 5.23.5 Two Phase Commit

Uma das principais características, e também das que apresentam maior dificuldade de implementação plena, para BD distribuídos é o Commit de duas fases ou two-phase commit (TPC).

Imagine uma transação sobre um banco de dados distribuído que atualize tabelas de 3 ou 4 máquinas. O que aconteceria à transação se uma das máquinas que teve alguma tabela já atualizada saísse do ar enquanto a transação prosseguia até certo ponto?

Ao tentar desfazer a parte da transação ocorrida sobre o equipamento em falha poderia acontecer de não serem revertidos os efeitos da transação sobre os dados daquele equipamento, mas dos demais sim. Isto colocaria o banco numa situação de exceção, ou erro transacional, ou ainda de falha da integridade transacional.

Para evitar este problema, e da mesma forma garantir que quando uma transação emita um COMMIT, todos os gerenciadores envolvidos efetuem as partes da transação com que estão envolvidos, faz-se necessário emitir um COMMIT para cada gerenciador. A transação só será efetivamente "commitada" no caso de todos os gerenciadores envolvidos responderem OK à solicitação.

#### 5.23.6 As 12 regras para sistemas distribuídos

Chirs J. Date é bastante conhecido pelo seu trabalho com bancos de dados. Talvez ele seja mais conhecido por seus trabalhos com bancos de dados relacionais e bancos de dados distribuídos, mas trabalha com vários aspectos da tecnologia de banco de dados. Ele trabalhou também com o Dr. Ted Codd, o pai dos sistemas relacionais. C. J. Date publicou doze regras que os bancos de dados distribuídos deveriam obedecer em um artigo de 1987, na InfoDB Magazine. Sendo elas:

1. **Autonomia Local** – Cada local em um sistema de banco de dados distribuído deve ser independente de outros.

2. **Não Ter confiança em uma instalação central** – se um banco de dados distribuído tivesse de confiar em uma instalação central única, este local poderia se tornar um ponto único de falha, algo a ser evitado em uma arquitetura de banco de dados distribuído.

3. **Operação Contínua** – um banco de dados distribuído nunca deve exigir tempo parado. Ele deve ser capaz de operar continuamente com recursos como de execução de cópias de segurança (backup) on-line e recuperação e arquivamento completo e incremental.

4. **Transparência a localização e independência de localização** – os usuários não devem precisar saber onde os dados estão armazenados. Deve parecer em locais diferentes. Isso deve ser transparente para os usuários.

5. **Independência a Fragmentação** – as tabelas (em bancos de dados distribuídos relacionais) podem ser divididas em fragmentos e armazenadas em locais diferentes. Isso deve ser transparente para os usuários.

6. **Independência a replicação** – os dados podem ser replicados por diferentes computadores na arquitetura distribuída.

7. **Processamento de consulta distribuído** – o desempenho de resultados de uma consulta deve ser independente do local de onde ela é executada.

8. **Gerenciamento de transações distribuído** – o sistema deve ser capaz de oferecer suporte a transações atômicas. As transações atômicas, ou atomicidade, significa que as transações devem ser realizadas completamente ou não serem feitas em absoluto.

9. **Independência de Hardware** – os dados deveriam ser acessíveis em uma ampla variedade de plataformas de hardware.

10. **Independência de sistemas Operacionais** – o banco de dados distribuído deve ser capaz de operar com vários sistemas operacionais.

11. **Independência de rede** – o banco de dados distribuído deve ser capaz de operar com uma série de protocolos de rede ou topologias de rede.

12. **Independência de SGBD** – o sistema de banco de dados distribuído ideal deveria ser capaz de oferecer suporte à interoperabilidade entre diferentes tipos de sistemas de SGBDs rodando em vários nós.

### 5.24 Agendamento de Tarefas

Algumas tarefas ou execuções de processos precisam ser feitas periodicamente, ou simplesmente agendadas para um determinado horário.

O Oracle possui algumas maneiras de automatizar estas atividades, uma delas é a submissão de atividades como JOBS.

JOBS são objetos associados ao schema que permitem agendamento de atividades, para trabalharmos com os jobs precisamos de permissão de execução na package DBMS\_JOB.

Operações deste pacote:

- DBMS\_JOB.SUBMIT( )
- DBMS\_JOB.REMOVE( )
- DBMS\_JOB.CHANGE( )
- DBMS\_JOB.WHAT( )
- DBMS\_JOB.NEXT\_DATE( )
- DBMS\_JOB.INTERVAL( )
- DBMS\_JOB.RUN( )

#### 5.24.1 Criando um Job

Exemplo:

A procedure abaixo irá apagar os clientes que foram inseridos e não possuam conta ou dívida com o banco.

```
create or replace procedure LIMPADADOS is
begin
    delete from cliente where cpf_cliente
        not in ((select cpf_cliente from devedor)
            union
            (select cpf_cliente from conta)
        );
end;
```

Esta job deve ser executado diariamente as 02:00.

Declare

```
--Variavel que recebera o id do job
job_num number;
```

Begin

```
DBMS_JOB.SUBMIT(job_num,
    'LIMPADADOS;',
    sysdate,
    'trunc(sysdate + 1) + 2/24');
```

```
COMMIT;
```

End;

O primeiro parametro de DBMS\_JOB.SUBMIT é o número do job agendado.

O segundo parametro o nome da procedure que se deseja agendar.

O terceiro a data da proxima execução do JOB.

O ultimo o intervalo de execução da JOB, no caso do exemplo no proximo dia as 02:00 horas.

Abaixo outros exemplos de intervalo :

- 'SYSDATE + 7'=>exatamente sete dias da última execução
- 'SYSDATE + 1/48'=>cada meia hora
- 'NEXT\_DAY(TRUNC(SYSDATE), 'MONDAY')+15/24'=>toda segunda-feira as 15:00
- 'NEXT\_DAY(ADD\_MONTHS(TRUNC(SYSDATE, 'Q'), 3), 'THURSDAY')'=>primeira quinta-feira de cada trimestre
- 'TRUNC(SYSDATE + 1)' => todo dia a meia noite

- 'TRUNC(SYSDATE + 1)+8/24' => todo dia as 08:00
- 'NEXT\_DAY(TRUNC(SYSDATE), "TUESDAY")+12/24'=>toda terça-feira ao meio dia
- 'TRUNC(LAST\_DAY(SYSDATE)+1)'=>primeiro dia de cada mês a meia noite
- 'TRUNC(ADD\_MONTHS(SYSDATE+2/24,3), 'Q')-1/24'=>último dia de cada trimestre as 23:00
- 'NEXT\_DAY(SYSDATE, "FRIDAY"))+9/24'=>cada segunda, quarta e sexta as 09:00

#### 5.24.2 Listando os Jobs agendados

Exemplo:

```
SELECT * FROM ALL_JOBS;
```

#### 5.24.3 Listando os Jobs em execução

Exemplo:

```
SELECT * FROM DBA_JOBS_RUNNING;
```

É necessário ter privilégios de DBA.

#### 5.24.4 Removendo um Job

Exemplo:

```
exec DBMS_JOB.remove(<ID_JOB>);
```

**ID\_JOB** – Id do job que se deseja remover.

## 6 Banco de Dados de Orientado a Objeto

São divididos em dois grupos sistemas gerenciadores de banco de dados puramente orientado a objetos (Pure Object Oriented DBMS-ODBMS) e sistema gerenciador de banco de dados objeto relacional (Object Relational DBMS-ORDBMS).

Os puramente orientados a objetos baseia-se somente no modelo de dados Orientado a Objetos. Usam declarações de classes muito semelhantes das linguagens orientadas a objetos. Ex. BD Jasmini.

Os sistemas de banco de dados objeto relacional SGBDOR, são bancos relacionais que possibilitam o armazenamento de objetos, permitem a relação de objetos, herança e identificadores de objeto. Os identificadores de objeto é um identificador interno do banco para cada objeto, são atribuídos somente pelo DBMS e não pelos usuários. Não tem padrão único de implementação como os BD relacionais. Exemplos de SGBDOR: Oracle, Postgres, Informix, BD2, Titanium.

SGBDs Objeto-Relacional combinam os benefícios do modelo Relacional com a capacidade de modelagem do modelo OO. Fornecem suporte para consultas complexas sobre dados complexos. Atendem aos requisitos das novas aplicações e da nova geração de aplicações de negócios.

### 6.1 Vantagens

Em relação a bancos de dados relacionais temos como vantagens qualidade do software, reutilização, portabilidade, facilidade de manutenção e escalabilidade.

No relacional diversas vezes a linguagem de programação é completamente diferente a utilizada na RDBMS.

Os dados dos bancos relacionais utilizam a linguagem SQL no qual permite que os sistemas relacionais desenvolvidos por muitos fornecedores possam se comunicar entre si e acessar banco de dados comuns.

Em contra partida, os bancos de dados orientados a objetos não possuem uma linguagem padrão dificultando a interoperabilidade entre os bancos de dados de diferentes fornecedores.

### 6.2 Características

Em um SGBDOO os objetos da base de dados são tratados como objetos da linguagem de programação, possuem características de e princípios do paradigma de orientação a objetos. Estas características serão brevemente descritas.

**Persistência**, os dados de um processo ou transação persistem após o término da execução do mesmo. A persistência é requisito evidente para bases de dados, a persistência deve permitir que qualquer objeto, independente de seu tipo, torne-se persistente.

**Objeto complexos**, suporte a objetos grande em tamanhos e a objetos estruturados, como tuplas, vetores e listas. Tendo a necessidade de suporte as operações que manipulam estes objetos.

**Identidade de objeto**, cada objeto da base possui um identificador único e imutável, gerado automaticamente pelo sistema.

**Encapsulamento**, o objeto da base de dados encapsula dados que definem sua estrutura interna e operações que definem seu comportamento, a estrutura interna e a definição do comportamento de um objeto ficam escondidas, e o objeto é acessado através das operações pré definidas para seu tipo.

**Tipos, Classes e Herança**, suporte a hierarquias de tipos ou hierarquias de classes através do conceito de herança, o que permitem que novos tipos sejam definidos a partir de tipos de classes pré definidos. Os subtipos de subclasses herdam os atributos e as rotinas das superclasses, podendo no entanto possuir atributos e rotinas próprias.

**Polimorfismo** também chamado de sobrecarga, permite que um mesmo nome de operação seja utilizado para implementações diferentes, dependendo do tipo de objeto ao qual a operação é aplicada.

**Ligação tardia (Late Binding)** realiza a tradução de nomes das operações em endereços de programas em tempo de execução. O binding realizado em tempo de compilação, ao contrário, é chamado de binding estático, o binding atrasado, embora seja lento e dificulte a checagem de tipos é necessário para viabilizar a utilização de sobrecarga de operações.

**Extensibilidade** é o conjunto de tipos pré definidos do sistema que deve ser extensível, permitindo que o programadores definam novos tipos. No entanto o tratamento de tipos definidos pelo usuário é

diferente dos que são definidos pelo sistema. Estas diferenças devem ser imperceptíveis para o programadores e para a aplicação.

### 6.3 Banco de Dados Oracle

O banco de dados Oracle com sua opção de objetos, permite a manipulação e criação de objetos assim podemos classificá-lo como um banco Objeto Relacional.

As entidades do mundo real são definidas pelos usuários, podendo possuir operações implementadas no servidor, todos os objetos criados terão um identificados único que nunca terá o valor alterado durante todo o ciclo de vida do objeto.

As referências são os ponteiros no banco de dado e definem relacionamentos entre objetos que torna o acesso navegacional mais rápido do que o acesso padrão.

Possui suporte a coleções na forma de arrays de tamanhos variáveis e tabelas aninhadas, essas coleções podem consistir de tipos nativos, tipos definidos pelo usuário e de referencias, podendo ser utilizado como atributos dos tipos de objetos.

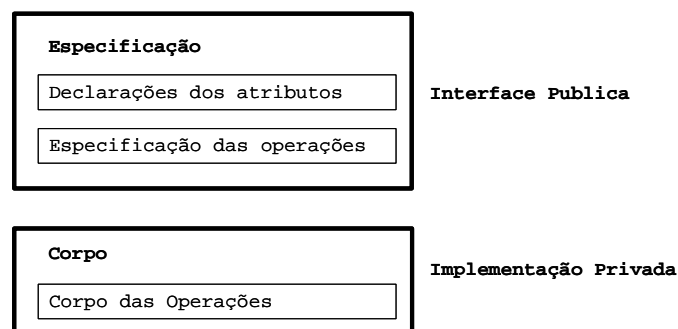
Outra característica do banco de dados Oracle é o cache de objetos, que são as cópias dos objetos que podem ser levados para um cache do cliente diminuindo o tráfego na rede. Podendo-se carregar os objetos complexos com um único acesso ao banco de dados.

As visões de objetos permitem criar abstrações de objetos sobre os banco de dados relacionais. Em adição ao mecanismo tradicional de visões.

#### 6.3.1 Tipos de Objetos

Um tipo de objeto em Oracle possui a seguinte estrutura:

**Figura 6 - Estrutura de tipo de objeto**



Exemplo de especificação da interface pública de um objeto

Sintaxe resumida:

```
CREATE [OR REPLACE] TYPE nome_tipo AS OBJECT (
    [lista de atributos]
    [lista de operações]
);
```

Exemplo:

```
create or replace type TTelefone as object(
    telefoneId number,
    numero varchar(10),
    tipo varchar(30),
    ddd varchar(3),
    Member function getfone return varchar
);
```

```
create or replace type body TTelefone as
    Member function getfone return varchar is
```

```

Begin
    return '(' || ddd || ')' || '-' || numero;
End;
end;

```

### 6.3.2 Operações

São funções ou procedimentos que são declarados na definição de um tipo de objeto. Exigem o uso de parênteses (mesmo sem parâmetros). O uso de ( ) é para diferenciar a operação de um procedimento ou função comum. Podem ser:

MEMBER ou STATIC

MAP ou ORDER (para ordenação)

Construtor

A operação mais comum é Member. Implementam as operações das instâncias do tipo. São invocados através da qualificação de objeto.operacao().

### 6.3.3 Herança de Tipos

O oracle suporta herança simples. Permite criar uma hierarquia de subtipos especializados. Os tipos derivados (subtipos) herdam os atributos e operações dos tipos ancestrais (supertipos). Os subtipos podem acrescentar novos atributos ou operações e/ou redefinir as operações dos super-tipos.

Os tipos e operações podem ser FINAL e NOT FINAL. A definição do tipo do objeto determina se um subtipo pode ser derivado. Por padrão os tipos de objeto são do tipo FINAL. Por isso, para permitir subtipos, deve ser obrigatoriamente adicionada à expressão NOT FINAL na declaração do tipo do objeto.

Por exemplo, abaixo apresenta uma expressão onde é declarado que o tipo TipoPessoa pode ser derivado, permitindo que sejam criados subtipos a partir dele.

```

create or replace type TPessoa as object(
    pessoaId number,
    nome varchar(50),
    email varchar(100),
    telefone TTelefone
) not final;

create or replace type TPessoaJuridica under TPessoa(
    cnpj varchar(16)
);

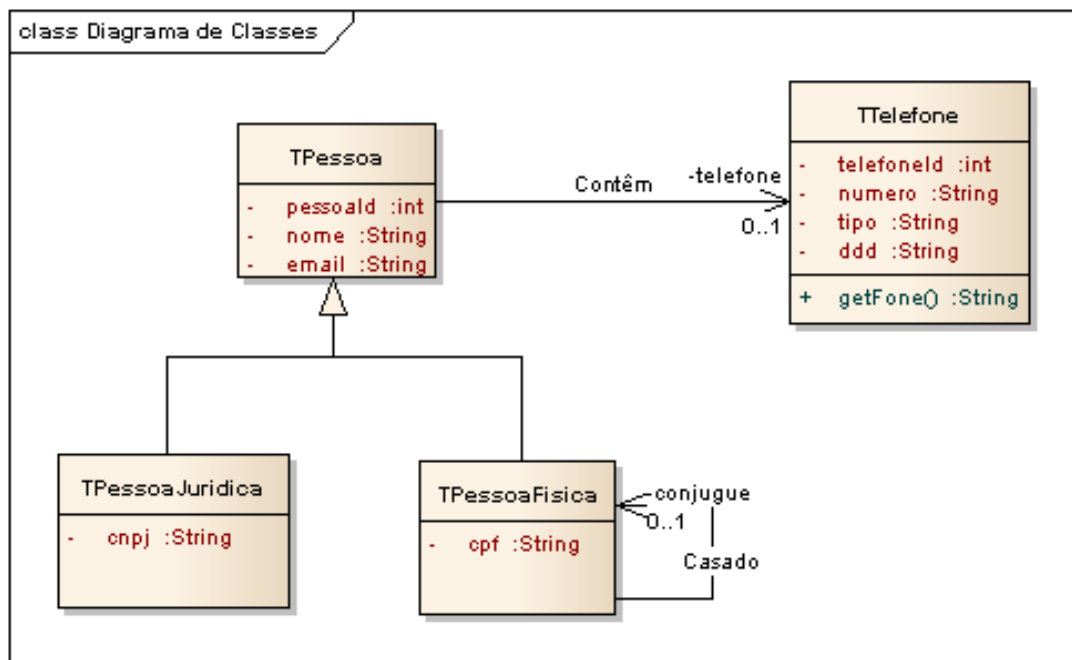
create or replace type TPessoaFisica under TPessoa(
    cpf varchar(11),
    conjugue ref TPessoaFisica
);

```

A Figura 7 representa o diagrama de classes dos tipos definidos anteriormente.



Figura 7 – Diagrama de Classes dos Tipos



### 6.3.4 Tabelas

Uma tabela de objetos difere de uma tabela relacional em vários aspectos. Cada linha de uma tabela de objetos possui um identificador de objeto o OID definido pelo oracle quando a linha é inserida na tabela. Um OID é um ponteiro para um objeto “linha” um “Row Object, ele permite que as linhas de “Row Objects” de uma tabela de objetos possam ser referenciadas em atributos de outros objetos ou em colunas de tabelas relacionais. Um atributo do tipo pré-definido REF é capaz de representar as referências de OID.

As tabelas são criadas a partir dos tipos. Novos campos podem ser adicionados às tabelas além dos definidos no tipo.

```

create Table PessoaJuridica of TPessoaJuridica;

create Table PessoaFisica of TPessoaFisica;

create Table Cliente (
    pessoa TPessoaFisica,
    detalhes varchar(4000)
);
  
```

### 6.3.5 Tipo REF

O tipo REF é um ponteiro lógico para um “Row Object”. É definido a partir do OID do objeto. Oferece acesso rápido/direto a objetos relacionados. Não garante integridade referencial, tem que usar referential constraint, neste caso REF pode apontar para qualquer objeto do tipo apontado.

Os REFs e coleções de REFs são utilizados na modelagem de associações entre objetos. Por ex. o relacionamento entre um Pedido e um cliente.

Constituem um mecanismo simples para navegar entre objetos. Pode-se utilizar a notação estendida de “pontos” para seguir os ponteiros sem a necessidade de junções explícitas.

Na especificação do tipo TPessoaFisica o atributo conjugue é uma referência ao próprio tipo.

### 6.3.6 Alterações de Tipos

As alterações de tipos é muito semelhante a alterações de atributos para as tabelas. Uso do comando ALTER TYPE, permite modificar ou evoluir um tipo objeto, sendo possível: adicionar e remover atributos, adicionar e remover operações, modificar um atributo numérico para aumentar o

length, precision, ou scale, modificar um atributo string para aumentar seu length. Para adicionar o atributo celular do tipo telefone ao tipo pessoa podemos utilizar o comando:

```
alter type Tpessoa add attribute celular TTelefone cascade;
```

Para remover o atributo celular do tipo pessoa utilize o comando:

```
alter type Tpessoa drop attribute celular cascade;
```

A opção CASCADE propaga a mudança para todos os tipos dependentes, com isto as alterações no tipo refletem em todas as estruturas que a utilizam.

Não se pode remover um subtipo antes de remover suas respectivas instâncias na tabela que armazena as tuplas daquele subtipo (substitutability).

Por exemplo, para o apagar o tipo PessoaJuridica:

```
DROP TYPE TPessoaJuridica VALIDATE;
```

Primeiro deve ser apagado as instancias em tabelas do seu tipo:

```
DELETE FROM PessoaJuridica WHERE p IS OF (TPessoaJuridica);
DROP TYPE TPessoaJuridica VALIDATE;
```

### 6.3.7 Construtor

Criado implicitamente ao criar um tipo de objeto. Deve ser exatamente igual ao nome do tipo. Pode haver mais de um construtor para um tipo.

Exemplo de inserção de dados em uma tabela criada a partir de tipo:

```
insert into PessoaJuridica
  values (1,'ACME SA','acme@gmail.com',
    TTelefone(1,'111','22','333'),'123456789000012');
```

Como a tabela for criada inteiramente a partir do tipo, não tem necessidade do construtor para TPessoaJuridica somente para TTelefone.

Exemplo de tabela criada com um atributo de tipo:

```
insert into cliente(pessoa, detalhes)
  values (TPessoaFisica(1,'Joao da Silva','joao@gmail.com',
    TTelefone(1,'123','12','123'),'12345678912',NULL),
    'cliente desde 1999');
```

Se você não possuir dados para um atributo insira **NULL**. No exemplo não foi inserido telefone e conjugue para o cliente.

```
insert into cliente(pessoa, detalhes)
  values (TPessoaFisica(2,'Pedro da Silva','pedro@gmail.com',
    NULL,'12345678912',NULL), 'cliente desde 1999');
```

Para criar uma referência vamos primeiro inserir uma pessoa física:

```
insert into PessoaFisica
  values (1,'Joao da Silva','joao@gmail.com',
    TTelefone(2,'134','12','123'),'123456789', NULL);
```

Com uma pessoa inserida agora vamos inserir uma segunda pessoa, tendo a primeira como seu conjugue. Usamos REF para recuperar a referência:

```
insert into PessoaFisica
select TPessoaFisica (2,'Maria da Silva', 'maria@gmail.com',
                     NULL,'123456789',REF(pf))
from PessoaFisica pf
where pf.pessoaId = 1;
```

### 6.3.8 Atualização e Exclusão

A atualização ocorre de forma muito semelhante ao modelo relacional. Para atualizar o telefone da pessoa Jurídica inserida anteriormente executamos o comando:

```
update PessoaJuridica pj
set pj.telefone = TTelefone(3,'123','123','321')
where pj.pessoaId = 1;
```

Para excluir o registro alterado execute o comando:

```
delete from PessoaJuridica pj
where pj.pessoaId = 1;
```

### 6.3.9 Consulta

Existem diferenças significativas no modo de utilização de uma tabela de objetos. Cada linha dentro de uma tabela de objetos possuirá um OID, e essas linhas poderão ser referenciadas como objetos.

```
select * from cliente;
```

o resultado:

PESSOA	DETALHES
1 [PROFBD2.TPESSOAFISICA]	cliente desde 1999
2 [PROFBD2.TPESSOAFISICA]	cliente desde 1999

ou,

PESSOA	DETALHES
PROFBD2.TPESSOAFISICA(1,'Joao da Silva', 'joao@gmail.com',PROFBD2.TTELEFONE(1,'123','12','123'), '12345678912',NULL)	cliente desde 1999
PROFBD2.TPESSOAFISICA(2,'Pedro da Silva', 'pedro@gmail.com',NULL,'12345678912',NULL)	cliente desde 1999

Outros exemplos de consulta.

```
select c.pessoa.nome
from cliente c;
```

```
select c.pessoa.nome, c.pessoa.telefone.numero
from cliente c;
```

```
select c.pessoa.nome
from cliente c
where c.pessoa.pessoaId = 1;
```

Nas consultas você pode chamar operações.

```
select c.pessoa.nome, c.pessoa.telefone.numero, c.pessoa.telefone.getFone()
from cliente c;
```

Com os dados relacionados através de REF as consultas podem ser realizadas mais facilmente:

```
select ref(pf)
from PessoaFisica pf;
```

O resultado da consulta:

```
PROFBD2.TPESSOAFISICA(1,'Joao da Silva','joao@gmail.com',
    PROFBD2.TTELEFONE(2,'134','12','123'),'123456789',NULL)
PROFBD2.TPESSOAFISICA(2,'Maria da Silva','maria@gmail.com',
    NULL,'123456789','oracle.sql.REF@3df04063')
```

Para consultar o nome da pessoa física e o nome do seu conjugue podemos escrever:

```
select pf.nome, pf.conjugue.nome
from PessoaFisica pf;
```

O resultado da consulta e apresentado abaixo perceba que não é necessário usar junção para trazer os dados relacionados.

nome	conjugue.nome
Joao da Silva	<b>NULL</b>
Maria da Silva	Joao da Silva

### 6.3.10 Constraints

As constraints podem ser especificadas normalmente como no modelo relacional.

```
create Table PessoaJuridica of TPessoaJuridica(
    constraint PK_PessoaJuridica primary key(pessoaId),
    constraint NN_PessaoJuridica_nome nome not null
);

create Table PessoaFisica of TPessoaFisica(
    constraint PK_PessoaFisica primary key(pessoaId),
    constraint NN_PessaoFisica_nome nome not null,
    constraint UK_PessoaFisica_email UNIQUE (email),
    constraint CK_PessoaFisica_pessoaId CHECK (pessoaId > 0),
    constraint FK_PessoaFisica_Conjugue FOREIGN KEY (conjugue)
        REFERENCES pessoaFisica
);

create Table Cliente (
    pessoa TPessoaFisica,
    detalhes varchar(4000) constraint NN_Cliente_detalhes not null,
    constraint PK_Cliente primary key(pessoa.pessoaId),
    constraint CK_Cliente_nome check (pessoa.nome is not null)
);
```

Para atributos tipo REF a integridade referencial é semelhante ao FOREIGN KEY. A constraint Primary Key não pode ser especificada para uma coluna do tipo REF.

## 7 Bibliografia

DATE, C.J., **Introdução a Sistemas de Bancos de Dados**. 7. ed. Rio de Janeiro. Ed. Campus, 2000.

ELMASRI, R.; NAVATHE, S. B., **Sistemas de Banco de Dados – Fundamentos Aplicações**, 3. ed. Livros Técnicos e Científicos, 2002

FREEMAN, R. **Oracle, referência para o DBA: técnicas essenciais para o dia-a-dia do DBA**. Rio de Janeiro: Elsevier, 2005.

KROENKE, D. M. **Banco de Dados Fundamentos, Projetos e Implementação**, Rio de Janeiro, Ed. Livros Técnicos e Científicos, 1998.

RAMALHO, J. A.. **Oracle10g**, São Paulo. Ed. Pioneira Thomsom Learning, 2005.

SILBERSCHATZ, A. ; KORTH, H.F. ; SUDARSHAN, S. **Sistema de Banco de Dados**. 5. ed. Rio de Janeiro: Elsevier, 2006

## 8 Apêndice 1 – Oracle

### 8.1 Características

#### 8.1.1 Limites do Oracle

**Quadro 11 - Limites do Oracle**

ITEM	LIMITE
Tabela na Base de dados	Não há limites.
Linhas por Tabelas	Não há limites
Colunas por tabelas	254
Índices por tabelas	Não há limites
Tabelas ou views joined em uma query	Não há limites
Níveis de ninho de subqueries	30
Caracteres em um nome	255
Colunas por índices	16

### 8.2 Tabelas do Dicionário de Dados

Os objetos do dicionário de dados que um usuário pode acessar, encontram-se na visão `DICTIONARY`, que é propriedade do usuário `SYS`. Ela possui o nome da tabela e uma breve descrição.

O exemplo abaixo lista as tabelas que armazenam dados sobre as tabelas.

```
SELECT table_name, substr(comments,1,60) comments
FROM sys.dictionary
WHERE table_name like upper('%TABLE%')
```

### 8.3 Visões do Dicionário de Dados

O dicionário de dados oferece informações detalhadas sobre cada aspecto do banco de dados. As tabelas base do dicionário de dados pertencem a `SYS` e residem no espaço de tabela `SYSTEM`. Essas tabelas não devem ser acessadas diretamente. Em vez disso, as informações devem ser acessadas por meio de algumas visões. Boa parte das visões de dicionário de dados inicia com um prefixo de `USER_`, `ALL` e `DBA_`. O prefixo detalha o escopo das informações apresentadas nessa visão.

#### 8.3.1 `USER_`

Visões que mostram informações provenientes do esquema do usuário atual. Por exemplo `USER_TABLES`, `USER_INDEXES`, `USER_VIEWS`, `USER_TRIGGERS`, `USER_SEQUENCES`, `USER_SOURCE`

#### 8.3.2 `ALL_`

Visões que mostram informações do esquema do usuário atual, bem como as informações de outros esquemas se o outro usuário atual tiver os privilégios apropriados sobre essas informações. Por exemplo `ALL_TABLES` mostra as informações sobre todas as tabelas que o usuário atual possui e as informações que o usuário pode acessar.

#### 8.3.3 `DBA_`

Visões que mostram informações para o banco de dados inteiro. No exemplo `DBA_TABLES` mostra informações sobre todas as tabelas no banco de dados.

## 8.4 Tabelas

### 8.4.1 Metadados

USER\_TABLES - Tabelas/Views das qual o usuário é dono.  
 ALL\_TABLES - Tabelas/Views das qual o usuário tem acesso.  
 DICTIONARY - Tabelas/Views que fazem parte do dicionário de dados.  
 USER\_TAB\_COLUMNS - Tabelas mostra todas as colunas definidas nas Tabelas/Views

### 8.4.2 Listando as tabelas que o usuário é dono

```
SELECT Table_Name
FROM USER_TABLES;
```

### 8.4.3 Listando as tabelas que o usuário tem acesso

```
SELECT Table_Name
FROM ALL_TABLES;
```

### 8.4.4 Mostrando a Estrutura de uma Tabela

```
DESC <Nome_Tabela>;
ou
DESCRIBE <Nome_Tabela>;
```

### 8.4.5 Mostrando todos os Dados de uma Tabela

```
SELECT * FROM <Nome_Tabela>;
```

### 8.4.6 Mostrando a Tabelas e o Atributo por tabela

```
SELECT Table_Name, Column_Name FROM USER_TAB_COLUMNS;
```

### 8.4.7 Apagando Tabelas

```
DROP TABLE <nome_tabela>;
```

### 8.4.8 Apagando Tabelas com Restrições

```
DROP TABLE <nome_tabela> CASCADE CONSTRAINTS;
```

## 8.5 Constraint (Integridade)

### 8.5.1 Metados

```
USER_CONSTRAINTS
```

### 8.5.2 Listando os Nomes as Restrições

```
SELECT constraint_name
FROM USER_CONSTRAINTS;
```

### 8.5.3 Listando os Nomes as Restrições e suas Tabelas

```
SELECT constraint_name, table_name
FROM USER_CONSTRAINTS;
```

## 8.6 Índices

### 8.6.1 Metadados

```
USER_INDEXES
```

### 8.6.2 Listando os índices criados

```
SELECT Index_name
FROM USER_INDEXES;
```

### 8.6.3 Listando as tabelas e seus índices

```
SELECT Table_name, Index_name
FROM USER_INDEXES;
```

## 8.7 Visões

### 8.7.1 Visões Normais

#### 8.7.1.1 Metadados

```
USER_VIEWS
```

#### 8.7.1.2 Listando as visões que o usuário é dono e tem acesso

```
SELECT View_Name
FROM USER_VIEWS;
```

### 8.7.2 Visões Materializadas

#### 8.7.2.1 Metadados

```
USER_MVIEWS
```

#### 8.7.2.2 Listando as visões materializadas que o usuário é dono e tem acesso

```
SELECT MView_Name
FROM USER_VIEWS;
```

## 8.8 Sequências

### 8.8.1 Metadados

```
USER_SEQUENCES
```

### 8.8.2 Listando as seqüências criadas pelo usuário

```
SELECT sequence_name
FROM USER_SEQUENCES;
```

## 8.9 Triggers

### 8.9.1 Metadados

```
USER_TRIGGERS
```

### 8.9.2 Listando as Triggers que o usuário é dono

```
SELECT Trigger_Name
FROM USER_TRIGGERS;
```

## 8.10 Procedimentos, Funções e Packages

### 8.10.1 Metadados

```
USER_SOURCE
```



### 8.10.2 Mostrando os Argumentos de uma Function ou Procedure

```
DESC <Nome_Procedimento>;
ou
DESCRIBE <Nome_Procedimento>;
```

### 8.10.3 Listando os Procedimentos criados(Procedures, Functions e Packages)

```
SELECT distinct(name)
FROM USER_SOURCE;
```

### 8.10.4 Mostrando os erros no Procedimentos (Procedures, Functions e Packages)

```
SHOW errors;
```

## 8.11 *Synônimos*

### 8.11.1 Metadados

```
USER_SYNONYMS
```

### 8.11.2 Listando os Sinônimos que o usuário é dono

```
SELECT Synonym_Name
FROM USER_SYNONYMS;
```

## 8.12 *Usuário*

### 8.12.1 Metadados

```
USER_USERS
USER_TS_QUOTAS
```

### 8.12.2 Mostrando o Usuário Conectado

```
SHOW user;
```

### 8.12.3 Alterando a Senha do Usuário

```
ALTER USER <nome_do_usuario> IDENTIFIED BY <nova_senha>;
```

### 8.12.4 Senhas e definidas no Oracle

**Quadro 12 - Usuários predefinidos Oracle**

Usuário	Senha	Papel
SCOTT	TIGER	CONNECT e RESOURCE
SYSTEM	MANAGER	DBA
SYS	SYS	CONNECT, RESOURCE, DBA, EXP_FULL_DATABASE e IMP_FULL_DATABASE
DEMO	DEMO	CONNECT e RESOURCE

### 8.12.5 Papéis Definidos no Oracle

**Quadro 13 - Papéis predefinidos Oracle**

Papel	Privilegio
CONNECT	Permite acesso ao banco de dados
RESOURCE	Permite o acesso ao banco de dados e permite a criação de tabelas, seqüências, procedures, triggers, índices e clusters

DBA	Todos os privilégios de sistema. Permite a concessão de privilégio para outros usuários.
EXP_FULL_DATABASE	Exportar o banco de dados
IMP_FULL_DATABASE	Importar o banco de dados

### 8.12.6 Sessões de Usuário no Banco

```
column username format a15;
column machine format a20;
column status format a8;
column logon_time format a16;

SELECT substr(vs.username,1,15) username,
       substr(vs.machine,1,20) machine,
       vs.status,
       vs.audsid,
       vs.sid,
       vs.serial#,
       vp.spid,
       to_char(vs.logon_time,'dd/mm/yyyy hh24:mi') logon_time
FROM   sys.v_$session vs,
       sys.v_$process vp
WHERE  vs.paddr = vp.addr
       and vs.username is not null
ORDER BY
       substr(vs.username,1,15),machine;
```

## 8.13 Linha de Comando SQL

### 8.13.1 Conectando ao Banco de Dados

```
connect;
```

ou

```
connect usuario@stringid;
```

ou

```
connect usuario@endereco_ip;
```

### 8.13.2 Desconectando do Banco de Dados

```
disconnect;
```

### 8.13.3 Saindo da Linha de Comando SQL

```
exit;
```

### 8.13.4 Limpando a tela

```
clear screen;
```

### 8.13.5 Modificando a linha de Exibição

```
set line 300;
```

### 8.13.6 Exibindo o tamanho da linha de Exibição

```
show linesize;
```

### 8.13.7 Modificando o tamanho de exibição de coluna

```
column username format a15;
column machine format a20;
column audsid format 9999;
column sid format 9999;
column serial# format 999;
column spid format 999;
column logon_time format a16;
```

### 8.13.8 Habilitando a saída

```
set serveroutput on;
```

### 8.13.9 Salvando Comandos em Arquivo

```
save <caminho + nome_arquivo>;
```

Obs. Extensão Default .SQL

### 8.13.10 Carregando Comandos em Arquivo

```
get <caminho + nome_arquivo>;
ou
@ <caminho + nome_arquivo>;
```

Get somente carrega o arquivo.

@ carrega e executa os comandos do arquivo.

### 8.13.11 Usando & para substituir variável

Usando o duplo &, você pode reusar uma variável sem colocar no prompt a cada vez.

Exemplo:

```
SELECT *
FROM &Nome_Tabela
```

Resultado:

Enter value for Nome\_Tabela: Cliente

CODIGO	NOME
1000	King
7782	John

## 8.14 TableSpace

Oracle armazena dados logicamente em tablespaces e fisicamente em arquivos de dados (datafiles). Os arquivos de dados e os tablespaces estão muito inter-relacionados, mas têm diferenças importantes:

Um banco de dados Oracle consiste em uma ou mais unidades de armazenamento lógicas denominadas tablespaces, que armazenam coletivamente todos os dados do banco de dados.

Cada tablespace em um banco de dados Oracle consiste em um ou mais arquivos denominados arquivos de dados (datafiles), que são estruturas físicas compatíveis com o sistema operacional no qual o Oracle é executado.

Os dados de um banco de dados são armazenados coletivamente nos arquivos de dados que constituem cada tablespace do banco de dados.

Como um banco de dados é um conjunto de arquivos de dados, é muito importante que entendamos como um banco de dados Oracle agrupa esses arquivos. Como dito anteriormente, o Oracle faz isso sob a proteção de um objeto de banco de dados chamado tablespace. Antes de poder inserir dados em um banco de dados Oracle, primeiro é necessário criar um tablespace e depois uma tabela dentro desse tablespace que conterá os dados. Podemos observar que na criação de um banco de dados utilizando o DBCA, o

Oracle como padrão sempre cria um tablespace de dados chamado USERS. Ao criar uma tabela é necessário incluir todas as informações sobre o tipo de dados que deseja manter. O código abaixo para criar a tabela CLIENTE ilustra como o Oracle armazena informações sobre o tipo de dado que irá registrar:

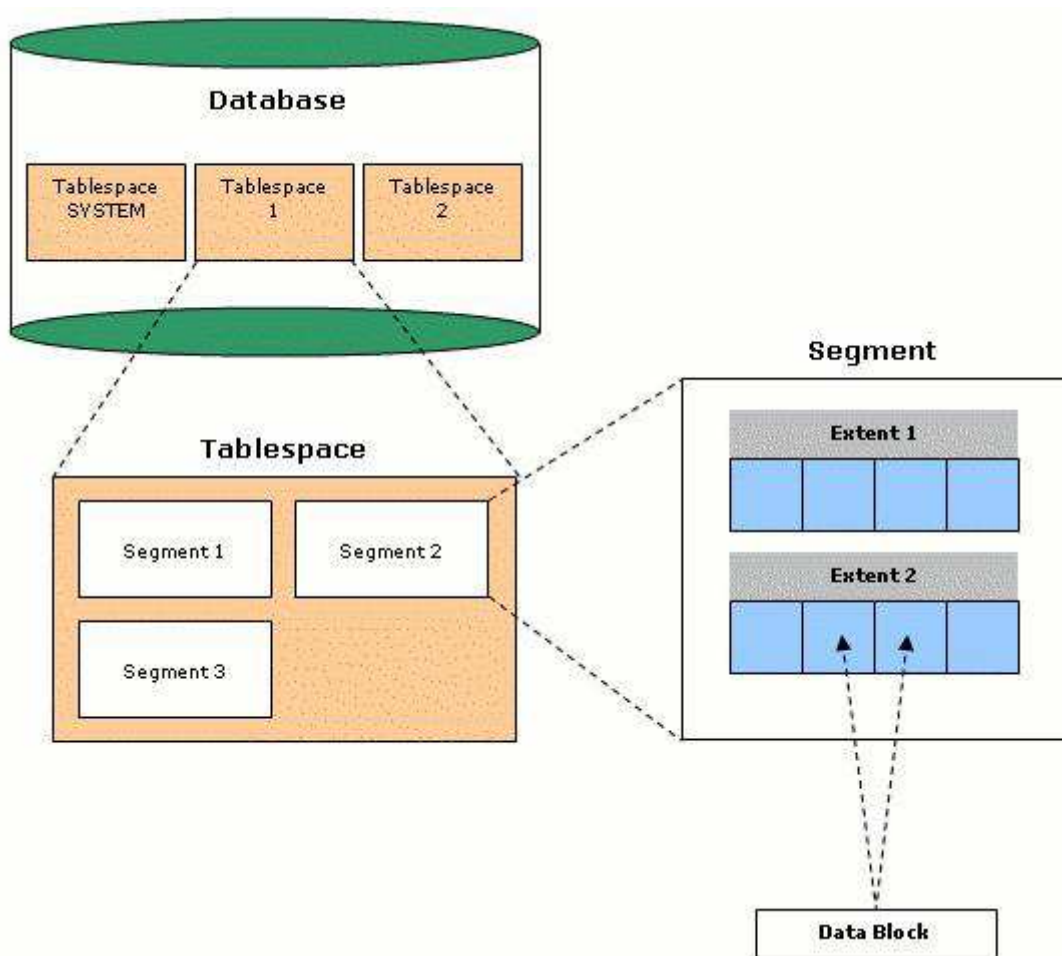
```
SQL> create table cliente
2  (cod_cliente    number constraint pk_cliente primary key,
3   nome           varchar2(60)  not null,
4   endereco       varchar2(100) not null,
5   telefone       number,
6   data_cadastro  date)
7  tablespace users;
Tabela criada.
```

```
SQL> desc cliente
Nome                               Nulo?      Tipo
-----
COD_CLIENTE                        NOT NULL   NUMBER
NOME                               NOT NULL   VARCHAR2(60)
ENDERECO                           NOT NULL   VARCHAR2(100)
TELEFONE                           NUMBER
DATA_CADASTRO                      DATE
```

```
SQL> select table_name,tablespace_name
2  from user_tables
3  where table_name='CLIENTE';
TABLE_NAME      TABLESPACE_NAME
-----
CLIENTE         USERS
```

Na instrução acima, foi criada uma tabela que é o meio mais comum de armazenar dados em um banco de dados. Os dados de um segmento de tabela são armazenados aleatoriamente no tablespace e o DBA tem pouco controle sobre a localização das linhas dos blocos de uma tabela. Por falar nisso, o que é um segmento? Os segmentos são objetos que ocupam espaço em um banco de dados. Existem vários tipos de segmentos como tabelas, índices, de undo, temporários, LOB, entre outros. Já uma extensão (extent), é um espaço usado por um segmento em um tablespace. Para terminar, um bloco Oracle consiste em um ou mais blocos do sistema operacional e seu tamanho é definido na criação do tablespace. Então a estrutura lógica de um banco de dados Oracle se resume em tablespaces que contém segmentos que contém extensões que contém blocos. A figura abaixo ilustra esta estrutura lógica:

Figura 8 - Estrutura lógica TableSpace



```
SQL> select segment_name,segment_type,tablespace_name,bytes,blocks,extents
2  from user_segments
3  where segment_name='CLIENTE';
```

SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	BYTES	BLOCKS	EXTENTS
CLIENTE	TABLE	SYSTEM	65536	8	1

O Oracle tem como default o tablespace SYSTEM para cada novo usuário. Você pode definir o tablespace padrão do usuário definido na sua criação (create user). O comando abaixo exibe o tablespace default do usuário.

```
SQL> select default_tablespace from user_users;
DEFAULT_TABLESPACE
-----
SYSTEM
```

Você pode alterar o tablespace default usando o comando abaixo.

```
SQL> alter database default tablespace users;
Alter Database Default bem-sucedido
```

O nome tablespace vem da necessidade de se agrupar os arquivos de dados. A melhor analogia para se explicar banco de dados, tablespace, arquivo de dados, tabelas e dados é a imagem de um fichário. Imagine um banco de dados como um fichário: as gavetas dentro do fichário são os tablespaces; as pastas nessas gavetas são os arquivos de dados; os papéis em cada pasta são as tabelas; a informação escrita no papel de cada pasta são os dados. Em resumo, o tablespace é um modo de agrupar arquivos de dados.

É aconselhável não misturar dados de aplicativos no mesmo tablespace. Então, ao criar tablespaces para seus aplicativos, dê a eles um nome descritivo (por exemplo, dados de um sistema de Compras podem ser mantidos no tablespace COMPRAS). Em resumo, aplicação separada corresponde a tablespace separado.

#### 8.14.1 O que é o tablespace USERS?

Como demonstrado anteriormente, este geralmente é o tablespace padrão para os usuários. Se um usuário criar um objeto, tal como uma tabela ou um índice, sem especificar o tablespace, o Oracle o cria no tablespace padrão do usuário, isso se o tablespace padrão do usuário foi definido para utilizar o tablespace USERS.

#### 8.14.2 O que é o tablespace SYSTEM?

O tablespace SYSTEM (tablespace de sistema) é uma parte obrigatória de todo banco de dados Oracle. É onde o Oracle armazena todas as informações necessárias para o seu próprio gerenciamento. Em resumo, SYSTEM é o tablespace mais crítico do banco de dados porque ele contém o dicionário de dados. Se por algum motivo ele se tornar indisponível, a instância do Oracle abortará. Por esse motivo, o tablespace SYSTEM nunca pode ser colocado offline, ao contrário de um tablespace comum como, por exemplo, o tablespace USERS.

#### 8.14.3 O que é o tablespace TEMP?

O tablespace TEMP (tablespace temporário) é onde o Oracle armazena todas as suas tabelas temporárias. É o quadro branco ou papel de rascunho do banco de dados. Assim como às vezes precisamos de um lugar para anotar alguns números para poder somá-los, o Oracle também precisa de algum espaço em disco temporário. O Oracle geralmente utiliza o tablespace temporário para armazenar objetos transitórios durante as classificações e agrupamentos de dados durante a execução de uma SQL contendo as cláusulas ORDER BY e GROUP BY. É importante dizer também que os dados de sessão das tabelas temporárias globais (Global Temporary Tables) também ficam no tablespace TEMP. Assim como o tablespace SYSTEM é o tablespace mais crítico do banco de dados, o tablespace TEMP é o menos crítico do banco de dados exatamente porque armazena apenas os segmentos temporários durante as operações de classificação de dados e, como tal, no caso de uma falha, ele pode simplesmente ser dropado e recriado, em vez de ser restaurado e recuperado.

#### 8.14.4 O que é o tablespace UNDO?

Todos os bancos de dados Oracle precisam de um local para armazenar informações a desfazer. O que isso significa? Esse tablespace que contém seus segmentos de reconstrução em versões anteriores ao Oracle 9i chamado de RBS (tablespace de rollback), possui a capacidade de recuperar transações incompletas ou abortadas. Um segmento de undo é usado para salvar o valor antigo quando um processo altera dados de um banco de dados. Ele armazena a localização dos dados e também os dados da forma como se encontravam antes da modificação. Basicamente, os objetivos dos segmentos de undo são:

**Rollback de transação:** Quando uma transação modifica uma linha de uma tabela, a imagem original das colunas modificadas é salva no segmento de undo, e se for feito o rollback da transação, o servidor Oracle restaurará os valores originais gravando os valores do segmento de undo novamente na linha.

**Recuperação de Transação:** Se ocorrer uma falha de instância enquanto houver transações em andamento, o servidor Oracle precisará desfazer as alterações não submetidas à commit quando o banco de dados for aberto novamente. Esse rollback faz parte da recuperação da transação. A recuperação só é possível porque as alterações feitas no segmento de undo também são protegidas pelos arquivos de redo log online.

**Consistência de Leitura:** Enquanto houver transações em andamento, outros usuários do banco de dados não deverão ver as alterações não submetidas à commit feitas nessas transações. Além disso, uma instrução não deverá ver as alterações submetidas à commit após o início da execução dessa instrução. Os valores antigos (dados de undo) dos segmentos de undo também são usados para oferecer aos leitores uma imagem consistente de uma instrução específica.

### 8.14.5 O que é o tablespace SYSAUX?

Este tablespace auxiliar não existe nas versões anteriores ao Oracle 10g e foi criado especialmente para aliviar o tablespace SYSTEM de segmentos associados a algumas aplicações do próprio banco de dados como o Oracle ultra search, Oracle Text e até mesmo segmentos relacionados ao funcionamento do Oracle Enterprise Manager entre outros. Como resultado da criação desse tablespace, alguns gargalos de I/O freqüentemente associados ao tablespace SYSTEM foram reduzidos ou eliminados. Vale a pena salientar, que o tablespace SYSAUX também não pode se colocado offline e é parte integrante obrigatório em todos os bancos de dados a partir do Oracle 10g. Existe uma view de dicionário de dados que mostra os ocupantes neste tablespace:

```
SQL> select occupant_name, schema_name, space_usage_kbytes
2 from v$sysaux_occupants;
```

OCCUPANT_NAME	SCHEMA_NAME	SPACE_USAGE_KBYTES
LOGMNR	SYSTEM	7488
LOGSTDBY	SYSTEM	0
STREAMS	SYS	192
AO	SYS	960
XSOQHIST	SYS	960
SM/AWR	SYS	68352
SM/ADVISOR	SYS	7360
SM/OPTSTAT	SYS	21120
SM/OTHER	SYS	3328
STATSPACK	PERFSTAT	0
ODM	DMSYS	5504
SDO	MDSYS	6080
WM	WMSYS	6656
ORDIM	ORDSYS	512
ORDIM/PLUGINS	ORDPLUGINS	0
ORDIM/SQLMM	SI_INFORMTN_SCHEMA	0
EM	SYSMAN	61632
TEXT	CTXSYS	4736
ULTRASEARCH	WKSYS	7296
JOB_SCHEDULER	SYS	256

Uma outra informação bastante útil que esta view oferece é o nome de uma procedure que o DBA pode utilizar para mover dados de um ocupante para um outro tablespace:

```
SQL> select occupant_name,move_procedure
2 from v$sysaux_occupants where occupant_name='LOGMNR';
```

OCCUPANT_NAME	MOVE_PROCEDURE
LOGMNR	SYS.DBMS_LOGMNR_D.SET_TABLESPACE

## 8.15 Gerenciamento de Espaço em Tablespaces

Os tablespaces alocam espaço em extensões (extents). Eles podem ser criados para usar um dos dois métodos de controle de espaço livre e utilizado:

**Tablespaces gerenciados localmente:** As extensões são gerenciadas no tablespace por bitmaps. Cada bitmap corresponde a um bloco ou a um grupo de blocos. Quando uma extensão é alocada ou liberada para reutilização, o servidor Oracle altera os valores do bitmap para mostrar o novo status dos blocos. A partir do Oracle 9i este gerenciamento local é o padrão.

**Tablespaces gerenciados por dicionário:** As extensões são gerenciadas pelo dicionário de dados. O servidor atualiza as tabelas apropriadas no dicionário de dados sempre que uma extensão é alocada ou desalocada.

Nas versões anteriores ao Oracle 8i, os extents de todos os tablespaces eram gerenciados centralmente por meio das tabelas do dicionário de dados, quando os extents são alocados ou desalocados em qualquer lugar do banco de dados, o Oracle atualiza as tabelas do dicionário de dados para registrar o novo mapa de armazenamento. A partir do Oracle 8i um novo recurso possibilitando o gerenciamento local dos extents dentro de um tablespace praticamente decretou a morte do tablespace gerenciado por dicionário de dados. Como dito anteriormente, o Oracle mantém um bitmap em cada arquivo de dados de um tablespace gerenciado localmente. Para se criar um tablespace gerenciado localmente, é necessário usar a cláusula `EXTENT MANAGEMENT LOCAL` como o comando `create tablespace`. Comparando com os tablespaces gerenciados por dicionário, os tablespaces gerenciados localmente têm um modo completamente diferente de dimensionar os extents. Os parâmetros de armazenamento `NEXT`, `PCTINCREASE`, `MINEXTENTS`, `MAXEXTENTS` e `DEFAULT_STORAGE` não são válidos nos casos dos tablespaces gerenciados localmente. Em vez disso, existe a opção de especificar um tamanho uniforme para todos os extents ou especificar apenas o tamanho do extent inicial e deixar que o Oracle determine automaticamente o tamanho de todos os extents subsequentes. Os extents uniformes ou dimensionados automaticamente podem ser selecionados especificando as opções `UNIFORM` ou `AUTOALLOCATE`, respectivamente, ao criar um tablespace gerenciado localmente com o comando `CREATE TABLESPACE`.

OBS: Os tablespaces gerenciados localmente ajudam a reduzir a overhead de gerenciamento de espaço eliminando a necessidade de várias gravações nas tabelas do dicionário de dados ou nos segmentos de rollback, o que ocorre necessariamente quando o espaço é gerenciado centralmente por meio do dicionário de dados. Segundo a Oracle, os tablespaces gerenciados por dicionário não serão mais suportados nas futuras versões do Oracle:

"Oracle strongly recommends that you create only locally managed tablespaces. Locally managed tablespaces are much more efficiently managed than dictionary-managed tablespaces. The creation of new dictionary-managed tablespaces is scheduled for desupport."

Outra informação importante é que um tablespace gerenciado por dicionário não pode ser criado caso o tablespace `SYSTEM` seja gerenciado localmente:

```
SQL> create tablespace tbs_test
2 logging
3 datafile /u01/oradata/BD01/test01.dbf' size 5m
4 extent management dictionary;
create tablespace tbs_test
*
ERRO na linha 1:
ORA-12913: Não é possível criar um tablespace gerenciado por dicionário
SQL> select extent_management
2 from dba_tablespaces
3 where tablespace_name='SYSTEM';
EXTENT_MANAGEMENT
-----
LOCAL
```

## 8.16 Propriedades do SGBD

Para exibir as propriedades do banco use a tabela `nls_database_parameters`.

```
SQL> select * from nls_database_parameters
```

PARAMETER	VALUE
NLS_LANGUAGE	AMERICAN
NLS_TERRITORY	AMERICA
NLS_CURRENCY	\$
NLS_ISO_CURRENCY	AMERICA



```

NLS_NUMERIC_CHARACTERS      . ,
NLS_CHARACTERSET             AL32UTF8
NLS_CALENDAR                 GREGORIAN
NLS_DATE_FORMAT              DD-MON-RR
NLS_DATE_LANGUAGE            AMERICAN
NLS_SORT                     BINARY
NLS_TIME_FORMAT              HH.MI.SSXFF AM
NLS_TIMESTAMP_FORMAT         DD-MON-RR HH.MI.SSXFF AM
NLS_TIME_TZ_FORMAT           HH.MI.SSXFF AM TZR
NLS_TIMESTAMP_TZ_FORMAT      DD-MON-RR HH.MI.SSXFF AM TZR
NLS_DUAL_CURRENCY            $
NLS_COMP                     BINARY
NLS_LENGTH_SEMANTICS         BYTE
NLS_NCHAR_CONV_EXCP          FALSE
NLS_NCHAR_CHARACTERSET       AL16UTF16
NLS_RDBMS_VERSION            10.2.0.1.0

20 rows selected

```

### **8.17 Versão do SGBD**

Para exibir as versões dos componentes do banco de dados.

```
SQL> select * from product_component_version;
```

## 9 Apêndice 2 – Oracle PL/SQL

### 9.1 Conceitos

O Oracle possui uma extensão da linguagem SQL chamada PL/SQL. A PL/SQL é uma linguagem estruturada usada para a criação de diversos objetos de banco de dados como triggers, stored procedures, packages e scripts para o SQL Plus.

A linguagem PL/SQL trabalha com o conceito de bloco estruturado. Além da criação de procedures e funções pode-se criar um bloco de comandos sem nome específico e executá-lo. As procedures e funções permitem que parâmetros sejam passados para ela, enquanto um bloco não tem essa capacidade nem tão pouco retorna qualquer valor como uma função.

### 9.2 Vantagens

#### 9.2.1 Portabilidade

Aplicações escritas em PL/SQL são portáveis para qualquer Máquina que rode ORACLE RDBMS com PL/SQL.

#### 9.2.2 Integração com RDBMS

Variáveis PL/SQL podem ser definidas a partir de definições das colunas das tabelas.  
Redução de manutenção das aplicações, pois estas adaptam-se as mudanças da Base de Dados.

#### 9.2.3 Capacidade Procedural

Comandos de controle de fluxo, comandos de repetições e tratamentos de erros;

#### 9.2.4 Produtividade

Desenvolvimento de Procedures e Triggers no Oracle Forms e Oracle Reports.  
Desenvolvimento de Database Triggers, Procedures e Functions a nível do Banco de Dados

### 9.3 Blocos

#### 9.3.1 Estrutura de Básica

Um bloco PL/SQL tem uma estrutura básica composta de três partes.  
Seção de Declaração (Opcional), em que todos os objetos são declarados.  
Seção de Execução, em que os comandos PL/SQL são colocados.  
Seção de Exceção (opcional), em que os erros são tratados.

#### BLOCO PL/SQL

```
DECLARE (Opcional)
    Variáveis, cursores, exceptions definidas pelo usuário

BEGIN (Obrigatório)
    - SQL
    - PL/SQL
    - EXCEPTION(Opcional)
    Ações que são executadas quando ocorrem os erros

END(obrigatório)
```

### 9.3.2 Tipos de Blocos

No quadro abaixo os tipos de blocos suportados em PL/SQL.

**Quadro 14 - Tipos de Blocos**

Anonymous	Procedure	Function
<pre> DECLARE  BEGIN     .....     Exception END;</pre>	<pre> PROCEDURE name is  BEGIN     .....     Exception END;</pre>	<pre> FUNCTION name return     datatype is  BEGIN     .....     Return value;     Exception END;</pre>

## 9.4 DataTypes

O tipos de dados servem para declarar desde atributos de tabelas a variáveis em blocos de PL/SQL.

### 9.4.1 Datatypes mais Utilizados

O Quadro abaixo apresenta os tipos de dados mais comumente utilizados.

**Quadro 15 - Tipos de Dados**

Char(n)	Tamanho Fixo, pode conter uma sequência de 1 a 255 bytes alfanuméricos;
Varchar2(n)	Tamanho Variável, pode conter uma sequência de 1 a 4000 bytes - alfanuméricos.
Long	Tamanho Variável até 2 Gigabytes alfanuméricos nota : só pode existir uma coluna long em cada tabela
Number(p,s)	Numérico com sinal e ponto decimal, sendo precisão de 1 a 38 dígitos
Raw	Binário - Variável até 255 bytes
Long Raw	Binário - Variável até 2 gigabytes - imagem
Date	Data c/ hora, minuto e segundo
Blob	dado binário até 4 Gigabyte
Clob	pode conter uma sequência até 4 Gigabytes - alfanuméricos.
Bfile	dado binário externo até 4 Gigabyte

### 9.4.2 Declaração

Exemplo:

DECLARE

```

NOME          CHAR(30);
SALARIO       NUMBER(11,2);
DEPARTAMENTO   NUMBER(4);
DATANASC      DATE;
SIM           BOOLEAN;
CONT          NUMBER(6) := 0;
PERC          CONSTANT NUMBER(4,2) := 36.00;
```

### 9.4.3 O atributo %TYPE

Atribui à variável que está sendo criada o mesmo tipo de dados usados pela coluna que está no banco de dados.

Exemplo:

```

DECLARE
    V_nome          empregado.nome%TYPE;
    V_saldo          number(7,2);
    V_saldo_min      V_saldo%TYPE :=10;

```

#### 9.4.4 O atributo %ROWTYPE

Declara uma variável composta equivalente à linha de uma tabela. Uma vez criada a variável, podem-se acessar os campos da tabela, usando-se o nome da variável seguido de um ponto e do nome do campo.

Exemplo:

```

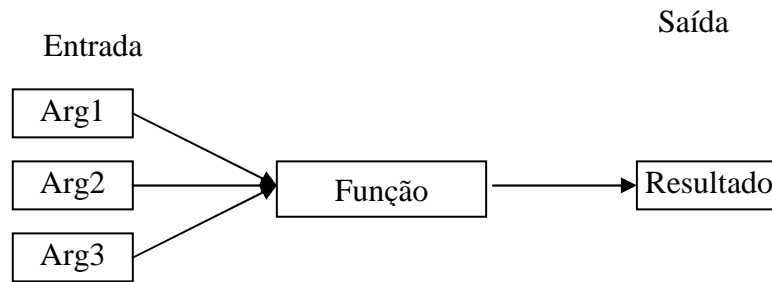
DECLARE
    linha           cliente%ROWTYPE;
Begin
    select * into linha from cliente where cliente_id = 1;
    dbms_output.put_line(linha.cliente_id || linha.nome);
End;

```

### 9.5 Funções

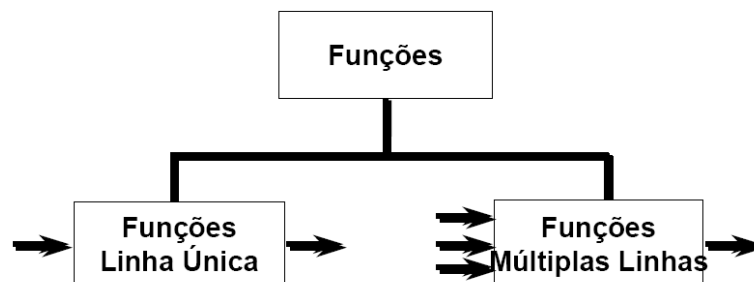
Possuem uma ou mais argumentos de entrada e um valor de saída.

**Figura 9 - Argumentos em Função**



Existem funções de dois tipos  
 Funções de Linha Única  
 Funções de Múltiplas Linhas

**Figura 10 - Tipos de Linhas em Função**

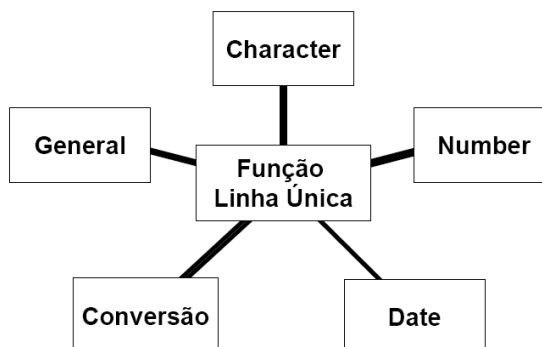


Funções Linha Única  
 Manipula itens de dados  
 Aceita argumentos e retorna um valor  
 Age sobre cada linha retornada  
 Retorna um resultado por linha  
 Pode modificar o tipo de dado  
 Pode ser agrupado

function\_name (column|expression, [arg1, arg2,...])

Podemos contar com o uso de funções matemáticas, data, conversão e caracteres.

**Figura 11 - Tipos de Função**



Exemplo:

```

DECLARE
    Cargo_atual char(10);
BEGIN
    SELECT upper(substr(cargo,1,10)) into cargo_atual
    FROM funcionario
    WHERE rg_funcionario = '2150';
END;
```

### 9.5.1 Funções Matemáticas

O quadro abaixo apresenta algumas funções matemáticas. As funções possuem nomes bastante significativos e muito semelhante a muitas linguagens de programação.

**Quadro 16 - Funções Matemáticas**

FUNÇÕES	EXEMPLO	RESULTADO
ABS	ABS(-10.0)	Retorna o valor absoluto de um número ou expressão matemática.
ACOS	ACOS(15)	Retorna o arco-coseno(em radianos) de um número ou expressão
ATAN	ATAN(15)	Retorna o arco-tangente(em radianos) de um número ou expressão
COS	COS(45)	Retorna o co-seno de um ângulo expresso em radianos.
EXP	EXP(18)	Retorna e (a base dos logaritmos naturais) elevado à uma potência especificada (e <sup>n</sup> ).
LN	LN(10)	Retorna o logaritmo natural de um número positivo.
LOG	LOG(10,2)	Retorna o logaritmo, base m, de n. log(m,n).
MOD	MOD(3,2)	Retorna o resto da divisão do primeiro argumento pelo segundo argumento.
POWER	POWER(2,3)	Retorna m elevado a n-ésima potência, power(m,n).
ROUND	ROUND(100.1234,2)	Retorna um número arredondo em um número de casas especificadas.
SIN	SIN(90)	Retorna o seno de um ângulo medido em radianos.

SQRT	SQRT(9)	Retorna a raiz quadrada de um número não negativo ou expressão matemática.
TAN	TAN(180)	Retorna a tangente de um ângulo que é especificado em radianos.
TRUNC	TRUNC(100.1234,2)	Retorna um número truncado em um número de casas especificadas.

### 9.5.2 Funções de Data

O Oracle armazena datas em um formato interno numérico: Século, ano, mês, dia, horas, minutos, segundos.

O formato padrão é DD-MON-YY.

SYSDATE é uma função que retorna a data e a hora.

DUAL é uma tabela fictícia utilizada para visualizar a data e a hora com o comando SYSDATE.

**Quadro 17 - Funções de Data**

FUNÇÕES	EXEMPLO	RESULTADO
ADD_MONTHS	ADD_MONTHS(HIREDATE,5)	Adiciona 5 meses na data HIREDATE
LAST_DAY	LAST_DAY(SYSDATE)	Retorna a data tomando como parâmetro o 'FMT'
MONTHS_BETWEEN	MONTHS_BETWEEN(HIREDATE,SYSDATE)	Calcula o número de meses BETWEEN entre as datas
NEXT_DAY	NEXT_DAY(HIREDATE,'FRIDAY')	Procura uma sexta-feira após HIREDATE
ROUND	ROUND(SYSDATE,'MONTH')	Retorna uma data arredonda para o segundo argumento.
SYSDATE	SYSDATE	Retorna a data e o horário atuais do sistema
SYSTIMESTAMP	SYSTIMESTAMP	Retorna a data e o horário atuais do sistema com TimeStamp
TIME	TIME	Retorna o horário atual do sistema operacional do sistema.
TRUNC	TRUNC(SYSDATE,FMT)	Trunca a data para a primeira data do 'FMT'

#### 9.5.2.1 Recuperando o nome dos clientes que nascem em um determinado dia.

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'DD') = 01;
```

#### 9.5.2.2 Recuperando o nome dos clientes que nascem em um determinado mês.

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'MM') = 08;
```

#### 9.5.2.3 Recuperando o nome dos clientes que nascem em um determinado ano.

```
SELECT nome_cliente
FROM cliente
WHERE to_char(Data_Nasc,'YYYY') = 1998;
```

### 9.5.3 Funções de conversão

Conversão pode ser de dois tipos:

Conversão Implícita

Conversão Explícita

### 9.5.3.1 Conversão Implícita

Para Atribuições, são convertidas de maneira automática:

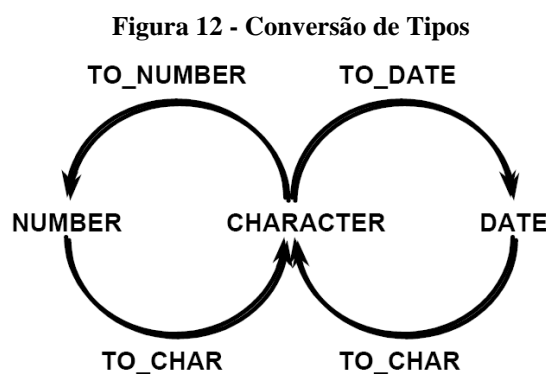
De	Para
VARCHAR2 ou CHAR	NUMBER
VARCHAR2 ou CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

Para comparações de expressões, são convertidas automaticamente:

De	Para
VARCHAR2 ou CHAR	NUMBER
VARCHAR2 ou CHAR	DATE

### 9.5.3.2 Conversão Explícita

A figura abaixo demonstra as conversões explícitas que são necessárias na conversão de tipos de dados em PL/SQL.



Para as conversões explícitas são necessárias funções. Abaixo o quadro apresenta algumas funções de conversão de tipos de dados.

**Quadro 18 - Funções de Conversão**

FUNÇÕES	EXEMPLO	RESULTADO
TO_CHAR	TO_CHAR(SYSDATE, 'fmt ')	Converte colunas do tipo number e data para char.
TO_DATE	TO_DATE('15/05/90', 'DD/MM/YY')	Converte colunas do tipo char para o formato data.
TO_NUMBER	TO_NUMBER(SUBSTR('\$150', 2, 3))	Converte as 3 ultimas (em formato char) para number.

Para converter data é necessário especificar a máscara da data. O quadro abaixo apresenta as máscaras mais comuns para conversão de data.

**Quadro 19 - Formato de Data**

Formato	Descrição
AM	AM ou PM
CC	Século
D	Dia da semana (1-7)
DAY	Dia da semana ('SUNDAY')

DD	Dia do mês (1-31)
DDD	Dia do ano
DY	Dia da semana abreviado ('SUN')
FM	Tira os Brancos ou Zeros da esquerda
HH	Hora do dia (0-12)
HH24	Hora do dia (0-24)
MI	Minutos da Hora
MM	Mês com 2 dígitos
MON	Mês abreviado ('NOV')
MONTH	Mês por extenso ('NOVEMBER')
PM	AM ou PM
RR	Ano com 2 dígitos - especial
RRRR	Ano com 4 dígitos
SS	Segundos do minuto (0 – 59)
SSSSS	Segundos do dia
W	Semana do Mês
WW	Semana do Ano
YEAR	Ano por extenso
YY	Ano com 2 dígitos
YYYY	Ano com 4 dígitos

Utilize os seguintes formatos com a função TO\_CHAR para mostrar um valor numérico como um caracter. O quadro abaixo apresenta as mascaras mais comuns para converter números.

Formato	Descrição
9	Representa um número
0	Força o display de zeros
\$	Coloca um sinal de dólar
L	Utiliza o sinal da moeda corrente
.	Coloca um ponto decimal
,	Coloca um identificador de milhar

#### 9.5.4 Funções de Caracteres

O quadro abaixo apresenta as funções de caracteres mais comuns.

**Quadro 20 - Funções de Caracteres**

FUNÇÕES	EXEMPLO	RESULTADO
ASCII	ASCII ('CASA')	Retorna a representação decimal do caractere dado.
CHR	CHR (45)	Retorna o caractere correspondente ao código decimal ANSI.
CONCAT	CONCAT('bom', 'dia')	Retorna uma string concatenando as duas strings passadas como argumento. Como alternativa pode-se ser usar o    double pipe.
INITCAP	INITCAP ('casa')	Retorna uma string cuja primeira letra foi convertida minúscula.
INSTR	INSTR ('string', 'i')	Retorna um número inteiro com a primeira ocorrência da letra do segundo argumento na string do primeiro argumento.
LOWER	LOWER ('CaSa')	Retorna uma string cujas letras maiúsculas



		foram convertidas para letras minúsculas.
LEFT	LEFT('Casa',2)	Retorna o número de caracteres mais à esquerda especificados a partir de uma string.
LENGTH	LENGTH('Casa')	Retorna o número de caracteres contidos em uma string.
LPAD	LPAD('senha',10,'*')	Retorna a string completando com * até o 10 caractere a esquerda.
LTRIM	LTRIM(' Casa')	Retorna uma string sem espaços em brancos na esquerda.
REPLACE	REPLACE('XUXXU', 'XXX', 'X')	Retorna a string com a substituição de todas as ocorrências do segundo argumento pelo valor do terceiro argumento.
RIGHT	RIGHT('Casa',2)	Retorna o número de caracteres mais à direita especificados a partir de uma string.
RPAD	RPAD('senha',10,'*')	Retorna a string completando com * até o 10 caracter a direita.
RTRIM	RTRIM('Casa ')	Retorna uma string sem espaços em brancos na direita.
SOUNDEX	SOUNDEX('Joao')	Retorna a expressão fonética de uma string. Para a String 'Joao' retorna J000.
STR	STR(10)	Retorna uma representação em forma de string de um número ou expressão.
STRING	STRING(10,45)	Retorna uma string de comprimento especificado que consiste de um único caractere.
SUBSTR	SUBSTR('String',1,3)	Retorna uma substring da posição inicial(1) até a posição final(3).
TRIM	TRIM(' Casa ')	Retorna uma string sem espaços em brancos na esquerda e direita.
UPPER	UPPER('casa')	Retorna uma string cujas letras minúsculas foram convertidas para letras maiúsculas.

#### 9.5.4.1 Consultar um literal que contenha espaços e somente em maiúsculo.

```
SELECT nome_cliente
FROM cliente
WHERE upper(trim(nome_cliente)) = 'PEDRO';
```

#### 9.5.5 Funções Diversas

Existem funções dos mais diversos tipos, abaixo o quadro lista alguma delas.

**Quadro 21 - Funções Diversas**

FUNÇÕES	EXEMPLO	RESULTADO
DECODE	DECODE(VARIAVEL, VAL1, EXPR1, VAL2, EXPR2, ...)	Retorna a expressão EXPR1 se a VARIAVEL for igual a VAL1, caso contrário procura se a VARIAVEL é igual a VAL2 e retorna a expressão VAL2 e assim por diante.
ISNULL	ISNULL(EXPRESSAO)	Retorna verdadeiro ou falso se a EXPRESSÃO quando avaliada retorna valor nulo.
ISNUMERIC	ISNUMERIC(EXPRESSAO)	Retorna verdadeiro ou falso se a EXPRESSÃO quando avaliada pode ser convertida para número.
NVL	NVL(EXPR1, EXPR2)	Se EXPR1 for avaliada como um valor não nulo, a função retorna EXPR1 caso contrário retorna EXPR2.

GREAST	SELECT GREATEST(1, 2, 3, 4) FROM DUAL	Retorno o maior valor de um conjunto, seja de valores literais ou numéricos.
LEAST	SELECT LEAST(1, 2, 3, 4) FROM DUAL	Retorno o menor valor de um conjunto, seja de valores literais ou numéricos.

### 9.5.6 Funções de Erro do SQL Oracle

O quadro abaixo apresenta algumas funções de tratamento de erros em PL/SQL.

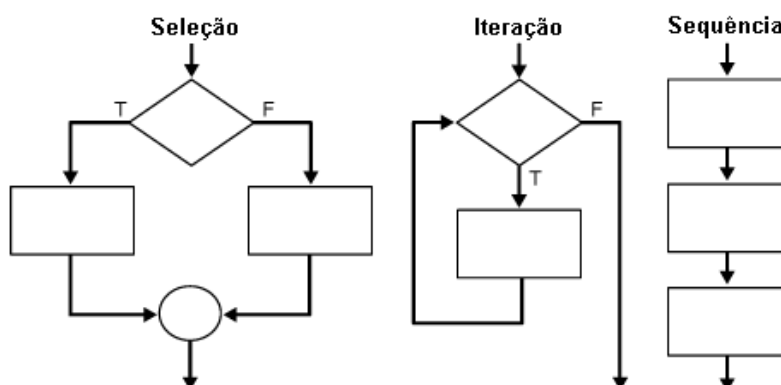
**Quadro 22 - Funções de Erro**

FUNÇÕES	EXEMPLO	RESULTADO
SQLERRCODE	SQLERRCODE	Fornece um código de erro nativo retornado pela instrução SQL executada mais recentemente.
SQLERRTEXT	SQLERRTEXT	Fornece o texto da mensagem de erro nativo retornado pela instrução SQL executada mais recentemente.
SQLROWCOUNT	SQLROWCOUNT	Retorna o número de linhas afetadas pela instrução SQL executada mais recentemente.

## 9.6 Estruturas de Controle

Como a maioria das linguagens procedurais, o PL/SQL possui comandos para controlar o fluxo de execução do programa. São eles que fazem a diferença realizando desvios, analisando condições e permitindo a tomada de decisões dentro do programa.

**Figura 13 - Estruturas de Controle**



### 9.6.1 Comando IF...THEN

O comando IF...THEN tem como finalidade avaliar uma condição e executa uma ou mais linhas de comandos somente se a condição analisada for verdadeira.

Sintaxes:

```
1. IF <condição> THEN
    <comandos>;
END IF;
```

```
2. IF <condição> THEN
    <comandos>;
ELSE
    <comandos>;
END IF;
```

```
3. IF <condição> THEN
```

```

        <comandos>;
ELSIF <condição> THEN
        <comandos>;
END IF;

```

```

4. IF <condição> THEN
    <comandos>;
ELSIF <condição> THEN
    <comandos>;
ELSE
    <comandos>;
END IF;

```

```

5. IF <condição> THEN
    IF <condição> THEN
        <comandos>;
    END IF;
END IF;

```

Exemplo

```

DECLARE
    QUANT NUMBER(3);
BEGIN
    SELECT ES.NR_QTD INTO QUANT
    FROM ESTOQUE ES
    WHERE CD_PROD = 30;
    IF QUANT > 0 AND QUANT < 3000 THEN
        UPDATE ESTOQUE SET NR_QTD = QUANT + 1
        WHERE CD_PROD = 30;
    ELSIF QUANT >= 3000 THEN
        INSERT INTO ALERTA(PROD,ERRO) VALUES(30,'MÁXIMO');
    ELSE
        INSERT INTO ALERTA(PROD,ERRO) VALUES(30,'MÍNIMO');
    END IF;
END;

```

### 9.6.2 Comando CASE..WHEN

O comando CASE...WHEN tem como finalidade avaliar múltiplas condições e executa uma ou mais linhas de comandos somente se uma condição analisada for verdadeira.

Sintaxe:

```

CASE <expressao>
    WHEN (condicao1) THEN comandos1;
    WHEN (condicao2) THEN comandos2;
    WHEN (condicao3) THEN comandos3;
    ...
    [ELSE comandoselse;]
END CASE;

```

Exemplo:

```

DECLARE
    hora number;
BEGIN
    SELECT to_char(sysdate,'HH24') INTO hora FROM dual;
    CASE hora
        WHEN (7) THEN dbms_output.put_line('Cafe da Manha');

```

```

        WHEN (12) THEN dbms_output.put_line('Almoco');
        WHEN (18) THEN dbms_output.put_line('Janta');
        ELSE dbms_output.put_line('Estudar');
    END CASE;
END;
```

### 9.6.3 Comando LOOP

O comando LOOP executa um grupo de comandos indefinidamente ou até que uma condição force a 'quebra' do LOOP e desvie a execução do programa para outro lugar. O comando LOOP é usado em conjunto com o comando EXIT, responsável pela parada de execução do LOOP.

Sintaxe:

```

LOOP
    <comandos>;
    EXIT;
    <comandos>;
END LOOP;
```

ou

```

LOOP
    <comandos>;
    EXIT [<nome_de_label>] WHEN <condição>
    <comandos>;
END LOOP;
```

Exemplo:

```

DECLARE
    X          NUMBER := 0;
    CONTADOR NUMBER := 0;
BEGIN
    LOOP
        X := X + 1000;
        CONTADOR := CONTADOR + 1;
        IF CONTADOR > 4 THEN
            EXIT;
        END IF;
        DBMS_OUTPUT.PUT_LINE (X || ' ' || CONTADOR || ' LOOP');
    END LOOP;
END;
```

### 9.6.4 Comando FOR...LOOP

O comando FOR...LOOP é uma variação do comando LOOP. Aqui os comandos são executados automaticamente até que uma condição avaliada retorne falsa. A opção REVERSE faz com que o laço seja inverso, ou seja do valor final até o valor inicial.

Sintaxe:

```

FOR <variável_contadora> IN [REVERSE] <valor_inicial> .. <valor_final> LOOP
    <comandos>
END LOOP;
```

Exemplo:

```

DECLARE
    A,B    NUMBER(3) := 0;
BEGIN
```

```

FOR A IN 1..25 LOOP
    B:= B + 1;
    DBMS_OUTPUT.PUT_LINE('LOOP - ' || B);
END LOOP;
END;

```

### 9.6.5 Comando WHILE

Essa estrutura analisa uma condição e somente se ela for verdadeira executa os comandos contidos dentro da estrutura.

Sintaxe:

```

WHILE <condição> LOOP
    <comandos>;
    <comandos>;
END LOOP;

```

Exemplo:

```

DECLARE
    X NUMBER(3);
    Y VARCHAR2(30);
    K DATE;
    J NUMBER(3);
BEGIN
    X:= 0;
    WHILE X<= 100 LOOP
        K:= SYSDATE-X;
        Y := 30;
        INSERT INTO TESTE VALUES (X,Y,K);
        X := X + 1;
    END LOOP;
    COMMIT;
END;

```

## 9.7 Operadores

### 9.7.1 Operador CASE

Uma operação CASE retorna um valor de um conjunto especificado de valores, dependendo de uma condição especificada

Exemplo:

```

Select nome_funcionario,
CASE
    WHEN salario_funcionario<1000 THEN 'Baixo'
    WHEN salario_funcionario>1000 AND salario_funcionario<10000 THEN 'Alto'
ELSE
    'Muito Alto'
END
From funcionario;

```

Muito similar pode ser feito com a função DECODE, com certas limitações. O DECODE só realiza comparação de igualdade.

Exemplo:

```

Select nome_funcionario,

```

```
        DECODE(Sexo,
                'M', 'Masculino',
                'F', 'Feminino')
From funcionario;
```

### 9.7.2 Operador CAST

O operador CAST converte um valor escalar especificado em um tipo de dados escalar especificado.

Exemplo:

```
Select CAST(nome_funcionario AS VARCHAR(100)), salario_funcionario
From empregado;
```

## 10 Apêndice 3 - Exemplo de um Banco de Dados

Relação EMPREGADO					
<u>EmpregadoId</u>	Nome	CPF	DeptId	SupervisorId	Salario
10101010	João Luiz	11111111	1	<i>NULO</i>	3.000,00
20202020	Fernando	22222222	2	10101010	2.500,00
30303030	Ricardo	33333333	2	10101010	2.300,00
40404040	Jorge	44444444	2	20202020	4.200,00
50505050	Renato	55555555	3	20202020	1.300,00

Relação DEPTO		
<u>DeptId</u>	Nome	GerentId
1	Contabilidade	10101010
2	Engenharia Civil	30303030
3	Engenharia Mecânica	20202020

Relação PROJETO		
<u>ProjetoId</u>	Nome	Localizacao
5	Financeiro 1	São Paulo
10	Motor 3	Rio Claro
20	Prédio Central	Campinas

Relação DEPENDENTE				
<u>EmpregadoId</u>	<u>Nome</u>	DtNascimento	Relacao	Sexo
10101010	Jorge	27/12/86	Filho	Masculino
10101010	Luiz	18/11/79	Filho	Masculino
20202020	Fernanda	14/02/69	Conjuge	Feminino
20202020	Angelo	10/02/95	Filho	Masculino
30303030	Adreia	01/05/90	Filho	Feminino

Relação DEPTO_PROJ	
<u>DeptId</u>	<u>ProjetoId</u>
2	5
3	10
2	20

Relação EMP_PROJ		
<u>EmpregadoId</u>	<u>ProjetoId</u>	Horas
20202020	5	10
20202020	10	25
30303030	5	35
40404040	20	50
50505050	20	35