

Marcos Alexandruk

SQL

Structured Query Language

rápido e fácil

Universidade Nove de Julho – UNINOVE

Rua Vergueiro, 235/249 – 12º andar

CEP: 01504-001 – Liberdade – São Paulo, SP – Brasil

Tel.: (11) 3385-9191 – editora@uninove.br

MARCOS ALEXANDRUK

SQL

STRUCTURED QUERY LANGUAGE

RÁPIDO E FÁCIL

São Paulo
2018



© 2018 UNINOVE

Todos os direitos reservados. A reprodução desta publicação, no todo ou em parte, constitui violação do copyright (Lei nº 9.610/98). Nenhuma parte desta publicação pode ser reproduzida por qualquer meio, sem a prévia autorização da UNINOVE.

Conselho Editorial Eduardo Storópoli
 Maria Cristina Barbosa Storópoli
 Marcos Alberto Bussab

Os conceitos emitidos neste livro são de inteira responsabilidade do autor.

Capa: Big Time Serviços Editoriais

Imagem da Capa: 123RF

Figuras 1 a 14: *Copyright Oracle and its affiliates. Use with permission*

O uso das 14 figuras foi autorizado pela Oracle

Editoração eletrônica: Big Time Serviços Editoriais

Revisão: Antonio Marcos Cavalheiro

Catálogo na Publicação (CIP)
Cristiane dos Santos Monteiro - CRB/8 7474

Alexandruk, Marcos

SQL – Structured Query Language: rápido e fácil / Marcos Alexandruk. —
São Paulo : Universidade Nove de Julho, UNINOVE, 2018.
232 p. il.

ISBN: 978-85-89852-66-1 (e-book)
ISBN: 978-85-89852-67-8 (impresso)

1. SQL (Linguagem de programação de computador). 2. Banco de dados. 3.
I. Autor.

CDU 004.655.3SQL

SUMÁRIO

Agradecimentos.....	13
Prefácio	15
Introdução	17

Capítulo 1 – BREVE HISTÓRIA DOS SISTEMAS DE BANCOS DE DADOS RELACIONAIS..... 19

1.1 Bancos de dados relacionais – conceitos básicos	21
1.2 SQL – Structured Query Language.....	23
1.3 Alguns dos principais sistemas de bancos de dados relacionais	23
1.3.1 Oracle Database.....	24
1.3.2 Microsoft SQL Server	25
1.3.3 IBM DB2	26
1.3.4 MySQL.....	27
1.3.5 PostgreSQL.....	28
1.3.6 SQLite	28
1.4 Edições do Oracle Database.....	29
1.4.1 Oracle Database Standard Edition One	29
1.4.2 Oracle Database Standard Edition.....	29
1.4.3 Oracle Database Enterprise Edition	30
1.4.4 Oracle Database Express Edition	30
1.4.5 Oracle Database Personal Edition	30
1.5 Interfaces para Usuários do Oracle Database.....	30
1.5.1 SQL Plus	30
1.5.1.1 DESCRIBE	31
1.5.1.2 Comandos de edição	32
1.5.1.3 Salvar, recuperar e executar arquivos	33
1.5.1.4 Formatar colunas	34
1.5.1.5 Variáveis	35
1.5.1.6 Ajuda do SQL*PLUS.....	36
1.5.2 SQL Developer	37
1.5.3 Oracle Application Express.....	39
Resumo	48
Exercícios.....	49

Capítulo 2 – TIPOS DE DADOS 51

2.1 NUMBER (p,s)	51
2.2 VARCHAR2 (tamanho)	52
2.3 CHAR	52
2.4 DATE	53
2.4.1 TIMESTAMP	53
2.5 LONG	53
2.6 CLOB	53
2.7 RAW	53
2.8 LONG RAW	54
2.9 BLOB	54
2.10 BFILE	54
2.11 ROWID	54
Resumo	54
Exercícios	55

Capítulo 3 – DDL – DATA DEFINITION LANGUAGE 56

3.1 CREATE TABLE	56
3.1.1 CONSTRAINTS	58
3.1.1.1 PRIMARY KEY	58
3.1.1.2 FOREIGN KEY	60
3.1.1.3 CHECK	63
3.1.1.4 UNIQUE	64
3.1.1.5 NOT NULL	65
3.1.1.6 DEFAULT	66
3.1.2 CRIAR UMA TABELA COM BASE EM OUTRA	66
3.2 ALTER TABLE	67
3.2.1 Adicionar uma coluna	67
3.2.2 Modificar uma coluna	68
3.2.3 Renomear uma coluna	69
3.2.4 Eliminar uma coluna	69
3.2.5 Adicionar uma constraint (CHAVE PRIMÁRIA)	70
3.2.6 Adicionar uma constraint (CHAVE ESTRANGEIRA)	70
3.2.7 Adicionar uma constraint (NOT NULL)	71
3.2.8 Eliminar uma constraint	71
3.2.9 Desabilitar uma constraint	71
3.10 Habilitar uma constraint	72
3.3 DROP TABLE	72

3.4 TRUNCATE.....	73
3.5 RENAME.....	73
3.6 SEQUENCE	73
Resumo	74
Exercícios.....	74

Capítulo 4 – DML – DATA MANIPULATION LANGUAGE.....76

4.1 SELECT	76
4.2 INSERT.....	78
4.3 DELETE.....	80
4.4 UPDATE.....	81
4.5 SELECT FOR UPDATE	82
4.6 MERGE.....	82
Resumo	84
Exercícios.....	84

Capítulo 5 – QUERIES (CONSULTAS)..... 86

5.1 Cláusula WHERE.....	86
5.2 Operadores.....	86
5.2.1 Operadores de comparação	86
5.2.2 Operadores lógicos	87
5.2.3 Operadores SQL.....	88
5.3 ALIAS.....	91
5.4 DISTINCT.....	92
5.5 ORDER BY.....	93
5.6 GROUP BY	94
5.7 Cláusula HAVING	94
5.8 CASE.....	95
Resumo	97
Exercícios.....	98

Capítulo 6 – FUNÇÕES..... 99

6.1 Funções de grupo.....	99
6.1.1 SUM	99
6.1.2 AVG.....	100
6.1.3 MAX	100
6.1.4 MIN.....	101

6.1.5 COUNT	101
6.1.6 MEDIAN	102
6.1.7 STDDEV	103
6.1.8 VARIANCE	103
6.2 Funções de linha	104
6.2.1 UPPER	105
6.2.2 LOWER	106
6.2.3 INITCAP	106
6.2.4 LPAD	106
6.2.5 RPAD	107
6.2.6 SUBSTR	109
6.2.7 REPLACE	110
6.2.8 TRANSLATE	111
6.2.9 CONCAT	112
6.10 LTRIM	112
6.11 RTRIM	113
6.12 TRIM	114
6.13 LENGTH	115
6.14 CHR	116
6.15 ASCII	116
6.3 Funções numéricas	116
6.3.1 ABS	117
6.3.2 CEIL	118
6.3.3 FLOOR	118
6.3.4 ROUND	119
6.3.5 TRUNC	120
6.3.6 MOD	121
6.3.7 POWER	121
6.3.8 SQRT	121
6.3.9 SIGN	122
6.4 Funções de conversão	123
6.4.1 TO_CHAR	123
6.4.2 TO_DATE	124
6.4.3 TO_NUMBER	125
6.5 Funções de expressões regulares	125
6.5.1 REGEXP_LIKE()	128
6.5.2 REGEXP_INSTR()	129
6.5.3 REGEXP_REPLACE()	129
6.5.4 REGEXP_SUBSTR()	129
6.5.5 REGEXP_COUNT()	130

6.6 Outras funções.....	130
6.6.1 NVL.....	131
6.6.2 NULLIF	131
6.6.3 DECODE.....	132
Resumo	132
Exercícios.....	133

Capítulo 7 – SUBQUERIES (SUBCONSULTAS)..... 135

7.1 Subconsultas de uma linha	135
7.2 Subconsultas de várias linhas	136
7.2.1 Utilizando ANY em consultas de várias linhas.....	137
7.2.2 Utilizando ALL em consultas de várias linhas	138
7.3 Subconsultas em uma cláusula FROM.....	140
7.4 EXISTS e NOT EXISTS em subconsultas.....	140
Resumo	141
Exercícios.....	142

Capítulo 8 – JOINS (JUNÇÕES)..... 144

8.2 Natural Join.....	146
8.3 Junção baseada em nomes de colunas.....	146
8.4 Outer Join.....	147
8.4.1 Left Outer Join.....	147
8.4.2 Right Outer Join	148
8.4.3 Full Outer Join.....	149
8.5 Non Equi Join	149
8.6 Self join	151
8.7 Cross Join	152
Resumo	153
Exercícios.....	154

Capítulo 9 – OPERAÇÕES DE CONJUNTO..... 156

9.1 UNION (União).....	156
9.2 INTERSECT (Interseção)	158
9.3 MINUS (Diferença)	158
Resumo	160
Exercícios.....	160

Capítulo 10 – CONSULTAS AVANÇADAS 162

10.1 ROLLUP	162
10.1.1 ROLLUP – usando GROUPING()	165
10.2 CUBE	166
10.2.1 CUBE usando GROUPING()	167
10.3 Consultas hierárquicas	168
Resumo	172
Exercícios	173

Capítulo 11 – DTL – DATA TRANSACT LANGUAGE..... 175

11.1 COMMIT	176
11.2 ROLLBACK	176
11.3 SAVEPOINTS	177
Resumo	179
Exercícios	179

Capítulo 12 – DCL – DATA CONTROL LANGUAGE..... 181

12.1 Privilégios de sistema	181
12.2 Privilégios de objeto	182
12.3 GRANT	182
12.4 REVOKE	184
Resumo	184
Exercícios	185

Capítulo 13 – INDEXES (ÍNDICES) 187

13.1 Índices únicos (exclusivos)	187
13.2 Índices não únicos (não exclusivos)	188
13.3 Renomear um índice	188
13.4 Eliminar um índice	188
Resumo	189
Exercícios	189

Capítulo 14 – VIEWS (VISÕES) 191

14.1 Visões regulares	191
14.1.1 Visões regulares – READ ONLY	192
14.1.2 Visões regulares – apelidos para colunas	193

14.1.3 Visões regulares – baseadas em junções	193
14.2 Visões materializadas.....	195
14.3 Visões de objetos	196
14.4 Visões "XML Type"	198
Resumo	199
Exercícios.....	200

Capítulo 15 – SEQUENCES (SEQUÊNCIAS)..... 202

15.1 Criando uma sequência.....	202
15.1.1 START WITH.....	203
15.1.2 INCREMENT BY.....	203
15.1.3 MINVALUE e MAXVALUE.....	203
15.1.4 CYCLE.....	203
15.1.5 CACHE.....	203
15.1.6 ORDER.....	203
15.2 Usando uma sequência.....	204
15.3 Sequência: consultando o dicionário de dados	205
15.4 Modificando uma sequência.....	205
15.5 Excluindo uma sequência	205
Resumo	205
Exercícios.....	206

Capítulo 16 – SYNONYM (SINÔNIMOS)..... 208

16.1 Substituindo um SYNONYM.....	209
16.2 Excluindo um SYNONYM	209
Resumo	210
Exercícios.....	210
Referências	213

APÊNDICE – PALAVRAS RESERVADAS ORACLE 215

O AUTOR..... 220

À minha esposa Keli e ao meu filho Victor,
pelo apoio incondicional e constante
incentivo.

Aos meus pais, Boris e Olga, pelo tempo que
dedicaram à minha educação e por terem me
preparado para enfrentar os desafios da vida.

AGRADECIMENTOS

Ao Prof. Marcos Alberto Bussab, Diretor dos Cursos de Informática, pelo incentivo e apoio.

Aos meus colegas, gestores acadêmicos: Prof. Aguinaldo Alberto de Sousa Junior, Prof. Cleber Vinícios Filippin, Prof. Evandro Carlos Teruel, Prof. Hebert Bratefixe Alquimim, Prof. Jakov Trofo Surjan, Prof. Jefferson dos Santos Marques, Prof. José Azanha da Silva Neto, Prof. Luciano Gillieron Gavinho, Prof. Marcus Vasconcelos de Castro, Prof. Nilson Salvetti, Prof. Ovídio Lopes da Cruz Netto, Prof. Thiago Gaziani Traue, pelo incentivo em todas as fases deste trabalho.

Ao corpo docente dos cursos da Diretoria dos Cursos de Informática pelo inestimável apoio.

*A educação é a arma mais poderosa que você
pode usar para mudar o mundo.*

NELSON MANDELA

PREFÁCIO

Desde a criação dos bancos de dados relacionais na década de 1970 assistimos a uma evolução tão acelerada da tecnologia que surpreendeu até mesmo as pessoas mais otimistas e entusiastas. A era da informação, que teve início na década de 1970 e se acentuou na década de 1980, com a difusão dos computadores pessoais, avanço das tecnologias de telecomunicação e uso civil da Internet, revolucionou a forma como as pessoas se comunicam e se relacionam e a forma como as empresas fazem negócios, abrindo caminho para a globalização.

Na base de toda essa evolução estão os Sistemas Gerenciadores de Banco de Dados (SQL Server, Oracle, DB2, MySQL, PostgreSQL etc.), que permitem que os dados sejam armazenados, organizados e possam ser acessados de diversas maneiras, gerando informações úteis para as pessoas e informações estratégicas para as empresas. A empresa Cisco Systems, no artigo "The Zettabyte Era: Trends and Analysis", publicado em 2017, prevê o crescimento do tráfego mensal global pelas redes de 96 Exabytes em 2016 para 278 Exabytes em 2021. Esse volume de dados está distribuído e armazenado em bases de dados ao redor do mundo e será necessária mão de obra especializada para lidar com eles e garantir consistência, segurança e disponibilidade. A linguagem SQL é usada para essa finalidade, para garantir a disponibilidade e segurança dos dados às pessoas, empresas e aplicações.

Segundo a consultoria de recrutamento e seleção Catho, um profissional da área de banco de dados em início de carreira chega a ganhar até R\$ 6.000,00. O valor atrativo se justifica pela falta de mão de obra especializada no segmento, que eleva o valor do salário do profissional capacitado.

É nesse cenário que o livro Structured Query Language – Rápido e Fácil, do Professor Marcos Alexandruk, se insere, como material de excelência para a capacitação profissional. A linguagem fácil e a contextualização com exemplos práticos, aliada à experiência didática do Professor Marcos Alexandruk, torna a leitura do livro prazerosa, prendendo a atenção do leitor do início ao fim. Estas características tornam a obra um material valioso para iniciantes, profissionais atuantes no mercado, estudantes universitários e autodidatas que pretendem obter certificações internacionais na área. É uma obra completa, que aborda a linguagem SQL de forma ampla e descomplicada.

Sobre o autor, Marcos Alexandruk, conheci-o há oito anos, quando, depois de uma carreira de sucesso no mercado de trabalho, atuava como Gestor e Professor da Universidade Nove de Julho. Já no primeiro contato admirei-me do profundo conhecimento e a serenidade como lidava com situações complexas e complicadas, convertendo-as, a partir de sua experiência, didática e poder de persuasão, em situações simples. Este livro expressa bem esta habilidade do Professor Marcos Alexandruk. Que os próximos não demorem, pois carecemos de autores dessa magnitude.

EVANDRO CARLOS TERUEL

INTRODUÇÃO

Durante os mais de trinta anos de vida profissional, dos quais dediquei com muito prazer os últimos doze à vida acadêmica, atuando como professor e coordenador em uma das maiores instituições de ensino superior do Brasil, tive a oportunidade de participar da formação de milhares de alunos em cursos de tecnologia e bacharelado.

Muitos deles atuam em empresas de pequeno, médio e grande porte como projetistas, programadores ou administradores de bancos de dados.

Uma das maiores satisfações que tenho é saber que contribuí de alguma forma para que eles conquistassem o lugar que almejavam. Tenho certeza que o mérito maior é de cada um deles, pois foi necessário esforço e dedicação para atingirem seus objetivos.

Nestes anos que tenho dedicado à vida acadêmica, elaborei material didático na forma de apostilas e resumos, além de conteúdo disponibilizado no AVA (Ambiente Virtual de Aprendizagem) da instituição na qual atuo.

Incentivado por colegas e outros gestores, resolvi elaborar este livro com o objetivo de colaborar na formação de tecnólogos e bacharéis da área da computação e também de ajudar qualquer pessoa que realmente estiver disposta a conhecer e utilizar, nas mais diversas áreas, os valiosos recursos da linguagem SQL.

Esta obra está estruturada de forma diferente de muitas outras, que primeiro apresentam como realizar consultas em bases de dados já existentes para, mais adiante, apresentarem como estas bases são criadas.

Procurei ser sucinto na parte teórica. Portanto, você observará que as explicações sobre o que cada comando da linguagem SQL realiza são bem objetivas. Por outro lado, apresento exemplos em cada item para que o leitor possa, se assim desejar, colocar em prática seus conhecimentos. Acredito que esta é a melhor maneira de assimilar uma linguagem.

Quando concluir a leitura e realizar os exercícios propostos acredito que você estará apto a executar as seguintes operações em bancos de dados relacionais:

- 1º – Criar uma base de dados relacional (criar, alterar e eliminar tabelas);
- 2º – Manipular dados na base criada (inserir, atualizar e excluir dados);
- 3º – Realizar consultas simples e complexas a partir de uma ou mais tabelas;
- 4º – Controlar as transações SQL;
- 5º – Administrar os privilégios dos usuários do banco de dados;
- 6º – Criar índices (para melhorar o desempenho de suas consultas);
- 7º – Criar visões de dados (consultas gravadas com base em uma ou mais tabelas).

Antes de passar aos exercícios, localizados no final de cada capítulo, apresentarei um resumo de tudo aquilo que foi abordado para melhor fixação.

Conforme será apresentado a seguir, há muitos SGBDRs (Sistemas de Gerenciamento de Banco de Dados Relacionais). Os comandos são muitas vezes exatamente os mesmos, independentemente do sistema de banco de dados. No entanto, é importante avisar que existem de fato pequenas "variações" conforme cada empresa resolveu implementar seu produto.

Os comandos apresentados neste livro são compatíveis na sua totalidade com o SGBD (Sistema de Gerenciamento de Banco de Dados) Oracle Database nas suas últimas versões (11g e 12c). Justifico a escolha por se tratar, conforme várias pesquisas, do produto mais amplamente utilizado pelas grandes corporações independentemente de seu segmento de mercado.

Embora reconheça que você provavelmente está ansioso para colocar tudo isso em prática, apresentarei antes algumas informações importantes sobre bancos de dados relacionais e sobre a linguagem SQL. Acredite: farei isso de uma forma breve.

CAPÍTULO 1 – BREVE HISTÓRIA DOS SISTEMAS DE BANCOS DE DADOS RELACIONAIS

No mesmo mês em que a seleção brasileira de futebol se sagrava tricampeã no México – junho 1970 –, o periódico Communications of the ACM, editado pela Association of Computer Machinery (ACM), publicava o artigo "A Relational Model of Data for Large Shared Data Banks" (em tradução livre: "Um Modelo Relacional de Dados para Grandes Bancos de Dados Compartilhados") de autoria do matemático e pesquisador da IBM, Edgar Frank Codd.

Em meados da década de 1970, Donald D. Chamberlin e Raymond F. Boyce, pesquisadores no IBM San Jose Research Laboratory, apresentaram uma linguagem baseada na álgebra relacional (usada como fundamento no artigo de Codd) denominada SEQUEL (Structured English Query Language). Algum tempo depois o nome da linguagem foi alterado simplesmente para SQL (Structured Query Language).

A partir do que foi apresentado por Codd e de outras pesquisas na área, a IBM – através do System R – e, logo após, outras empresas, passaram a implementar soluções através de sistemas denominados "Relational Database Management System" (RDBMS), ou, como vieram a ser conhecidos em português: "Sistemas de Gerenciamento de Banco de Dados Relacionais" (SGBDR).

Em junho de 1979, a Relational Software, Inc. (atualmente Oracle) apresentou ao mercado o primeiro produto comercial com base na linguagem SQL, o Oracle V2 (Version 2), para computadores VAX. (Você pode estar se perguntando: Por que versão 2? Houve uma versão anterior? A resposta é: sim. Porém, no início a Relational atendia principalmente agências governamentais americanas. A versão 2, conforme explicado, foi a primeira a ser distribuída para o mercado em geral.)

Nas décadas de 1970 e 1980 outras empresas passaram a desenvolver seus próprios sistemas de bancos de dados relacionais com base na SQL. Portanto, tornou-se necessário "padronizar" a linguagem.

É neste contexto que, a partir de 1986, a ANSI (American National Standards Institute) e, mais tarde, a ISO (International Organization

for Standardization) passaram a "padronizar" a linguagem SQL. Nas décadas seguintes as duas organizações reeditaram suas normas com o objetivo de atualizá-las à medida que a linguagem apresentava mais recursos como, por exemplo, orientação a objetos.

Embora os produtos lançados por diferentes empresas apresentem algumas características específicas, o fato da linguagem SQL ter sido "padronizada" representa uma grande vantagem para todos aqueles que querem trabalhar com bancos de dados relacionais. (Imagine se cada empresa resolvesse apresentar uma linguagem diferente para que os usuários interagissem com seus respectivos sistemas de bancos de dados relacionais.)

Este deve, sem dúvida, ser um fator para motivá-lo a aprender e conhecer com profundidade os valiosos recursos da linguagem SQL.

Mesmo que você não seja um profissional da área de banco de dados, conhecer SQL pode ser de grande ajuda no seu dia a dia profissional. Observe a seguir alguns exemplos.

O profissional da área de redes de computadores pode imaginar que não é tão necessário conhecer banco de dados. No entanto, pare e pense: o que faz um servidor de DNS (Domain Name System)? Isto é da sua "área", certo? Vamos explicar para aqueles que não são da área de redes: os servidores DNS são responsáveis por traduzir para os números IP os endereços dos sites digitados nos navegadores. E como eles fazem isso? A resposta é simples: "consultando" uma tabela de correspondências para determinar qual o número IP (e o número da porta) do site que o usuário deseja acessar.

Àqueles que trabalham na área de desenvolvimento, independentemente da linguagem que utilizem, é desnecessário explicar por que conhecer SQL é fundamental. A vasta maioria das aplicações utilizam bancos de dados. E dentre estas aplicações a maior parte ainda utiliza os bancos de dados relacionais. (Sim há outros tipos de bancos de dados, além dos relacionais, mas está fora do escopo deste livro abordá-los).

Administradores e outros profissionais poderão elaborar relatórios valiosos para apoio à tomada de decisões utilizando filtros e funções de agregação disponibilizados através da SQL.

O objetivo deste livro é apresentar uma visão prática da SQL. Preferimos apresentar explicações breves sobre cada recurso da linguagem e focar mais em exemplos.

Portanto, para que o leitor tire maior proveito, sugerimos que instale o software em seu equipamento ou que utilize a plataforma online disponibilizada pela Oracle, que será apresentada mais adiante.

Esperamos que ao final da leitura, quando tiver executado cada exemplo e realizado os exercícios propostos, o leitor tenha adquirido um bom conhecimento da SQL.

1.1 BANCOS DE DADOS RELACIONAIS – CONCEITOS BÁSICOS

É muito simples entender os conceitos básicos que envolvem os bancos de dados relacionais. A forma como os dados são armazenados e recuperados através dos bancos de dados relacionais fizeram deste modelo o mais utilizado nas últimas décadas.

A estrutura básica dos bancos de dados relacionais são as tabelas (também conhecidas como "relações"). Uma tabela é uma estrutura bidimensional formada por colunas e por linhas. Apresentam, portanto, uma estrutura similar às planilhas que muito provavelmente você já utilizou em aplicativos como o Microsoft Excel.

Outra característica importante dos bancos de dados relacionais é a capacidade de relacionar dados entre duas ou mais tabelas, isto é, criar "relacionamentos" entre as tabelas. Isto é implementado através de campos ou colunas com valores comuns.

Nos próximos capítulos vamos apresentar muitos exemplos de tabelas que se relacionam. No entanto, para melhorar a compreensão apresentaremos a seguir um exemplo de duas tabelas que se relacionam.

TABELA: CLIENTE

ID_CLIENTE	NOME_CLIENTE	FONE_CLIENTE
1001	ANTONIO ALVARES	5555-1111
1002	BEATRIZ BARBOSA	5555-2222

TABELA: PEDIDO

NR_PEDIDO	ID_CLIENTE	DT_PEDIDO
0001	1001	11/07/2016
0002	1002	11/07/2016
0003	1001	12/07/2016
0004	1002	12/07/2016

Quando observamos os dados acima, verificamos que determinados valores da coluna ID_CLIENTE da tabela CLIENTE repetem-se na coluna ID_CLIENTE da tabela PEDIDO. Esta é a forma utilizada nos bancos de dados relacionais para implementar o relacionamento entre as tabelas. Desta forma podemos determinar, por exemplo, que o cliente ANTONIO ALVARES realizou duas compras que correspondem aos pedidos 0001 e 0003.

É muito comum, no entanto, que aqueles que não estão habituados a trabalhar com bancos de dados relacionais (e que costumam organizar seus dados em planilhas eletrônicas) fiquem pensando: "Mas por que não colocar todos estes dados em uma única tabela?". "Não seria mais prático?".

O que estão propondo, tomando-se como base os dados acima, é o seguinte:

NR_PEDIDO	DT_PEDIDO	ID_CLIENTE	NOME_CLIENTE	FONE_CLIENTE
0001	11/07/2016	1001	ANTONIO ALVARES	5555-1111
0002	11/07/2016	1002	BEATRIZ BARBOSA	5555-2222
0003	12/07/2016	1003	ANTONIO ALVARES	5555-1111
0004	12/07/2016	1004	BEATRIZ BARBOSA	5555-2222

Podemos observar que há dados redundantes – nomes e fones dos clientes são repetidos a cada pedido novo. Isto gera, a princípio, dois problemas: desperdício de espaço para armazenamento dos dados e dificuldade para atualização dos dados. (Por exemplo, se o telefone da

BEATRIZ BARBOSA for alterado, será necessário alterá-lo provavelmente em muitas linhas.)

É exatamente por este e outros motivos (que serão considerados posteriormente) que um passo importante no projeto de um banco de dados é o que chamamos de "normalização de tabelas". Sim, há normas já estabelecidas que determinam se as tabelas de um banco de dados estão normalizadas. Mas normalização de tabelas está além do escopo deste livro. Nosso objetivo foi apenas o de demonstrar porque em um banco de dados bem projetado os dados que pertencem a "entidades" diferentes ficam armazenados em tabelas distintas.

1.2 SQL – STRUCTURED QUERY LANGUAGE

A SQL (Structure Query Language) apresenta os subgrupos descritos a seguir:

- DDL (Data Definition Language);
- DML (Data Manipulation Language);
- DTL (Data Transact Language);
- DCL (Data Control Language).

Alguns autores apresentam um quinto subgrupo, denominado DQL (Data Query Language), que inclui o comando SELECT. Preferimos, assim como muitos outros autores, incluir o comando SELECT no subgrupo DML (Data Manipulation Language). Porém, antes de abordarmos em detalhes cada um destes subgrupos, apresentaremos brevemente a seguir alguns dos principais sistemas de gerenciamento de banco de dados relacionais.

1.3 ALGUNS DOS PRINCIPAIS SISTEMAS DE BANCOS DE DADOS RELACIONAIS

Os primeiros Sistemas de Gerenciamento de Bancos de Dados Relacionais (SGBDR's) surgiram à partir do início da década de 1970.

O site DB-Engines (<http://db-engines.com/en/ranking>) publica mensalmente o ranking dos SGBDs, conforme a popularidade. A pesquisa realizada em setembro de 2016 (que inclui alguns outros

bancos além dos relacionais) apresentava o seguinte resultado (apenas os 10 primeiros):

1. Oracle;
2. MySQL;
3. Microsoft SQL Server;
4. PostgreSQL;
5. MongoDB;
6. IBM DB2;
7. Cassandra;
8. Microsoft Access;
9. SQLite;
10. Redis.

A escolha dos sistemas de bancos de dados apresentados a seguir levou em conta outros quesitos além da popularidade. Por exemplo: o MongoDB aparece em quinto lugar, mas não vamos abordá-lo, pois não se trata de um banco estritamente relacional. Por outro lado, incluímos o SQLite, que aparece em nono lugar, por sua importância principalmente em aplicações móveis devido ao seu tamanho. De qualquer modo, não seria possível falar sobre todos os sistemas gerenciadores de bancos de dados da atualidade. A lista do site DB-Engines apresentava "apenas" 315 sistemas (em setembro de 2016).

1.3.1 ORACLE DATABASE

Fundada em agosto de 1977 por Larry Ellison, Bob Miner, Ed Oates com a colaboração de Bruce Scott, "Oracle" foi inicialmente o nome do projeto para um de seus clientes: a CIA (Central Intelligence Agency). A empresa que desenvolveu o Oracle chamava-se "Systems Development Labs" (SDL). Em 1978 a SDL passou a ser denominada Relational Software Inc (RSI) para comercializar o seu novo banco de dados.

O primeiro SGBDR comercial foi desenvolvido usando PDP-11 – linguagem assembler. Embora a primeira versão do banco de dados já estivesse disponível em 1977, não foi colocada à venda até 1979 com

o lançamento do Oracle versão 2. A Força Aérea dos EUA e a CIA foram os primeiros clientes a usar o Oracle 2.

Em 1982, o nome da empresa foi alterado de RSI para Oracle Systems Corporation, de modo a coincidir com o nome do banco de dados.

Até o momento em que este livro foi escrito, a versão mais atual do Oracle Database é a 12c, lançada em 2013. Esta versão apresenta as seguintes novidades:

- New Multitenant and Pluggable database concept
- Adaptive Query Optimization
- Online Stats Gathering
- Temporary UNDO
- In Database Archiving
- Invisible Columns
- PGA Aggregate Limit Setting
- DDL Logging
- Flash ASM

Todos estes são recursos avançados e este livro não tem o objetivo de explaná-los. Porém, caso tenha interesse, poderá conhecê-los melhor acessando a documentação online disponível no site da Oracle: <http://docs.oracle.com/database/121/index.htm>.

1.3.2 MICROSOFT SQL SERVER

O SQL Server foi desenvolvido pela empresa Sybase na década de 1980 para sistemas UNIX [SILBERSCHATZ]. No início da década de 1990, a Microsoft se juntou à Sybase para criar uma versão deste sistema de gerenciamento de banco de dados que fosse compatível com o seu sistema operacional da época: o Windows NT. As versões lançadas pela Microsoft a partir de 1994 tornaram-se independentes da Sybase. Visto que a Microsoft havia negociado direitos exclusivos para todas as versões SQL Server escritas para o seu sistema operacional, em 1996, a Sybase mudou o nome de seu produto para Adaptive Server Enterprise, visando evitar conflitos com a Microsoft.

As principais "editions" do Microsoft SQL Server 2016 são as seguintes:

- Enterprise: atende a aplicações de missão crítica, que necessitem de alta disponibilidade, armazenamento de dados em larga escala e alta performance. Não apresenta limites quanto ao número de núcleos para processamento, memória limitada apenas pelo sistema operacional e 524 PB de dados.
- Standard: oferece as principais funcionalidades de gerenciamento de dados e business intelligence com mínimos recursos de TI. Suporta processamento até 24 núcleos (licenciamento por núcleo, em pacotes de 2 núcleos), 128 GB de memória máxima alocada por instância e 524 PB de dados.
- Express: ideal para a implantação de bancos de dados pequenos em ambientes de produção. Embora oferecido gratuitamente, o produto apresenta as seguintes limitações: processadores de até 4 núcleos, 1 GB de memória máxima alocada por instância e 10 GB de dados.
- Developer: oferece gratuitamente o conjunto completo de recursos da versão Enterprise que permite aos desenvolvedores criar, testar e demonstrar aplicações em um ambiente de não produção.

A administração do SQL Server pode ser realizada através de ferramentas com interface gráfica, como o Management Studio. Esta ferramenta fornece um ambiente adequado para administrar todos os serviços relacionados ao SQL Server: Database Engine, Analysis Services, Reporting Services, Integration Services etc.

1.3.3 IBM DB2

A origem do DB2 remonta ao projeto denominado System R, desenvolvido no San Jose Research Laboratory, da IBM. Porém, o produto com o nome DB2 foi lançado em 1984 para plataforma de mainframe da IBM. A seguir, a empresa passou a disponibilizar o seu sistema de gerenciamento de banco de dados para outras plataformas que incluem o Linux e o Windows.

As principais "editions" do IBM DB2 10.5 são as seguintes:

- DB2 Express-C: não apresenta suporte técnico da IBM. Suporta processamento até 2 núcleos, 16 GB de memória e 15 TB de dados (por database).
- DB2 Express: suporta processamento até 8 núcleos, 64 GB de memória e 15 TB de dados (por database).
- DB2 Workgroup: suporta processamento até 16 núcleos, 128 GB de memória e 15 TB de dados (por database).
- DB2 Advanced Workgroup: suporta processamento até 16 núcleos, 128 GB de memória e 15 TB de dados (por database).
- DB2 Enterprise: suporta processamento sem limite (teórico) de núcleos, sem limite (teórico) de memória e sem limite (teórico) de dados.
- DB2 Advanced Enterprise: suporta processamento sem limite (teórico) de núcleos, sem limite (teórico) de memória e sem limite (teórico) de dados.

O IBM Data Studio, disponível gratuitamente, oferece um ambiente integrado para administração e desenvolvimento de banco de dados do DB2 para os sistemas operacionais UNIX, Linux e Windows e ferramentas de desenvolvimento colaborativas para o z/OS.

1.3.4 MySQL

O MySQL foi desenvolvido por uma empresa da Suécia, a MySQL AB, fundada por Michael Widenius, David Axmark e Allan Larsson. O desenvolvimento original do MySQL por Widenius e Axmark começou em 1994. A primeira versão do MySQL apareceu em 23 de maio de 1995. Inicialmente foi criada para uso pessoal a partir do mSQL, considerado muito lento e inflexível. O MySQL apresentava, portanto, uma nova interface SQL, mantendo a mesma API do mSQL.

A versão do MySQL para Windows (95 e NT) foi lançada em 8 de janeiro de 1998. Em 2008, a Sun Microsystems adquiriu a MySQL AB. A Oracle adquiriu a Sun Microsystems em 27 de janeiro de 2010. Portanto,

o MySQL atualmente faz parte da linha de produtos oferecidos ao mercado pela Oracle Corporation.

Durante os últimos anos o MySQL foi o SGBD disponibilizado com mais frequência pelos provedores de hospedagem de sites da internet.

Em janeiro de 2009, antes da aquisição do MySQL por parte da Oracle, Michael Widenius iniciou um "garfo" do MySQL – o MariaDB – com licenciamento conforme a GPL. O banco de dados MariaDB baseia-se na mesma base de código do MySQL 5.5 e pretende manter a compatibilidade com as versões fornecidas pela Oracle.

1.3.5 POSTGRESQL

Na década de 1980, a Defense Advanced Research Projects Agency (DARPA), o Army Research Office (ARO), a National Science Foundation (NSF) e a ESL, Inc. patrocinaram o projeto POSTGRES, liderado pelo Professor Michael Stonebraker, da Universidade da Califórnia, em Berkeley.

A primeira versão do sistema se tornou operacional em 1987 e foi exibida no ano seguinte na Conferência ACM-SIGMOD. A empresa Illustra Information Technologies, posteriormente incorporada pela Informix, que agora pertence à IBM, passou a comercializar o produto nos anos seguintes.

Em 1994, Andrew Yu e Jolly Chen adicionaram um interpretador da linguagem SQL ao Postgres95. O código do Postgres95 foi totalmente escrito na linguagem C e o seu tamanho foi reduzido em 25%.

Em setembro de 2016, o Grupo de Desenvolvimento Global do PostgreSQL anunciou o lançamento do PostgreSQL 9.3, a última versão do sistema de banco de dados de código aberto. Esta versão expande a confiabilidade, disponibilidade e habilidade em integrar com outros bancos de dados.

1.3.6 SQLITE

O SQLite, projeto de domínio público criado por Richard Hipp, é um sistema de gerenciamento de dados que consiste em uma pequena biblioteca (menor que 300 KB) escrita na linguagem C. Atualmente, na sua versão 3, o SQLite permite a gestão de base de dados de até 2TB.

Diferente dos outros sistemas apresentados, que trabalham em uma arquitetura básica de "cliente-servidor", o "engine" do SQLite não é um

processo independente com o qual o aplicativo se comunica. O programa utiliza as funcionalidades do SQLite através de chamadas simples a sub-rotina e funções. Todo o conjunto: definições, tabelas, índices e os dados são guardados em apenas um arquivo na máquina "cliente".

Nesta obra, é importante que você saiba, utilizaremos o Oracle Database para apresentação dos exemplos e para os exercícios práticos. A escolha é facilmente justificável: além de aparecer em primeiro lugar na lista que leva em consideração a popularidade dos SGBDs, é o produto mais utilizado por empresas de grande porte em todo o mundo. Esta explicação é importante, pois embora a linguagem SQL seja padronizada, conforme mencionamos pela ANSI e pela ISO, há pequenas diferenças entre os principais sistemas de bancos de dados. Observe, portanto, quais são as "editions" do Oracle Database.

1.4 EDIÇÕES DO ORACLE DATABASE

O Oracle Database está disponível em cinco edições, cada uma adequada para diferentes cenários de desenvolvimento e implantação. A Oracle também oferece várias opções de banco de dados, pacotes e outros produtos que aprimoram os recursos do Oracle Database para propósitos específicos.

1.4.1 ORACLE DATABASE STANDARD EDITION ONE

O Oracle Database Standard Edition One oferece facilidade de uso para os aplicativos de grupo de trabalho, departamento e Web applications. A partir de ambientes de servidor único para pequenas empresas e para ambientes de filiais altamente distribuídas, o Oracle Database Standard Edition One inclui todas as facilidades necessárias para a criação de aplicativos críticos para o negócio.

1.4.2 ORACLE DATABASE STANDARD EDITION

O Oracle Database Standard Edition oferece as mesmas facilidades de uso do Standard Edition One, com suporte para máquinas maiores e clustering de serviços com Oracle Real Application Clusters (Oracle RAC). (O Oracle RAC não está incluído na Edição Standard de versões anteriores ao Oracle Database 10g, nem é uma opção disponível com essas versões anteriores.)

1.4.3 ORACLE DATABASE ENTERPRISE EDITION

O Oracle Database Enterprise Edition fornece desempenho, disponibilidade, escalabilidade e segurança necessárias para aplicativos de missão crítica, como aplicativos de processamento de transações on-line (OLTP) de alto volume, armazéns de dados com demanda intensiva e aplicativos de Internet exigentes. O Oracle Database Enterprise Edition contém todos os componentes do Oracle Database e pode ser aprimorado com a compra de pacotes adicionais.

1.4.4 ORACLE DATABASE EXPRESS EDITION

O Oracle Database Express Edition (Oracle Database XE) é uma edição básica do Oracle Database que é rápida de baixar, simples de instalar e gerenciar, e é livre para desenvolver, implantar e distribuir. O Oracle Database XE facilita a atualização para as outras edições do Oracle sem migrações dispendiosas e complexas. O Oracle Database XE armazena até 4GB de dados, usando até 1GB de memória e usando apenas uma CPU na máquina host. O suporte para esta edição é fornecido por um fórum on-line.

1.4.5 ORACLE DATABASE PERSONAL EDITION

O Oracle Database Personal Edition suporta ambientes de desenvolvimento e implantação de usuário único que requerem compatibilidade total com o Oracle Database Standard Edition One, o Oracle Database Standard Edition e o Oracle Database Enterprise Edition. Inclui todos os componentes do Enterprise Edition, bem como todas as opções que estão disponíveis com o Enterprise Edition, com exceção da opção Oracle Real Application Clusters, que não pode ser utilizado com o Personal Edition. O Personal Edition está disponível apenas em plataformas Windows. Pacotes de Gerenciamento também não estão incluídos nesta edição.

1.5 INTERFACES PARA USUÁRIOS DO ORACLE DATABASE

1.5.1 SQL PLUS

A ferramenta SQL*Plus provê um ambiente tipo linha de comando (texto) para que possamos enviar comando ao banco de dados. Através

do SQL*Plus podemos executar, desde que tenhamos os privilégios necessários, comandos de todos os subgrupos da linguagem SQL.

Há também alguns comandos específicos do ambiente SQL*Plus que detalharemos a seguir que facilitam o trabalho dos usuários do Oracle Database.

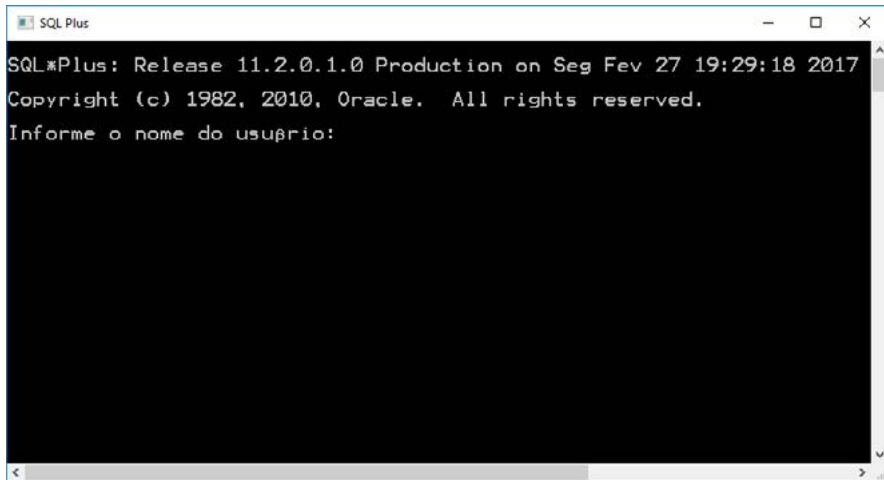


Figura 1: Interface do SQL*Plus
Fonte: Oracle Database – SQL Plus.

Você deverá informar o nome do usuário e a respectiva senha ao entrar no SQL*Plus. A Oracle provê um usuário com privilégios básicos denominado scott. Antes de terminar a instalação do Oracle Database sugerimos que desbloqueie a conta scott e escolha uma senha. Importante: a conta scott deve ser mantida bloqueada em bancos de produção por razão de segurança.

1.5.1.1 DESCRIBE

O comando DESCRIBE exibe a estrutura da tabela, isto é, informa quais são os nomes das colunas, o tipo de dado e o tamanho de cada coluna e também se a coluna pode ou não armazenar valores nulos.

A utilização do comando DESCRIBE é muito simples, conforme você pode observar a seguir:

DESCRIBE NOME_DA_TABELA;

Exemplo:

DESCRIBE CLIENTE;

O SQL*Plus aceita também a forma abreviada do comando DESCRIBE:

DESC CLIENTE;

1.5.1.2 COMANDOS DE EDIÇÃO

O SQL*Plus armazena a instrução anterior em um buffer. Isso torna possível editar as linhas que compõem a instrução SQL armazenadas no buffer. A tabela a seguir apresenta os principais comandos de edição e sua respectiva descrição.

Comando	Forma abreviada	Descrição
APPEND	A	Anexa um texto no final da linha.
CHANGE /antigo / novo	C	Altera o texto de antigo para novo na linha atual.
CLEAR BUFFER	CL BUFF	Apaga todas as linhas do buffer.
DEL		Exclui a linha atual.
DEL n		Exclui a linha especificada em n.
LIST	L	Lista todas as linhas presentes no buffer.
LIST n	L n	Lista a linha especificada em n.
RUN	R ou /	Executa a instrução armazenada no buffer.
n		Torna corrente a linha especificada em n.

Fonte: Autor.

1.5.1.3 SALVAR, RECUPERAR E EXECUTAR ARQUIVOS

O SQL*Plus possibilita salvar, recuperar e executar arquivos (scripts) que contenham instruções SQL. A tabela a seguir apresenta alguns dos comandos utilizados para estes propósitos.

Comando	Forma abreviada	Descrição
SAVE nome_arquivo [APPEND REPLACE]	SAV	Salva o conteúdo do buffer do SQL*Plus em um arquivo. APPEND anexa o conteúdo do buffer a um arquivo existente. REPLACE sobrescreve um arquivo existente.
GET nome_arquivo		Recupera o conteúdo de um arquivo para o buffer do SQL*Plus.
START nome_arquivo	STA	Recupera o conteúdo de um arquivo para o buffer do SQL*Plus e tenta executar o conteúdo do buffer.
EDIT	ED	Copia o conteúdo do buffer do SQL*Plus em um arquivo denominado afiedit.buf e inicia o editor de texto padrão do sistema operacional. O conteúdo do arquivo editado é copiado no buffer do SQL*Plus quando o usuário sai do editor.
EDIT nome_arquivo	ED nome_arquivo	Esta opção permite que o usuário especifique um nome de arquivo para editar.

SPOOL nome_arquivo	SPO_nome_arquivo	Copia a saída do SQL*Plus para o arquivo.
SPOOL OFF	SPO OFF	Interrompe a cópia da saída do SQL*Plus no arquivo e fecha o arquivo.

Fonte: Autor.

1.5.1.4 FORMATAR COLUNAS

O SQL *Plus disponibiliza o comando COLUMN para formatar a exibição de cabeçalhos e dados de colunas. Há várias opções para este comando, conforme você pode observar na tabela a seguir.

Opção	Forma abreviada	Descrição
FORMAT formato	FOR	Define o formato de exibição da coluna ou do apelido de coluna conforme a string <i>formato</i> .
HEADING cabeçalho	HEA	Define o cabeçalho da coluna ou do apelido de coluna conforme a string <i>cabeçalho</i> .
JUSTIFY LEFT CENTER RIGHT	JUS	Coloca a saída da coluna na esquerda, no centro ou na direita.
WRAPPED	WRA	Coloca o final de uma string na próxima linha de saída. Palavras individuais podem ser divididas em várias linhas.
WORD_WRAPPPED	WOR	Semelhante à WRAPPED, porém palavras individuais não são divididas em várias linhas.
CLEAR	CLE	Limpa a formatação das colunas (configura a formatação de volta ao padrão).

Fonte: Autor.

1.5.1.5 VARIÁVEIS

O SQL*Plus permite criar variáveis de substituição que podem ser utilizadas no lugar de valores reais em instruções SQL. Há dois tipos de variáveis de substituição:

- a. Variável temporária: válida apenas para a instrução SQL em que é utilizada (não persiste).
- b. Variável definida: persiste até que seja redefinida, removida explicitamente ou ao encerrar o SQL*Plus.

Variáveis Temporárias

Determinamos as variáveis temporárias utilizando o caractere '&' (E comercial) seguido do nome da variável. Por exemplo: &V_CODIGO_CLIENTE determina uma variável denominada V_CODIGO_CLIENTE.

Variáveis Definidas

Variáveis definidas podem ser utilizadas várias vezes dentro de uma instrução SQL. Definimos uma variável deste tipo utilizando o comando DEFINE. Quando não necessitamos mais utilizá-la podemos removê-la usando o comando UNDEFINE. O exemplo a seguir define uma variável denominada V_CODIGO_CLIENTE e atribui a ela o valor 1001:

```
DEFINE V_CODIGO_CLIENTE = 1001
```

É possível inclusive conhecer todas as variáveis de uma determinada sessão digitando apenas DEFINE no prompt do SQL*Plus.

ACCEPT

O comando ACCEPT pode ser utilizado para configurar uma variável existente, atribuindo-lhe um novo valor, ou para definir uma nova variável e inicializá-la com um valor. O comando permite especificar, além do nome da variável, o tipo de dado (CHAR, NUMBER ou DATE), o formato (veja a seguir) e um prompt (texto exibido pelo SQL*Plus para o usuário digitar o valor da variável). Observe o exemplo a seguir:

```
ACCEPT V_CODIGO_CLIENTE NUMBER FORMAT 9999 PROMPT 'Código: '
Código: 1001
```

Alguns exemplos de formatos para as variáveis são:

- 9999 – Número com quatro dígitos
- DD-MOM-YYYY – Data no formato 15-JAN-2016
- A20 – 20 caracteres

1.5.1.6 AJUDA DO SQL*PLUS

Você poderá obter ajuda do SQL*Plus utilizando o comando HELP. A lista de tópicos pode ser obtida através de HELP INDEX, conforme segue:

@	DISCONNECT	RESERVED WORDS (SQL)
@@	EDIT	RESERVED WORDS (PL/SQL)
/	EXECUTE	RUN
ACCEPT	EXIT	SAVE
APPEND	GET	SET
ARCHIVE LOG	HELP	SHOW
ATTRIBUTE	HOST	SHUTDOWN
BREAK	INPUT	SPOOL
BTITLE	LIST	SQLPLUS
CHANGE	PASSWORD	START
CLEAR	PAUSE	STARTUP
COLUMN	PRINT	STORE
COMPUTE	PROMPT	TIMING
CONNECT	QUIT	TTITLE
COPY	RECOVER	UNDEFINE
DEFINE	REMARK	VARIABLE
DEL	REPFOOTER	WHENEVER OSERROR
DESCRIBE	REPHEADER	WHENEVER SQLERROR

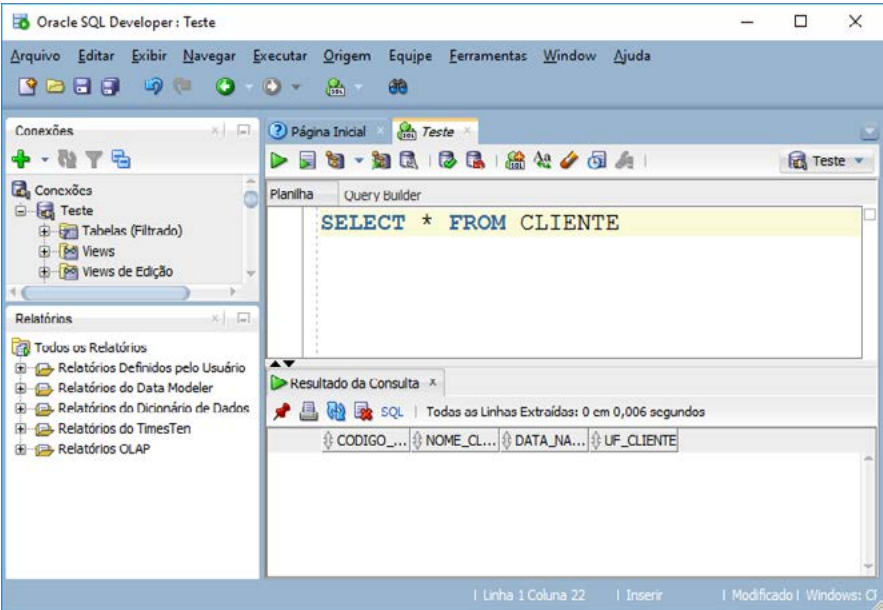


Figura 2 – Interface SQL Developer
Fonte: Oracle Database – SQL Plus.

Uma nova janela é aberta conforme a figura a seguir:

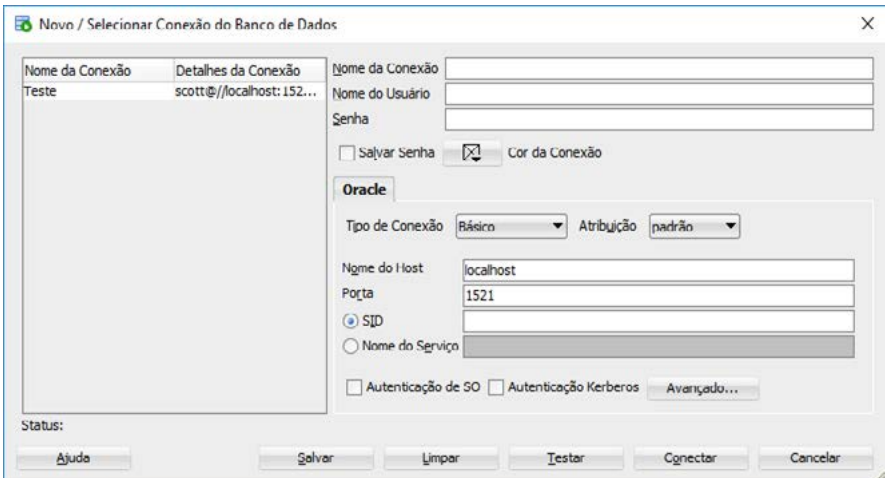


Figura 3 – SQL Developer – conexão com o Banco de Dados
Fonte: Oracle Database – SQL Plus.

Você deverá escolher um nome para a conexão e, em seguida, informar o nome do usuário do banco (scott, por exemplo) e sua respectiva senha. Abaixo deverá informar também o Nome do Host (localhost, a menos que você tenha informado outro no momento da instalação), a Porta (1521, é o padrão) e o SID (orcl é o padrão para maioria das edições do Oracle Database ou XE para Express Editon). Verifique antes se a conexão foi bem sucedida clicando no botão Testar e, em seguida, clique no botão Conectar.

O SQL Developer voltará à tela principal. Escolha o ícone verde identificado com o rótulo SQL e digite os comandos SQL que desejar. A saída dos comandos aparecerá na área logo abaixo, conforme você pode observar na Figura 2.

1.5.3 ORACLE APPLICATION EXPRESS

O Oracle Application Express, disponível gratuitamente em <https://apex.oracle.com/en/>, permite que você crie, desenvolva e implemente aplicativos baseados em banco de dados, usando apenas o navegador da Web. A plataforma contém recursos como visualizar as tabelas criadas, os comandos executados recentemente, importar e executar scripts etc. É uma excelente opção caso você não deseje instalar o Oracle Database em seu equipamento ou se desejar executar os exemplos apresentados neste livro em locais que disponibilizem computadores sem instalação do banco de dados da Oracle. Observe, a seguir, como é simples utilizar esta valiosa ferramenta web para melhorar seus conhecimentos da linguagem SQL.

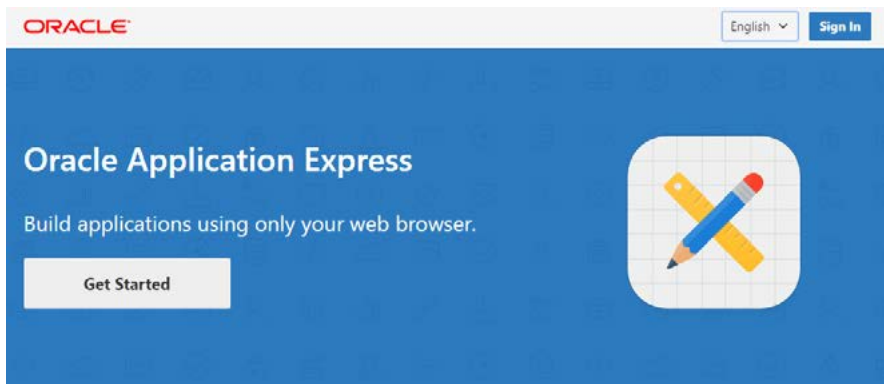


Figura 4 – Oracle Apex

Fonte: Disponível em: <https://apex.oracle.com/en/>

Criar uma conta no Oracle Application Express é muito simples. O primeiro passo é clicar no botão **Get Started**. A seguir é a apresentada a seguinte tela:

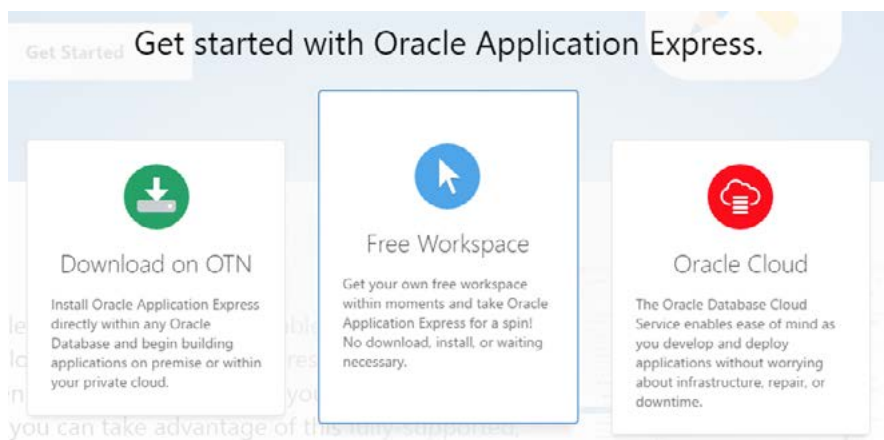


Figura 5 – Opções disponíveis no Oracle Application Express

Fonte: Disponível em: <https://apex.oracle.com/en/>

Selecione a segunda opção: Free Workspace.

A próxima tela perguntará "Que tipo de Workspace você deseja solicitar?" e apresentará as seguintes opções:

- Application Development
- Packaged Apps Only

Você deverá selecionar a opção **Application Development**, conforme a figura a seguir, e clicar no botão Next.

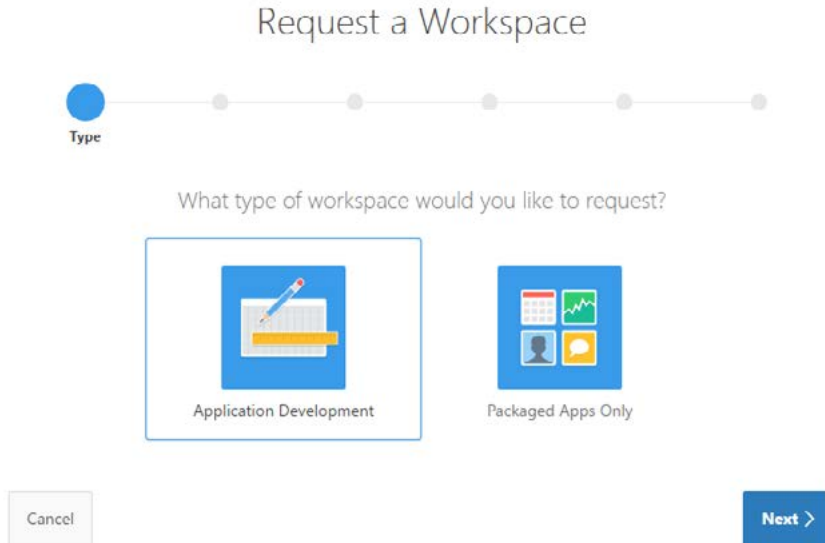


Figura 6 – Oracle Application Express – Tipos de Workspace
 Fonte: Disponível em: <https://apex.oracle.com/en/>

Preencha os dados a seguir e clique em **Next**.

Figura 7 – Oracle Application Express – Identificação do usuário
 Fonte: Disponível em: <https://apex.oracle.com/en/>

O nome do Workspace a ser escolhido na tela anterior deverá ser exclusivo, isto é, não poderá ser idêntico ao escolhido anteriormente por outro usuário. O e-mail deverá ser válido, pois a Oracle enviará para o e-mail informado uma mensagem de confirmação com um link para ativação do Workspace.

Figura 8 – Oracle Application Express – Schema e Espaço de Alocação de Dados
Fonte: Disponível em: <https://apex.oracle.com/en/>

O schema (esquema) é o usuário que possuirá os objetos de banco de dados. Quando o serviço for provisionado, você receberá o nome do seu esquema e sua senha de login. Uma vez que seu espaço de trabalho for criado, você poderá adicionar outros esquemas a ele. Os nomes de esquema válidos devem estar em conformidade com as seguintes diretrizes:

- Deve ter de 1 a 30 bytes de comprimento;
- Não pode conter aspas;
- Deve começar com um caractere alfabético;
- Só pode conter caracteres alfanuméricos, com exceção de `_`, `$` e `#`;
- Não pode ser uma palavra reservada Oracle.

Você poderá escolher entre dois espaços de alocação para seus dados: 10 MB ou 25 MB. Visto tratar-se de um serviço gratuito para as duas opções, sugerimos que escolha 25 MB.

A próxima tela apresentará três perguntas que você deverá obrigatoriamente responder para seguir para o próximo passo.

- Can Oracle contact you about your use of Oracle Application Express? (A Oracle pode entrar em contato com você sobre o uso do Oracle Application Express?)
- Are you new to Oracle Application Express? (Você é novo no Oracle Application Express?)
- Would your organization be willing to be a reference for Oracle Application Express? (Sua organização estaria disposta a ser uma referência para o Oracle Application Express?)

A seguir será apresentado o Oracle Application Express Agreement (Contrato de Serviço do Oracle Application Express). Leia todo o contrato de serviço e se estiver de acordo você deverá aceitar os termos (selecione I accept the terms) para prosseguir para próxima etapa.

Destacamos a seguir dois detalhes do Contrato de Serviço do Oracle Application Express que merecem especial atenção:

- O Serviço Oracle Application Express pode ser usado apenas para fins não relacionados à produção.
- Você reconhece que a Oracle não tem nenhuma obrigação de entrega e não enviará cópias de qualquer programa Oracle para você como parte dos serviços.

Clicando em Next você chegará a tela final que apresentará as seguintes informações, conforme informações fornecidas nas etapas anteriores:

- Workspace Name
- Description

- First Name
- Last Name
- Administrator Email
- Schema Name
- Database Size (MB)

Confira os dados apresentados e se tudo estiver correto clique em Submit Request.

Request a Workspace

Please verify your workspace request.

Workspace Name

Description

First Name

Last Name

Administrator Email

Schema Name

Database Size (MB)

< Cancel Submit Request

Figura 9 – Oracle Application Express – Verificação dos dados

Fonte: Disponível em: <https://apex.oracle.com/en/>

A tela final informa que o processo foi bem-sucedido e que será enviado um e-mail para o endereço fornecido por você com as informações para que você faça o login no Oracle Application Express.

Observe a seguir os passos para utilização deste ambiente para prática da linguagem SQL. Lembramos que o Oracle Application Express deve ser utilizado apenas para testes, jamais como ambiente de produção.

Após responder à mensagem enviada pela Oracle e confirmando a criação de seu Workspace você deverá fazer o login, conforme os dados fornecidos nas etapas anteriores.

A tela a seguir apresenta o botão Sign In no canto superior direito. Clique neste botão em seu browser para fazer o login.

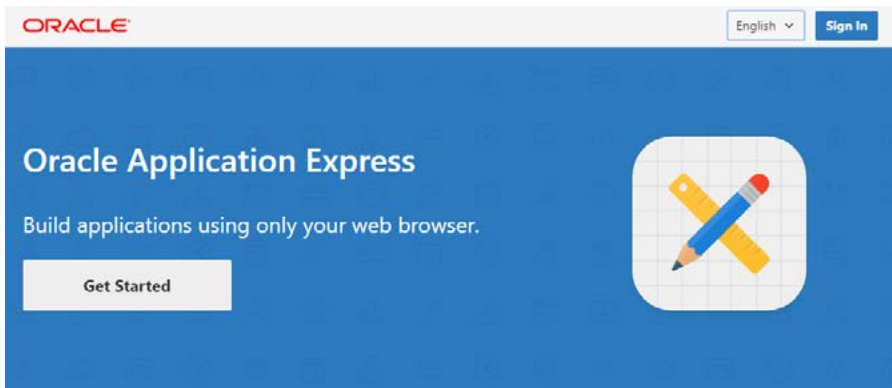
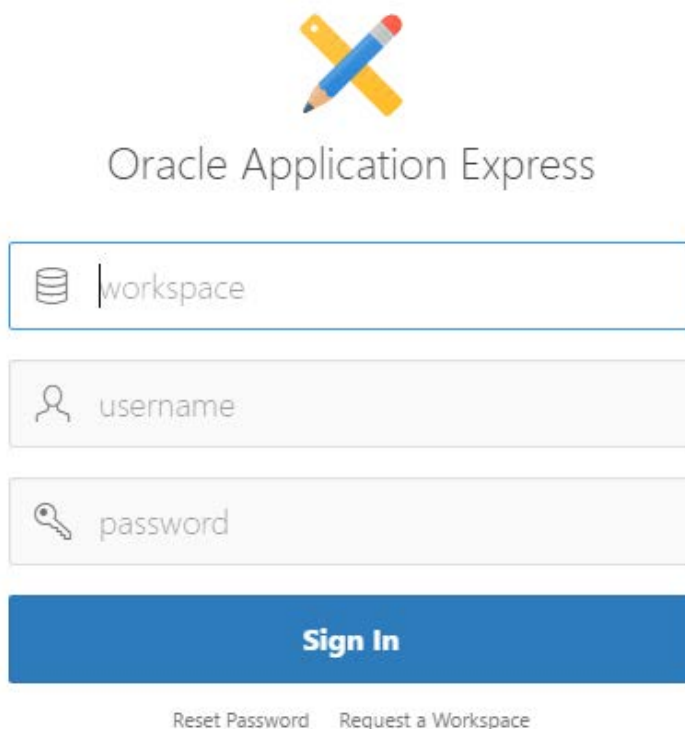


Figura 10 – Oracle Application Express – Sign In
Fonte: Disponível em: <https://apex.oracle.com/en/>

Informe na tela de Sign In os seguintes dados, conforme definidos nas etapas anteriores:

- Workspace
- Username
- Password



The image shows the Oracle Application Express login interface. At the top is the Oracle APEX logo, which consists of two crossed pencils, one yellow and one blue. Below the logo is the text "Oracle Application Express". There are three input fields: the first is for the workspace name, with a database icon and the placeholder text "workspace"; the second is for the username, with a person icon and the placeholder text "username"; the third is for the password, with a key icon and the placeholder text "password". Below these fields is a large blue "Sign In" button. At the bottom, there are two links: "Reset Password" and "Request a Workspace".

Figura 11- Oracle Application Express – Dados para Sign In

Fonte: Disponível em: <https://apex.oracle.com/en/>

A próxima tela apresentará quatro opções. Selecione a segunda: SQL Workshop, conforme a próxima figura.

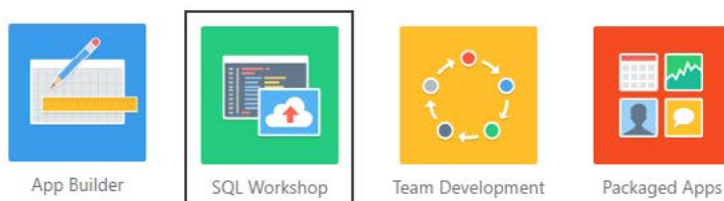


Figura 12 – Oracle Application Express – SQL Workshop

Fonte: Disponível em: <https://apex.oracle.com/en/>

O Oracle Application Express apresentará a seguir cinco opções. Selecione SQL Commands, conforme apresentado na próxima figura.

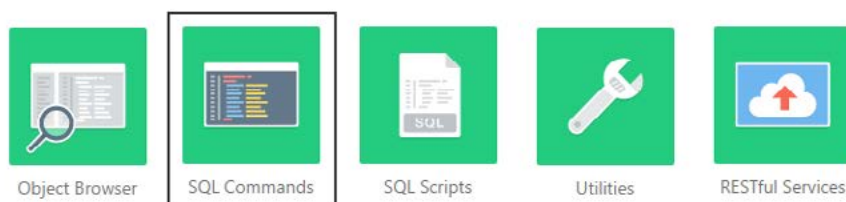


Figura 13 – Oracle Application Express – SQL Commands

Fonte: Disponível em: <https://apex.oracle.com/en/>

A figura a seguir apresenta uma visão parcial da tela SQL Commands.

Rows: 10 | Clear Command | Find Tables | Save | Run

SELECT * FROM CLIENTE;

CODIGO_CLIENTE	NOME_CLIENTE	DATA_NASC_CLIENTE	UF_CLIENTE
1001	ANTONIO ALVARES	03/28/1986	RS
1002	BEATRIZ BARBOSA	06/15/1991	SC

Figura 14 – Oracle Application Express – Consulta ao Banco de Dados.

Fonte: Disponível em: <https://apex.oracle.com/en/>

A interface apresenta duas áreas principais. A área superior é destinada à digitação dos comandos SQL e a área inferior à apresentação dos resultados. Após digitar o comando SQL, você deverá pressionar o botão Run, localizado no canto superior direito para que o comando seja executado.

Neste capítulo apresentamos três opções para que você coloque em prática tudo o que abordaremos até o final deste livro:

- SQL*Plus;
- SQL Developer;
- Oracle Application Express.

Escolha a que achar mais apropriada para o seu caso e execute cada um dos exemplos apresentados. Faça também os exercícios disponibilizados no final de cada capítulo. Para tornar-se "fluyente" na linguagem SQL, assim como em qualquer outra linguagem, o segredo é um só: PRÁTICA.

RESUMO

A estrutura básica dos bancos de dados relacionais são as tabelas ("relações"). Tabelas são estruturas bidimensionais formadas por colunas e por linhas (similar às planilhas do Microsoft Excel).

Outra característica importante dos bancos de dados relacionais é a capacidade de relacionar dados entre duas ou mais tabelas, isto é, criar "relacionamentos" entre as tabelas. Esta implementação ocorre através de campos ou colunas com valores comuns.

Os primeiros Sistemas de Gerenciamento de Bancos de Dados Relacionais surgiram a partir do início da década de 1970.

Em junho de 1970, o matemático e pesquisador da IBM, Edgar Frank Codd, publicou no periódico Communications of the ACM, o artigo "A Relational Model of Data for Large Shared Data Banks".

Em meados da década de 1970, Donald D. Chamberlin e Raymond F. Boyce, pesquisadores no IBM San Jose Research Laboratory, apresentaram uma linguagem baseada na álgebra relacional (usada como fundamento no artigo de Codd) denominada SEQUEL (Structured English Query Language). Algum tempo depois o nome da linguagem foi alterado simplesmente para SQL (Structured Query Language).

A SQL (Structure Query Language) apresenta os seguintes subgrupos:

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DTL (Data Transact Language)
- DCL (Data Control Language)

Os subgrupos que formam a linguagem SQL serão considerados em detalhes nos próximos capítulos.

EXERCÍCIOS

Fundada em agosto de 1977 por Larry Ellison, Bob Miner, Ed Oates, com a colaboração de Bruce Scott. No início, chamava-se "Systems Development Labs" (SDL). Em 1978, passou a ser denominada Relational Software Inc. (RSI) para comercializar o seu novo banco de dados. Esta descrição corresponde a qual das seguintes desenvolvedoras de sistemas de bancos de dados?

- a) IBM
- b) Microsoft
- c) MySQL
- d) Oracle
- e) PostgreSQL

É uma edição básica do Oracle Database, rápida de baixar, simples de instalar e gerenciar, e é livre para desenvolver, implantar e distribuir. Facilita a atualização para as outras edições do Oracle sem migrações dispendiosas e complexas. Armazena até 4GB de dados, usando até 1GB de memória e usando apenas uma CPU na máquina host. O suporte para esta edição é fornecido por um fórum on-line. A descrição corresponde a qual das seguintes edições do Oracle Database?

- a) Personal Edition
- b) Express Edition

- c) Standard Edition
- d) Standard Edition One
- e) Enterprise Edition

CAPÍTULO 2 – TIPOS DE DADOS

Um banco de dados relacional armazena uma coleção de dados organizados que se relacionam. Portanto, um cuidado especial necessário é a escolha do tipo correto de dados de acordo com o que será armazenado em cada coluna das tabelas do banco.

Vamos analisar brevemente alguns exemplos:

Número de vezes que a seleção brasileira de futebol se sagrou campeã mundial. A resposta corresponde a um número inteiro. Não teria sentido inserir em uma coluna de tabela, criada para esta finalidade, valores como 5.2 ou abc.

Data de nascimento de uma pessoa. A informação deve corresponder a dia, mês e ano (conforme costumamos utilizar no Brasil) ou mês, dia e ano (conforme costumam utilizar os americanos e os ingleses). Não teria sentido armazenar em uma coluna, criada para este propósito, valores como 99 ou xyz.

Portanto, para organizar melhor o banco e também para evitar que valores indesejáveis, como os citados acima, sejam inseridos em determinada coluna, devemos informar no momento em que estamos criando uma tabela os nomes das colunas e os seus respectivos tipos de dados.

Quando declaramos, no momento de criação da tabela, os tipos dos dados, precisamos na maioria dos casos declarar, a seguir, entre parênteses, o tamanho (fixo ou variável, dependendo do caso) dos dados que serão armazenados.

Vejam, a seguir, quais são os principais tipos de dados utilizados pelo Oracle Database.

2.1 NUMBER (p,s)

Utilizado para armazenar valores numéricos com precisão p e escala s. A precisão (p) é o número total de dígitos e a escala (s) é o número total de dígitos à direita da vírgula (ou ponto, dependendo do sistema utilizado) decimal. A precisão pode variar de 1 até 38 e a escala pode variar de -84 a 127.

Exemplos:

NUMBER (4): Armazena números inteiros com até 4 dígitos. Ex: 5792

NUMBER (5,2): Armazena números com até 5 dígitos com 2 casa decimais. Ex: 416.25

2.2 VARCHAR2 (TAMANHO)

Utilizado para armazenar caracteres alfanuméricos com tamanho variável. O tamanho mínimo é 1 (caractere) e o máximo 4000 (caracteres).

Exemplo:

VARCHAR2 (20): Armazena até 20 caracteres alfanuméricos.

2.3 CHAR

Utilizado para armazenar caracteres alfanuméricos com tamanho fixo. O tamanho default e mínimo é 1 (caractere) e o máximo 2000 (caracteres).

Exemplo:

CHAR (4): Armazena 4 caracteres alfanuméricos.

Devo utilizar VARCHAR2 ou CHAR?

Depende. Por exemplo, se você precisar armazenar nomes de clientes em determinada coluna, a opção a ser escolhida deverá ser VARCHAR2, pois nomes de pessoas ou de empresas têm tamanhos variáveis. O tipo CHAR sempre ocupará o espaço total informado, portanto se você optasse por este tipo haveria (além de outros inconvenientes) desperdício de espaço de armazenamento. No entanto, se você precisar armazenar a sigla de um estado brasileiro ou a placa de um veículo, deverá optar pelo tipo CHAR, pois ambos conterão dados de tamanho fixo: 2 caracteres alfabéticos para estados brasileiros e 7 caracteres alfanuméricos para placas de veículos.

2.4 DATE

Utilizado para armazenar valores referentes a datas e horários. É possível armazenar datas entre 1º de janeiro de 4712 a.C. até 31 de dezembro de 9999 d.C.

Estes são os tipos de dados mais comuns encontrados na maioria dos bancos de dados de uso comercial. Observe, a seguir, alguns outros tipos de dados disponíveis no Oracle e em outros SGBDRs.

2.4.1 TIMESTAMP

Este tipo de dado é uma extensão do tipo DATE, que permite armazenar (além de dia, mês e ano) hora, minuto, segundo e frações de segundos.

2.5 LONG

Armazena caracteres com extensão variável até 2 GB.

Restrições quanto ao uso do tipo LONG:

- É possível utilizar apenas uma coluna definida como LONG por tabela.
- Não é possível utilizar restrições em uma coluna do tipo LONG.
- Uma coluna definida como LONG não é copiada quando uma tabela é criada usando uma subconsulta. (Um capítulo posterior abordará a criação de tabelas utilizando consultas.)
- Uma coluna definida como LONG não pode ser incluída em uma cláusula GROUP BY ou ORDER BY. (Este assunto também será abordado em um capítulo posterior.)

2.6 CLOB

Armazena caracteres com extensão variável até 4 GB.

2.7 RAW

Armazena dados binários brutos com tamanho máximo igual a 2000. É necessário informar o tamanho.

2.8 LONG RAW

Armazena dados binários brutos de tamanho variável até 2 GB.

2.9 BLOB

Armazena dados binários até 4 GB.

2.10 BFILE

Armazena em um arquivo externo dados binários até 4 GB.

2.11 ROWID

Utiliza um sistema numérico de base 64 para representar o endereço exclusivo de uma linha da tabela.

RESUMO

É necessário um cuidado especial para escolher o tipo correto de dados de acordo com o que será armazenado em cada coluna das tabelas do banco.

Os principais tipos de dados que podem ser definidos no Oracle são os seguintes:

- NUMBER: Armazena números inteiros e com ponto flutuante com precisão que pode variar de 1 a 38 e escala de -84 a 127.
- VARCHAR2: Armazena dados de tamanho variável de 1 a 4000 caracteres.
- CHAR: Armazena dados de tamanho fixo de 1 (default) a 2000 caracteres.
- DATE: Armazena datas e horários de 01/01/4712 a.C a 31/12/9999 d.C.
- LONG: Armazena dados de caracteres de extensão variável até 2 GB.
- CLOB: Armazena dados de caracteres até 4 GB.
- RAW: Armazena dados binários brutos com tamanho máximo igual a 2000.
- LONG RAW: Armazena dados binários de tamanho variável até 2 GB.

- **BLOB:** Armazena dados binários até 4 GB.
- **BFILE:** Armazena dados binários armazenados em arquivos externos até 4 GB.
- **ROWID:** Armazena dados em base 64 representando o endereço exclusivo de uma linha da tabela.

EXERCÍCIOS

O departamento comercial de uma empresa solicitou a elaboração de uma tabela denominada PRODUTO para armazenar toda a linha de produtos que serão comercializados. Uma das colunas deverá armazenar os valores dos produtos. É sabido que os valores dos produtos se situam na faixa de 10.99 a 9800.00. Qual dos seguintes tipos de dados e seu respectivo tamanho deverá ser utilizado?

- a) VARCHAR2 (4)
- b) INT (4)
- c) INT (6)
- d) NUMBER (4,2)
- e) NUMBER (6,2)

A tabela PRODUTO, descrita no exercício anterior, também precisa armazenar a descrição dos produtos comercializados. Sabe-se que a quantidade de caracteres dos dados que serão armazenados nesta coluna é variável. Qual das seguintes opções deverá ser utilizada para armazenar este tipo de dado?

- a) CHAR
- b) VARCHAR2
- c) RAW
- d) LONG RAW
- e) BLOB

CAPÍTULO 3 – DDL – DATA DEFINITION LANGUAGE

A documentação oficial da Oracle disponível na internet lista 16 comandos para o subgrupo da SQL denominado DDL – Data Definition Language ou (em português) Linguagem de Definição de Dados. Neste capítulo abordaremos os relacionados com as seguintes operações realizadas em bancos de dados relacionais:

- Criação de tabelas;
- Alteração (na estrutura) de tabelas;
- Renomeação de tabelas;
- Exclusão de tabelas.

3.1 CREATE TABLE

A criação de uma nova tabela em um banco de dados é realizada através do comando CREATE TABLE.

É necessário informar o seguinte ao criar uma tabela:

- Nome da tabela;
- Tipo de dado e seu respectivo tamanho para cada coluna da tabela.

Cuidados ao escolher os nomes das tabelas, nomes de colunas e nomes de outros objetos do banco de dados:

- O primeiro caractere deve ser alfabético (A-Z ou a-z);
- Após o primeiro caractere pode-se utilizar números;
- Não são permitidos caracteres especiais;
- Apenas os seguintes caracteres não alfanuméricos são permitidos: #, \$ e _.

- Não são permitidos nomes que coincidam com as palavras reservadas do Oracle Database. (Consulte a lista de palavras reservadas no Apêndice.)

Veja alguns exemplos para melhor esclarecimento:

- PEDIDO (nome válido)
- PEDIDOCIENTE (nome válido)
- PEDIDO CLIENTE (nome não válido, pois contém espaço)
- PEDIDO*CLIENTE (nome não válido, pois contém caractere considerado especial)
- PEDIDO_CLIENTE (nome válido, pois o _ não é considerado caractere especial)

Além disso, conforme será apresentado nos próximos itens, devemos incluir determinadas "constraints" (ou "restrições") para o bom funcionamento do banco.

Vamos à prática. Imagine que você precisasse armazenar informações de seus clientes em um banco de dados relacional como o Oracle Database. Informações como o código, nome, data de nascimento e estado (UF) onde o cliente reside.

Vamos definir o nome da tabela para armazenar os dados acima como: CLIENTE.

A seguir vamos determinar os nomes das colunas da tabela e seus respectivos tipos e tamanhos de dados:

- CODIGO_CLIENTE NUMBER (4)
- NOME_CLIENTE VARCHAR2 (50)
- DATA_NASC_CLIENTE DATE
- UF_CLIENTE CHAR (2)

Quando criarmos a tabela, o tamanho de cada coluna deverá ser informado sempre entre parênteses, conforme mostrado acima.

Observe também que não é necessário informar o tamanho para a coluna que armazenará dados do tipo data.

A partir das informações acima podemos criar nossa tabela CLIENTE. Observe:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) ,
  NOME_CLIENTE VARCHAR2 (50) ,
  DATA_NASC_CLIENTE DATE ,
  UF_CLIENTE CHAR (2)
) ;
```

Após o nome da tabela e antes do nome da primeira coluna devemos abrir parênteses que deverão ser fechados no final. Utilizamos vírgulas para separar as colunas da tabela. Não utilizamos vírgula após a última coluna, salvo se for necessário definir constraints, conforme veremos a seguir.

Você poderá praticar os comandos acima e todos os outros apresentados neste e nos próximos capítulos utilizando as ferramentas SQL*Plus ou SQL Developer do Oracle.

Vejamos a seguir como o uso de "restrições" ou "constraints" são importantes para garantir a integridade dos dados armazenados no banco.

3.1.1 CONSTRAINTS

3.1.1.1 PRIMARY KEY

Utilizamos constraints do tipo PRIMARY KEY (CHAVE PRIMÁRIA) para identificar com exclusividade cada linha da tabela. Por que isso é importante? Imagine se a Receita Federal cadastrasse dois contribuintes com o mesmo CPF ou se uma empresa atribuísse o mesmo código para clientes diferentes. Que confusão! Quantos problemas!

Dados referentes aos códigos inseridos na tabela CLIENTE, criada anteriormente, poderiam apresentar este problema, visto que não há nenhuma constraint associada à coluna CODIGO para garantir que apenas valores exclusivos sejam ali inseridos.

Vamos recriar a tabela CLIENTE, desta vez com uma constraint denominada CLIENTE_PK do tipo PRIMARY KEY. Apresentaremos também duas formas para declarar a PRIMARY KEY: INLINE (na mesma linha da declaração da coluna) e OUT OF LINE (após a declaração de todas as linhas da tabela).

IMPORTANTE: Se você estiver praticando os comandos apresentados, será necessário eliminar a tabela CLIENTE criada anteriormente. Não é possível ter duas tabelas com o mesmo nome no mesmo schema de um banco de dados.

Para eliminar a tabela CLIENTE você deverá utilizar o comando a seguir:

```
DROP TABLE CLIENTE CASCADE CONSTRAINTS ;
```

O comando DROP TABLE será apresentado em detalhes logo mais neste capítulo.

IN LINE (Declarando o nome da constraint):

```
CREATE TABLE CLIENTE (  
  CODIGO_CLIENTE NUMBER (4) CONSTRAINT CLIENTE_PK PRIMARY  
  KEY,  
  NOME_CLIENTE VARCHAR2 (50) ,  
  DATA_NASC_CLIENTE DATE,  
  UF_CLIENTE CHAR (2)  
) ;
```

IN LINE (Sem declarar o nome da constraint):

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) PRIMARY KEY,
  NOME_CLIENTE VARCHAR2 (50),
  DATA_NASC_CLIENTE DATE,
  UF_CLIENTE CHAR (2)
);
```

OUT OF LINE:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4),
  NOME_CLIENTE VARCHAR2 (50),
  DATA_NASC_CLIENTE DATE,
  UF_CLIENTE CHAR (2),
  CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE)
);
```

Neste último modo, primeiro inserimos uma vírgula após o tamanho referente à última coluna (UF), a seguir utilizamos a palavra reservada CONSTRAINT. O nome da constraint aparece logo a seguir (CLIENTE_PK) juntamente com o tipo (PRIMARY KEY). E, finalmente, informamos o nome da coluna que recebe a restrição (CODIGO_CLIENTE).

No exemplo apresentado, a chave primária é simples, isto é, formada por apenas uma coluna. Porém, há casos em que a chave primária é composta, formada por mais de uma coluna. Quando isto ocorre, os nomes das colunas que compõem a chave primária devem ser separados por vírgula.

3.1.1.2 FOREIGN KEY

Um banco de dados é formado por um conjunto de tabelas que se relacionam. O relacionamento em bancos de dados relacionais é implementado através de colunas que contém valores comuns que podem estar em uma mesma tabela (autorrelacionamento) ou (o que é mais comum) em tabelas diferentes.

Porém, é necessário garantir a integridade relacional destes relacionamentos de dados. Como isto deve ser feito? Através de outra constraint denominada FOREIGN KEY (CHAVE ESTRANGEIRA).

Talvez você não conheça ainda o termo "integridade relacional". Pois saiba que ele exerce um papel fundamental na consistência dos dados do banco. Observe os exemplos a seguir:

Imagine que, além da tabela CLIENTE, nosso banco também tivesse uma tabela PEDIDO com a seguinte estrutura:

- NR_PEDIDO
- DATA_PEDIDO
- VALOR_PEDIDO
- CODIGO_CLIENTE

Observamos que a tabela acima tem uma coluna com dados que devem ser comuns à tabela CLIENTE. Estes dados estão contidos nas colunas que receberam o mesmo nome nas duas tabelas: CODIGO_CLIENTE.

Como garantir, no entanto, que apenas clientes que foram inseridos previamente na tabela CLIENTE possam realizar pedidos ou como garantir que clientes que realizaram pedidos não sejam excluídos da tabela CLIENTE?

É exatamente para resolver isso que utilizamos a FOREIGN KEY ou CHAVE ESTRANGEIRA.

Para melhor entendimento criaremos a seguir a tabela PEDIDO com suas duas constraints (PRIMARY KEY e FOREIGN KEY):

```
CREATE TABLE PEDIDO (
NR_PEDIDO NUMBER (5) ,
DATA_PEDIDO DATE,
VALOR_PEDIDO NUMBER (6,2) ,
CODIGO_CLIENTE NUMBER (4) ,
CONSTRAINT PEDIDO_PK PRIMARY KEY (NR_PEDIDO) ,
CONSTRAINT PEDIDO_CLIENTE_FK FOREIGN KEY (CODIGO_CLIENTE)
REFERENCES CLIENTE (CODIGO_CLIENTE)
);
```

Observe que a criação da tabela PEDIDO seguiu a mesma sequência da tabela CLIENTE, criada anteriormente. A diferença ficou por conta da inclusão da FOREIGN KEY.

Para criarmos uma FOREIGN KEY devemos utilizar também a palavra reservada CONSTRAINT. A seguir devemos informar o nome da chave (PEDIDO_CLIENTE_FK no exemplo acima). O tipo de chave – FOREIGN KEY – e o nome da coluna – CODIGO_CLIENTE – aparecem na sequência. A seguir, devemos utilizar a palavra reservada REFERENCE que indicará o nome da tabela para a qual estamos criando o relacionamento (tabela CLIENTE no caso acima). E entre parênteses informamos em qual coluna da tabela CLIENTE ocorre o relacionamento (coluna CODIGO_CLIENTE no exemplo apresentado).

Conforme mencionamos, a FOREIGN KEY garante que os clientes que realizaram pedidos não sejam excluídos da tabela CLIENTE. Caso isso ocorresse nosso banco ficaria com dados inconsistentes. No entanto, há situações em que desejamos, por exemplo, eliminar os dados de certos clientes (contidos na tabela CLIENTE) e como consequência eliminar também os dados dos pedidos que correspondem a estes clientes. Esta opção pode ser considerada ao criar a FOREIGN KEY. Neste caso, devemos acrescentar a seguinte opção à FOREIGN KEY: ON DELETE CASCADE. Observe como esta opção pode ser incluída ao criar a tabela PEDIDO:

```
CREATE TABLE PEDIDO (
  NR_PEDIDO NUMBER (5) ,
  DATA_PEDIDO DATE ,
  VALOR_PEDIDO NUMBER (6,2) ,
  CODIGO_CLIENTE NUMBER (4) ,
  CONSTRAINT PEDIDO_PK PRIMARY KEY (NR_PEDIDO) ,
  CONSTRAINT PEDIDO_CLIENTE_FK FOREIGN KEY (CODIGO_CLIENTE)
  REFERENCES CLIENTE (CODIGO_CLIENTE) ON DELETE CASCADE
) ;
```

Em outras situações, precisamos que ao excluir determinado cliente da tabela CLIENTE os valores correspondentes à FOREIGN KEY da tabela PEDIDO sejam alterados para NULL (ausência de dados). Neste caso, devemos acrescentar a seguinte opção à FOREIGN KEY: ON DELETE SET

NULL. Para que esta opção funcione é obvio que a coluna FOREIGN KEY da tabela pedido não tenha uma restrição do tipo NOT NULL. (A restrição NOT NULL será abordada logo adiante neste capítulo.) Observe agora como esta opção pode ser incluída ao criar a tabela PEDIDO.

```
CREATE TABLE PEDIDO (
NR_PEDIDO NUMBER (5) ,
DATA_PEDIDO DATE ,
VALOR_PEDIDO NUMBER (6,2) ,
CODIGO_CLIENTE NUMBER (4) ,
CONSTRAINT PEDIDO_PK PRIMARY KEY (NR_PEDIDO) ,
CONSTRAINT PEDIDO_CLIENTE_FK FOREIGN KEY (CODIGO_CLIENTE)
REFERENCES CLIENTE (CODIGO_CLIENTE) ON DELETE SET NULL
) ;
```

3.1.1.3 CHECK

Digamos que uma determinada empresa franqueada realize vendas apenas para clientes da região sul de nosso país, ou seja, para clientes que residam nos estados RS (Rio Grande do Sul), SC (Santa Catarina) e PR (Paraná). Como verificar e impedir que alguém insira na coluna UF da tabela CLIENTE criada anteriormente, por exemplo, 'SP' (São Paulo)?

É para restrições deste tipo que existe outra constraint denominada CHECK. Esta constraint verifica se o valor que está sendo inserido em determinada coluna do banco corresponde a um conjunto de valores previamente determinado.

Os valores a serem verificados podem fazer parte de um conjunto de valores específicos como os citados acima ou de faixas (ranges) de valores que podem ser determinados com a utilização de vários operadores diferentes (=, <, <=, >, >=, etc.).

LEMBRETE: Não esqueça de eliminar a tabela CLIENTE anterior (utilizando o comando DROP TABLE) caso queira praticar o exemplo a seguir utilizando o SQL*Plus ou o SQL Developer.

IN LINE:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) ,
  NOME_CLIENTE VARCHAR2 (50) ,
  DATA_NASC_CLIENTE DATE ,
  UF_CLIENTE CHAR (2) CHECK (UF_CLIENTE IN ('RS', 'SC', 'PR')) ,
  CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE)
);
```

OUT OF LINE:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) ,
  NOME_CLIENTE VARCHAR2 (50) ,
  DATA_NASC_CLIENTE DATE ,
  UF_CLIENTE CHAR (2) ,
  CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE) ,
  CONSTRAINT UF_CK CHECK (UF_CLIENTE IN ('RS', 'SC', 'PR'))
);
```

3.1.1.4 UNIQUE

Apenas uma PRIMARY KEY pode estar associada a cada tabela. O que devemos fazer caso queiramos garantir valores exclusivos em outras colunas que não compõem a PRIMARY KEY? Neste caso utilizamos a restrição UNIQUE.

O exemplo a seguir demonstra como utilizar esta restrição para garantir que não sejam inseridos na coluna NOME_CLIENTE dois nomes idênticos:

IN LINE:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) ,
  NOME_CLIENTE VARCHAR2 (50) UNIQUE,
```



```
DATA_NASC_CLIENTE DATE,
CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE)
);
```

OUT OF LINE:

```
CREATE TABLE CLIENTE (
CODIGO_CLIENTE NUMBER (4),
NOME_CLIENTE VARCHAR2 (50),
DATA_NASC_CLIENTE DATE,
CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE),
CONSTRAINT CODIGO_UN UNIQUE (NOME_CLIENTE)
);
```

Visto que o Oracle indexa automaticamente as colunas com restrições do tipo PRIMARY KEY e UNIQUE, consultas realizadas com base em valores desta coluna serão mais rápidas do que as realizadas com base em valores de outras colunas. No capítulo que abordará o item ÍNDICES falaremos sobre isso com mais detalhes.

3.1.1.5 NOT NULL

Há situações em que precisamos garantir que valores NULOS não sejam inseridos em determinadas colunas do banco. Para este objetivo devemos utilizar a restrição NOT NULL.

Queremos garantir, por exemplo, que não sejam inseridos valores NULOS na coluna NOME_CLIENTE de nossa tabela CLIENTE.

Observe a seguir como podemos garantir isso utilizando a restrição NOT NULL:

```
CREATE TABLE CLIENTE (
CODIGO_CLIENTE NUMBER (4),
NOME_CLIENTE VARCHAR2 (50) NOT NULL,
DATA_NASC_CLIENTE DATE,
UF_CLIENTE CHAR (2) CHECK (UF_CLIENTE IN ('RS', 'SC', 'PR')),
CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE)
);
```

NOTA: A constraint NOT NULL somente pode ser declarada no modo IN LINE, conforme demonstrado. A tentativa de declará-la no modo OUT OF LINE apresentará um erro.

A restrição NOT NULL não garante que um nome válido de cliente seja inserido na coluna acima. Ela apenas garante que algo diferente de NULL (ausência de valor) seja inserido na coluna NOME_CLIENTE.

3.1.1.6 DEFAULT

A cláusula DEFAULT não é uma constraint, pois não impõe nenhuma restrição à tabela. Porém, pode ser incluída no momento de criação da tabela para que um valor "default" (padrão) seja inserido na coluna caso este não seja informado no momento em que uma nova linha for inserida na tabela.

No exemplo a seguir, caso seja omitida a data do pedido durante a inserção de uma nova linha na tabela, será inserida a SYSDATE (data do sistema) na coluna DATA_PEDIDO.

```
CREATE TABLE PEDIDO (
NR_PEDIDO NUMBER (5) ,
DATA_PEDIDO DATE DEFAULT SYSDATE,
VALOR_PEDIDO NUMBER (6,2) ,
CODIGO_CLIENTE NUMBER (4) ,
CONSTRAINT PEDIDO_PK PRIMARY KEY (NR_PEDIDO) ,
CONSTRAINT PEDIDO_CLIENTE_FK FOREIGN KEY (CODIGO_CLIENTE)
REFERENCES CLIENTE (CODIGO_CLIENTE) ON DELETE SET NULL
);
```

3.1.2 CRIAR UMA TABELA COM BASE EM OUTRA

É possível também criar uma nova tabela com base na estrutura ou nos dados de outra anteriormente criada.

Observe o exemplo a seguir:

```
CREATE TABLE CLIENTE_2 AS SELECT * FROM CLIENTE;
```

Através do comando acima criamos uma nova tabela denominada CLIENTE_2 com a mesma estrutura da tabela CLIENTE. Se a tabela CLIENTE contiver dados, estes também serão "copiados" para a tabela CLIENTE_2.

3.2 ALTER TABLE

Muitas vezes é necessário realizar alterações em tabelas já criadas. Utilizando o comando ALTER TABLE podemos realizar alterações tais como:

- Adicionar uma ou mais colunas na tabela;
- Modificar o tamanho de uma coluna;
- Renomear uma coluna;
- Eliminar uma coluna;
- Adicionar constraints;
- Eliminar constraints;
- Desabilitar constraints;
- Habilitar constraints.

Vejamos a seguir na prática como podemos realizar cada uma das alterações apresentadas acima.

3.2.1 ADICIONAR UMA COLUNA

Tomaremos novamente como base a tabela CLIENTE anteriormente criada. O que devemos fazer para acrescentar uma coluna para armazenar os e-mails dos clientes?

Devemos definir antes: o nome da coluna, o tipo de dados que armazenará e, se necessário, o seu respectivo tamanho.

- EMAIL_CLIENTE VARCHAR2 (50)

Uma vez que definimos o nome da coluna como EMAIL_CLIENTE, o tipo de dado como VARCHAR2 e o tamanho limitado a 50 caracteres, podemos alterar nossa tabela CLIENTE conforme segue:

```
ALTER TABLE CLIENTE
ADD EMAIL_CLIENTE VARCHAR2 (50) ;
```

ADD significa que queremos adicionar (acrescentar) a coluna EMAIL_CLIENTE;

Você poderá comprovar a alteração em sua tabela CLIENTE utilizando o seguinte comando no SQL*PLUS:

```
DESCRIBE CLIENTE ;
```

Poderá, opcionalmente, utilizar a forma abreviada:

```
DESC CLIENTE ;
```

3.2.2 MODIFICAR UMA COLUNA

Outra situação com a qual um profissional da área de banco e dados pode se deparar é a necessidade de alterar o tamanho de uma coluna.

A tabela CLIENTE, por exemplo, apresenta a coluna NOME_CLIENTE com tipo VARCHAR2 e tamanho limitado a 50 caracteres. O que fazer, porém, se descobrirmos que alguns clientes têm nomes com mais de 50 caracteres?

Neste caso podemos aumentar o tamanho da coluna NOME_CLIENTE, conforme apresentado a seguir:

```
ALTER TABLE CLIENTE
MODIFY NOME_CLIENTE VARCHAR2 (80) ;
```

MODIFY indica que estamos modificando o tamanho da coluna NOME_CLIENTE.

Pode-se também diminuir o tamanho de uma coluna, mas esta opção deve ser realizada com muita cautela para que não ocorra perda de parte dos dados.

3.2.3 RENOMEAR UMA COLUNA

Há situações não muito frequentes em que é necessário alterar o nome de uma coluna. Por exemplo, quando criamos uma tabela e descobrimos que erramos ao digitar o nome de uma coluna.

Digamos que por algum motivo precisássemos renomear a coluna UF_CLIENTE para ESTADO_CLIENTE. Que comando deveríamos utilizar para este propósito? Observe:

```
ALTER TABLE CLIENTE  
RENAME COLUMN UF_CLIENTE TO ESTADO_CLIENTE;
```

RENAME indica que desejamos renomear a coluna da tabela CLIENTE de UF_CLIENTE para ESTADO_CLIENTE.

Repare que antes de informarmos o nome da coluna que deverá ser renomeada utilizamos a palavra que corresponde à COLUNA, em inglês: COLUMN. Entre o antigo e o novo nome da coluna utilizamos a preposição que corresponde à PARA, em inglês: TO.

Mais uma vez, se desejar comprovar a alteração em sua tabela CLIENTE, poderá utilizar o seguinte comando no SQL*PLUS:

```
DESCRIBE CLIENTE;
```

3.2.4 ELIMINAR UMA COLUNA

É possível que em algumas outras situações cheguemos à conclusão de que determinados dados não fazem mais sentido serem mantidos em um banco.

Por exemplo, o fax foi durante anos um meio muito comum de comunicação, principalmente entre empresas. Atualmente, pouquíssimas empresas ou pessoas ainda o utilizam. É possível que alguém solicite a um profissional de banco de dados a eliminação de uma coluna da tabela destinada a armazenar estes dados.

No exemplo que utilizamos até o momento, a tabela CLIENTE, vamos eliminar a coluna que corresponde à data de nascimento dos clientes:

```
DATA_NASC_CLIENTE:
ALTER TABLE CLIENTE
DROP COLUMN DATA_NASC_CLIENTE;
```

DROP indica que desejamos eliminar da tabela CLIENTE a coluna denominada DATA_NASC_CLIENTE.

Além de alterações em tabelas o comando ALTER é utilizado para realizar outras alterações nos objetos de um banco de dados relacional. Apresentaremos a seguir como ALTER pode ser utilizado para alterar as constraints de um banco.

Visto que os comandos que apresentaremos são muito similares aos que você acabou de ver, seremos mais objetivos e apresentaremos apenas descrição de cada comando e um exemplo de como deve ser aplicado.

3.2.5 ADICIONAR UMA CONSTRAINT (CHAVE PRIMÁRIA)

Digamos que ao criar a tabela CLIENTE não tivéssemos incluído a constraint PRIMARY KEY. Como incluí-la após a criação da tabela? Observe:

```
ALTER TABLE CLIENTE
ADD CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE);
```

É importante que você saiba que podemos ter apenas uma chave primária em cada tabela que pode ser, conforme mencionamos, simples (formada por apenas uma coluna) ou composta (formada por mais de uma coluna).

3.2.6 ADICIONAR UMA CONSTRAINT (CHAVE ESTRANGEIRA)

Imagine agora que ao criar a tabela PEDIDO tivéssemos esquecido de incluir a constraint FOREIGN KEY para relacioná-la com a tabela CLIENTE. Como podemos incluí-la após a criação da tabela? Veja a seguir:

```
ALTER TABLE PEDIDO
```

```
ADD CONSTRAINT PEDIDO_CLIENTE_FK FOREIGN KEY (CODIGO_CLIENTE)
REFERENCES CLIENTE (CODIGO_CLIENTE) ;
```

3.2.7 ADICIONAR UMA CONSTRAINT (NOT NULL)

A constraint NOT NULL não pode ser declarada no modo OUT OF LINE. Por isso, não podemos adicionar uma constraint deste tipo utilizando a instrução ADD (adicionar), pois equivaleria a uma tentativa de declará-la no modo OUT OF LINE. Devemos, portanto, utilizar a instrução MODIFY. Desta forma modificamos (alteramos) a coluna e incluindo a restrição do tipo NOT NULL.

```
ALTER TABLE CLIENTE
```

```
MODIFY NOME_CLIENTE CONSTRAINT NOME_CLIENTE_NN NOT NULL;
```

3.2.8 ELIMINAR UMA CONSTRAINT

Caso seja necessário eliminar uma constraint devemos utilizar o seguinte comando (aplicado à tabela PEDIDO para eliminar a constraint PEDIDO_CLIENTE_FK):

```
ALTER TABLE PEDIDO
```

```
DROP CONSTRAINT PEDIDO_CLIENTE_FK;
```

3.2.9 DESABILITAR UMA CONSTRAINT

Desabilitar uma constraint indica que ela não atuará sobre as operações que serão realizadas no banco de dados, porém ela continuará a existir e poderá ser reabilitada, conforme veremos no próximo item.

Observe o exemplo a seguir utilizado para desabilitar a constraint CLIENTE_PK da tabela CLIENTE:

```
ALTER TABLE CLIENTE DISABLE CONSTRAINT CLIENTE_PK;
```

Deve-se tomar muito cuidado ao inserir, alterar ou excluir dados enquanto uma constraint estiver desabilitada, pois qualquer uma destas operações poderá causar problemas de inconsistências no banco.

3.10 HABILITAR UMA CONSTRAINT

Habilitar (ou "reabilitar") uma constraint indica que ela atuará (novamente) sobre as operações que serão realizadas no banco de dados.

No exemplo a seguir habilitamos a constraint CLIENTE_PK da tabela CLIENTE:

```
ALTER TABLE CLIENTE ENABLE CONSTRAINT CLIENTE_PK;
```

Visto que um dos propósitos da constraint acima era evitar que valores duplicados fossem inseridos na coluna CODIGO_CLIENTE, o que ocorreria se, durante o período em que ela esteve desabilitada, valores duplicados fossem inseridos nesta coluna? Neste caso o banco enviaria um aviso de erro e a constraint não seria habilitada até que os valores da coluna CODIGO_CLIENTE se tornassem novamente exclusivos.

3.3 DROP TABLE

Algumas tabelas perdem com o tempo sua utilidade. Outras devem ser excluídas, pois já cumpriram sua função temporária. Portanto, quando necessitamos excluir uma tabela de um banco de dados utilizamos o comando DROP TABLE.

O exemplo a seguir demonstra como podemos eliminar a tabela PEDIDO criada no início deste capítulo:

```
DROP TABLE PEDIDO CASCADE CONSTRAINTS;
```

O comando acima é utilizado para eliminar não somente a tabela, mas também as constraints a ela associadas. Esta é a finalidade do complemento do comando: CASCADE CONSTRAINTS.

3.4 TRUNCATE

O comando DDL TRUNCATE é utilizado para "cortar" uma tabela. O que isto significa? Significa que a estrutura da tabela permanecerá sem nenhuma alteração, porém os dados contidos na tabela serão definitivamente eliminados.

Para eliminar os dados da tabela CLIENTE, preservando a estrutura da mesma, utilizamos o seguinte comando:

```
TRUNCATE TABLE CLIENTE;
```

3.5 RENAME

Visto que um banco de dados apresenta uma estrutura sujeita a alterações com o passar do tempo, às vezes torna-se necessário renomear uma ou mais tabelas. Porém, conforme observamos, uma tabela pode estar relacionada a uma ou mais tabelas. Por este motivo, a operação de renomear tabelas também requer muito cuidado.

O comando a seguir demonstra como renomear a tabela CLIENTE para CLIENTES:

```
RENAME CLIENTE TO CLIENTES;
```

Conforme observado, após a palavra reservada RENAME informamos o nome atual da tabela e após a preposição TO (em inglês) informamos o novo nome da tabela.

3.6 SEQUENCE

A instrução CREATE SEQUENCE faz parte do subgrupo DDL e é utilizada para criar uma sequência, que é um objeto de banco de dados do qual os usuários podem gerar valores inteiros exclusivos. Você pode, por exemplo, usar sequências para gerar automaticamente valores de chave primária. Alguns conceitos preliminares deverão ser passados para melhor compreensão do uso de sequências. Por este motivo, escolhemos criar um capítulo mais adiante para abordar este item.

RESUMO

O subgrupo da SQL denominado DDL – Data Definition Language – é utilizado para operações em bancos de dados, tais como: criação de tabelas, alteração na estrutura de tabelas, renomeação de tabelas e exclusão de tabelas.

O comando CREATE TABLE é utilizado para criação de uma nova tabela em um banco de dados. É necessário informar o nome da tabela e tipo de dado e seu respectivo tamanho para cada coluna da tabela.

A constraint PRIMARY KEY (CHAVE PRIMÁRIA) é utilizada para identificar com exclusividade cada linha da tabela.

O relacionamento em bancos de dados relacionais é implementado através de colunas que contém valores comuns que podem estar em uma mesma tabela (autorrelacionamento) ou (o que é mais comum) em tabelas diferentes. A integridade relacional dos relacionamentos de dados é garantida através de outra constraint denominada FOREIGN KEY (CHAVE ESTRANGEIRA).

O comando ALTER TABLE é utilizado para realizar alterações nas estruturas das tabelas, tais como: adicionar uma ou mais colunas, modificar o tamanho de uma coluna, renomear uma coluna, adicionar ou eliminar uma coluna, habilitar ou desabilitar constraints.

Utilizamos o comando DROP TABLE quando necessitamos excluir uma tabela de um banco de dados.

O comando TRUNCATE é utilizado para "cortar" uma tabela. TRUNCATE mantém a estrutura da tabela sem nenhuma alteração, porém os dados contidos na tabela são definitivamente eliminados.

EXERCÍCIOS

A coluna ID_CLI de uma tabela denominada CLIENTE precisa ser renomeada para ID_CLIENTE. Qual dos seguintes comandos deverá ser utilizado para renomear a coluna?

- a) ALTER CLIENTE RENAME ID_CLI TO ID_CLIENTE;
- b) ALTER TABLE CLIENTE COLUMN ID_CLI TO ID_CLIENTE;
- c) ALTER TABLE CLIENTE RENAME COLUMN ID_CLI TO ID_CLIENTE;

- d) ALTER TABLE RENAME COLUMN ID_CLI TO ID_CLIENTE;
- e) ALTER TABLE CLIENTE RENAME ID_CLI TO ID_CLIENTE;

A tabela CLIENTE_SAO_PAULO deverá ser renomeada para CLIENTE_SP. Qual dos seguintes comandos DDL deverá ser utilizado para realizar isso?

- a) RENAME CLIENTE_SAO_PAULO TO CLIENTE_SP;
- b) ALTER TABLE CLIENTE_SAO_PAULO TO CLIENTE_SP;
- c) DROP TABLE CLIENTE_SAO_PAULO TO CLIENTE_SP;
- d) ALTER TABLE RENAME CLIENTE_SAO_PAULO TO CLIENTE_SP;
- e) RENAME TABLE CLIENTE_SAO_PAULO TO CLIENTE_SP;

CAPÍTULO 4 – DML – DATA MANIPULATION LANGUAGE

Os comandos do subgrupo denominado DML (Data Manipulation Language) ou Linguagem de Manipulação de Dados têm como objetivo recuperar dados (realizar consultas), inserir novas linhas, alterar linhas existentes e remover linhas do banco de dados.

Os comandos que fazem parte deste subgrupo são:

- SELECT
- INSERT
- UPDATE
- DELETE
- MERGE

4.1 SELECT

O comando SELECT é utilizado para realização de consultas em tabelas ou visões. Através das consultas podemos obter todos os dados de uma tabela ou apenas aqueles que desejamos utilizando, para isso, filtros. Veremos como aplicar filtros ou condições às consultas no próximo capítulo.

Quando desejamos obter dados contidos em uma tabela devemos informar os nomes das colunas onde estes dados estão localizados e o nome da tabela. Consultas mais complexas, denominadas subqueries, envolvem mais de uma tabela e serão abordadas mais à frente.

Voltemos à nossa tabela CLIENTE. Caso você não tenha criado esta tabela, apresentamos a seguir novamente o comando utilizado para criar a última versão, conforme o capítulo anterior:

```
CREATE TABLE CLIENTE (
  CODIGO_CLIENTE NUMBER (4) ,
  NOME_CLIENTE VARCHAR2 (50) UNIQUE,
  DATA_NASC_CLIENTE DATE,
```

```
UF_CLIENTE CHAR (2) CHECK (UF_CLIENTE IN ('RS','SC','PR')),
CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO_CLIENTE)
);
```

Observe que a tabela tem quatro colunas: CODIGO_CLIENTE, NOME_CLIENTE, DATA_NASC_CLIENTE e UF_CLIENTE.

Por enquanto, nossa tabela ainda não tem nenhuma linha, isto é, nenhum dado foi inserido. Nos próximos itens veremos como inserir, alterar e eliminar dados de nossas tabelas. Portanto, os comandos utilizados a seguir apresentarão como resposta que NENHUM DADO FOI ENCONTRADO.

Veja a seguir como realizar uma consulta para obter os nomes dos clientes utilizando para isso a tabela CLIENTE:

```
SELECT NOME_CLIENTE FROM CLIENTE;
```

Conforme mencionamos, quando desejamos obter dados contidos em uma tabela devemos informar os nomes das colunas onde estes dados estão localizados e o nome da tabela. No entanto, conforme você pode observar, isso deve ser feito utilizando uma sequência determinada pela linguagem SQL. Primeiro utilizamos a palavra reservada SELECT, depois informamos o nome da coluna (NOME_CLIENTE), a seguir utilizamos a preposição (em inglês) FROM e, finalmente, informamos o nome da tabela (CLIENTE).

Quando precisamos obter dados de mais de uma coluna, devemos simplesmente declarar os nomes das colunas separados por vírgula. Veja a seguir, por exemplo, como obter os nomes e as datas de nascimento a partir da tabela CLIENTE:

```
SELECT NOME_CLIENTE, DATA_NASC_CLIENTE FROM CLIENTE;
```

Caso seja necessário obter os dados contidos em todas as colunas de uma tabela, podemos simplificar nossa consulta utilizando no lugar dos nomes de todas as colunas o caractere * (asterisco). Veja a seguir como obter todos os dados contidos na tabela CLIENTE:

```
SELECT * FROM CLIENTE;
```

As consultas que acabamos de realizar não trouxeram como resultado nenhuma linha, pois ainda não foram inseridos dados na tabela CLIENTE. Vejamos no item a seguir como isso deve ser realizado utilizando a linguagem SQL.

4.2 INSERT

O comando DML utilizado para inserir ou incluir novos dados em uma tabela ou visão é o INSERT.

Para incluir uma nova linha em uma tabela, além da palavra reservada INSERT, utilizamos a preposição (em inglês) INTO, o nome da tabela, os nomes das colunas (que deverão ser declaradas entre parênteses e dependendo da situação pode ser opcional), a palavra VALUES e, finalizando, os valores que deverão ser incluídos (também entre parênteses).

Voltando à tabela CLIENTE, notamos que ao incluir uma nova linha devemos informar: CODIGO_CLIENTE, NOME_CLIENTE, DATA_NASC_CLIENTE e UF_CLIENTE.

Digamos que os seguintes dados deverão ser incluídos na tabela cliente:

CODIGO_CLIENTE: 1001

NOME_CLIENTE: ANTONIO ALVARES

DATA_NASC_CLIENTE: 28/03/1986

UF_CLIENTE: RS

O comando a ser utilizado para incluir estes dados na tabela CLIENTE deverá ser o seguinte:

```
INSERT INTO CLIENTE
(CODIGO_CLIENTE, NOME_CLIENTE, DATA_NASC_CLIENTE,
UF_CLIENTE)
VALUES
(1001, 'ANTONIO ALVARES', '28/03/1986', 'RS');
```

Conforme você pode observar, o valor 1001 aparece sem aspas simples. Isso ocorre porque refere-se a um valor numérico (os valores para esta coluna foram definidos como NUMBER). Por outro lado, todos os demais valores aparecem entre aspas simples porque não são valores numéricos. Quando definimos os valores como VARCHAR2, CHAR, DATE etc. devemos utilizar aspas simples, como observado acima.

Outro detalhe importante a ser observado é a sequência em que os valores foram apresentados: primeiro 1001, que corresponde ao CODIGO_CLIENTE, depois 'ANTONIO ALVARES', que corresponde ao NOME_CLIENTE e assim por diante.

Quando informamos os dados referentes a todas as colunas da tabela e na exata sequência como foram declarados no momento em que a tabela foi criada, podemos omitir os nomes das colunas ao inserir uma nova linha na tabela. Observe o próximo exemplo:

```
INSERT INTO CLIENTE
VALUES
(1002, 'BEATRIZ BARBOSA', '15/06/1991', 'SC');
```

Caso seja necessário omitir o valor correspondente a determinada coluna, desde que esta coluna não tenha uma restrição do tipo NOT NULL, podemos utilizar uma das seguintes formas:

1ª Quando declaramos os nomes das colunas no comando INSERT:

```
INSERT INTO CLIENTE
(CODIGO_CLIENTE, NOME_CLIENTE, UF_CLIENTE)
VALUES
(1003, 'CLAUDIO CARDOSO', 'PR');
```

2ª Quando omitimos os nomes das colunas no comando INSERT:

```
INSERT INTO CLIENTE
VALUES
(1003, 'CLAUDIO CARDOSO', '', 'PR');
```

Note que omitimos o valor correspondente à `DATA_NASC_CLIENTE`. Quando apresentamos a 2ª forma, utilizamos duas aspas simples no lugar do valor que está sendo omitido. As duas aspas deverão ser inseridas juntas, sem espaço (ou qualquer outro caractere) entre elas.

Antes de apresentar o próximo item que tratará da exclusão de linhas de tabelas, vamos considerar um tópico muito importante: integridade referencial. É verdade que já abordamos este assunto quando falamos da constraint `FOREIGN KEY`. No entanto, vejamos na prática como os bancos de dados mantêm a coerência entre os dados de tabelas que apresentam relacionamentos.

Observamos que a tabela `PEDIDO`, apresentada anteriormente, relaciona-se com a tabela `CLIENTE` através da coluna `CODIGO_CLIENTE`. Este relacionamento garante, por exemplo, que não vamos incluir nenhum pedido na tabela `PEDIDO` para um cliente que não exista na tabela `CLIENTE`.

Caso tomemos como base os exemplos apresentados neste capítulo, a tabela `CLIENTE` contém três linhas que correspondem aos clientes com códigos 1001, 1002 e 1003. Portanto, qualquer tentativa de incluir um código diferente destes no campo `CODIGO_CLIENTE` da tabela `PEDIDO` não será bem-sucedida. Nestes casos, o mecanismo do banco que verifica a integridade referencial checa se o valor a ser inserido na coluna que corresponde à chave estrangeira na "tabela filha" (tabela `PEDIDO`, conforme exemplo) existe na coluna que corresponde à chave primária na "tabela mãe" (tabela `CLIENTE`, conforme exemplo) e não permite a inserção caso o valor correspondente não seja encontrado.

4.3 DELETE

A exclusão de linhas de uma tabela é realizada através do comando `DELETE`. Podemos excluir uma ou mais linhas de uma tabela utilizando este comando.

Após a palavra reservada `DELETE`, utilizamos a preposição (em inglês) `FROM` que deverá ser seguida pelo nome da tabela. Quando desejamos excluir uma ou mais linhas, após a palavra `WHERE` (onde) devemos informar a condição para que a exclusão ocorra. Se a condição for verdadeira, a linha será excluída da tabela. O exemplo a seguir apresenta

a exclusão de uma linha (do cliente cujo CODIGO_CLIENTE é igual a 1003) da tabela CLIENTE:

```
DELETE FROM CLIENTE WHERE CODIGO_CLIENTE = 1003;
```

Neste caso, o mecanismo do banco que verifica a integridade referencial checa se o valor correspondente à chave primária na "tabela mãe" (tabela CLIENTE, conforme exemplo) existe na coluna que corresponde à chave estrangeira na "tabela filha" (tabela PEDIDO, conforme exemplo) e não permite a exclusão caso o valor correspondente seja encontrado.

Para excluir todas as linhas de uma tabela é preciso apenas utilizar o comando DELETE seguido da preposição FROM e do nome da tabela. O exemplo apresentado a seguir exclui, através de um único comando, todas as linhas da tabela CLIENTE.

```
DELETE FROM CLIENTE;
```

4.4 UPDATE

Um banco de dados apresenta uma coleção de dados relacionados que constantemente passam por atualizações. Portanto, quando precisamos atualizar os dados de uma tabela utilizamos o comando UPDATE.

Quando utilizamos o comando UPDATE devemos informar na sequência o nome da tabela que será seguido pela palavra SET e do nome da coluna onde ocorrerá a alteração ou atualização dos dados. A seguir será preciso informar qual será o novo valor e, muito importante, em qual linha a atualização deverá ser realizada. Utilizamos como referência para identificar a linha a ser atualizada a chave primária, pois assim sempre teremos certeza de atualizar a linha correta.

Observe a seguir como atualizar a data de nascimento (que hipoteticamente foi inserida por engano) do cliente cujo CODIGO_CLIENTE é igual a 1001.

```
UPDATE CLIENTE
SET DATA_NASC_CLIENTE = '19/07/1986'
WHERE CODIGO_CLIENTE = 1001;
```

Vale aqui um alerta. Caso a última linha – WHERE CODIGO_CLIENTE = 1001 – seja omitida, TODAS AS LINHAS da tabela CLIENTE serão atualizadas. Consequentemente, todos os clientes passarão a ter como data de nascimento 19/07/1986.

Além do operador de igualdade, outros também poderão ser utilizados, conforme veremos em capítulos à frente.

4.5 SELECT FOR UPDATE

O comando SELECT com a cláusula FOR UPDATE bloqueia todas as linhas selecionadas. Nenhuma alteração poderá ser realizada em sessões diferentes daquela que emitiu o comando. O bloqueio será mantido até que a sessão que emitiu o comando realize o COMMIT ou o ROLLBACK. (Para mais detalhes sobre os comandos COMMIT e ROLLBACK consulte o capítulo 11: DTL – Data Transact Language).

```
SELECT * FROM NOME_DA_TABELA FOR UPDATE;
```

No exemplo apresentado a seguir as linhas com ID_CLIENTE entre 1001 e 1005 estarão bloqueadas para as outras sessões até que seja emitido o COMMIT ou o ROLLBACK.

```
SELECT * FROM CLIENTE
WHERE CODIGO_CLIENTE BETWEEN 1001 AND 1005 FOR UPDATE;
```

4.6 MERGE

O comando MERGE é utilizado para mesclar dados de duas ou mais tabelas. É, sem dúvida, o mais complexo dentre os comandos DML. Porém, ele pode combinar em uma única instrução os comandos INSERT, UPDATE e até, em certos casos, DELETE.

Imagine que além da tabela CLIENTE, criada anteriormente, exista uma outra tabela com estrutura idêntica ou semelhante denominada CLIENTE_2. A tabela CLIENTE_2 apresenta dados de alguns clientes que também constam na CLIENTE. Porém os dados destes clientes estão mais atualizados na tabela CLIENTE_2. Além disso, na tabela CLIENTE_2 há dados de clientes que ainda não são encontrados na tabela CLIENTE. Este cenário demonstra a importância do comando MERGE.

No exemplo apresentado a seguir veremos como é possível atualizar a tabela CLIENTE a partir de dados mais atualizados encontrados na tabela CLIENTE_2. E não apenas isso, mas também como inserir na tabela CLIENTE os dados que constam na tabela CLIENTE_2 que não são encontrados na tabela CLIENTE.

Em linhas gerais, informamos primeiro que queremos mesclar os dados da tabela CLIENTE, referenciada através do apelido C, com os dados da tabela CLIENTE_2, referenciada através do apelido C_2. A seguir é colocada a condição: os códigos dos clientes devem ser iguais nas duas tabelas (C.CODIGO_CLIENTE = C_2.CODIGO_CLIENTE). Quando isso ocorrer (MATCHED), os dados da tabela CLIENTE serão atualizados conforme os dados da tabela CLIENTE_2. Por outro lado, quando não forem encontrados valores correspondentes aos códigos de clientes na tabela CLIENTE que sejam idênticos aos da tabela CLIENTE_2, os dados encontrados na tabela CLIENTE_2 serão inseridos na tabela CLIENTE.

```

MERGE INTO CLIENTE C
USING CLIENTE_2 C_2 ON (
C.CODIGO_CLIENTE = C_2.CODIGO_CLIENTE)
WHEN MATCHED THEN
UPDATE
SET
C.NOME_CLIENTE = C_2.NOME_CLIENTE,
C.DATA_NASC_CLIENTE = C_2.DATA_NASC_CLIENTE,
C.UF_CLIENTE = C_2.UF_CLIENTE
WHEN NOT MATCHED THEN
INSERT
(C.CODIGO_CLIENTE, C.NOME_CLIENTE,
C.DATA_NASC_CLIENTE, C.UF_CLIENTE)
VALUES
(C_2.CODIGO_CLIENTE, C_2.NOME_CLIENTE,
C_2.DATA_NASC_CLIENTE, C_2.UF_CLIENTE);

```

O resultado será, portanto, uma tabela CLIENTE completamente atualizada. Trata-se de um comando mais complexo, porém muitíssimo útil em situações conforme a que acabamos de apresentar.

RESUMO

Os comandos que fazem parte do subgrupo DML (Data Manipulation Language) são os seguintes:

- **SELECT:** Utilizado para realização de consultas em uma ou mais tabelas.
- **INSERT:** Utilizado para inserção de novas linhas em tabelas.
- **UPDATE:** Utilizado para atualização dos valores contidos nas linhas das tabelas.
- **DELETE:** Utilizado para exclusão de linhas das tabelas.
- **MERGE:** Utilizado para mesclagem de dados entre duas ou mais tabelas.

EXERCÍCIOS

A tabela ALUNO é composta por duas colunas, RA NUMBER (4) e NOME VARCHAR2 (40). Qual dos seguintes comandos deverá ser utilizado para se inserir uma nova linha na tabela?

- INSERT TO ALUNO VALUES (1001,'ANTONIO ALVARES');
- INSERT INTO ALUNO VALUES (1001,'ANTONIO ALVARES');
- INSERT INTO ALUNO VALUES (1001,ANTONIO ALVARES);
- INSERT FROM ALUNO VALUES (1001,'ANTONIO ALVARES');
- INSERT VALUES INTO ALUNO (1001,'ANTONIO ALVARES');

A linha inserida no exercício anterior precisa ser atualizada. O nome do aluno deve ser alterado de ANTONIO ALVARES para CLAUDIO CARDOSO. Qual dos seguintes comandos deve ser utilizado para realizar esta atualização na tabela?

- ALTER ALUNO SET NOME = 'CLAUDIO CARDOSO' WHERE RA = 1001;
- RENAME ALUNO NOME = 'CLAUDIO CARDOSO' WHERE RA = 1001;
- UPDATE ALUNO WHERE RA = 1001 TO 'CLAUDIO CARDOSO';

- d) UPDATE ALUNO NOME = 'CLAUDIO CARDOSO' WHERE RA = 1001;
- e) UPDATE ALUNO SET NOME = 'CLAUDIO CARDOSO' WHERE RA = 1001;

CAPÍTULO 5 – QUERIES (CONSULTAS)

Queries ou consultas às tabelas de bancos de dados relacionais são realizadas, conforme vimos, através do comando SELECT. As consultas realizadas no capítulo anterior utilizaram apenas o operador de igualdade e apresentaram apenas uma condição simples após a cláusula WHERE. Neste capítulo utilizaremos consultas mais elaboradas, porém baseadas em uma única tabela. Consultas envolvendo duas ou mais tabelas serão abordadas mais à frente quando tratarmos de junções de tabelas (JOINS) e subconsultas (SUB QUERIES).

5.1 CLÁUSULA WHERE

A cláusula WHERE (onde, em inglês) é utilizada como parte da sintaxe da linguagem SQL e antecede as condições que deverão ser verificadas para apresentação de dados de determinada consulta. Uma ou mais condição pode ser verificada. Há também diversos operadores que podem ser utilizados como veremos a seguir.

5.2 OPERADORES

A linguagem SQL apresenta vários tipos de operadores que podem ser agrupados conforme segue:

- Operadores de comparação;
- Operadores lógicos;
- Operadores SQL.

5.2.1 OPERADORES DE COMPARAÇÃO

A linguagem SQL define seis operadores de comparação, conforme você pode observar na tabela a seguir.

OPERADORES DE COMPARAÇÃO	
OPERADOR	DESCRIÇÃO
=	Igual

<	Menor
<=	Menor ou igual
>	Maior
>=	Maior ou igual
< >	Diferente

Fonte: Autor.

A consulta a seguir apresentará os nomes dos clientes (NOME_CLIENTE) cujos códigos (CODIGO_CLIENTE) sejam maior ou igual a 1001.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE >= 1001;
```

5.2.2 OPERADORES LÓGICOS

A linguagem SQL define três operadores lógicos, conforme apresentados na tabela a seguir.

OPERADORES LÓGICOS	
OPERADOR	DESCRIÇÃO
AND	Corresponde ao "E" lógico. Para que a expressão seja satisfeita as duas condições apresentadas devem ser verdadeiras.
OR	Corresponde ao "OU" lógico. Para que a expressão seja satisfeita uma das duas condições deve ser verdadeira.
NOT	Operador lógico de negação.

Fonte: Autor.

A consulta a seguir apresentará os nomes dos clientes (NOME_CLIENTE) cujos códigos (CODIGO_CLIENTE) sejam maior ou igual a 1001 e cujo estado (UF_CLIENTE) seja igual a RS (Rio Grande do Sul). Portanto, clientes com código maior que 1001, mas que não sejam do RS não aparecerão como resultado desta consulta. Tampouco clientes do RS com código menor que 1001 aparecerão como resultado da consulta.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE >= 1001
AND
UF_CLIENTE = 'RS' ;
```

Por outro lado, a consulta a seguir apresentará os nomes dos clientes (NOME_CLIENTE) cujos códigos (CODIGO_CLIENTE) sejam maior ou igual a 1001 ou cujo estado (UF_CLIENTE) seja igual a RS (Rio Grande do Sul). Portanto, clientes com código maior que 1001, mesmo que não sejam do RS aparecerão como resultado desta consulta. Também clientes do RS com código menor que 1001 aparecerão como resultado da consulta.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE >= 1001
OR
UF_CLIENTE = 'RS' ;
```

Conforme observado, precisamos tomar muito cuidado ao utilizar os operadores lógicos para que possamos obter os resultados corretos nas nossas consultas.

5.2.3 OPERADORES SQL

A SQL define diversos operadores específicos da linguagem. A tabela a seguir apresenta os operadores SQL.

OPERADORES SQL	
OPERADOR	DESCRIÇÃO
IS NULL	Verifica se o conteúdo da coluna é NULL (ausência de valores)
IS NOT NULL	Negação do operador IS NULL

LIKE	<p>Compara cadeia de caracteres utilizando padrões de comparação:</p> <p>% substitui zero, um ou mais caracteres</p> <p>_ substitui um caractere</p> <p>Exemplos:</p> <p>LIKE 'A%' inicia com a letra A</p> <p>LIKE '%A' termina com a letra A</p> <p>LIKE '%A%' tem a letra A em qualquer posição</p> <p>LIKE 'A_' string de dois caracteres, inicia com a letra A</p> <p>LIKE '_A' string de dois caracteres, termina com a letra A</p> <p>LIKE '_A_' string de três caracteres, letra A na segunda posição</p> <p>LIKE '%A_' tem a letra A na penúltima posição</p> <p>LIKE '_A%' tem a letra A na segunda posição</p>
NOT LIKE	Negação do operador LIKE
IN	Verifica se um valor pertence a um conjunto de valores
NOT IN	Negação do operador IN
BETWEEN	<p>Determina um intervalo de busca entre dois valores:</p> <p>BETWEEN 'valor1' AND 'valor2'</p>
NOT BETWEEN	Negação do operador BETWEEN
EXISTS	<p>Testa se uma consulta retorna algum resultado ou não, retornando true (verdadeiro) caso exista pelo menos uma linha e se não obtiver nenhuma linha o resultado é false (falso).</p>
NOT EXISTS	Negação do operador EXISTS

Fonte: Autor.

O exemplo a seguir apresenta os nomes dos clientes (NOME_CLIENTE) cujas datas de nascimento não foram inseridas na coluna correspondente (DATA_NASC_CLIENTE).

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE DATA_NASC_CLIENTE IS NULL;
```

O próximo exemplo apresenta os nomes de todos os clientes cujos nomes começam com o caractere 'A'.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE NOME_CLIENTE LIKE 'A%';
```

E este exemplo apresenta os nomes de todos os clientes cujos nomes terminam com o caractere 'A'.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE NOME_CLIENTE LIKE '%A';
```

O próximo apresenta os nomes dos clientes que contenham o caractere 'A' em qualquer posição do nome.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE NOME_CLIENTE LIKE '%A%';
```

Por outro lado, o próximo apresenta os nomes dos clientes que não contenham o caractere 'A' em qualquer posição do nome.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE NOME_CLIENTE NOT LIKE '%A%';
```

E o exemplo a seguir apresenta os nomes dos clientes que contenham o caractere 'A' na segunda posição do nome (segundo caractere do nome).

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE NOME_CLIENTE LIKE '_A%';
```

A seguinte consulta apresentará os nomes dos clientes (NOME_CLIENTE) que nasceram em um dos seguintes estados (UF_CLIENTE) 'RS', 'SC', 'PR'.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE UF_CLIENTE IN ('RS', 'SC', 'PR');
```

A consulta a seguir apresentará os nomes dos clientes (NOME_CLIENTE) que nasceram entre 1ª de janeiro de 1986 e 31 de dezembro de 1990.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE DATA_NASC_CLIENTE BETWEEN '01/01/1986' AND
'31/12/1990';
```

A tabela de operadores SQL apresenta ainda o operador EXISTS e a sua negação NOT EXISTS. Utilizaremos estes operadores mais adiante no capítulo que abordará subconsultas (SUB QUERIES).

5.3 ALIAS

Alias ou apelidos são nomes alternativos, temporários e muitas vezes abreviados utilizados para designar tabelas ou colunas de tabelas. A princípio pode parecer estranho utilizar um nome (ou até um simples caractere) alternativo no lugar do nome de uma tabela ou coluna de tabela. No entanto, o uso de apelidos torna muito práticas consultas, principalmente as que envolvem mais de uma tabela. Até este momento utilizamos apelidos ou alias apenas uma vez quando apresentamos o comando MERGE no capítulo anterior. No entanto, voltaremos a falar sobre apelidos mais à frente, demonstrando inclusive como em algumas situações seu uso é indispensável, como ocorre quando trabalhamos com um tipo de junção denominado SELF JOIN.

Observe um exemplo simples onde utilizamos um apelido em substituição ao nome de uma coluna e outro apelido em substituição ao nome de uma tabela.

```
SELECT C.NOME_CLIENTE NM FROM CLIENTE C;
```

No exemplo apresentado utilizamos os caracteres NM como alias para a coluna NOME_CLIENTE e C para a tabela CLIENTE (embora consultas simples como esta não exijam a utilização de alias). Note que os apelidos aparecem logo após o nome da coluna ou da tabela. A utilização de

C.NOME_CLIENTE indica que NOME_CLIENTE é uma coluna da tabela CLIENTE (o alias 'C' aparece logo a seguir na consulta).

5.4 DISTINCT

A cláusula DISTINCT é utilizada em uma consulta quando desejamos que valores distintos (não repetidos) sejam apresentados como resultado, mesmo quando aparecem mais de uma vez na tabela.

Imagine que na tabela CLIENTE tenhamos três clientes do RS (Rio Grande do Sul), dois de SC (Santa Catarina) e dois do PR (Paraná). Veja, a seguir, um exemplo de como poderiam ficar nossas consultas:

1º – SEM a cláusula DISCTINCT:

```
SELECT UF_CLIENTE FROM CLIENTE;
--
UF
--
RS
SC
RS
PR
SC
RS
PR
```

2º – COM a cláusula DISCTINCT:

```
SELECT DISTINCT UF_CLIENTE FROM CLIENTE;
--
UF
--
RS
SC
PR
```

Conforme observado, os valores repetidos foram omitidos na segunda consulta, naquela que utilizamos a cláusula DISTINCT.

5.5 ORDER BY

Quando realizamos consultas muitas vezes desejamos que os dados sejam ordenados de forma crescente ou decrescente. A menos que informemos isso no momento da consulta os dados aparecerão sem nenhuma ordenação. A cláusula ORDER BY existe justamente para isso, para que os dados de uma consulta possam aparecer de forma ordenada. Veja, a seguir, um exemplo de como poderiam ficar nossas consultas:

1º – SEM a cláusula ORDER BY:

```
SELECT NOME_CLIENTE FROM CLIENTE;
```

```
-----
```

```
NOME_CLIENTE
```

```
-----
```

```
BEATRIZ BARBOSA
```

```
CLAUDIO CARDOSO
```

```
ANTONIO ALVARES
```

2º – COM a cláusula ORDER BY (ordem crescente):

```
SELECT NOME_CLIENTE FROM CLIENTE ORDER BY NOME_CLIENTE;
```

```
-----
```

```
NOME_CLIENTE
```

```
-----
```

```
ANTONIO ALVARES
```

```
BEATRIZ BARBOSA
```

```
CLAUDIO CARDOSO
```

3º – COM a cláusula ORDER BY (ordem decrescente):

```
SELECT NOME_CLIENTE FROM CLIENTE ORDER BY NOME_CLIENTE
DESC;
```

```
-----
NOME_CLIENTE
-----
```

```
CLAUDIO CARDOSO
```

```
BEATRIZ BARBOSA
```

```
ANTONIO ALVARES
```

O segundo exemplo apresenta os dados em ordem crescente ou ascendente. Poderíamos acrescentar ASC após a cláusula ORDER BY, mas isso seria desnecessário uma vez que este é o "default" (padrão) quando trabalhamos com ordenação. O terceiro exemplo apresenta os dados em ordem decrescente ou descendente, por isso acrescentamos DESC após a cláusula ORDER BY, no final do comando.

5.6 GROUP BY

O agrupamento de dados oriundos de diferentes linhas de uma tabela também ocorre com frequência quando trabalhamos com bancos de dados relacionais. A cláusula GROUP BY supre exatamente esta necessidade. Através dela podemos agrupar dados em relatórios que podem atender as necessidades de diferentes usuários.

Quando realizamos o agrupamento de dados utilizamos as chamadas FUNÇÕES DE AGRUPAMENTO. Estas funções permitem que apresentemos, por exemplo, o valor total dos pedidos de cada cliente, o valor médio de seus pedidos etc. No próximo capítulo vamos apresentar cinco diferentes funções de agrupamento. Voltaremos, portanto, a abordar a cláusula GROUP BY logo mais.

5.7 CLÁUSULA HAVING

Conforme vimos anteriormente, quando desejamos filtrar os dados de uma consulta de acordo com determinadas condições, utilizamos a cláusula WHERE. A cláusula WHERE é utilizada quando desejamos filtrar valores que estão sendo considerados linha a linha, porém não funciona

quando desejamos filtrar valores que foram agrupados. Portanto, a linguagem SQL oferece a cláusula HAVING para que possamos filtrar valores que foram agrupados a partir de várias linhas de uma tabela. Falaremos novamente deste assunto no próximo capítulo na seção FUNÇÕES DE GRUPO.

5.8 CASE

A instrução CASE implementa uma estrutura de controle em uma consulta (SELECT) ou atualização (UPDATE). Quando utilizamos a cláusula CASE em comandos SQL podemos economizar diversas linhas de código, pois não é necessário criar blocos de programação (por exemplo, PL/SQL) para testar condições. Observe os exemplos a seguir.

Exemplo 1, com base na tabela CLIENTE, quando realizamos a seguinte consulta:

```
SELECT NOME_CLIENTE, UF_CLIENTE,
CASE UF_CLIENTE
WHEN 'RS' THEN 'RIO GRANDE DO SUL'
WHEN 'SC' THEN 'SANTA CATARINA'
ELSE 'OUTRO ESTADO'
END NOME_ESTADO
FROM CLIENTE;
```

O resultado da consulta será:

```
NOME_CLIENTE      UF NOME_ESTADO
-----  --  -----
ANTONIO ALVARES  RS  RIO GRANDE DO SUL
BEATRIZ BARBOSA  SC  SANTA CATARINA
CLAUDIO CARDOSO  PR  OUTRO ESTADO
```

Observe que foi acrescentado após o END do comando o nome que queremos para a coluna temporária (NOME_ESTADO). O banco de dados faz a comparação com base nas condições descritas em CASE, ou seja, quando a UF_CLIENTE for igual a RS, será atribuído a NOME_ESTADO

o valor RIO GRANDE DO SUL; se for igual a SC, será atribuído SANTA CATARINA e se não satisfizer a nenhuma das condições anteriores será atribuído o valor OUTRO ESTADO.

Exemplo 2, com base na tabela PEDIDO, quando realizamos a seguinte consulta:

```
SELECT NR_PEDIDO, VALOR_PEDIDO,
CASE
WHEN VALOR_PEDIDO <= 1000.00 THEN VALOR_PEDIDO * 0.9
WHEN VALOR_PEDIDO <= 2000.00 THEN VALOR_PEDIDO * 0.8
ELSE VALOR_PEDIDO * 0.7
END VALOR_PROMOCIONAL
FROM PEDIDO;
```

O resultado da consulta será o seguinte:

NR_PEDIDO	VALOR_PEDIDO	VALOR_PROMOCIONAL
1	1000	900
2	2000	1600
3	3000	2100

Neste exemplo o banco de dados faz a comparação com base nas condições descritas em CASE. Quando o VALOR_PEDIDO for menor ou igual a 1000, o VALOR_PEDIDO será multiplicado por 0.9; se for menor ou igual a 2000, o VALOR_PEDIDO será multiplicado por 0.8 e se não satisfizer a nenhuma das condições anteriores o VALOR_PEDIDO será multiplicado por 0.7. Os resultados aparecerão na coluna temporária VALOR_PROMOCIONAL.

Exemplo 3, com base na tabela PEDIDO, quando realizamos a seguinte atualização:

```
UPDATE PEDIDO
SET VALOR_PEDIDO =
CASE
```



```

WHEN VALOR_PEDIDO <= 1000.00 THEN VALOR_PEDIDO * 0.9
WHEN VALOR_PEDIDO <= 2000.00 THEN VALOR_PEDIDO * 0.8
ELSE VALOR_PEDIDO * 0.7
END ;

```

O resultado da atualização será o seguinte:

NR_PEDIDO	VALOR_PEDIDO
1	900
2	1600
3	2100

Neste exemplo o banco também faz a comparação com base nas condições apresentadas no Exemplo 2. Porém os resultados são usados para atualização da coluna VALOR_PEDIDO. Note, portanto, que diferente dos exemplos anteriores nos quais não ocorreram alterações de valores nas tabelas, neste último caso os valores foram alterados ou atualizados, conforme as condições apresentadas na tabela PEDIDO.

RESUMO

As consultas às tabelas de bancos de dados relacionais são realizadas através do comando SELECT.

A cláusula WHERE antecede as condições que deverão ser verificadas para apresentação de dados de determinada consulta. Uma ou mais condição pode ser verificada.

Diversos operadores que podem ser utilizados:

- Operadores de comparação: =, <, <=, >, >=, <>
- Operadores lógicos: AND, OR, NOT
- Operadores SQL: IS NULL, LIKE, IN, BETWEEN, EXISTS

Alias ou apelidos são nomes alternativos utilizados para designar tabelas ou colunas de tabelas.

A cláusula **DISTINCT** é utilizada em uma consulta quando desejamos que o resultado de uma consulta não apresente valores repetidos, mesmo quando aparecem mais de uma vez na tabela.

A cláusula **ORDER BY** é utilizada para que os dados de uma consulta sejam apresentados de forma ordenada.

A cláusula **GROUP BY** é utilizada com as funções de agrupamento para agrupar dados em relatórios que podem atender as necessidades de diferentes usuários.

A cláusula **HAVING** é utilizada para filtrar valores que foram agrupados a partir de várias linhas de uma tabela.

A instrução **CASE** implementa uma estrutura de controle em uma consulta ou em uma atualização.

EXERCÍCIOS

A tabela **ALUNO** é composta por duas colunas, **RA** e **NOME**. Qual dos seguintes comandos deverá ser utilizado para apresentar os nomes dos alunos em ordem decrescente?

- a) `SELECT NOME FROM ALUNO GROUP BY DESC NOME;`
- b) `SELECT NOME FROM ALUNO GROUP BY NOME DESC;`
- c) `SELECT NOME FROM ALUNO ORDER DESC BY NOME;`
- d) `SELECT NOME FROM ALUNO ORDER NOME DESC;`
- e) `SELECT NOME FROM ALUNO ORDER BY NOME DESC;`

A tabela **CLIENTE** apresenta uma coluna denominada **NOME**. Qual dos seguintes comandos deve ser utilizado para selecionar todos os clientes cujos nomes contenham o caractere **Y** em qualquer posição (começo, meio ou fim)?

- a) `SELECT NOME FROM CLIENTE WHERE NOME = '%Y%';`
- b) `SELECT NOME FROM CLIENTE WHERE NOME = '_Y_';`
- c) `SELECT NOME FROM CLIENTE WHERE NOME LIKE '%Y%';`
- d) `SELECT NOME FROM CLIENTE WHERE NOME LIKE '_Y_';`
- e) `SELECT NOME FROM CLIENTE WHERE NOME LIKE 'Y';`

CAPÍTULO 6 – FUNÇÕES

Neste capítulo apresentaremos as funções nativas da SQL, conforme implementadas no Oracle Database. As funções estão agrupadas conforme segue:

- Funções de grupo;
- Funções de linha;
- Funções numéricas;
- Funções de conversão;
- Funções de expressões regulares;
- Outras funções.

6.1 FUNÇÕES DE GRUPO

Funções de grupo (também conhecidas como funções de agregação) são funções nativas SQL que operam em grupos de linhas e retornam um valor para todo o grupo. Estas funções são: SUM, AVG, MAX, MIN, COUNT, MEDIAN, STDDEV e VARIANCE.

6.1.1 SUM

A função SUM retorna a soma por grupo das colunas selecionadas. Imagine, por exemplo, uma tabela denominada PEDIDO com os seguintes dados:

NR_PEDIDO	VALOR	UF
1	2500	SP
2	1200	RJ
3	1600	SP
4	1800	RJ

Podemos obter o total por UF (Estado) dos pedidos utilizando a função SUM. Observe:

```
SELECT UF, SUM(VALOR) FROM PEDIDO GROUP BY UF;
```

A consulta apresentará a seguinte resposta:

UF	SUM (VALOR)
RJ	3000
SP	4100

A função SUM pode ser utilizada também para apresentar o valor total de TODOS os pedidos. Procedimentos semelhantes podem ser aplicados às outras funções de grupo descritas a seguir.

```
SELECT SUM (VALOR) FROM PEDIDO;
SUM (VALOR)
-----
          7100
```

6.1.2 AVG

A função AVG retorna a média por grupo das colunas selecionadas.

Considerando ainda a tabela PEDIDO, podemos obter a média dos pedidos por UF utilizando a função AVG. Observe:

```
SELECT UF, AVG (VALOR) FROM PEDIDO GROUP BY UF;
```

A consulta apresentará a seguinte resposta:

UF	AVG (VALOR)
RJ	1500
SP	2050

6.1.3 MAX

A função MAX retorna o maior valor por grupo das colunas selecionadas.

Podemos obter o maior valor de pedido por UF com base na tabela PEDIDO utilizando a função MAX, conforme apresentado a seguir.

```
SELECT UF, MAX (VALOR) FROM PEDIDO GROUP BY UF;
```

A consulta apresentará a seguinte resposta:

UF	MAX (VALOR)
RJ	1800
SP	2500

6.1.4 MIN

A função MIN retorna o menor valor por grupo das colunas selecionadas. Podemos obter o menor valor de pedido por UF com base na tabela PEDIDO utilizando a função MIN, conforme segue.

```
SELECT UF, MIN (VALOR) FROM PEDIDO GROUP BY UF;
```

A consulta apresentará a seguinte resposta:

UF	MIN (VALOR)
RJ	1200
SP	1600

6.1.5 COUNT

A função COUNT retorna o número de linhas por grupo das colunas selecionadas.

Podemos obter a quantidade de pedidos (ou o número de pedidos emitidos) por UF com base na tabela PEDIDO utilizando a função COUNT, conforme apresentado a seguir.

```
SELECT UF, COUNT (NR_PEDIDO) FROM PEDIDO GROUP BY UF;
```

A consulta apresentará a seguinte resposta:

```
UF COUNT (NR_PEDIDO)
```

```
-- -----
RJ                2
SP                2
```

Observe que utilizamos a coluna NR_PEDIDO como parâmetro para a função COUNT. O resultado seria o mesmo se utilizássemos a coluna VALOR desde que tivéssemos como base os dados da tabela PEDIDO. No entanto, isso nem sempre ocorrerá. Quando selecionamos uma coluna que contém um ou mais valores nulos (NULL) estes valores são desprezados se utilizarmos a função COUNT. Portanto, quando precisamos saber a quantidade de linhas de uma tabela é recomendável que utilizemos o caractere (*) ao invés de nomes de colunas. Observe como podemos obter de forma segura a quantidade de linhas da tabela CLIENTE:

```
SELECT COUNT (*) FROM PEDIDO;
```

A consulta apresentará a seguinte resposta:

```
COUNT (*)
```

```
-----
4
```

6.1.6 MEDIAN

A função MEDIAN retorna o valor correspondente à mediana por grupo das colunas selecionadas.

Imagine que na coluna VALOR_SALARIO de uma tabela denominada SALARIO tivéssemos os seguintes valores:

```
VALOR_SALARIO
```

```
-----
2000
1000
5000
4000
3000
```

A consulta:

```
SELECT MEDIAN (VALOR_SALARIO) FROM SALARIO;
```

Apresentará a seguinte resposta:

```
-----  
3000
```

Lembre-se: Quando temos um número ímpar de elementos (valores numéricos), a função mediana realiza a ordenação dos mesmos e retorna o elemento central como resultado. Quando temos um número par de elementos, a função mediana retorna a média dos dois elementos centrais.

6.1.7 STDDEV

A função STDDEV retorna o valor correspondente ao desvio padrão por grupo das colunas selecionadas.

A consulta:

```
SELECT STDDEV (VALOR_SALARIO) FROM SALARIO;
```

Apresentará a seguinte resposta:

```
-----  
1581.1388
```

Lembre-se: O desvio padrão é definido como a raiz quadrada da variância. (Veja o próximo item.)

6.1.8 VARIANCE

A função VARIANCE retorna o valor correspondente à variância por grupo das colunas selecionadas.

A consulta:

```
SELECT VARIANCE (VALOR_SALARIO) FROM SALARIO;
```

Apresentará a seguinte resposta:

```
-----  
250000
```

Lembre-se: A variância é definida como a dispersão ou variação de um grupo de valores numéricos em uma amostra. Ela é igual ao quadrado do desvio padrão.

6.2 FUNÇÕES DE LINHA

As funções de linha atuam sobre cada uma das linhas resultantes de uma consulta. Para cada linha produzem um valor, conforme os argumentos recebidos. As funções de linha podem ser encadeadas com outras funções. A documentação online fornecida pela Oracle apresenta 21 funções de linha, conforme a tabela a seguir:

CHR	NLS_LOWER	RTRIM
CONCAT	NLSSORT	SOUNDEX
INITCAP	NLS_UPPER	SUBSTR
LOWER	REGEXP_REPLACE	TRANSLATE
LPAD	REGEXP_SUBSTR	TREAT
LTRIM	REPLACE	TRIM
NLS_INITCAP	RPAD	UPPER

Fonte: Disponível em: http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions001.htm

Nesta seção abordaremos 13 funções de linha que consideramos de uso mais frequente, conforme apresentadas a seguir:

- UPPER
- LOWER
- INITCAP
- LPAD
- SUBSTR
- REPLACE
- CONCAT
- LTRIM
- RTRIM
- TRIM
- CHR
- ASCII*
- TRANSLATE

* A função ASCII não é listada na documentação acima para funções de linha. Porém, incluímos esta função nesta seção, pois realiza o oposto da função CHR. Veja mais detalhes consultando os itens 6.2.11 e 6.2.12.

Tabela DUAL:

A tabela DUAL é criada automaticamente pelo Oracle Database e é acessível a todos os usuários. A tabela tem uma coluna, DUMMY, definida como VARCHAR2 (1) e contém uma linha com um valor X. Realizar uma consulta a partir da tabela DUAL é útil para calcular uma expressão constante com a instrução SELECT. Como a tabela DUAL tem apenas uma linha, a constante é retornada apenas uma vez. Deste ponto em diante, utilizaremos com muita frequência a tabela DUAL na apresentação das próximas funções SQL nativas. Você poderá, evidentemente, aplicar os conceitos aprendidos outras tabelas.

6.2.1 UPPER

A função UPPER retorna todos os caracteres da string em maiúsculas.

O comando:

```
SELECT UPPER('aNtOnIo aLvArEs') FROM DUAL;
```

Retornará:

```
ANTONIO ALVARES
```

6.2.2 LOWER

A função LOWER retorna todos os caracteres da string em minúsculas.

O comando:

```
SELECT LOWER('aNtOnIo aLvArEs') FROM DUAL;
```

Retornará:

```
antonio alvares
```

6.2.3 INITCAP

A função INITCAP retorna os primeiros caracteres de cada palavra da string em maiúsculas e os caracteres seguintes em minúsculas.

O comando:

```
SELECT INITCAP('aNtOnIo aLvArEs') FROM DUAL;
```

Retornará:

```
Antonio Alvares
```

6.2.4 LPAD

A função LPAD tem como objetivo o preenchimento à esquerda (Left Padding) e pode ser utilizada com dois ou três parâmetros. O último parâmetro (string_de_preenchimento) é opcional.

```
LPAD (string, tamanho_do_preenchimento,
      string_de_preenchimento)
```

O comando:

```
SELECT LPAD('TESTE',10,'*') FROM DUAL;
```

Retornará:

```
-----
*****TESTE
```

O comando (que omite o último parâmetro da função):

```
SELECT LPAD('TESTE',10) FROM DUAL;
```

Retornará:

```
-----
      TESTE
```

A string será truncada quando o valor correspondente ao tamanho_do_preenchimento for menor que a quantidade de caracteres da string. Observe o resultado do comando a seguir:

```
SELECT LPAD('TESTE',4) FROM DUAL;
```

Retornará:

```
----
TEST
```

6.2.5 RPAD

A função RPAD tem como objetivo o preenchimento à direita (Right Padding) e também pode ser utilizada com dois ou três parâmetros.

Assim como ocorre com a função LPAD, o último parâmetro (string_de_preenchimento) é opcional.

```
RPAD (string, tamanho_do_preenchimento,
      string_de_preenchimento)
```

O comando:

```
SELECT RPAD('TESTE',10,'*') FROM DUAL;
```

Retornará:

```
-----
TESTE*****
```

O comando (que omite o último parâmetro da função):

```
SELECT RPAD('TESTE',10) FROM DUAL;
```

Retornará:

```
-----
TESTE
```

Assim como ocorre com a função LPAD, a string também será truncada quando o valor correspondente ao tamanho_do_preenchimento for menor que a quantidade de caracteres da string. Veja o resultado do comando a seguir:

```
SELECT RPAD('TESTE',4) FROM DUAL;
```

Retornará:

```
----
TEST
```

6.2.6 SUBSTR

Retorna a quantidade de caracteres da string definidos no terceiro parâmetro (comprimento) a partir da posição definida no segundo parâmetro (posição_início). Quando o terceiro parâmetro for omitido, levará em conta todo o comprimento da string.

SUBSTR(string, posicao_inicio, comprimento)

O comando abaixo apresentará uma 'substring' a partir da segunda posição da string TESTE com comprimento igual a 3.

```
SELECT SUBSTR('TESTE',2,3) FROM DUAL;
```

Retornará:

```
---  
EST
```

O comando abaixo apresentará uma 'substring' a partir da segunda posição da string TESTE. Visto que não foi especificado o tamanho (terceiro parâmetro da função) levará em conta todo o comprimento da string.

```
SELECT SUBSTR('TESTE',2) FROM DUAL;
```

Retornará:

```
----  
ESTE
```

O comando abaixo utiliza um valor negativo para o segundo parâmetro da função (posicao_inicio) e apresentará uma 'substring' a partir da terceira posição da string TESTE, porém contada da direita para a

esquerda. Visto que não foi especificado o tamanho (terceiro parâmetro da função) levará em conta todo o comprimento da string.

```
SELECT SUBSTR('TESTE',-3) FROM DUAL;
```

Retornará:

```
---
STE
```

6.2.7 REPLACE

A função REPLACE substitui na string (primeiro parâmetro) todas as ocorrências definidas na string_a_substituir (segundo parâmetro) pela string_de_substituicao (terceiro parâmetro) da função. Caso não seja definido o terceiro parâmetro, nada será colocado no lugar, isto é, os valores da string_a_substituir (segundo parâmetro) serão simplesmente "retirados" da string (primeiro parâmetro).

```
REPLACE(string, string_a_substituir,
string_de_substituicao)
```

O comando abaixo substituirá a parte da 'string' que corresponde a AAA. Visto que não foi utilizado o terceiro parâmetro (string_de_substituicao) nada será colocado no lugar.

```
SELECT REPLACE('AAATESTE', 'AAA') FROM DUAL;
```

Retornará:

```
-----
TESTE
```

O comando abaixo substituirá a parte da 'string' que corresponde a AAA. Será utilizado o terceiro parâmetro (string_de_substituicao) no seu lugar.

```
SELECT REPLACE('AAATESTE','AAA','BBB') FROM DUAL;
```

Retornará:

```
-----
BBBTESTE
```

6.2.8 TRANSLATE

A função TRANSLATE substitui na string (primeiro parâmetro) todas as ocorrências definidas na string_a_substituir (segundo parâmetro) pela string_de_substituicao (terceiro parâmetro) da função. Porém, a substituição é realizada levando-se em conta cada caractere.

```
TRANSLATE(string, string_a_substituir,
string_de_substituicao)
```

O comando apresentado a seguir:

```
SELECT TRANSLATE('123TESTE123','123','456') FROM DUAL;
```

Retornará:

```
-----
456TESTE456
```

O comando apresentado a seguir:

```
SELECT TRANSLATE('SOGRA','SG','CB') FROM DUAL;
```

Retornará:

```
-----
COBRA
```

6.2.9 CONCAT

A função CONCAT é utilizada para concatenar valores de duas strings.

CONCAT(string1, string2)

O comando abaixo concatenará a 'string' que corresponde a A com a 'string' que corresponde a B.

```
SELECT CONCAT('A', 'B') FROM DUAL;
```

Retornará:

```
--  
AB
```

Caso seja necessário concatenar mais de duas strings deve-se utilizar a função CONCAT de forma encadeada, conforme segue.

CONCAT(CONCAT(string1, string2), string3)

O comando abaixo concatenará três 'strings' que correspondem respectivamente a A, B e C.

```
SELECT CONCAT(CONCAT('A', 'B'), 'C') FROM DUAL;
```

Retornará:

```
---  
ABC
```

6.10 LTRIM

A função LTRIM remove todos os caracteres, especificados no segundo parâmetro, do lado esquerdo de uma string (primeiro parâmetro).


```
LTRIM(string, trim_string)
```

O comando apresentado a seguir:

```
SELECT LTRIM('  TESTE') FROM DUAL;
```

Retornará:

```
-----
TESTE
```

O comando apresentado a seguir:

```
SELECT LTRIM('123TESTE', '123') FROM DUAL;
```

Retornará:

```
-----
TESTE
```

6.11 RTRIM

A função LTRIM remove todos os caracteres especificados no segundo parâmetro, do lado direito de uma string (primeiro parâmetro).

```
RTRIM( string, trim_string)
```

O comando apresentado a seguir:

```
SELECT RTRIM('TESTE  ') FROM DUAL;
```

Retornará:

TESTE

O comando apresentado a seguir:

```
SELECT RTRIM('TESTE123','123') FROM DUAL;
```

Retornará:

TESTE

6.12 TRIM

A função TRIM remove todos os caracteres especificados de uma string, que poderão estar no início (LEADING), no final (TRAILING) ou em ambos (BOTH) os lados da string.

```
TRIM(LEADING | TRAILING | BOTH trim_character FROM string)
```

O comando apresentado a seguir:

```
SELECT TRIM(' ' FROM '   TESTE   ') FROM DUAL;
```

Retornará:

TESTE

O comando apresentado a seguir:

```
SELECT TRIM(LEADING '0' FROM '000TESTE') FROM DUAL;;
```

Retornará:

```
-----
TESTE
```

O comando apresentado a seguir:

```
SELECT TRIM(TRAILING '0' FROM 'TESTE000') FROM DUAL;
```

Retornará:

```
-----
TESTE
```

O comando apresentado a seguir:

```
SELECT TRIM(BOTH '0' FROM '000TESTE000') FROM DUAL;
```

Retornará:

```
-----
TESTE
```

6.13 LENGTH

A função LENGTH retorna a quantidade de caracteres da string.

O comando apresentado a seguir:

```
SELECT LENGTH('TESTE') FROM DUAL;
```

Retornará:

```
-----
5
```

6.14 CHR

A função CHR retorna o caractere conforme o código ASCII com base no parâmetro numérico (NUMBER na base 10).

O comando apresentado a seguir:

```
SELECT CHR(65) FROM DUAL;
```

Retornará:

```
-  
A
```

6.15 ASCII

A função ASCII faz o oposto da função CHR, apresenta o código ASCII (NUMBER na base 10) com base no parâmetro (CHAR).

O comando apresentado a seguir:

```
SELECT ASCII('A') FROM DUAL;
```

Retornará:

```
--  
65
```

6.3 FUNÇÕES NUMÉRICAS

As funções numéricas recebem entradas numéricas e retornam valores numéricos, conforme os argumentos recebidos. A documentação da Oracle apresenta 26 funções numéricas, conforme a tabela a seguir:

ABS	EXP	SIGN
ACOS	FLOOR	SIN

ASIN	LN	SINH
ATAN	LOG	SQRT
ATAN2	MOD	TAN
BITAND	NANVL	TANH
CEIL	POWER	TRUNC
COS	REMAINDER	WIDTH_BUCKET
COSH	ROUND	

Disponível em: http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions001.htm

Apresentaremos nesta seção nove funções numéricas que consideramos de uso mais frequente. Não abordaremos, por exemplo, funções de uso mais específico, como as funções trigonométricas SIN, COS, TAN etc.

6.3.1 ABS

A função ABS retorna o valor absoluto de um número. Retornará um valor absoluto independentemente se o número for positivo ou negativo.

Observe os dois exemplos:

```
SELECT ABS(10) FROM DUAL;
```

Retornará:

```
---
```

```
10
```

```
SELECT ABS(-10) FROM DUAL;
```

Retornará:

```
---
```

```
10
```

6.3.2 CEIL

A função CEIL retorna próximo número inteiro maior ou igual ao número do argumento.

Observe os exemplos a seguir:

```
SELECT CEIL(10) FROM DUAL;
```

Retornará:

```
---
```

```
10
```

```
SELECT CEIL(10.4) FROM DUAL;
```

Retornará:

```
---
```

```
11
```

```
SELECT CEIL(-10.4) FROM DUAL;
```

Retornará:

```
---
```

```
-10
```

6.3.3 FLOOR

A função FLOOR retorna o próximo número inteiro menor ou igual ao número do argumento.

Observe os exemplos a seguir:

```
SELECT FLOOR(10) FROM DUAL;
```

Retornará:

10

```
SELECT FLOOR(10.4) FROM DUAL;
```

Retornará:

10

```
SELECT FLOOR(-10.4) FROM DUAL;
```

Retornará:

-11

6.3.4 ROUND

A função ROUND retorna o arredondamento do número do primeiro argumento. A função pode receber um ou dois argumentos. Quando receber apenas o número a ser arredondado, retornará um inteiro (conforme arredondamento). Quando receber um segundo argumento (que corresponde ao número de casas decimais a ser mantido) retornará um número de ponto flutuante (número com "casas decimais").

Observe os exemplos a seguir:

```
SELECT ROUND(10) FROM DUAL;
```

Retornará:

10

```
SELECT ROUND (10.45) FROM DUAL;
```

Retornará:

```
---
```

```
10
```

```
SELECT ROUND (10.45,1) FROM DUAL;
```

Retornará:

```
----
```

```
10.5
```

6.3.5 TRUNC

A função TRUNC retorna o número do primeiro argumento truncado (cortado). Esta função também pode receber um ou dois argumentos. Quando receber apenas o número a ser cortado, retornará um inteiro (sem arredondamento). Quando receber um segundo argumento (que corresponde ao número de casas decimais a ser mantido) retornará um número de ponto flutuante (sem arredondamento).

Observe os exemplos a seguir:

```
SELECT TRUNC (10) FROM DUAL;
```

Retornará:

```
---
```

```
10
```

```
SELECT TRUNC (10.45) FROM DUAL;
```

Retornará:

```
---
```

```
10
```



```
SELECT TRUNC(10.45,1) FROM DUAL;
```

Retornará:

```
----
10.4
```

6.3.6 MOD

A função MOD retorna o resto da divisão. A função recebe dois argumentos: o primeiro é o dividendo e o segundo o divisor.

O exemplo a seguir:

```
SELECT MOD(10,3) FROM DUAL;
```

Retornará o resto da divisão de 10 por 3:

```
---
1
```

6.3.7 POWER

A função POWER retorna o valor da potenciação. A função recebe dois argumentos: o primeiro é a base e o segundo o expoente.

O exemplo a seguir:

```
SELECT POWER(10,2) FROM DUAL;
```

Retornará o valor correspondente a 10 elevado ao quadrado:

```
---
100
```

6.3.8 SQRT

A função SQRT recebe um valor e retorna a raiz quadrada.

O exemplo a seguir:

```
SELECT SQRT(10) FROM DUAL;
```

Retornará o valor correspondente à raiz quadrada de 10:

```
-----
3.16227766
```

6.3.9 SIGN

A função SIGN recebe um valor e retorna 1 para valores maiores que 0 (zero), 0 (zero) quando o valor for igual a 0 (zero) e -1 quando o valor for menor que 0 (zero).

Observe os exemplos a seguir:

```
SELECT SIGN(10) FROM DUAL;
```

Retornará:

```
---
1
SELECT SIGN (0) FROM DUAL;
```

Retornará:

```
---
0
SELECT SIGN(-10) FROM DUAL;
```

Retornará:

```
----
-1
```

6.4 FUNÇÕES DE CONVERSÃO

As funções de conversão convertem um valor de um tipo para outro tipo. A Oracle em sua documentação 35 funções de conversão, conforme a tabela a seguir:

ASCIISTR	ROWIDTOCHAR	TO_NCHAR
BIN_TO_NUM	ROWIDTONCHAR	TO_NCLOB
CAST	SCN_TO_TIMESTAMP	TO_NUMBER
CHARTOROWID	TIMESTAMP_TO_SCN	TO_DSINTERVAL
COMPOSE	TO_BINARY_DOUBLE	TO_SINGLE_BYTE
CONVERT	TO_BINARY_FLOAT	TO_TIMESTAMP
DECOMPOSE	TO_CHAR	TO_TIMESTAMP_TZ
HEXTORAW	TO_CLOB	TO_YMINTERVAL
NUMTODSINTERVAL	TO_DATE	TO_YMINTERVAL
NUMTOYMINTERVAL	TO_DSINTERVAL	TRANSLATE ... USING
RAWTOHEX	TO_LOB	UNISTR
RAWTONHEX	TO_MULTI_BYTE	

Fonte: Disponível em: http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions001.htm

Apresentaremos nesta seção as seguintes funções de conversão que consideramos de uso mais frequente:

- TO_CHAR
- TO_DATE
- TO_NUMBER

6.4.1 TO_CHAR

A função TO_CHAR converte um valor dos tipos DATE ou NUMBER para um valor do tipo CHAR.

O exemplo a seguir retorna o dia do mês, conforme a data do sistema:

```
SELECT TO_CHAR(SYSDATE, 'DD') FROM DUAL;
```

Retornará (por exemplo):

```
--
25
```

O próximo exemplo retorna o mês por extenso, conforme a data do sistema:

```
SELECT TO_CHAR(SYSDATE, 'MONTH') FROM DUAL;
```

Retornará (por exemplo):

```
-----
JANEIRO
```

O exemplo a seguir retorna o valor conforme notação do sistema inglês (separando os milhares com , (vírgulas) e os valores decimais com . (ponto).

```
SELECT TO_CHAR(1250, '999,999.99') FROM DUAL;
```

Retornará:

```
-----
1,250.00
```

6.4.2 TO_DATE

A função TO_DATE converte uma string do tipo CHAR para o tipo DATE.

```
SELECT TO_DATE(SYSDATE, 'MM/DD/YY') FROM DUAL;
```

Retornará (por exemplo):

```
-----
01/25/17
```

É recomendável que você utilize a função TO_DATE para inserir datas em tabelas quando não souber o formato configurado no servidor (DD/MM/YYYY ou MM/DD/YYYY). O exemplo a seguir assegura que a data sempre será inserida corretamente na tabela TESTE.

```
INSERT INTO TESTE (DATA_NASC)
VALUES (TO_DATE('01/25/1980', 'MM/DD/YYYY'));
```

6.4.3 TO_NUMBER

A função TO_NUMBER converte uma string (válida) em um valor do tipo NUMBER.

O exemplo a seguir apresenta como primeiro argumento uma string de caracteres que deverá ser convertida para o tipo NUMBER. O segundo argumento apresenta a "máscara" utilizada para string de caracteres que será levada em consideração para conversão do valor em número.

```
SELECT TO_NUMBER('$1,250.00', '$999,999.99') FROM DUAL;
```

Retorna:

```
----
1250
```

6.5 FUNÇÕES DE EXPRESSÕES REGULARES

Expressões regulares são sequências de caracteres compostos de literais e meta caracteres que descrevem um padrão em um texto (ou string).

Antes de apresentarmos as funções que utilizam expressões regulares, explicaremos brevemente alguns conceitos necessários para compreensão dos exemplos apresentados nesta seção. O assunto Expressões Regulares é extenso e você poderá encontrar livros que

foram publicados com o objetivo de abordar de forma mais completa este assunto.

A tabela a seguir apresenta alguns metacaracteres utilizados em expressões regulares:

Metacaracteres	Significados
\	Corresponde a um caractere especial ou literal ou realiza uma referência retroativa.
^	Corresponde à posição no início da string.
\$	Corresponde à posição no final da string.
*	Corresponde ao caractere anterior, zero ou mais vezes.
?	Corresponde ao caractere anterior, zero ou uma vez.
+	Corresponde ao caractere anterior, uma ou mais vezes.
{n}	Corresponde a um caractere n vezes (n é sempre um valor inteiro).
{n,m}	Corresponde a um caractere no mínimo n vezes e no máximo m vezes (n e m são sempre valores inteiros).
x y	Corresponde a x ou a y (x e y são um ou mais caracteres).
[abc]	Corresponde a qualquer dos caracteres incluídos.
[a-z]	Corresponde a qualquer caractere do intervalo informado.

[:]	Corresponde a qualquer caractere de uma classe, conforme segue: [:alphanum:] 0-9, A-Z e a-z [:alpha:] A-Z e a-z [:lower:] a-z [:upper:] A-Z [:blank:] espaço ou tabulação [:space:] espaço em branco [:graph:] caracteres não em branco [:print:] semelhante a [:graph:] porém inclui o caractere de espaço [:punct:] caracteres de pontuação: ', ", etc. [:xdigit:] hexadecimais: 0-9, A-F, a-f
[..]	Elemento de comparação.
[==]	Classes de equivalência.

Fonte: Autor.

Exemplo 1:

[^]198[1-3]\$

- [^] metacaractere que corresponde à posição inicial da string
- \$ metacaractere que corresponde à posição final da string
- [^]198 corresponde a uma string que começa com 198
- [1-3] corresponde a uma string que termina com 1, 2 ou 3

Portanto, [^]198[1-3]\$ corresponde a 1981, 1982 e 1983.

Exemplo 2:

D[[:alpha:]]{4}

Representa uma string (ou substring, conforme o caso) que começa com D e é seguida por 4 (quatro) caracteres alfabéticos (a-z ou A-Z). Exemplo: DADOS.

O Oracle apresenta diversas funções que utilizam expressões regulares. Nesta seção vamos abordar as seguintes funções:

- REGEXP_LIKE;
- REGEXP_INSTR;
- REGEXP_REPLACE;
- REGEXP_SUBSTR;
- REGEXP_COUNT.

6.5.1 REGEXP_LIKE()

A função REGEXP_LIKE procura no primeiro parâmetro a expressão regular conforme definida no parâmetro padrão. Pode receber ainda uma opção de correspondência, conforme segue:

- 'c': correspondência com diferenciação de minúscula e maiúscula (padrão);
- 'i': correspondência sem diferenciação de minúscula e maiúscula (padrão);
- 'n': permite utilizar o operador de correspondência com qualquer caractere;
- 'm': trata o primeiro parâmetro como uma linha múltipla;

Exemplo:

A consulta a seguir retorna os clientes cujos nomes começam com 'a' ou 'A':

```
SELECT ID_CLIENTE, NOME_CLIENTE FROM CLIENTE
WHERE REGEXP_LIKE (NOME_CLIENTE, '^A', 'i');
```


6.5.2 REGEXP_INSTR()

A função REGEXP_INSTR procura no primeiro parâmetro a expressão regular conforme definida no parâmetro padrão e retorna a posição em que o padrão ocorre. As posições começam em 1 (um).

Exemplo:

A consulta a seguir retorna 15, que é a posição correspondente à expressão regular: 'B[[:alpha:]]{4}' (que corresponde a BANCO, pois começa com B e a seguir apresenta mais quatro caracteres alfabéticos):

```
SELECT REGEXP_INSTR('TECNOLOGIA EM BANCO DE DADOS',
  'B[[:alpha:]]{4}') AS RESULTADO FROM DUAL;
RESULTADO
-----
          15
```

6.5.3 REGEXP_REPLACE()

A função REGEXP_REPLACE procura no primeiro parâmetro a expressão regular conforme definida no parâmetro padrão e substitui pela 'string substituta' (que corresponde ao último parâmetro da função).

Exemplo:

A consulta a seguir substitui a string SETE, que corresponde à expressão regular: 'S[[:alpha:]]{3}' pela string 'DEZ', que corresponde ao último parâmetro da função:

```
SELECT REGEXP_REPLACE('A NOTA DO ALUNO É SETE',
  'S[[:alpha:]]{3}', 'DEZ') AS RESULTADO FROM DUAL;
RESULTADO
-----
A NOTA DO ALUNO É DEZ
```

6.5.4 REGEXP_SUBSTR()

A função REGEXP_SUBSTR procura no primeiro parâmetro a expressão regular conforme definida no parâmetro padrão e a seguir retorna a substring correspondente à expressão regular.

Exemplo:

A consulta a seguir retorna 'DADOS', que é a posição correspondente à expressão regular: 'D[[:alpha:]]{4}' (que corresponde a DADOS, pois começa com D e a seguir apresenta mais quatro caracteres alfabéticos):

```
SELECT REGEXP_SUBSTR('TECNOLOGIA EM BANCO DE DADOS',
'D[[:alpha:]]{4}') AS RESULTADO FROM DUAL;
```

RESULTADO

DADOS

6.5.5 REGEXP_COUNT()

A função REGEXP_COUNT procura no primeiro parâmetro quantas vezes a expressão regular, conforme definida no parâmetro padrão, é encontrada nele (isto é, no primeiro parâmetro).

Exemplo:

A consulta a seguir retorna 1 (um), que é quantidade de vezes que o padrão que corresponde à expressão regular: 'B[[:alpha:]]{4}' (que corresponde a BANCO, pois começa com B e a seguir apresenta mais quatro caracteres alfabéticos) é encontrado no primeiro parâmetro 'TECNOLOGIA EM BANCO DE DADOS':

```
SELECT REGEXP_COUNT('TECNOLOGIA EM BANCO DE DADOS',
'B[[:alpha:]]{4}') AS RESULTADO FROM DUAL;
```

RESULTADO

1

6.6 OUTRAS FUNÇÕES

O Oracle apresenta ainda outras funções de uso frequente. Nesta seção vamos abordar as funções:

- NVL
- NULLIF
- DECODE.

6.6.1 NVL

A função NVL retorna um valor não nulo quando o termo nulo (NULL) for encontrado.

O comando utilizado a seguir retornará 0 (zero) quando o termo nulo for encontrado:

```
SELECT NVL(NOME_DA_COLUNA,0) FROM NOME_DA_TABELA;
```

6.6.2 NULLIF

A função NULLIF compara dois valores e retorna NULL se os dois valores forem iguais, senão retorna o primeiro valor.

```
SELECT NULLIF(NOME_DA_COLUNA_1,NOME_DA_COLUNA_2)
FROM NOME_DA_TABELA;
```

Você também poderá testar a função NULLIF informando os dois valores no momento da consulta. Observe os dois exemplos a seguir:

```
SELECT NULLIF(15,15) FROM DUAL;
```

Retorna (NULL):

```
----
```

NOTA: Visto que o comando anterior retorna NULL, nada poderá ser visualizado no resultado da consulta.

```
SELECT NULLIF(15,10) FROM DUAL;
```

Retorna:

```
--
```

```
15
```

6.6.3 DECODE

A função DECODE implementa a lógica condicional if-then-else. A função apresenta quatro argumentos. Realiza a comparação dos dois primeiros. Se estes forem iguais, apresentará como saída o terceiro argumento. Caso contrário apresentará o quarto argumento como saída. Observe os dois exemplos a seguir.

O comando:

```
SELECT DECODE (15,15, 'IGUAIS', 'DIFERENTES') FROM DUAL;
```

Retorna:

```
-----
IGUAIS
```

O comando:

```
SELECT DECODE (15,10, 'IGUAIS', 'DIFERENTES') FROM DUAL;
```

Retorna:

```
-----
DIFERENTES
```

RESUMO

As funções nativas da SQL, conforme implementadas no Oracle Database, estão agrupadas conforme segue:

- **Funções de grupo:** São também conhecidas como **funções de agregação** e operam em grupos de linhas retornando um valor para todo o grupo. Estas funções são: SUM, AVG, MAX, MIN, COUNT, MEDIAN, STDDEV e VARIANCE.
- **Funções de linha:** Atuam sobre cada uma das linhas resultantes de uma consulta. Para cada linha produzem um valor, conforme os argumentos recebidos. As funções de linha podem ser encadeadas com outras funções. Algumas destas funções são: UPPER, LOWER,

INITCAP, LPAD, SUBSTR, REPLACE, CONCAT, LTRIM, RTRIM, TRIM, CHR, ASCII e TRANSLATE.

- Funções numéricas: As funções numéricas recebem entradas numéricas e retornam valores numéricos, conforme os argumentos recebidos. Algumas funções numéricas são: ABS, CEIL, FLOOR, ROUND, TRUNC, MOD, POWER, SQRT e SIGN.
- Funções de conversão: As funções de conversão convertem um valor de um tipo para outro tipo. Algumas das funções de conversão mais utilizadas são: TO_CHAR, TO_DATE, TO_NUMBER.
- Funções de expressões regulares: Expressões regulares são sequências de caracteres compostos de literais e meta caracteres que descrevem um padrão em um texto. O Oracle apresenta diversas funções que utilizam expressões regulares: REGEXP_LIKE, REGEXP_INSTR, REGEXP_REPLACE, REGEXP_SUBSTR e REGEXP_COUNT.
- Outras funções: Algumas outras funções de uso frequente são: NVL, NULLIF e DECODE.

EXERCÍCIOS

A tabela PEDIDO contém valores de pedidos de vários clientes. Qual das consultas, apresentadas a seguir, apresenta a soma dos pedidos emitidos por cada cliente?

TABELA: PEDIDO

NR_PED	VALOR	ID_CLI
1	1600.00	1001
2	1500.00	1002
3	1200.00	1001
4	1000.00	1002

- `SELECT ID_CLI, SUM(VALOR) FROM PEDIDO GROUP BY NR_PED;`
- `SELECT ID_CLI, SUM(VALOR) FROM PEDIDO GROUP BY ID_CLI;`
- `SELECT ID_CLI, SUM(ID_CLI) FROM PEDIDO GROUP BY VALOR;`
- `SELECT ID_CLI, SUM(NR_PED) FROM PEDIDO GROUP BY VALOR;`
- `SELECT ID_CLI, SUM(NR_PED) FROM PEDIDO GROUP BY ID_CLI;`

Qual será o resultado da consulta apresentada a seguir?

```
SELECT ROUND (SQRT (5) ) FROM DUAL ;
```

- a) A consulta retornará 2, pois primeiro extrairá a raiz quadrada e depois fará o arredondamento para o valor inteiro.
- b) A consulta retornará 2,2360 pois primeiro fará o arredondamento para o valor inteiro e depois extrairá a raiz quadrada.
- c) A consulta apresentará um erro, pois não é possível encadear duas funções: ROUND e SQRT.
- d) A consulta retornará um erro, pois não existe uma tabela denominada DUAL.
- e) A consulta retornará 5, pois não é possível encadear as duas funções: ROUND e SQRT.

CAPÍTULO 7 – SUBQUERIES (SUBCONSULTAS)

Subquery ou subconsulta pode ser definida de forma simples como "uma consulta dentro de outra consulta". Essas subconsultas podem ser criadas a partir da cláusula WHERE, da cláusula FROM ou da cláusula SELECT.

A maioria das subconsultas utilizam a cláusula WHERE. Essas subconsultas também são chamadas de subconsultas aninhadas. O Oracle Database permite até 255 níveis de subconsultas na cláusula WHERE.

7.1 SUBCONSULTAS DE UMA LINHA

As subconsultas de uma única linha retornam 0 (zero) ou 1 (uma) linha para uma instrução SQL externa. O exemplo a seguir apresenta as duas tabelas CLIENTE e PEDIDO:

TABELA: CLIENTE			TABELA: PEDIDO	
CODIGO_CLIENTE	NOME_CLIENTE		NR_PEDIDO	CODIGO_CLIENTE
1001	ANTONIO ALVARES		1	1002
1002	BEATRIZ BARBOSA		2	1001
1003	CLAUDIO CARDOSO		3	1004
1004	DANIELA DAMASIO		4	1003

Utilizaremos uma subconsulta de uma linha para apresentar o nome do cliente (NOME_CLIENTE) que fez o pedido cujo número de pedido (NR_PEDIDO) é igual a 3.

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE =
  (SELECT CODIGO_CLIENTE FROM PEDIDO
   WHERE NR_PEDIDO = 3) ;
```

Entenda as duas etapas da consulta:

1. SELECT CODIGO_CLIENTE FROM PEDIDO WHERE NR_PEDIDO = 3;

A resposta é 1004 e este resultado é utilizado na segunda etapa:

2. SELECT NOME_CLIENTE FROM CLIENTE WHERE CODIGO_CLIENTE = 1004;

A seguir o resultado:

```
-----
DANIELA DAMASIO
```

Você observou que na primeira etapa a consulta retornou apenas uma linha? É daí que vem a denominação subconsulta de uma linha. Utilizamos o operador '=' no exemplo apresentado. Porém outros operadores poderiam ter sido utilizados <, <=, >, >= e <>. No entanto, em qualquer caso, a primeira etapa deveria retornar 0 (zero) ou 1 (uma) linha. Quando houver a possibilidade de retornar mais de uma linha, devemos utilizar as subconsultas de várias linhas, conforme veremos a seguir.

7.2 SUBCONSULTAS DE VÁRIAS LINHAS

As subconsultas de várias linhas podem retornar mais de uma linha na instrução SQL externa. Utilizamos frequentemente os operadores IN, ANY ou ALL para verificar se o valor de uma coluna está contido em uma lista de valores.

O exemplo a seguir toma como base as mesmas tabelas (CLIENTE e PEDIDO) utilizadas no item anterior (CONSULTAS DE UMA LINHA).

Utilizaremos uma subconsulta de várias linhas para apresentar os nomes dos clientes (NOME_CLIENTE) que fizeram pedidos cujos números de pedidos (NR_PEDIDO) estão contidos na lista que apresenta os valores 1, 2 e 4.


```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE IN
  (SELECT CODIGO_CLIENTE FROM PEDIDO
   WHERE NR_PEDIDO IN (1, 2, 4));
```

Entenda as duas etapas da consulta:

1. SELECT CODIGO_CLIENTE FROM PEDIDO WHERE NR_PEDIDO IN (1, 2, 4);

A resposta apresenta os valores 1002, 1001 e 1003, que serão utilizados na segunda etapa:

2. SELECT NOME_CLIENTE FROM CLIENTE WHERE CODIGO_CLIENTE IN 1002, 1001, 1003;

A seguir o resultado:

```
-----
ANTONIO ALVARES
BEATRIZ BARBOSA
CLAUDIO CARDOSO
```

7.2.1 UTILIZANDO ANY EM CONSULTAS DE VÁRIAS LINHAS

Utilizamos o operador ANY para comparar um valor com qualquer valor presente em uma lista. Antes do operador ANY devemos utilizar um dos seguintes operadores =, <, <=, >, >=, <>.

O exemplo a seguir:

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE < ANY
  (SELECT CODIGO_CLIENTE FROM PEDIDO
   WHERE NR_PEDIDO IN (1, 4));
```

Apresenta o seguinte resultado:

ANTONIO ALVARES
BEATRIZ BARBOSA

Entenda as duas etapas da consulta:

1. SELECT CODIGO_CLIENTE FROM PEDIDO WHERE NR_PEDIDO IN (1, 4);

A resposta apresenta os valores 1002 e 1003, que serão utilizados na segunda etapa:

2. SELECT NOME_CLIENTE FROM CLIENTE WHERE CODIGO_CLIENTE < ANY 1002, 1003;

A resposta apresenta os nomes dos clientes (NOME_CLIENTE) cujos códigos (CODIGO_CLIENTE) sejam menores que 1002 ou menores que 1003.

Observe a tabela a seguir para entender melhor:

1001	É menor que 1002. Portanto atende a condição
1002	Não é menor que 1002, mas é menor que 1003. Portanto atende a condição
1003	Não é menor que 1002 e não é menor que 1003. Portanto não atende a condição
1004	Não é menor que 1002 e não é menor que 1003. Portanto não atende a condição

7.2.2 UTILIZANDO ALL EM CONSULTAS DE VÁRIAS LINHAS

Utilizamos o operador ALL para comparar um valor com todos os valores presentes em uma lista. Antes do operador ALL devemos utilizar um dos seguintes operadores =, <, <=, >, >=, <>.

O exemplo a seguir:

```
SELECT NOME_CLIENTE FROM CLIENTE
WHERE CODIGO_CLIENTE < ALL
  (SELECT CODIGO_CLIENTE FROM PEDIDO
   WHERE NR_PEDIDO IN (1, 4));
```

Apresenta o seguinte resultado:

```
-----
ANTONIO ALVARES
```

Entenda as duas etapas da consulta:

1. SELECT CODIGO_CLIENTE FROM PEDIDO WHERE NR_PEDIDO IN (1, 4);

A resposta apresenta os valores 1002 e 1003, que serão utilizados na segunda etapa:

2. SELECT NOME_CLIENTE FROM CLIENTE WHERE CODIGO_CLIENTE < ALL 1002, 1003;

A resposta apresenta os nomes dos clientes (NOME_CLIENTE) cujos códigos (CODIGO_CLIENTE) sejam menores que 1002 e menores que 1003.

Observe a tabela a seguir para entender melhor:

1001	É menor que 1002 e menor que 1003. Portanto atende a condição.
1002	Não é menor que 1002. Portanto não atende a condição.
1003	Não é menor que 1002. Portanto não atende a condição.
1004	Não é menor que 1002. Portanto não atende a condição.

Quando trabalhamos com o operador ALL, se o valor comparado não satisfizer a uma única condição em relação aos valores da lista será automaticamente excluído do resultado.

7.3 SUBCONSULTAS EM UMA CLÁUSULA FROM

Este tipo de subconsulta é também conhecido como VIEW INLINE, pois a subconsulta fornece dados em linha utilizando a cláusula FROM.

O exemplo a seguir toma como base as mesmas tabelas (CLIENTE e PEDIDO) para criar uma view (visão) inline que apresenta o código do cliente (CODIGO_CLIENTE), alias ID_CLI, e a respectiva quantidade de pedido, alias QUANT_PED. Os dados obtidos através da view são então passados para a consulta com a cláusula FROM.

```
SELECT ID_CLI, QUANT_PED FROM
  (SELECT CODIGO_CLIENTE ID_CLI, COUNT(CODIGO_CLIENTE)
   QUANT_PED
   FROM PEDIDO GROUP BY CODIGO_CLIENTE) ;
```

Observe o resultado a seguir:

```
-----
1003  1
1002  1
1001  1
1004  1
```

7.4 EXISTS E NOT EXISTS EM SUBCONSULTAS

O operador EXISTS verifica a existência de linhas retornadas a partir de uma subconsulta. O exemplo a seguir apresenta uma tabela denominada DISCIPLINA na qual encontramos o código das disciplinas (COD_DISC), os seus respectivos nomes (NOME_DISC) e uma coluna com o código da disciplina que é pré-requisito. Observe que nem todas as disciplinas têm pré-requisitos.

TABELA: DISCIPLINA

COD_DISC	NOME_DISC	COD_PRE_REQ
-----	-----	-----
D1001	FÍSICA 1	
D1002	FÍSICA 2	D1001
D1003	QUÍMICA 1	
D1004	QUÍMICA 2	D1003

Utilizamos, a seguir, uma subconsulta com o operador EXISTS para retornar os códigos das disciplinas (COD_DISC) e os nomes das disciplinas (NOME_DISC) que são pré-requisitos para outras disciplinas.

```
SELECT COD_DISC, NOME_DISC FROM DISCIPLINA D1
WHERE EXISTS
  (SELECT COD_PRE_REQ FROM DISCIPLINA D2
   WHERE D2.COD_PRE_REQ = D1.COD_DISC) ;
```

Veja que utilizamos o alias D1 para designar a tabela DISCIPLINA na consulta externa e D2 para designá-la na consulta interna. Observe a seguir o resultado da consulta:

```
D1001  FÍSICA 1
D1003  QUÍMICA 1
```

O operador NOT EXISTS apresenta o resultado oposto ao apresentado acima. Portanto, poderia ser utilizado para apresentar as disciplinas que não são pré-requisitos para outras.

RESUMO

Uma subquery pode ser definida como "uma consulta dentro de outra consulta". As subqueries ou subconsultas podem ser criadas a partir da cláusula WHERE, da cláusula FROM ou da cláusula SELECT.

- Subconsultas de uma única linha: retornam 0 (zero) ou 1 (uma) linha para uma instrução SQL externa.
- Subconsultas de várias linhas: podem retornar mais de uma linha na instrução SQL externa. São utilizados os operadores IN, ANY ou ALL para verificar se o valor de uma coluna está contido em uma lista de valores.

Subconsultas de várias linhas:

- Utilizamos o operador ANY para comparar um valor com **qualquer** valor presente em uma lista. Antes do operador ANY devemos utilizar um dos seguintes operadores =, <, <=, >, >=, <>.
- Utilizamos o operador ALL para comparar um valor com **todos** os valores presentes em uma lista. Antes do operador ALL devemos utilizar um dos seguintes operadores =, <, <=, >, >=, <>.

O operador EXISTS verifica a existência de linhas retornadas a partir de uma subconsulta.

EXERCÍCIOS

Observe as tabelas ALUNO e CURSO:

TABELA: ALUNO

RA	NOME_ALUNO	COD_CURSO
1001	ANTONIO ALVARES	11
1002	BEATRIZ BARBOSA	12
1003	CLAUDIO CARDOSO	11
1004	DANIELA DAMASIO	12

TABELA: CURSO

COD_CURSO	NOME_CURSO
11	WORD
12	EXCEL
13	POWER POINT
14	ACCESS

Qual das seguintes subconsultas retornará nomes dos alunos matriculados no curso EXCEL?

- a) `SELECT NOME_ALUNO FROM ALUNO WHERE COD_CURSO =
(SELECT NOME_CURSO FROM CURSO WHERE COD_CURSO = '12');`
- b) `SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO =
(SELECT NOME_ALUNO FROM ALUNO WHERE COD_CURSO = '12');`
- c) `SELECT RA FROM ALUNO WHERE COD_CURSO =
(SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO = 'EXCEL');`
- d) `SELECT NOME_ALUNO FROM ALUNO WHERE RA =
(SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO = 'EXCEL');`
- e) `SELECT NOME_ALUNO FROM ALUNO WHERE COD_CURSO =
(SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO = 'EXCEL');`

Considerando ainda as tabelas ALUNO e CURSO, qual das seguintes subconsultas retornará nomes dos cursos que não têm nenhum aluno matriculado?

- a) `SELECT NOME_CURSO FROM CURSO WHERE COD_CURSO <>
(SELECT COD_CURSO FROM ALUNO);`
- b) `SELECT NOME_CURSO FROM CURSO WHERE COD_CURSO NOT EXISTS
(SELECT COD_CURSO FROM ALUNO);`
- c) `SELECT NOME_CURSO FROM CURSO WHERE COD_CURSO NOT IN
(SELECT COD_CURSO FROM ALUNO);`
- d) `SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO NOT IN
(SELECT COD_CURSO FROM ALUNO);`
- e) `SELECT COD_CURSO FROM CURSO WHERE NOME_CURSO NOT IN
(SELECT NOME_CURSO FROM ALUNO);`

CAPÍTULO 8 – JOINS (JUNÇÕES)

Junções (joins, em inglês) são consultas SQL usadas para recuperar dados de várias tabelas. O Oracle oferece os seguintes tipos de joins:

- EQUI JOIN
- NATURAL JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- NON EQUI JOIN
- SELF JOIN*
- CROSS JOIN

** A junção do tipo SELF JOIN recupera dados de uma mesma tabela.*

8.1 EQUI JOIN

A junção do tipo EQUI JOIN apresenta todos dos registros onde são encontradas correspondências (igualdade de valores) entre campos comuns de duas ou mais tabelas.

Observe as duas tabelas, CLIENTE e PEDIDO, apresentadas a seguir:

TABELA: CLIENTE

CODIGO_CLIENTE	NOME_CLIENTE	FONE_CLIENTE
1001	ANTONIO ALVARES	5555-1111
1002	BEATRIZ BARBOSA	5555-2222
1003	CLAUDIO CARDOSO	5555-3333

TABELA: PEDIDO

NR_PEDIDO	CODIGO_CLIENTE	DT_PEDIDO
0001	1001	11/07/2016
0002	1002	11/07/2016
0003	1001	12/07/2016
0004		12/07/2016

Podemos usar uma junção do tipo EQUI JOIN para responder a seguinte questão: quais os nomes dos clientes que efetuaram pedidos e os números dos respectivos pedidos emitidos pelos clientes?

Observe a seguir como criar a EQUI JOIN:

```
SELECT C.NOME_CLIENTE, P.NR_PEDIDO
FROM CLIENTE C
INNER JOIN PEDIDO P
ON C.CODIGO_CLIENTE = P.CODIGO_CLIENTE;
```

Observação: C e P foram os alias (apelidos) utilizados respectivamente para as tabelas CLIENTE e PEDIDO.

O resultado desta consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
ANTONIO ALVARES	0001
BEATRIZ BARBOSA	0002
ANTONIO ALVARES	0003

Nota: a sintaxe utilizada nas consultas apresentadas neste e nos próximos tópicos deste capítulo é compatível com as versões mais recentes da linguagem SQL.

8.2 NATURAL JOIN

Quando as tabelas sobre as quais queremos realizar a junção apresentam colunas com o mesmo nome e para que, nesses casos, não seja necessário explicitar o nome das colunas, utilizamos a JUNÇÃO NATURAL. As tabelas CLIENTE e PEDIDO apresentam o mesmo nome de coluna: ID_CLIENTE, portanto podemos fazer uso de uma junção do tipo NATURAL JOIN, conforme segue:

```
SELECT C.NOME_CLIENTE, P.NR_PEDIDO
FROM CLIENTE C
NATURAL INNER JOIN PEDIDO P;
```

O resultado desta consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
-----	-----
ANTONIO ALVARES	0001
BEATRIZ BARBOSA	0002
ANTONIO ALVARES	0003

8.3 JUNÇÃO BASEADA EM NOMES DE COLUNAS

Na JUNÇÃO NATURAL todas as colunas de mesmo nome nas tabelas são utilizadas para realizar a junção. Porém, na JUNÇÃO BASEADA EM NOMES DE COLUNAS apresentada a seguir, somente são utilizadas as colunas listadas na cláusula USING.

```
SELECT C.NOME_CLIENTE, P.NR_PEDIDO
FROM CLIENTE C
INNER JOIN PEDIDO P
USING (CODIGO_CLIENTE);
```

O resultado desta consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
-----	-----
Antonio Alvares	0001
Beatriz Barbosa	0002
Antonio Alvares	0003

8.4 OUTER JOIN

A OUTER JOIN além de mostrar registros cujos campos em comum estejam presentes nas duas tabelas, ainda mostra os que faltam, isto é, aqueles que não têm correspondência entre as duas tabelas.

Há três tipos de OUTER JOINS:

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

8.4.1 LEFT OUTER JOIN

A junção do tipo LEFT OUTER JOIN apresenta todos dos registros onde são encontradas correspondências (igualdade de valores) entre campos comuns das tabelas e também os registros da primeira tabela citada na consulta (considerada "esquerda") para os quais não há correspondência (ou igualdade de valores) em relação à outra tabela.

Exemplo: apresentar os nomes de TODOS os clientes e os números dos pedidos dos respectivos clientes:

```
SELECT C.NOME_CLIENTE, P.NR_PEDIDO
FROM CLIENTE C
LEFT OUTER JOIN PEDIDO P
ON C.CODIGO_CLIENTE = P.CODIGO_CLIENTE;
```

O resultado da consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
-----	-----
ANTONIO ALVARES	0001
BEATRIZ BARBOSA	0002
ANTONIO ALVARES	0003
CLAUDIO CARDOSO	

Observação: a primeira tabela da consulta é considerada esquerda.

O cliente CLAUDIO CARDOSO não realizou nenhum pedido, portanto não há correspondência de seu ID_CLIENTE com nenhuma linha da tabela PEDIDO. No entanto, ele aparece no resultado da consulta, pois pertence à primeira tabela citada na junção (que é considerada "esquerda").

8.4.2 RIGHT OUTER JOIN

A junção do tipo RIGHT OUTER JOIN apresenta todos registros onde são encontradas correspondências (igualdade de valores) entre campos comuns das tabelas e também os registros da segunda tabela citada na consulta (considerada "direita") para os quais não há correspondência (ou igualdade de valores) em relação à outra tabela.

Exemplo: Apresentar os nomes dos clientes e os números de TODOS os pedidos:

```
SELECT C.NOME_CLIENTE, P.NR_PEDIDO
FROM CLIENTE C
RIGHT OUTER JOIN PEDIDO P
ON C.CODIGO_CLIENTE = P.CODIGO_CLIENTE;
```

O resultado da consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
-----	-----
ANTONIO ALVARES	0001
BEATRIZ BARBOSA	0002
ANTONIO ALVARES	0003
	0004

O pedido cujo NR_PEDIDO é igual a "0004" não corresponde a nenhum CODIGO_CLIENTE. No entanto, ele aparece no resultado da consulta, pois pertence à segunda tabela citada na junção (que é considerada "direita").

8.4.3 FULL OUTER JOIN

A junção do tipo FULL OUTER JOIN apresenta todos os registros onde são encontradas correspondências e também aqueles (em ambas tabelas) onde não são encontradas correspondências.

Exemplo: Apresentar os nomes de TODOS os clientes e os números de TODOS os pedidos:

```
SELECT C1.NOME_CURSO, C2.NOME_CURSO "PRÉ-REQUISITO"
FROM CURSO C1
INNER JOIN CURSO C2
ON C1.COD_PRE_REQ = C2.COD_CURSO;
```

O resultado da consulta será o seguinte:

NOME_CLIENTE	NR_PEDIDO
-----	-----
ANTONIO ALVARES	0001
BEATRIZ BARBOSA	0002
ANTONIO ALVARES	0003
CLAUDIO CARDOSO	0004

O cliente CLAUDIO CARDOSO não realizou nenhum pedido e o pedido cujo número é igual a "0004" não está relacionado a nenhum cliente. Porém, em uma junção do tipo FULL OUTER JOIN ambos aparecem no resultado da consulta.

8.5 NON EQUI JOIN

A junção do tipo NON EQUI JOIN reúne campos de tabelas que não apresentam necessariamente valores comuns.

Observe as duas tabelas, PRODUTO e CLASSE, apresentadas a seguir:

TABELA: PRODUTO

COD_PROD	VALOR_PROD
101	75.00
102	120.00
103	30.00

TABELA: CLASSE

FAIXA	VALOR_MIN	VALOR_MAX
A	0.01	50.00
B	50.01	100.00
C	100.01	150.00

Devemos usar uma junção do tipo NON EQUI JOIN para responder a seguinte questão: a qual faixa (A, B ou C) cada produto está relacionado, conforme os valores mínimos e máximos da tabela CLASSE?

```
SELECT P.COD_PROD, P.VALOR_PROD, C.FAIXA
FROM PRODUTO P
INNER JOIN CLASSE C
ON P.VALOR_PROD BETWEEN C.VALOR_MIN AND C.VALOR_MAX;
```

O resultado da consulta será o seguinte:

COD_PROD	VALOR_PROD	FAIXA
101	75.00	B
102	120.00	C
103	30.00	A

Utilizamos o operador relacional BETWEEN no exemplo apresentado. No entanto, podemos utilizar em JUNÇÕES do tipo NON EQUI JOIN qualquer operador relacional, exceto o operador = (igual).

8.6 SELF JOIN

A junção do tipo SELF JOIN apresenta todos registros onde são encontradas correspondências (igualdade de valores) entre campos com valores comuns de uma mesma tabela.

TABELA: CURSO

COD_CURSO	NOME_CURSO	COD_PRE_REQ
11	WINDOWS BÁSICO	
12	WINDOWS AVANÇADO	11
13	WORD BÁSICO	
14	WORD AVANÇADO	13
15	EXCEL BÁSICO	
16	EXCEL AVANÇADO	15

Quando interpretamos a tabela acima observamos, por exemplo, que o curso WINDOWS BÁSICO (COD_CURSO igual a 11) é pré-requisito para o curso WINDOWS avançado. Observe, a seguir, como podemos deixar isso mais claro utilizando uma junção do tipo SELF JOIN:

```
SELECT C1.NOME_CURSO, C2.NOME_CURSO "PRÉ-REQUISITO"
FROM CURSO C1
INNER JOIN CURSO C2
ON C1.COD_PRE_REQ = C2.COD_CURSO;
```

Note que utilizamos dois apelidos diferentes para a tabela CURSO (C1 e C2).

O resultado da consulta será o seguinte:

NOME_CURSO	PRÉ-REQUISITO
WINDOWS AVANÇADO	WINDOWS BÁSICO
WORD AVANÇADO	WORD BÁSICO
EXCEL AVANÇADO	EXCEL BÁSICO

8.7 CROSS JOIN

Uma junção do tipo CROSS JOIN (junção cruzada ou produto cartesiano) apresenta todas as combinações possíveis entre elementos de duas tabelas.

No exemplo apresentado a seguir os times de São Paulo deverão enfrentar os times do Rio de Janeiro. Utilizamos uma junção do tipo CROSS JOIN para gerar todos os tipos de combinações entre os times dos dois estados.

TABELA: TIME_SP

COD_TIME	NOME_TIME
101	CORINTHIANS
102	PALMEIRAS
103	SANTOS
104	SÃO PAULO

TABELA: TIME_RJ

COD_TIME	NOME_TIME
201	BOTAFOGO
202	FLAMENGO
203	FLUMINENSE
204	VASCO

Agora utilizaremos a junção do tipo CROSS JOIN para gerar as combinações entre os times de São Paulo e do Rio de Janeiro:


```
SELECT SP.NOME_TIME "TIME SP",RJ.NOME_TIME "TIME RJ"
FROM TIME_SP SP
CROSS JOIN TIME_RJ RJ;
```

O resultado da consulta será o seguinte:

TIME SP	TIME RJ
-----	-----
CORINTHIANS	BOTAFOGO
CORINTHIANS	FLAMENGO
CORINTHIANS	FLUMINENSE
CORINTHIANS	VASCO
PALMEIRAS	BOTAFOGO
PALMEIRAS	FLAMENGO
PALMEIRAS	FLUMINENSE
PALMEIRAS	VASCO
SANTOS	BOTAFOGO
SANTOS	FLAMENGO
SANTOS	FLUMINENSE
SANTOS	VASCO
SÃO PAULO	BOTAFOGO
SÃO PAULO	FLAMENGO
SÃO PAULO	FLUMINENSE
SÃO PAULO	VASCO

RESUMO

Joins ou junções são consultas SQL usadas para recuperar dados de várias tabelas. O Oracle oferece os seguintes tipos de joins:

- Equi join: Apresenta todos registros onde há igualdade de valores entre campos comuns de duas ou mais tabelas.
- Natural join: Utilizadas quando as tabelas apresentam colunas com o mesmo nome e para que não seja necessário explicitar o nome das colunas.

- Left Outer Join: Apresenta todos registros onde há igualdade de valores entre campos comuns das tabelas e também os registros da primeira tabela citada na consulta (considerada "esquerda") para os quais não há igualdade de valores em relação à outra tabela.
- Right Outer Join: Apresenta todos registros onde há igualdade de valores entre campos comuns das tabelas e também os registros da segunda tabela citada na consulta (considerada "direita") para os quais não há igualdade de valores em relação à outra tabela.
- Full Outer Join: Apresenta todos registros onde são encontradas correspondências e também aqueles, em ambas tabelas, onde não há igualdade de valores.
- Non Equi Join: Reúne campos de tabelas que não apresentam necessariamente valores comuns.
- Self Join: Apresenta todos registros onde há igualdade de valores entre campos com valores comuns de uma mesma tabela.
- Cross Join: Apresenta todas as combinações possíveis entre elementos de duas tabelas.

EXERCÍCIOS

Observe as tabelas ALUNO e CURSO:

TABELA: ALUNO

RA	NOME_ALUNO	COD_CURSO
1001	ANTONIO ALVARES	11
1002	BEATRIZ BARBOSA	12
1003	CLAUDIO CARDOSO	11
1004	DANIELA DAMASIO	12

TABELA: CURSO

COD_CURSO	NOME_CURSO
11	WORD
12	EXCEL
13	POWER POINT
14	ACCESS

Qual das seguintes joins retornará nomes dos alunos e os nomes de seus respectivos cursos?

- a) `SELECT C.NOME_ALUNO, A.NOME_CURSO
FROM ALUNO A INNER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- b) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A INNER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- c) `SELECT C.NOME_ALUNO, A.NOME_CURSO
FROM ALUNO A INNER JOIN CURSO C
ON C.COD_CURSO = A.COD_CURSO;`
- d) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO C OUTER JOIN CURSO A
ON A.COD_CURSO = C.COD_CURSO;`
- e) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A OUTER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`

Considerando ainda as tabelas ALUNO e CURSO, qual das seguintes joins retornará nomes dos alunos, os nomes de seus respectivos cursos e os nomes dos cursos que não têm nenhum aluno matriculado?

- a) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A INNER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- b) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A CROSS JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- c) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A LEFT OUTER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- d) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A RIGHT OUTER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`
- e) `SELECT A.NOME_ALUNO, C.NOME_CURSO
FROM ALUNO A NATURAL INNER JOIN CURSO C
ON A.COD_CURSO = C.COD_CURSO;`

CAPÍTULO 9 – OPERAÇÕES DE CONJUNTO

As operações de conjunto permitem combinar as linhas retornadas de consultas. O número de colunas e os tipos de dados das colunas devem corresponder para que as operações de conjunto sejam bem-sucedidas. Porém, os nomes das colunas não precisam corresponder, isto é, não precisam ser idênticos.

9.1 UNION (UNIÃO)

A operação UNION retorna todas as linhas não duplicadas recuperadas através das consultas.

Imagine, por exemplo, que você tivesse os dados de clientes armazenados em duas tabelas diferentes. Há clientes que aparecem em apenas uma das tabelas e outros que aparecem nas duas, conforme você pode observar a seguir nas tabelas CLIENTE_A e CLIENTE_B:

TABELA: CLIENTE_A

CODIGO_CLIENTE	NOME_CLIENTE	DT_NASC_CLIENTE
1001	ANTONIO ALVARES	28/03/1986
1002	BEATRIZ BARBOSA	15/06/1991
1003	CLAUDIO CARDOSO	10/12/1995
1004	DANIELA DAMASIO	19/05/1993

TABELA: CLIENTE_B

ID_CLI	NOME_CLI	UF_CLI
5011	BEATRIZ BARBOSA	SC
5012	CLAUDIO CARDOSO	PR
5013	FABIANA FONSECA	SP
5014	ERNESTO ESTEVES	MG

A consulta a seguir utiliza a operação de conjunto UNION para retornar todas as ocorrências não duplicadas da coluna NOME_CLIENTE da tabela CLIENTE_A da coluna NOME_CLI da tabela CLIENTE_B:

```
SELECT NOME_CLIENTE FROM CLIENTE_A
UNION
SELECT NOME_CLI FROM CLIENTE_B;
```

Observe:

```
-----
ANTONIO ALVARES
BEATRIZ BARBOSA
CLAUDIO CARDOSO
DANIELA DAMASIO
FABIANA FONSECA
ERNESTO ESTEVES
```

A variação da operação, UNION ALL, retorna todas as linhas recuperadas através das consultas, inclusive as duplicadas.

A consulta:

```
SELECT NOME_CLIENTE FROM CLIENTE_A
UNION ALL
SELECT NOME_CLI FROM CLIENTE_B;
```

Retornará:

```
-----
ANTONIO ALVARES
BEATRIZ BARBOSA
CLAUDIO CARDOSO
DANIELA DAMASIO
BEATRIZ BARBOSA
CLAUDIO CARDOSO
FABIANA FONSECA
ERNESTO ESTEVES
```

9.2 INTERSECT (INTERSEÇÃO)

A operação INTERSECT retorna todas as linhas comuns às duas tabelas recuperadas através das consultas.

A consulta a seguir utiliza a operação de conjunto INTERSECT para retornar todas as ocorrências comuns entre a coluna NOME_CLIENTE da tabela CLIENTE_A e a coluna NOME_CLI da tabela CLIENTE_B:

```
SELECT NOME_CLIENTE FROM CLIENTE_A
INTERSECT
SELECT NOME_CLI FROM CLIENTE_B;
```

Observe:

```
-----
BEATRIZ BARBOSA
CLAUDIO CARDOSO
```

9.3 MINUS (DIFERENÇA)

A operação MINUS retorna todas as linhas da primeira consulta que não foram subtraídas pela segunda consulta. Colocando de forma mais clara: todas as linhas da primeira consulta que não aparecem na segunda.

A consulta a seguir utiliza a operação de conjunto MINUS para retornar todas as ocorrências da coluna NOME_CLIENTE da tabela CLIENTE_A que não aparecem na coluna NOME_CLI da tabela CLIENTE_B:

```
SELECT NOME_CLIENTE FROM CLIENTE_A
MINUS
SELECT NOME_CLI FROM CLIENTE_B;
```

Observe:

```
-----
ANTONIO ALVARES
DANIELA DAMASIO
```

Antes de finalizar este capítulo faremos duas considerações que julgamos importantes.

1. A consulta a seguir é válida:

```
SELECT CODIGO_CLIENTE, NOME_CLIENTE FROM CLIENTE_A
UNION
SELECT ID_CLI, NOME_CLI FROM CLIENTE_B;
```

Porém, retornará as seguintes linhas:

```
-----
1001      ANTONIO ALVARES
1002      BEATRIZ BARBOSA
1003      CLAUDIO CARDOSO
1004      DANIELA DAMASIO
5011      BEATRIZ BARBOSA
5012      CLAUDIO CARDOSO
5013      FABIANA FONSECA
5014      ERNESTO ESTEVES
```

Explicação:

As seguintes linhas não são iguais:

```
1002      BEATRIZ BARBOSA
1003      CLAUDIO CARDOSO
5011      BEATRIZ BARBOSA
5012      CLAUDIO CARDOSO
```

2. A consulta a seguir não é válida:

```
SELECT NOME_CLIENTE, DT_NASC_CLIENTE FROM CLIENTE_A
UNION
SELECT NOME_CLI, UF_CLI FROM CLIENTE_B;
```

Explicação:

O tipo de dado DT_NASC_CLIENTE (DATE) não é correspondente com UF_CLI (CHAR). Portanto não será possível aplicar, neste caso, a operação UNION.

RESUMO

Operações de conjunto permitem combinar as linhas retornadas de consultas. Para que as operações de conjunto sejam bem-sucedidas o número de colunas e os tipos de dados das colunas devem corresponder.

As operações de conjunto suportadas pela linguagem SQL são:

- **UNION:** retorna todas as linhas não duplicadas recuperadas pelas consultas. A variação **UNION ALL** retorna todas as linhas recuperadas pelas consultas, inclusive as duplicadas.
- **INTERSECT:** retorna todas as linhas comuns recuperadas pelas consultas.
- **MINUS:** retorna todas as linhas da primeira consulta que não aparecem na segunda.

EXERCÍCIOS

Observe as tabelas A e B:

-----	-----
TABELA: A	TABELA: B
-----	-----
NOME	NOME
-----	-----
ANTONIO ALVARES	ANTONIO ALVARES
BEATRIZ BARBOSA	CLAUDIO CARDOSO
CLAUDIO CARDOSO	ERNESTO ESTEVES
DANIELA DAMASIO	GERALDO GONZAGA

Qual das consultas, que utilizam operações de conjunto, retornará os seguintes valores?

ANTONIO ALVARES

CLAUDIO CARDOSO

- a) `SELECT NOME FROM A MINUS SELECT NOME FROM B ;`
- b) `SELECT NOME FROM B MINUS SELECT NOME FROM A ;`
- c) `SELECT NOME FROM A UNION SELECT NOME FROM B ;`
- d) `SELECT NOME FROM A UNION ALL SELECT NOME FROM B ;`
- e) `SELECT NOME FROM A INTERSECT SELECT NOME FROM B ;`

Qual das seguintes operações de conjunto retorna todas as linhas da primeira consulta que não aparecem na segunda?

- a) UNION
- b) UNION ALL
- c) MINUS
- d) INTERSECT
- e) Nenhuma operação de conjunto pode realizar isso.

CAPÍTULO 10 – CONSULTAS AVANÇADAS

Neste capítulo apresentaremos alguns recursos importantes da linguagem SQL que permitem que realizemos consultas mais avançadas. Abordaremos os seguintes recursos:

- ROLLUP
- CUBE
- CONSULTAS HIERÁRQUICAS

Você encontrará nos tópicos a seguir alguns exemplos práticos que o ajudarão a utilizar estes recursos da linguagem para resolver algumas situações apresentadas com frequência em muitas organizações.

10.1 ROLLUP

A cláusula ROLLUP, utilizada com GROUP BY, retorna uma linha com o subtotal de cada grupo de linhas e uma linha contendo o total de todos os grupos.

Criaremos, a seguir, uma tabela denominada VEICULO para apresentar de forma prática consultas que utilizam as cláusulas ROLLUP E CUBE:

```
CREATE TABLE VEICULO (
  PLACA CHAR(7),
  MODELO VARCHAR2 (10),
  COR VARCHAR2 (10),
  VALOR NUMBER (7,2),
  CONSTRAINT VEICULO_PK PRIMARY KEY (PLACA) );
```

Vamos inserir as seguintes linhas na tabela VEICULO:

```
INSERT INTO VEICULO VALUES ('XYZ1111', 'FOCUS', 'PRATA', 30000);
```

```

INSERT INTO VEICULO VALUES ('XYZ2222','PALIO','PRA
TA',20000);
INSERT INTO VEICULO VALUES ('XYZ3333','FOCUS','PRA
TA',35000);
INSERT INTO VEICULO VALUES ('XYZ4444','PALIO','PRA
TA',35000);
INSERT INTO VEICULO VALUES ('XYZ5555','FOCUS','PRE
TO',50000);
INSERT INTO VEICULO VALUES ('XYZ6666','PALIO','PRE
TO',40000);
INSERT INTO VEICULO VALUES ('XYZ7777','FOCUS','PRE
TO',60000);
INSERT INTO VEICULO VALUES ('XYZ8888','PALIO','PRE
TO',45000);
COMMIT;

```

Consultando a tabela VEICULO teremos o seguinte resultado:

```
SELECT * FROM VEICULO;
```

Tabela: VEICULO

PLACA	MODELO	COR	VALOR
XYZ1111	FOCUS	PRATA	30000
XYZ2222	PALIO	PRATA	20000
XYZ3333	FOCUS	PRATA	35000
XYZ4444	PALIO	PRATA	25000
XYZ5555	FOCUS	PRETO	50000
XYZ6666	PALIO	PRETO	40000
XYZ7777	FOCUS	PRETO	60000
XYZ8888	PALIO	PRETO	45000

Criaremos agora uma consulta que deverá apresentar uma linha com o subtotal de cada grupo de linhas:

```

SELECT MODELO, COR, SUM(VALOR)
FROM VEICULO
GROUP BY MODELO, COR;

```

O resultado da consulta, utilizando apenas GROUP BY, será o seguinte (acrescentamos comentários em cada linha para que você entenda melhor os valores apresentados):

MODELO	COR	SUM (VALOR)				
-----	-----	-----				
FOCUS	PRATA	65000	--	TOTAL	FOCUS	PRATA
FOCUS	PRETO	110000	--	TOTAL	FOCUS	PRETO
FOCUS		175000	--	TOTAL	TODOS	FOCUS
PALIO	PRATA	55000	--	TOTAL	PALIO	PRATA
PALIO	PRETO	85000	--	TOTAL	PALIO	PRETO
PALIO		140000	--	TOTAL	TODOS	PALIO

Observe agora como utilizamos a cláusula ROLLUP para retornar uma linha com o subtotal de cada grupo e uma linha com o total de todos os grupos.

```
SELECT MODELO, COR, SUM (VALOR)
FROM VEICULO
GROUP BY ROLLUP (MODELO, COR) ;
```

O resultado da consulta, utilizando a cláusula ROLLUP, será o seguinte (acrescentamos comentários em cada linha para que você entenda melhor os valores apresentados):

MODELO	COR	SUM (VALOR)				
-----	-----	-----				
FOCUS	PRATA	65000	--	TOTAL	FOCUS	PRATA
FOCUS	PRETO	110000	--	TOTAL	FOCUS	PRETO
FOCUS		175000	--	TOTAL	TODOS	FOCUS
PALIO	PRATA	55000	--	TOTAL	PALIO	PRATA
PALIO	PRETO	85000	--	TOTAL	PALIO	PRETO
PALIO		140000	--	TOTAL	TODOS	PALIO
		315000	--	TOTAL	GERAL	

10.1.1 ROLLUP – USANDO GROUPING()

A função GROUPING () acrescentará uma coluna às consultas que retornarão apenas dois valores: 0 (zero) quando o valor da coluna não é nulo e 1 (um) quando o valor da coluna é nulo. Observe a consulta a seguir:

```
SELECT GROUPING(MODELO) , MODELO, COR, SUM (VALOR)
FROM VEICULO
GROUP BY ROLLUP (MODELO, COR)
ORDER BY MODELO;
```

O resultado da consulta será o apresentado a seguir:

GROUPING (MODELO)	GROUPING (COR)	MODELO	COR	SUM (VALOR)
0	0	0 FOCUS	PRATA	65000
0	0	0 FOCUS	PRETO	110000
0	1	1 FOCUS		175000
0	0	0 PALIO	PRATA	55000
0	0	0 PALIO	PRETO	85000
0	1	1 PALIO		140000
1	1			315000

Observe que a função GROUPING() retornou 0 (zero) para as linhas que contêm valores para MODELO e 1 (um) para as linhas em que MODELO é nulo.

Podemos melhorar o resultado de nossas consultas utilizando a expressão CASE para converter o valor 1 (um) em algo mais significativo. Observe a consulta a seguir

```
SELECT
CASE GROUPING (MODELO)
  WHEN 1 THEN 'TODOS MODELOS'
  ELSE MODELO
END AS MOD,
```

```

CASE GROUPING (COR)
  WHEN 1 THEN 'TODAS CORES'
  ELSE COR
  END AS COR_V,
SUM (VALOR)
FROM VEICULO
GROUP BY ROLLUP (MODELO, COR)
ORDER BY MODELO, COR;

```

O resultado à consulta será o seguinte:

MOD	COR_V	SUM (VALOR)
-----	-----	-----
FOCUS	PRATA	65000
FOCUS	PRETO	110000
FOCUS	TODAS CORES	175000
PALIO	PRATA	55000
PALIO	PRETO	85000
PALIO	TODAS CORES	140000
TODOS MODELOS	TODAS CORES	315000

Conforme você observou, quando a função GROUPING retornou 1 para a coluna COR, substituímos por 'TODAS CORES' e quando retornou 1 para a coluna MODELO, substituímos por 'TODOS MODELOS'.

10.2 CUBE

A cláusula CUBE, utilizada com GROUP BY, retorna linhas com o subtotal para todas as combinações de colunas e uma linha contendo o total geral.

Utilizaremos, a seguir, a tabela VEICULO criada anteriormente para apresentar de forma prática consultas que utilizam a cláusula CUBE:

```

SELECT MODELO, COR, SUM (VALOR)
FROM VEICULO
GROUP BY CUBE (MODELO, COR) ;

```

O resultado da consulta, utilizando a cláusula CUBE, será o seguinte (acrescentamos comentários em cada linha para que você entenda melhor os valores apresentados):

MODELO	COR	SUM (VALOR)
		315000 -- TOTAL GERAL
	PRATA	120000 -- TOTAL TODOS PRATA
	PRETO	195000 -- TOTAL TODOS PRETO
FOCUS		175000 -- TOTAL FOCUS
FOCUS	PRATA	65000 -- TOTAL FOCUS PRATA
FOCUS	PRETO	110000 -- TOTAL FOCUS PRETO
PALIO		140000 -- TOTAL PALIO
PALIO	PRATA	55000 -- TOTAL PALIO PRATA
PALIO	PRETO	85000 -- TOTAL PALIO PRETO

10.2.1 CUBE USANDO GROUPING()

A função GROUPING (), conforme mencionamos no tópico anterior, acrescentará uma coluna às consultas realizadas que retornarão os valores: 0 (zero) quando o valor da coluna não é nulo e 1 (um) quando o valor da coluna é nulo.

Podemos melhorar, como observado no tópico anterior, o resultado das consultas utilizando a expressão CASE para converter o valor 1 (um) em algo mais significativo. Observe a consulta a seguir:

```
SELECT
CASE GROUPING (MODELO)
  WHEN 1 THEN 'TODOS MODELOS'
  ELSE MODELO
END AS MOD,
CASE GROUPING (COR)
  WHEN 1 THEN 'TODAS CORES'
  ELSE COR
END AS COR_V,
SUM (VALOR)
```

```

FROM VEICULO
GROUP BY CUBE (MODELO, COR)
ORDER BY MODELO, COR;

```

O resultado da consulta pode ser observado a seguir:

MOD	COR_V	SUM (VALOR)
-----	-----	-----
FOCUS	PRATA	65000
FOCUS	PRETO	110000
FOCUS	TODAS CORES	175000
PALIO	PRATA	55000
PALIO	PRETO	85000
PALIO	TODAS CORES	140000
TODOS MODELOS	PRATA	120000
TODOS MODELOS	PRETO	195000
TODOS MODELOS	TODAS CORES	315000

10.3 CONSULTAS HIERÁRQUICAS

É comum encontrarmos dados organizados em uma estrutura hierárquica, por exemplo, os funcionários em uma empresa: o supervisor está subordinado a um gerente, que está subordinado a um diretor, que está subordinado ao CEO.

Podemos registrar esta hierarquia em um banco de dados relacional criando uma tabela conforme apresentado a seguir:

```

CREATE TABLE FUNCIONARIO (
ID_FUNCIONARIO    NUMBER (4) ,
ID_GERENTE        NUMBER (4) ,
NOME_FUNCIONARIO  VARCHAR2 (30) ,
CARGO              VARCHAR2 (15) ,
CONSTRAINT FUNCIONARIO_PK PRIMARY KEY (ID_FUNCIONARIO)) ;

```

Vamos inserir, por exemplo, os seguintes dados na tabela criada:


```

INSERT INTO FUNCIONARIO VALUES
(1001, '', 'ANTONIO ALVARES', 'CEO');
INSERT INTO FUNCIONARIO VALUES
(1002, 1001, 'BEATRIZ BARBOSA', 'DIRETOR');
INSERT INTO FUNCIONARIO VALUES
(1003, 1002, 'CLAUDIO CARDOSO', 'GERENTE');
INSERT INTO FUNCIONARIO VALUES
(1004, 1001, 'DANIELA DAMASIO', 'DIRETOR');
INSERT INTO FUNCIONARIO VALUES
(1005, 1004, 'ERNESTO ESTEVES', 'GERENTE');
INSERT INTO FUNCIONARIO VALUES
(1006, 1004, 'FABIANA FONSECA', 'GERENTE');
INSERT INTO FUNCIONARIO VALUES
(1007, 1005, 'GERALDO GONZAGA', 'SUPERVISOR');
INSERT INTO FUNCIONARIO VALUES
(1008, 1006, 'HORACIA HUNGARO', 'SUPERVISOR');
INSERT INTO FUNCIONARIO VALUES
(1009, 1001, 'IGNACIO INDIANO', 'DIRETOR');
INSERT INTO FUNCIONARIO VALUES
(1010, 1009, 'JESUINA JARDINA', 'GERENTE');

```

Realizando a consulta seguinte na tabela FUNCIONÁRIO

```
SELECT * FROM FUNCIONARIO;
```

Obteremos o seguinte resultado:

ID_FUNCIONARIO	ID_GERENTE	NOME_FUNCIONARIO	CARGO
1001		ANTONIO ALVARES	CEO
1002	1001	BEATRIZ BARBOSA	DIRETOR
1003	1002	CLAUDIO CARDOSO	GERENTE
1004	1001	DANIELA DAMASIO	DIRETOR
1005	1004	ERNESTO ESTEVES	GERENTE
1006	1004	FABIANA FONSECA	GERENTE

1007	1005	GERALDO GONZAGA	SUPERVISOR
1008	1006	HORACIA HUNGARO	SUPERVISOR
1009	1001	IGNACIO INDIANO	DIRETOR
1010	1009	JESUINA JARDINA	GERENTE

Observe, no entanto, que não é fácil identificar o relacionamento dos funcionários a partir dos dados apresentados na consulta que acabamos de realizar. A figura a seguir apresenta de forma gráfica as relações dos funcionários.

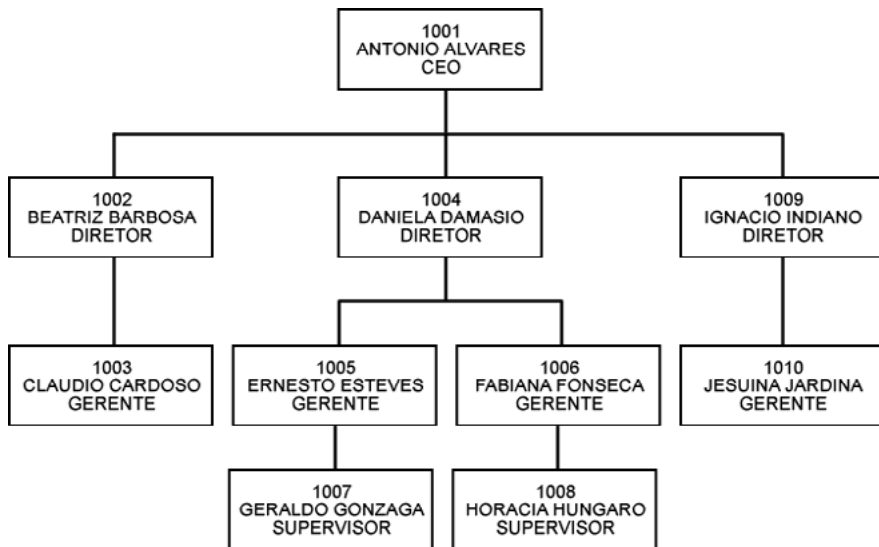


Figura 15: Relacionamentos conforme dados da tabela FUNCIONARIO

Fonte: Autor.

Os elementos da figura, também denominado nós, formam uma árvore. Os nós da árvore recebem as seguintes denominações, conforme a posição que ocupam:

- Raiz: Nó que está no topo da árvore. (ANTONIO ALVARES)
- Pai: Nó que tem debaixo dele um ou mais nós. (Por exemplo, ANTONIO ALVARES é 'pai' dos seguintes nós: BEATRIZ BARBOSA, DANIELA DAMASIO e IGNACIO INDIANO).

- **Filho:** Nó que tem acima dele um outro nó. (Por exemplo, o nó CLAUDIO CARDOSO é nó 'filho' de BEATRIZ BARBOSA).
- **Folha:** Nó que não tem debaixo dele outros nós. (Por exemplo: GERALDO GONZAGA e HORACIA HUNGARO são nós que recebem a denominação 'folha').

Consultas hierárquicas utilizam cláusulas como START WITH e CONNECT BY para percorrer os nós de uma estrutura do tipo árvore. Observe o exemplo apresentado a seguir:

```
SELECT LEVEL, ID_FUNCIONARIO, ID_GERENTE, NOME_FUNCIONARIO
FROM FUNCIONARIO
START WITH ID_FUNCIONARIO = 1001
CONNECT BY PRIOR ID_FUNCIONARIO = ID_GERENTE
ORDER BY LEVEL;
```

- **LEVEL:** Pseudocoluna que indica o nível do nó na árvore: 1 para o nó 'raiz' e assim por diante.
- **START WITH:** Nesta cláusula devemos informar onde a consulta hierárquica deverá ser iniciada. No exemplo apresentado a consulta foi iniciada onde ID_FUCNIONARIO = 1001.
- **CONNECT BY:** Nesta cláusula devemos especificar a relação entre os nós 'pai' e 'filho'. No exemplo apresentado o ID_GERENTE do 'filho' aponta para o ID_FUNCIONARIO do 'pai'.

Observe a seguir o resultado da consulta:

LEVEL	ID_FUNCIONARIO	ID_GERENTE	NOME_FUNCIONARIO
1	1001		ANTONIO ALVARES
2	1002	1001	BEATRIZ BARBOSA
2	1004	1001	DANIELA DAMASIO
2	1009	1001	IGNACIO INDIANO
3	1003	1002	CLAUDIO CARDOSO

3	1005	1004	ERNESTO ESTEVES
3	1006	1004	FABIANA FONSECA
3	1010	1009	JESUINA JARDINA
4	1007	1005	GERALDO GONZAGA
4	1008	1006	HORACIA HUNGARO

Não é necessário que a consulta inicie no nó 'raiz' para percorrer a árvore, você poderá percorrê-la a partir de qualquer nó com a cláusula **START WITH**. Na consulta a seguir começamos a percorrer a árvore a partir do nó DANIELA DAMASIO.

```
SELECT LEVEL, ID_FUNCIONARIO, ID_GERENTE, NOME_FUNCIONARIO
FROM FUNCIONARIO
START WITH NOME_FUNCIONARIO = 'DANIELA DAMASIO'
CONNECT BY PRIOR ID_FUNCIONARIO = ID_GERENTE
ORDER BY LEVEL;
```

O resultado da consulta será o seguinte:

LEVEL	ID_FUNCIONARIO	ID_GERENTE	NOME_FUNCIONARIO
-----	-----	-----	-----
1	1004	1001	DANIELA DAMASIO
2	1005	1004	ERNESTO ESTEVES
2	1006	1004	FABIANA FONSECA
3	1007	1005	GERALDO GONZAGA
3	1008	1006	HORACIA HUNGARO

RESUMO

Neste capítulo abordamos os seguintes recursos da linguagem SQL que permitem a realização de consultas avançadas:

- **ROLLUP**: Quando utilizada com **GROUP BY**, retorna uma linha com o subtotal de cada grupo de linhas e uma linha contendo o total de todos os grupos.

- CUBE: Quando utilizada com GROUP BY, retorna linhas com o subtotal para todas as combinações de colunas e uma linha contendo o total geral.
- CONSULTAS HIERÁRQUICAS: Apresentam dados organizados em uma estrutura hierárquica. Exemplo: hierarquia em uma organização.

EXERCÍCIOS

Consultando uma tabela denominada PEDIDO obtivemos o seguinte resultado:

```
SELECT * FROM PEDIDO;
```

NR_PED	VENDEDOR	GERENTE	VALOR
1	DANIELA	ANTONIO	2000
2	ERNESTO	BEATRIZ	3000
3	FABIANA	CLAUDIO	2500
4	DANIELA	ANTONIO	4500
5	ERNESTO	BEATRIZ	4000
6	FABIANA	CLAUDIO	3500

Qual das seguintes consulta retornará o resultado apresentado a seguir?

VENDEDOR	GERENTE	SUM (VALOR)
ERNESTO	BEATRIZ	7000
FABIANA	CLAUDIO	6000
DANIELA	ANTONIO	4500

- `SELECT VENDEDOR, GERENTE, VALOR FROM PEDIDO
GROUP BY VENDEDOR, GERENTE;`
- `SELECT VENDEDOR, GERENTE, SUM FROM PEDIDO
GROUP BY VENDEDOR, GERENTE;`
- `SELECT VENDEDOR, GERENTE, SUM(VALOR)
FROM PEDIDO;`

- d) `SELECT VENDEDOR, GERENTE, SUM(VALOR) FROM PEDIDO
WHERE VENDEDOR = GERENTE;`
- e) `SELECT VENDEDOR, GERENTE, SUM(VALOR) FROM PEDIDO
GROUP BY VENDEDOR, GERENTE;`

Considerando ainda a tabela PEDIDO, qual das consultas retornará o seguinte resultado?

VENDEDOR	GERENTE	SUM (VALOR)
-----	-----	-----
		17500
	ANTONIO	4500
	BEATRIZ	7000
	CLAUDIO	6000
DANIELA		4500
DANIELA	ANTONIO	4500
ERNESTO		7000
ERNESTO	BEATRIZ	7000
FABIANA		6000
FABIANA	CLAUDIO	6000

- a) `SELECT VENDEDOR, GERENTE, VALOR FROM PEDIDO
GROUP BY CUBE (VENDEDOR, GERENTE);`
- b) `SELECT VENDEDOR, GERENTE, SUM(VALOR) FROM PEDIDO
GROUP BY CUBE (VENDEDOR, GERENTE);`
- c) `SELECT VENDEDOR, GERENTE, SUM FROM PEDIDO
GROUP BY CUBE (VENDEDOR, GERENTE);`
- d) `SELECT VENDEDOR, GERENTE, SUM(VALOR) FROM PEDIDO
GROUP BY ROLLUP (VENDEDOR, GERENTE);`
- e) `SELECT VENDEDOR, GERENTE, VALOR FROM PEDIDO
GROUP BY ROLLUP (VENDEDOR, GERENTE);`

CAPÍTULO 11 – DTL – DATA TRANSACT LANGUAGE

A Linguagem de Transação de Dados (Data Transact Language, em inglês) permite o controle de transações no banco de dados. Mas, o que são transações no contexto dos bancos de dados? Recebe o nome de transação um conjunto inseparável de instruções SQL cujos resultados devem se tornar permanentes no banco de dados. Caso isso não seja possível, todas as instruções serão desfeitas. Este controle é fundamental para que os dados permaneçam consistentes no banco.

Imagine a seguinte situação: você está transferindo R\$ 1.000,00 da conta corrente A para a conta poupança B. Repare que isso implica em dois "upgrades".

O primeiro upgrade será algo como:

```
UPGRADE TAB_1 SET SALDO = SALDO - 1000 WHERE CONTA = 'A' ;
```

E o segundo upgrade algo como:

```
UPGRADE TAB_2 SET SALDO = SALDO + 1000 WHERE CONTA = 'B' ;
```

Se apenas o primeiro upgrade fosse realizado e, em seguida, por um problema qualquer, o segundo upgrade fosse descartado, imagine a confusão. Onde estariam os R\$ 1.000,00? Sumiram da conta A, mas não apareceram na conta B!

Este simples exemplo permite que compreendamos como é importante o controle de transações.

O subgrupo da SQL denominado Data Transaction Language compreende três comandos principais:

- COMMIT
- ROLLBACK
- SAVEPOINT

Vamos apresentar a seguir cada um deles e explicar como podemos utilizá-los para controlar as transações em um banco de dados relacional.

11.1 COMMIT

Utilizamos a instrução COMMIT para registrar permanentemente no banco de dados as instruções SQL de uma transação.

Observe como utilizamos a instrução COMMIT para confirmar as instruções SQL conforme a transação a seguir:

```
UPGRADE TAB_1 SET SALDO = SALDO - 1000 WHERE CONTA = 'A';  
UPGRADE TAB_2 SET SALDO = SALDO + 1000 WHERE CONTA = 'B';  
COMMIT;
```

NOTA: No SQL*Plus ocorre um COMMIT implícito nas seguintes situações:

- Quando o programa é fechado normalmente com o comando EXIT.
- Quando uma ou mais instruções DML (INSERT, UPDATE ou DELETE) é sucedida por um comando DDL (CREATE, ALTER ou DROP) ou por um comando DCL (GRANT).

11.2 ROLLBACK

Utilizamos a instrução ROLLBACK para descartar do banco de dados as instruções SQL de uma transação.

Observe como utilizamos a instrução ROLLBACK para descartar as instruções SQL apresentadas a seguir:

```
UPGRADE TAB_1 SET SALDO = SALDO - 1000 WHERE CONTA = 'A';  
UPGRADE TAB_2 SET SALDO = SALDO + 1000 WHERE CONTA = 'B';  
ROLLBACK;
```

As duas instruções acima serão descartadas do banco de dados.

NOTA: No SQL*Plus ocorre um ROLLBACK implícito na seguinte situação:

- Quando o programa é fechado de forma anormal (por exemplo, se o usuário clica na opção de fechar a janela do SQL*Plus, localizada no canto superior direito).

11.3 SAVEPOINTS

Savepoints (ou pontos de salvamento) são utilizados para decompor transações muito longas. Portanto, caso ocorram erros após um savepoint, a transação não precisará ser revertida até o seu início.

Observe a tabela apresentada a seguir:

```

-----
TABELA: PRODUTO
-----
COD_PROD NOME_PROD VALOR_PROD
-----
P1001     PRODUTO 1      80.00
P1002     PRODUTO 2      40.00
P1003     PRODUTO 3      70.00
P1004     PRODUTO 4      30.00

```

Imagine que você tenha que aumentar os valores dos produtos acima conforme segue:

- Produtos com valor até 50.00 devem ter um aumento de 20%
- Produtos com valor superior a 50.00 devem ter um aumento de 15%

Uma solução possível seria a apresentada a seguir:

```

UPDATE PRODUTO SET VALOR_PROD = VALOR_PROD * 1.20
WHERE VALOR_PROD <= 50.00;

```

```
UPDATE PRODUTO SET VALOR_PROD = VALOR_PROD * 1.15
WHERE VALOR_PROD > 50.00;
COMMIT;
```

Porém, o que aconteceria se você errasse, por exemplo, parte da transação, conforme apresentado a seguir:

```
UPDATE PRODUTO SET VALOR_PROD = VALOR_PROD * 1.10
WHERE VALOR_PROD > 50.00;
```

Os produtos com valores superiores teriam um aumento de 10%, o que não está de acordo com o que foi solicitado. Neste caso, a transação apresentada anteriormente deveria ser revertida na sua totalidade.

Observe agora como poderíamos utilizar um ponto de salvamento (SAVEPOINT) para evitar isso:

```
UPDATE PRODUTO SET VALOR_PROD = VALOR_PROD * 1.20
WHERE VALOR_PROD <= 50.00;
SAVEPOINT PONTO_1
UPDATE PRODUTO SET VALOR_PROD = VALOR_PROD * 1.15
WHERE VALOR_PROD > 50.00;
```

Neste exemplo criamos um ponto de salvamento após a atualização de valores dos produtos com valores até 50.00.

Portanto, se ocorresse algum problema com a parte seguinte da transação, esta poderia ser parcialmente revertida, conforme segue:

```
ROLLBACK TO SAVEPOINT PONTO_1;
```

A partir deste momento, apenas a parte final da transação precisaria ser restabelecida.

RESUMO

As três instruções que compõem o subgrupo denominado DTL (Data Transact Language) são as seguintes:

- COMMIT: Confirma as instruções DML que fazem parte da transação.
- ROLLBACK: Descarta as instruções DML que fariam parte de uma transação.
- SAVEPOINT: Insere um "ponto de salvamento" para que não seja necessário descartar todos os comandos de uma transação. Esta poderá ser revertida até um determinado "ponto de salvamento".

EXERCÍCIOS

Observe as instruções SQL a seguir e assinale a alternativa correta:

```
CREATE TABLE ALUNO (RA NUMBER(4), NOME VARCHAR2(30));
INSERT INTO ALUNO VALUES (1001, 'FULANO DE TAL');
SAVEPOINT PONTO_1;
UPDATE ALUNO SET NOME = 'SICRANO DE TAL' WHERE RA = 1001;
SAVEPOINT PONTO_2;
UPDATE ALUNO SET NOME = 'BELTRANO DE TAL' WHERE RA = 1001;
ROLLBACK;
SELECT NOME FROM ALUNO;
```

- Será apresentado: FULANO DE TAL.
- Será apresentado: SICRANO DE TAL.
- Será apresentado: BELTRANO DE TAL.
- Será apresentado: 'Não há linhas selecionadas'.
- Será apresentada uma mensagem de erro.

Observe as instruções SQL a seguir e assinale a alternativa correta:

```
CREATE TABLE CLIENTE (ID INT, NOME VARCHAR2(30));  
INSERT INTO CLIENTE VALUES (1, 'ANTONIO ALVARES');  
COMMIT;  
UPDATE CLIENTE SET NOME = 'BEATRIZ BARBOSA' WHERE ID = 1;  
CREATE TABLE TESTE (COLUNA_1 INT);  
UPDATE CLIENTE SET NOME = 'CLAUDIO CARDOSO' WHERE ID = 1;  
ROLLBACK;  
SELECT NOME FROM CLIENTE;
```

- a) Será apresentado: ANTONIO ALVARES.
- b) Será apresentado: BEATRIZ BARBOSA.
- c) Será apresentado: CLAUDIO CARDOSO.
- d) Será apresentado: 'Não há linhas selecionadas'.
- e) Será apresentada uma mensagem de erro.

CAPÍTULO 12 – DCL – DATA CONTROL LANGUAGE

A Linguagem de Controle de Dados (Data Control Language, em inglês) é um subconjunto da linguagem SQL, utilizado para controlar o acesso aos dados em um banco.

Os comandos do subconjunto são os seguintes:

- **GRANT:** Concede privilégios de sistema e de objetos a usuários do banco de dados.
- **REVOKE:** Revoga ou retira os privilégios do usuário do banco de dados.

12.1 PRIVILÉGIOS DE SISTEMA

Os privilégios de sistema permitem ao usuário realizar certas operações no banco de dados. Observe a seguir alguns dos privilégios de sistema mais utilizados:

- **CREATE SESSION:** Permite ao usuário conectar-se em um banco de dados.
- **CREATE TABLE:** Permite ao usuário criar tabelas em seu próprio schema.
- **CREATE ANY TABLE:** Permite ao usuário criar tabelas em qualquer schema do banco de dados.
- **DROP TABLE:** Permite ao usuário excluir tabelas de seu próprio schema.
- **DROP ANY TABLE:** Permite ao usuário excluir tabelas em qualquer schema do banco de dados.
- **CREATE VIEW:** Permite ao usuário criar visões (views) em seu próprio schema.
- **CREATE USER:** Permite ao usuário criar novos usuários no banco de dados.
- **DROP USER:** Permite ao usuário eliminar outros usuários do banco de dados.

12.2 PRIVILÉGIOS DE OBJETO

Os privilégios de objeto permitem que o usuário realize operações em objetos específicos (tabelas, visões, índices etc.) do banco de dados. Os principais privilégios de objetos são os seguintes:

- SELECT: Permite que o usuário realize consultas em objetos do banco de dados.
- INSERT: Permite que o usuário realize inserções de dados nos objetos.
- UPDATE: Permite que o usuário realize alterações nos dados dos objetos.
- DELETE: Permite que o usuário realize exclusões de dados nos objetos.
- EXECUTE: Permite que o usuário execute um procedimento armazenado (stored procedure).

12.3 GRANT

A instrução GRANT concede um privilégio de sistema ou de objeto a um usuário do banco de dados. A seguir, vamos criar um novo usuário no banco de dados e, em seguida, conceder-lhe alguns privilégios de sistema e de objeto.

NOTA: É preciso conectar-se ao banco através de um usuário com privilégios suficientes para criar um novo usuário. Portanto, vamos nos conectar como usuário system. No exemplo a seguir utilizaremos a senha padrão (manager) do usuário system em bancos de teste. (Pode ser que você tenha escolhido uma senha diferente para o usuário system quando realizou a instalação do Oracle.)

```
CONNECT system/manager
```

A seguir vamos criar um usuário de teste chamado fulano com a senha abc123:

```
CREATE USER fulano IDENTIFIED BY abc123
DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
```

Agora vamos conceder ao usuário fulano o privilégio de sistema CREATE SESSION para que ele possa conectar-se ao banco de dados:

```
GRANT CREATE SESSION TO fulano;
```

Você poderá conectar o usuário fulano ao banco de dados através do seguinte comando:

```
CONNECT fulano/abc123
```

Porém, o usuário que acabamos de criar não tem nenhum outro privilégio no banco. Caso você tente criar alguma tabela utilizando este usuário, observará que o banco emitirá uma mensagem de erro. Vamos, portanto, conceder alguns outros privilégios ao usuário fulano. Porém, não esqueça de conectar-se novamente como usuário system, conforme segue:

```
CONNECT system/manager
```

A instrução a seguir concede um privilégio de sistema e permite que o usuário fulano crie tabelas em seu próprio schema:

```
GRANT CREATE TABLE TO fulano;
```

A próxima instrução concede um privilégio de objeto e permite que o usuário fulano faça consultas na tabela emp que pertence ao usuário scott:

```
GRANT SELECT ON scott.emp TO fulano;
```

Você poderá realizar alguns testes para confirmar os privilégios concedidos. No entanto, não esqueça de conectar-se como usuário fulano para realizar os testes.

12.4 REVOKE

A instrução REVOKE retira um privilégio de sistema ou de objeto de um usuário do banco de dados.

Você poderá, por exemplo, retirar os privilégios anteriormente concedidos ao usuário fulano. Observe como fazer isso:

Retirar o privilégio do usuário fulano realizar consulta na tabela emp do usuário scott:

```
REVOKE SELECT ON scott.emp FROM fulano;
```

Retirar o privilégio do usuário fulano criar novas tabelas em seu schema:

```
REVOKE CREATE TABLE FROM fulano;
```

Poderá também eliminar o usuário fulano e seus objetos do banco de dados:

```
DROP USER fulano CASCADE;
```

NOTA: A opção CASCADE elimina também os objetos do usuário do banco de dados.

RESUMO

O subconjunto da SQL denominado DCL (Data Control Language) é composto basicamente pelos comandos:

- GRANT: Concede um privilégio de sistema ou de objeto a um usuário do banco de dados.
- REVOKE: Revoga ou retira os privilégios de sistema ou de objeto do usuário do banco de dados.

Alguns dos principais privilégios de sistema são:

- CREATE SESSION: Permite conectar-se em um banco de dados.
- CREATE TABLE: Permite criar tabelas em seu próprio schema.
- DROP TABLE: Permite excluir tabelas de seu próprio schema.
- CREATE VIEW: Permite criar visões (views) em seu próprio schema.
- CREATE USER: Permite criar novos usuários no banco de dados.
- DROP USER: Permite eliminar usuários do banco de dados.

Alguns dos principais privilégios de objeto são:

- SELECT: Permite realizar consultas em objetos do banco de dados.
- INSERT: Permite realizar inserções de dados nos objetos.
- UPDATE: Permite realizar alterações nos dados dos objetos.
- DELETE: Permite realizar exclusões de dados nos objetos.
- EXECUTE: Permite executar um procedimento armazenado (stored procedure).

EXERCÍCIOS

Qual dos seguintes comandos concederá ao usuário FULANO o privilégio de criar tabelas em qualquer SCHEMA do banco de dados?

- `GRANT CREATE TABLE TO FULANO;`
- `GRANT CREATE ALL TABLE TO FULANO;`
- `GRANT CREATE PUBLIC TABLE TO FULANO;`
- `GRANT CREATE ANY TABLE TO FULANO;`
- `GRANT CREATE TABLE TO FULANO WITH PUBLIC OPTION;`

O usuário FULANO é o 'proprietário' de uma tabela denominada CLIENTE. Qual dos seguintes comando deverá ser executado por

FULANO para que o usuário BELTRANO possa realizar atualizações nos dados da tabela CLIENTE?

- a) GRANT UPDATE TO BELTRANO ON CLIENTE;
- b) GRANT UPDATE ON CLIENTE TO BELTRANO;
- c) GRANT UPDATE ON BELTRANO.CLIENTE TO FULANO;
- d) GRANT UPDATE ON FULANO.CLIENTE TO BELTRANO;
- e) GRANT UPDATE ON FULANO TO BELTRANO.CLIENTE;

CAPÍTULO 13 – INDEXES (ÍNDICES)

Índices (Indexes, em inglês) permitem acesso mais rápido a determinadas linhas de uma tabela quando um pequeno subconjunto de linhas for selecionado. Portanto, os índices armazenam os valores das colunas que estão sendo indexadas juntamente com o RowID físico da respectiva linha, exceto no caso das tabelas organizadas por índice, que utilizam a Primary Key como um RowID lógico.

Nota: O RowID físico utiliza um sistema numérico de base 64 para representar o endereço exclusivo de uma linha da tabela.

O Oracle dispõe de vários tipos de índices, específicos para cada tipo de tabela, método de acesso ou ambiente de aplicação. Neste capítulo vamos abordar dois deles:

- Índices únicos (exclusivos)
- Índices não únicos (não exclusivos)

13.1 ÍNDICES ÚNICOS (EXCLUSIVOS)

Os índices exclusivos são criados automaticamente quando você definir uma restrição (constraint) PRIMARY KEY ou UNIQUE. Este tipo de índice garante que não existirão valores duplicados na coluna ou nas colunas indexadas.

Nota: É possível criar manualmente um índice exclusivo, no entanto, recomenda-se a criação de uma restrição (PRIMARY KEY ou UNIQUE) que crie implicitamente um índice exclusivo.

O exemplo a seguir apresenta a criação de dois índices exclusivos utilizando-se as restrições PRIMARY KEY (na coluna CODIGO) e UNIQUE (na coluna CPF):

```
CREATE TABLE CLIENTE (
  CODIGO NUMBER(4) ,
  NOME VARCHAR2(40) ,
  CPF CHAR(11) ,
```

```
CONSTRAINT CLIENTE_PK PRIMARY KEY (CODIGO) ,
CONSTRAINT CLIENTE_UN UNIQUE (CPF)
);
```

13.2 ÍNDICES NÃO ÚNICOS (NÃO EXCLUSIVOS)

Os índices não únicos, apesar de não imporem exclusividade de valores, aceleram o acesso aos dados quando a consulta for realizada utilizando-se como parâmetro a coluna ou as colunas indexadas.

Pode-se, por exemplo, criar um índice deste tipo na coluna NOME de uma tabela denominada CLIENTE para localizar mais rapidamente os dados de um cliente a partir de seu nome. Para criar um índice utilizamos o comando CREATE INDEX:

```
CREATE INDEX nome_do_indice
ON nome_da_tabela (nome_da_coluna);
```

O exemplo a seguir cria um índice na coluna NOME da tabela CLIENTE:

```
CREATE INDEX CLIENTE_IDX ON CLIENTE (NOME);
```

13.3 RENOMEAR UM ÍNDICE

Para renomear um índice utilizamos o comando ALTER INDEX:

```
ALTER INDEX nome_do_indice RENAME TO novo_nome_do_indice;
```

No exemplo a seguir estamos renomeando o índice anteriormente criado:

```
ALTER INDEX CLIENTE_IDX RENAME TO CLIENTE_NOME_IDX;
```

13.4 ELIMINAR UM ÍNDICE

Para eliminar um índice utilizamos o comando DROP INDEX. No exemplo a seguir estamos eliminando o índice CLIENTE_NOME_IDX:

```
DROP INDEX CLIENTE_NOME_IDX;
```

RESUMO

Os índices permitem acesso mais rápido às linhas de uma tabela quando selecionamos um pequeno subconjunto de linhas.

O Oracle dispõe de vários tipos de índices. Neste capítulo abordamos dois deles:

- Índices únicos (exclusivos): criados automaticamente quando você define uma constraint PRIMARY KEY ou UNIQUE.
- Índices não únicos (não exclusivos): não impõem exclusividade de valores, porém aceleram o acesso aos dados quando a consulta utiliza como parâmetro a coluna ou as colunas indexadas.

Para criar um índice utilizamos o comando CREATE INDEX:

```
CREATE INDEX nome_do_indice ON nome_da_tabela (nome_da_coluna);
```

Para renomear um índice utilizamos o comando ALTER INDEX:

```
ALTER INDEX nome_do_indice RENAME TO novo_nome_do_indice;
```

Para eliminar um índice utilizamos o comando DROP INDEX:

```
DROP INDEX nome_do_indice;
```

EXERCÍCIOS

Observe os comandos SQL a seguir e responda:

```
CREATE TABLE CLIENTE (
  ID_CLIENTE INT,
  NOME VARCHAR2(30),
  CONSTRAINT CLIENTE_PK PRIMARY KEY (ID_CLIENTE));
CREATE INDEX CLIENTE_IDX ON CLIENTE (ID_CLIENTE);
```

- a) Será apresentada uma mensagem de erro ao tentar criar o índice CLIENTE_IDX, pois a coluna ID_CLIENTE já está indexada.
- b) Será apresentada uma mensagem de erro ao tentar criar o índice CLIENTE_IDX, pois a sintaxe empregada está errada.
- c) Os dois comandos CREATE TABLE e CREATE INDEX estão errados.
- d) Serão criados dois índices através da constraint CLIENTE_PK e através do comando CREATE INDEX.
- e) O índice CLIENTE_IDX substituirá a constraint CLIENTE_PK.

Observe os comandos SQL a seguir e responda:

```
CREATE TABLE PEDIDO (
NR_PEDIDO INT,
DT_EMISSAO DATE,
ID_CLIENTE INT,
CONSTRAINT PEDIDO_PK PRIMARY KEY (NR_PEDIDO),
CONSTRAINT CLIENTE_PEDIDO_FK FOREIGN KEY (ID_CLIENTE)
REFERENCES CLIENTE (ID_CLIENTE);
CREATE INDEX PEDIDO_IDX ON PEDIDO (ID_CLIENTE);
```

- a) Será apresentada uma mensagem de erro ao tentar criar o índice PEDIDO_IDX, pois a coluna ID_CLIENTE da tabela PEDIDO foi declarada como FOREIGN KEY e, portanto, já está indexada.
- b) O índice PEDIDO_IDX será criado normalmente, pois colunas que recebem a constraint FOREIGN KEY não são automaticamente indexadas.
- c) O índice PEDIDO_IDX será criado normalmente e a constraint CLIENTE_PEDIDO_FK será automaticamente desabilitada, pois uma coluna declarada como FOREIGN KEY não pode ser indexada.
- d) Será apresentada uma mensagem de erro e o índice PEDIDO_IDX não será criado, pois uma coluna declarada como FOREIGN KEY não pode ser indexada.
- e) Será apresentada uma mensagem de erro ao tentar criar o índice PEDIDO_IDX, pois a sintaxe empregada está errada.

CAPÍTULO 14 – VIEWS (VISÕES)

Uma visão (view, em inglês) é uma representação lógica de uma ou mais tabelas. Uma visão deriva seus dados de tabelas denominadas "tabelas base". (As tabelas base, na prática, podem ser tabelas ou outras visões.)

As consultas em visões são realizadas da mesma maneira que as consultas em tabelas. É possível também realizar (com algumas restrições) operações DML (INSERT, UPDATE e DELETE) nas "tabelas base" através das visões. Todas as operações executadas em uma visão afetam as tabelas base. As visões são também conhecidas como "consultas armazenadas".

O Oracle dispõe dos seguintes tipos de visões:

- Visões Regulares
- Visões Materializadas
- Visões de Objetos
- Visões "XMLType"

14.1 VISÕES REGULARES

Uma visão regular armazena apenas sua definição ou consulta no dicionário de dados, não alocando, portanto, espaço em um segmento para armazenamento dos dados.

Visões regulares podem ser utilizadas para ocultar a complexidade de determinadas consultas ao banco de dados ou para impor segurança.

O exemplo a seguir apresenta a criação da tabela FUNCIONARIO com quatro colunas: MATRICULA, NOME, DEPARTAMENTO e SALARIO.

Na sequência criamos uma VISÃO REGULAR com base na tabela FUNCIONARIO. Observe, porém, que a coluna SALARIO não foi incluída na visão. Quando realizamos a consulta com base na visão, mesmo utilizando o caractere coringa (*), os dados referentes à coluna SALARIO não aparecem. Esta é uma forma simples e prática de apresentar uma visão de dados diferente para cada usuário, conforme suas atribuições dentro de uma organização.

```
CREATE TABLE FUNCIONARIO (
MATRICULA NUMBER(4) ,
NOME VARCHAR2(30) ,
DEPARTAMENTO VARCHAR2(20) ,
SALARIO NUMBER(7,2) );
```

```
INSERT INTO FUNCIONARIO
VALUES (1001, 'ANTONIO ALVARES', 'ENGENHARIA', 5300);
INSERT INTO FUNCIONARIO
VALUES (1002, 'BEATRIZ BARBOSA', 'MARKETING', 4800);
INSERT INTO FUNCIONARIO
VALUES (1003, 'CLAUDIO CARDOSO', 'JURÍDICO', 5100);
```

```
CREATE VIEW FUNCIONARIO_VIEW AS
SELECT MATRICULA, NOME, DEPARTAMENTO
FROM FUNCIONARIO;
```

```
SELECT * FROM FUNCIONARIO_VIEW;
```

MATR	NOME	DEPARTAMENTO
1001	ANTONIO ALVARES	ENGENHARIA
1002	BEATRIZ BARBOSA	MARKETING
1003	CLAUDIO CARDOSO	JURÍDICO

14.1.1 Visões REGULARES – READ ONLY

Há situações em desejamos que os dados das visões sejam utilizados apenas para consultas e não sejam alterados ou excluídos. Para isso podemos utilizar a cláusula READ ONLY ao criar a visão. Observe o exemplo a seguir:

```
CREATE VIEW FUNCIONARIO_VIEW AS
SELECT MATRICULA, NOME, DEPARTAMENTO
FROM FUNCIONARIO
WITH READ ONLY CONSTRAINT FUNC_VIEW_READ_ONLY;
```


14.1.2 VISÕES REGULARES – APELIDOS PARA COLUNAS

Podemos ocultar detalhes sobre a estrutura da tabela base utilizando apelidos para as colunas. Neste caso os nomes das colunas da visão corresponderão a nomes diferentes na tabela base. Observe duas formas diferentes de fazer isso:

Exemplo 1:

```
CREATE VIEW FUNCIONARIO_VIEW (MAT_FUNC, NOME_FUNC, DEPT_FUNC) AS
SELECT MATRICULA, NOME, DEPARTAMENTO
FROM FUNCIONARIO;
```

Exemplo 2:

```
CREATE VIEW FUNCIONARIO_VIEW AS
SELECT
MATRICULA AS MAT_FUNC,
NOME AS NOME_FUNC,
DEPARTAMENTO AS DEPT_FUNC
FROM FUNCIONARIO;
```

14.1.3 VISÕES REGULARES – BASEADAS EM JUNÇÕES

Outra função importante das visões regulares é a simplificação de consultas complexas. Este objetivo pode ser alcançado quando elaboramos visões com base em duas ou mais tabelas utilizando junções (joins). Observe os dados obtidos à partir de duas tabelas CLIENTE e PEDIDO.

TABELA: CLIENTE			TABELA: PEDIDO		
CODCLI	NOME	UF	NR	VALOR	CODCLI
1001	FULANO	SP	1	4800	1002
1002	BELTRANO	RJ	2	3600	1003
1003	SICRANO	SP	3	5500	1001

A view a seguir foi elaborada com base nas duas tabelas: CLIENTE e PEDIDO:

```
CREATE VIEW CLIENTE_PEDIDO_VIEW AS
SELECT C.CODCLI, C.NOME, C.UF,
P.NR, P.VALOR
FROM CLIENTE C
INNER JOIN PEDIDO P
ON C.CODCLI = P.CODCLI;
```

A consulta a seguir:

```
SELECT * FROM CLIENTE_PEDIDO_VIEW;
```

Apresentará o seguinte resultado:

CODCLI	NOME	UF	NR	VALOR
1002	FULANO	SP	1	4800
1003	BELTRANO	RJ	2	3600
1001	SICRANO	SP	3	5500

Veja primeiramente um exemplo de consulta que utiliza os dados obtidos a partir das duas tabelas base: CLIENTE e PEDIDO:

```
SELECT C.UF, SUM(P.VALOR)
FROM CLIENTE C
INNER JOIN PEDIDO P
ON C.CODCLI = P.CODCLI
GROUP BY C.UF;
```

Observe agora, quando utilizamos a visão CLIENTE_PEDIDO_VIEW a consulta fica bem mais simples:

```
SELECT UF, SUM (VALOR)
FROM CLIENTE_PEDIDO_VIEW
GROUP BY UF;
```

14.2 VISÕES MATERIALIZADAS

Uma visão materializada (Materialized View, em inglês) armazena sua definição ou consulta no dicionário de dados, porém, diferentemente das visões regulares, aloca espaço em um segmento para armazenamento dos dados. Este tipo de visão pode, por exemplo, replicar uma cópia somente leitura da tabela base para outro banco de dados.

Visões materializadas utilizam um log de visão materializada associado às tabelas base para realizar atualizações incrementais. Caso não se utilize este recurso, será preciso realizar uma atualização completa quando for necessária a atualização dos dados na visão materializada.

Criaremos uma visão materializada com base na tabela ALUNO apresentada a seguir:

TABELA: ALUNO

RA	NOME
1001	ANTONIO ALVARES
1002	BEATRIZ BARBOSA
1003	CLAUDIO CARDOSO

Antes da criação da visão materializada, a tabela base (master table) deve ser associada a um materialized view log. Os logs de visões materializadas são usados para sincronização entre a tabela base e a visão.

```
CREATE MATERIALIZED VIEW LOG ON ALUNO;
```

Exemplo 1: Visão materializada atualizada MANUALMENTE:

```
CREATE MATERIALIZED VIEW ALUNO_VIEW_1
BUILD IMMEDIATE
AS SELECT * FROM ALUNO;
```

A seguir vamos incluir uma nova linha na tabela ALUNO:

```
INSERT INTO ALUNO (RA, NOME)
VALUES (1004, 'DANIELA DAMASIO');
COMMIT;
```

Agora devemos utilizar a procedure DBMS_MVIEW.REFRESH para atualizar a view. (Antes da atualização a última inserção aparecerá apenas na tabela base. Não será apresentada em uma eventual consulta à view ALUNO_VIEW_1).

```
CALL DBMS_MVIEW.REFRESH ('ALUNO_VIEW_1', 'F');
```

O parâmetro F significa FAST e implica em incrementar na view a linha que foi inserida na tabela.

Exemplo 2: Visão materializada atualizada AUTOMATICAMENTE:

```
CREATE MATERIALIZED VIEW ALUNO_VIEW_2
BUILD IMMEDIATE
REFRESH FORCE START WITH SYSDATE NEXT SYSDATE + 1/86400
AS SELECT * FROM ALUNO;
```

Nota: A atualização ocorrerá automaticamente em 1 segundo (1 dia = 86400 segundos).

14.3 VISÕES DE OBJETOS

Uma visão de objeto é uma tabela virtual de objeto. Cada linha na visão é um objeto, que é uma instância de um tipo de objeto. Um tipo de objeto é um tipo de dado definido pelo usuário.

As visões de objeto permitem que as aplicações orientadas a objetos vejam os dados como uma coleção de objetos que têm atributos e

métodos, facilitando a migração de um ambiente puramente relacional para um ambiente orientado a objetos.

É possível utilizar as visões de objeto para recuperar, atualizar, inserir e excluir dados relacionais como se estes fossem armazenados como um tipo de objeto. Pode-se também definir visões com colunas, que são tipos de dados de objetos, tais como objetos e coleções (tabelas aninhadas e varrays).

As visões de objetos fornecem um caminho de migração gradual para dados herdados. Fornecem coexistência de aplicações relacionais e orientadas a objetos. Facilitam a introdução de aplicativos orientados a objetos para dados relacionais existentes sem que seja necessária uma mudança drástica de um paradigma para outro. Você poderá buscar dados relacionais no cache de objetos do lado do cliente e mapeá-los, por exemplo, em estruturas C++, para que os aplicativos 3GL possam manipulá-los como se fossem estruturas nativas. Fornecem flexibilidade, pois permitem usar diferentes representações de objetos na memória para diferentes aplicativos sem alterar a forma como você armazena os dados no banco de dados.

Usar visões de objeto pode levar a um melhor desempenho. Os dados relacionais que compõem uma "linha" de uma visão de objeto atravessam a rede como uma unidade, diminuindo potencialmente o tráfego.

Observe o exemplo a seguir:

```
CREATE TABLE FUNCIONARIO (
  MATRICULA NUMBER (5) ,
  NOME VARCHAR2 (40) ,
  SALARIO NUMBER (9,2) ,
  CARGO VARCHAR2 (20) ) ;

CREATE TYPE FUNCIONARIO_T AS OBJECT (
  MATRICULA_NR NUMBER (5) ,
  NOME VARCHAR2 (40) ,
  SALARIO NUMBER (9,2) ,
  CARGO VARCHAR2 (20) ) ;
/
CREATE VIEW FUNCIONARIO_VIEW OF FUNCIONARIO_T
```

```
WITH OBJECT OID (MATRICULA_NR) AS
SELECT F.MATRICULA, F.NOME, F.SALARIO, F.CARGO
FROM FUNCIONARIO F;
```

Cada linha da visão FUNCIONARIO_VIEW contém um objeto do tipo FUNCIONARIO_T. Cada linha tem um identificador de objeto exclusivo.

A cláusula WITH OBJECT OID é utilizada para especificar uma exibição de objeto de nível superior ou raiz. Permite que você especifique os atributos do tipo de objeto que será usado como uma chave para identificar cada linha na exibição de objeto. Na maioria dos casos, esses atributos correspondem à chave primária da tabela base. Você deve garantir que a lista de atributos é exclusiva e identifica exatamente uma linha na visualização.

14.4 VISÕES "XML TYPE"

Em alguns casos, você pode ter uma tabela sobre a qual você gostaria de criar uma exibição XMLType. O exemplo a seguir cria uma tabela denominada ALUNO e, em seguida, cria uma exibição XMLType dessa tabela:

```
CREATE TABLE ALUNO (
RA NUMBER,
NOME VARCHAR2 (40));

INSERT INTO ALUNO VALUES (1234, 'ANTONIO ALVARES');
INSERT INTO ALUNO VALUES (2345, 'BEATRIZ BARBOSA');

COMMIT;

CREATE VIEW ALUNO_VIEW_XML AS
SELECT XMLELEMENT (
"ALUNO",
XMLELEMENT("RA", RA),
XMLELEMENT("NOME", NOME)
)
AS XML_ALUNO
FROM ALUNO;
```

A consulta a seguir:

```
SELECT * FROM ALUNO_VIEW_XML;
```

Retornará o seguinte resultado:

XML_ALUNO

```
-----
<ALUNO><RA>1234</RA><NOME>ANTONIO ALVARES</NOME></ALUNO>
<ALUNO><RA>2345</RA><NOME>BEATRIZ BARBOSA</NOME></ALUNO>
```

RESUMO

Visões, também conhecidas como "consultas armazenadas", são representações lógicas de uma ou mais tabelas (denominadas "tabelas base").

O Oracle dispõe dos seguintes tipos de visões:

- Visões Regulares: armazenam apenas sua definição ou consulta no dicionário de dados, não alocam espaço em um segmento para armazenamento dos dados.
- Visões Materializadas: armazenam sua definição ou consulta no dicionário de dados, porém, diferentemente das visões regulares, alocam espaço em um segmento para armazenamento dos dados.
- Visões de Objetos: são tabelas virtuais de objeto. Cada linha na visão é um objeto, que é uma instância de um tipo de objeto que, por sua vez, é um tipo de dado definido pelo usuário.
- Visões "XMLType": apresentam os dados conforme a estrutura de um documento do tipo XML (eXtensible Markup Language).

EXERCÍCIOS

Uma consulta aos dados da tabela PRODUTO apresentou o seguinte resultado:

TABELA: PRODUTO

CODIGO	DESCRICAO	VALOR	CLASSE
101	PRODUTO 01	30.00	A
102	PRODUTO 02	10.00	B
103	PRODUTO 03	20.00	A
104	PRODUTO 04	30.00	B
105	PRODUTO 05	40.00	A
106	PRODUTO 06	20.00	B

Qual dos seguintes comandos criará uma visão que apresentará a CLASSE dos produtos e a MÉDIA DOS VALORES de cada CLASSE?

- `CREATE VIEW CLASSE, AVG (VALOR) VALOR_MEDIO
FROM PRODUTO AS PRODUTO_VW
GROUP BY CLASSE;`
- `CREATE PRODUTO_VW AS VIEW OF
SELECT CLASSE, AVG (VALOR) VALOR_MEDIO FROM PRODUTO
GROUP BY CLASSE;`
- `CREATE PRODUTO_VW AS
SELECT CLASSE, AVG (VALOR) VALOR_MEDIO FROM PRODUTO
GROUP BY CLASSE;`
- `CREATE VIEW AS PRODUTO_VW OF
SELECT CLASSE, AVG (VALOR) VALOR_MEDIO FROM PRODUTO
GROUP BY CLASSE;`
- `CREATE VIEW PRODUTO_VW AS
SELECT CLASSE, AVG (VALOR) VALOR_MEDIO FROM PRODUTO
GROUP BY CLASSE;`

Observe o comando abaixo, que cria uma view materializada denominada PRODUTO_VW_2, e assinale a alternativa correta:

```
CREATE MATERIALIZED VIEW PRODUTO_VW_2  
BUILD IMMEDIATE  
REFRESH FORCE START WITH SYSDATE NEXT SYSDATE + 1/1440  
AS SELECT * FROM PRODUTO;
```

- a) A view PRODUTO_VW_2 será atualizada automaticamente a cada minuto.
- b) A view PRODUTO_VW_2 deverá ser atualizada manualmente a cada minuto.
- c) A view PRODUTO_VW_2 deverá ser atualizada manualmente 1440 vezes por semana.
- d) A view PRODUTO_VW_2 deverá ser atualizada automaticamente 1440 vezes por mês.
- e) A view PRODUTO_VW_2 deverá ser atualizada manualmente todos os dias às 14:40h.

CAPÍTULO 15 – SEQUENCES (SEQUÊNCIAS)

Uma sequência é um objeto de banco de dados que gera uma série de números inteiros. Uma aplicação comum para este tipo de objeto é o preenchimento de uma coluna de chave primária com tipo de dados numérico. Neste capítulo você aprenderá a:

- Criar uma sequência;
- Usar uma sequência;
- Obter informações sobre uma sequência através de consulta ao dicionário de dados;
- Modificar uma sequência;
- Excluir uma sequência.

15.1 CRIANDO UMA SEQUÊNCIA

Utilizamos o comando `CREATE SEQUENCE` para criar uma nova sequência no banco de dados. Há várias cláusulas que podem ser acrescentadas para atender melhor a necessidades específicas. No exemplo a seguir estamos criando uma sequência denominada `CLIENTE_SEQ`. Incluímos, no momento da criação, cláusulas que serão explicadas nos próximos subtópicos.

```
CREATE SEQUENCE CLIENTE_SEQ
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 1000
CYCLE
CACHE 10
ORDER;
```

15.1.1 START WITH

A cláusula START WITH especifica o valor (número inteiro) de início da sequência. O valor padrão é 1 (um).

15.1.2 INCREMENT BY

INCREMENT BY determina qual o valor de incremento. O padrão também é 1 (um). O valor a ser informado deve sempre ser um número inteiro.

15.1.3 MINVALUE E MAXVALUE

MINVALUE e MAXVALUE determinam respectivamente o valor mínimo e o valor máximo da sequência. O valor mínimo deve ser menor ou igual ao valor atribuído à cláusula START WITH. O valor máximo deve ser maior ou igual ao valor atribuído à cláusula START WITH e maior do que o valor atribuído à cláusula MINVALUE.

15.1.4 CYCLE

CYCLE determina que a sequência ao atingir o valor correspondente ao MAXVALUE voltará novamente em um novo ciclo e assumirá o valor correspondente ao MINVALUE. Caso CYCLE não apareça explicitamente quando a sequência for criada, valerá o padrão: NOCYCLE.

15.1.5 CACHE

CACHE determina a quantidade de valores inteiros que deverá ser mantida na memória do servidor. O padrão é 20 e o valor mínimo é 2. O número máximo a ser informado está limitado conforme a seguinte fórmula:

$$\text{CEIL}((\text{número_máximo} - \text{número_mínimo}) / \text{ABS}(\text{número_incremento}))$$

15.1.6 ORDER

ORDER garante que os números inteiros sejam gerados na ordem da solicitação. A cláusula é mais utilizada quando trabalhamos com Real Application Cluster (ambiente no qual vários servidores compartilham a mesma memória). O oposto (NOORDER) é o padrão e não garante que os números inteiros sejam gerados na ordem da solicitação.

15.2 USANDO UMA SEQUÊNCIA

Uma sequência apresenta duas pseudocolunas: CURRVAL (valor atual) e NEXTVAL (próximo valor). Observe as consultas a seguir.

Para apresentar o próximo valor (NEXTVAL) utilize:

```
SELECT CLIENTE_SEQ.NEXTVAL FROM DUAL;
```

Para apresentar o valor atual (CURRVAL) utilize:

```
SELECT CLIENTE_SEQ.CURRVAL FROM DUAL;
```

NOTA: Ao criar uma nova sequência ou no início de uma nova sessão, NEXTVAL deve ser referenciado antes de CURRVAL.

Veja, a seguir, como utilizar a sequência CLIENTE_SEQ, apresentada anteriormente, para preencher automaticamente a coluna (CODIGO_CLIENTE) da tabela CLIENTE quando realizamos a inserção de uma nova linha na tabela. Assumiremos (para simplificar) que a tabela CLIENTE tem apenas duas colunas: CODIGO_CLIENTE e NOME_CLIENTE.

```
INSERT INTO CLIENTE (CODIGO_CLIENTE,NOME_CLIENTE) VALUES
(CLIENTE_SEQ.NEXTVAL, 'ANTONIO ALVARES');
```

A consulta a seguir:

```
SELECT * FROM CLIENTE;
```

Apresentará a seguinte resposta:

```
- -----
1 ANTONIO ALVARES
```

15.3 SEQUÊNCIA: CONSULTANDO O DICIONÁRIO DE DADOS

Poderá consultar o dicionário de dados para obter informações referentes às sequências criadas por você (seu usuário de banco de dados). Utilize para isso o seguinte comando:

```
SELECT * FROM USER_SEQUENCES;
```

15.4 MODIFICANDO UMA SEQUÊNCIA

Devemos utilizar o comando ALTER SEQUENCE para modificar uma sequência. Podemos alterar qualquer cláusula dentro dos limites apresentados no início deste capítulo. O exemplo a seguir apresenta a alteração do valor de incremento de 1 para 2:

```
ALTER SEQUENCE CLIENTE_SEQ  
INCREMENT BY 2;
```

15.5 EXCLUINDO UMA SEQUÊNCIA

Utilizamos o comando DROP SEQUENCE para excluir uma sequência. O exemplo a seguir apresenta a exclusão da sequência CLIENTE_SEQ, criada no início deste capítulo:

```
DROP SEQUENCE CLIENTE_SEQ;
```

RESUMO

Sequências são objetos de banco de dados que geram uma série de números inteiros e são utilizadas, por exemplo, para o preenchimento de uma coluna de chave primária com tipo de dados numérico.

Utilizamos o comando CREATE SEQUENCE para criar uma nova sequência.

As cláusulas utilizadas em uma sequência são as seguintes:

- **START WITH:** Especifica o valor (número inteiro) de início da sequência. O valor padrão é 1 (um).

- **INCREMENT BY:** Determina qual o valor de incremento. O padrão também é 1 (um). O valor a ser informado deve sempre ser um número inteiro.
- **MINVALUE e MAXVALUE:** Determinam respectivamente o valor mínimo e o valor máximo da sequência.
- **CYCLE:** Determina que a sequência ao atingir o valor correspondente ao MAXVALUE voltará novamente em um novo ciclo e assumirá o valor correspondente ao MINVALUE.
- **CACHE:** Determina a quantidade de valores inteiros que deverá ser mantida na memória do servidor. O padrão é 20 e o valor mínimo é 2.
- **ORDER:** Garante que os números inteiros sejam gerados na ordem da solicitação. (A cláusula é mais utilizada quando trabalhamos com Real Application Cluster.)

EXERCÍCIOS

Qual será o resultado dos seguintes comandos SQL?

```
CREATE SEQUENCE TESTE_SEQ
START WITH 1
INCREMENT BY 3;
SELECT TESTE_SEQ.CURRVAL FROM DUAL;
```

- O comando SELECT apresentará o valor 1.
- O comando SELECT apresentará o valor 3.
- O comando SELECT apresentará o valor 4.
- O comando SELECT falhará, pois NEXTVAL deve ser referenciado antes de CURRVAL.
- O último comando SELECT falhará, pois a sequência somente pode ser referenciada em um comando INSERT.

Observe os comandos SQL a seguir e responda:

```

CREATE TABLE CLIENTE (
  ID_CLIENTE INT,
  NOME VARCHAR2(30) ,
  CONSTRAINT CLIENTE_PK PRIMARY KEY(ID_CLIENTE)) ;
CREATE SEQUENCE CLIENTE_SEQ
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 3
CACHE 2
CYCLE ;
INSERT INTO CLIENTE VALUES (CLIENTE_SEQ.NEXTVAL, 'ANTONIO') ;
INSERT INTO CLIENTE VALUES (CLIENTE_SEQ.NEXTVAL, 'BEATRIZ') ;
INSERT INTO CLIENTE VALUES (CLIENTE_SEQ.NEXTVAL, 'CLAUDIO') ;
INSERT INTO CLIENTE VALUES (CLIENTE_SEQ.NEXTVAL, 'DANIELA') ;

```

- a) Todos os comandos serão executados sem nenhum erro.
- b) Será apresentada uma mensagem de erro ao tentar criar a sequência CLIENTE.SEQ, pois a sintaxe empregada está errada.
- c) A última linha (das inserções) apresentará um erro, pois viola a chave primária CLIENTE_PK.
- d) Será apresentada uma mensagem de erro ao tentar inserir os dados na tabela, pois a forma de inserir os valores referentes à sequência criada está errada.
- e) A última linha (das inserções) ignorará a cláusula CYCLE, para não violar a chave primária CLIENTE_PK, e continuará a inserir a sequência numérica atribuindo à cliente DANIELA o ID_CLIENTE igual a 4.

CAPÍTULO 16 – SYNONYM (SINÔNIMOS)

Sinônimos são utilizados para simplificar o acesso a objetos do banco de dados, tais como tabelas e visões. Sinônimos tornam possível:

- Criar uma referência simples para uma tabela ou visão que pertença a outro usuário;
- Reduzir nomes longos de objetos do banco de dados.

A sintaxe para criação de sinônimos é a seguinte:

```
CREATE SYNONYM nome_do_sinonimo FOR nome_do_objeto;
```

O exemplo abaixo cria um sinônimo CLIENTE_SP para a tabela CLIENTE_SAO_PAULO:

```
CREATE SYNONYM CLIENTE_SP FOR CLIENTE_SAO_PAULO;
```

Há ocasiões em que um ou mais usuários do banco de dados necessitarão acessar objetos de outro usuário. O DBA poderá criar, para atender a esta necessidade, um sinônimo público utilizando a opção PUBLIC.

O exemplo a seguir apresenta a criação de um sinônimo que permitirá que qualquer usuário com privilégios suficientes acesse a tabela CLIENTE do usuário SCOTT, sem que para isso tenha que referenciar o SCHEMA (ou nome do usuário proprietário da tabela):

```
CREATE PUBLIC SYNONYM CLIENTE FOR SCOTT.CLIENTE;
```

Observe, a seguir, a diferença entre as duas consultas que seriam realizadas caso um usuário diferente de SCOTT precisasse consultar todos os dados da tabela CLIENTE do usuário SCOTT (desde que tivesse privilégio para isso):

1. Sem o sinônimo anteriormente criado:

```
SELECT * FROM SCOTT.CLIENTE;
```

2. Com o sinônimo anteriormente criado:

```
SELECT * FROM CLIENTE;
```

16.1 SUBSTITUINDO UM SYNONYM

Podemos substituir o sinônimo criado anteriormente através do seguinte comando:

```
CREATE OR REPLACE SYNONYM nome_do_sinonimo  
FOR nome_do_objeto;
```

Podemos substituir um sinônimo com a opção PUBLIC através do seguinte comando:

```
CREATE OR REPLACE PUBLIC SYNONYM CLIENTE FOR SCOTT.CLIENTE;
```

16.2 EXCLUINDO UM SYNONYM

Podemos excluir o sinônimo criado anteriormente através do seguinte comando:

```
DROP SYNONYM nome_do_sinonimo;
```

Observe o exemplo a seguir:

```
DROP SYNONYM CLIENTE;
```

Podemos excluir também um sinônimo com a opção PUBLIC através do seguinte comando:

```
DROP PUBLIC SYNONYM nome_do_sinonimo;
```

RESUMO

Sinônimos são utilizados para simplificar o acesso a objetos do banco de dados, tais como tabelas e visões.

A sintaxe para criação de sinônimos é a seguinte:

```
CREATE SYNONYM nome_do_sinonimo FOR nome_do_objeto;
```

Sinônimos podem ser especialmente úteis quando precisamos acessar objetos de outros usuários. O DBA poderá criar, para atender a esta necessidade, um sinônimo público utilizando a opção PUBLIC. Desta forma torna-se possível acessar os objetos de outros usuários utilizando simplesmente o seu sinônimo.

A exclusão de um sinônimo é realizada através do seguinte comando:

```
DROP SYNONYM nome_do_sinonimo;
```

EXERCÍCIOS

Um usuário com privilégios SYSDBA emitiu o seguinte comando:

```
CREATE PUBLIC SYNONYM FUNC FOR FULANO.FUNCIONARIO;
```

Após a emissão deste comando criou o seguinte usuário:

```
CREATE USER BELTRANO IDENTIFIED BY ABC123;
```

A seguir concedeu o seguinte privilégio ao usuário FULANO:

```
GRANT CREATE SESSION TO BELTRANO;
```

Podemos, portanto, afirmar que:

- a) O usuário BELTRANO não poderá consultar os dados da tabela FUNCIONARIO, pois a criação do usuário (BELTRANO) ocorreu após a criação do synonym.
- b) Apenas o usuário FULANO poderá fazer uso do synonym para consultar os dados da tabela FUNCIONARIO.
- c) O usuário BELTRANO poderá consultar a tabela FUNCIONARIO sem utilizar o synonym. Deverá, portanto, informar schema e nome do objeto para acessá-la.
- d) O usuário BELTRANO poderá consultar os dados da tabela, pois o privilégio CREATE SESSION é suficiente para isso.
- e) O usuário BELTRANO não poderá consultar os dados da tabela utilizando o synonym, pois não recebeu o privilégio SELECT para este objeto.

O usuário SICRANO com privilégio CREATE SYNONYM emitiu o seguinte comando:

CREATE SYNONYM CLI FOR CLIENTE;

Admitindo-se que a tabela CLIENTE faz parte do schema de SICRANO, podemos afirmar:

- a) O comando falhará, pois o usuário SICRANO não referenciou o seu schema (SICRANO.CLIENTE) no momento da criação do synonym.
- b) O comando será executado normalmente e, a partir deste momento, todos os usuários do banco poderão consultar os dados da tabela CLIENTE utilizando o synonym.
- c) O comando será executado normalmente e, a partir deste momento, apenas os usuários do banco que receberem os privilégios necessários poderão consultar os dados da tabela CLIENTE referenciando o schema e o synonym (SICRANO.CLI).
- d) O comando será executado normalmente e, a partir deste momento, apenas os usuários do banco que receberem os

privilégios necessários poderão consultar os dados da tabela CLIENTE referenciando apenas o synonym (CLI).

- e) O comando será executado normalmente e, a partir deste momento, nenhum outro usuário, além de SICRANO, mesmo que tenha privilégios para isso, poderá consultar os dados da tabela CLIENTE utilizando o synonym.

REFERÊNCIAS

BC BURLESON CONSULTING. *The History of Oracle*. Disponível em: <http://www.dba-oracle.com/t_history_oracle.htm>. Acesso em: 16 fev. 2018.

BRYLA, Bob. LONEY, Kevin. *Oracle Database 11g manual do DBA*. Porto Alegre: Bookman, 2009.

COMUNIDADE BRASILEIRA DE POSTGRE SQL. *Postgre SQL10 lançado*. Disponível em: <<https://www.postgresql.org.br>>. Acesso em: 16 fev. 2018.

DaDBM. DATA AND DATABASE MANAGEMENT – DBMS BLOG. Disponível em: <<http://www.dadbm.com/roadmap-oracle-database-releases>>. Acesso em: 16 fev. 2018.

DATE, Christopher J. *Introdução a sistemas de bancos de dados*. Rio de Janeiro: Elsevier, 2004.

DB-ENGINES. Disponível em: <<http://db-engines.com/en/ranking>>. Acesso em: 17 set. 2016.

ELMASRI, Ramez; NAVATHE, Shamkant B. *Sistemas de banco de dados*. 4. ed. São Paulo: Pearson Addison Wesley, 2010.

IBM. *Simplifique a administração do banco de dados e acelere o desenvolvimento de aplicativos e aumente a colaboração*. Disponível em: <<http://www-03.ibm.com/software/products/pt/data-studio>>. Acesso em: 16 fev. 2018.

MICROSOFT. *Execute o SQL Server na sua plataforma favorita*. Disponível em: <<https://www.microsoft.com/pt-br/sql-server/sql-server-2017-editions>>. Acesso em: 20 fev. 2018.

O'HEAM, Steve. *OCA Oracle Database: SQL Certified Expert Exam Guide*. NewYork: Mc Graw Hill, 2010.

ORACLE HELP CENTER. *Database SQL Reference. Selecting from the DUAL Table*. Disponível em: <https://docs.oracle.com/cd/B19306_01/server.102/b14200/queries009.htm>. Acesso em: 16 fev. 2018.

ORACLE HELP CENTER. *Database SQL Reference. SQL Functions*. Disponível em: <http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions001.htm>. Acesso em: 16 fev. 2018.

ORACLE HELP CENTER. *Feature Availability by Edition*. Disponível em: <https://docs.oracle.com/cd/B28359_01/license.111/b28287/editions.htm#DBLIC116>. Acesso em: 16 fev. 2018.

ORACLE HELP CENTER. *Oracle Database Online Documentation 12c Release 1 (12.1)*. Disponível em: <<http://docs.oracle.com/database/121/index.htm>>. Acesso em: 17 set. 2016.

ORACLE® DATABASE SQL REFERENCE. *Data Definition Language (DDL) Statements*. Disponível em: <https://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_1001.htm#i2099120>. Acesso em: 16 fev. 2018.

PostgreSQL. *Documentação do PostgreSQL 8.0.0: uma breve história do PostgreSQL*. Disponível em: <<http://pgdocptbr.sourceforge.net/pg80/history.html>>. Acesso em: 25 set. 2016.

PRICE, Jason. *Oracle Database 11g SQL*. Porto Alegre: Bookman, 2009.

RADA, Amyris; MELNYK, Roman. *Compare the distributed DB2 10.5 database servers*. Disponível em: <<http://www.ibm.com/developerworks/data/library/techarticle/dm-1311db2compare>>. Acesso em: 16 fev. 2018.

SILBERSCHATZ, A.; KORTH, H.; SUBARSHAN, S. *Sistema de banco de dados*. 5. ed. Rio de Janeiro: Campus, 2006.

WATSON, John; RAMKLASS, Roopesh. *OCA Oracle Database 11g Fundamentos SQL I*. Rio de Janeiro: Alta Books, 2010.

APÊNDICE – PALAVRAS RESERVADAS ORACLE

ACCESS	ACCOUNT	ACTIVATE
ADD	ADMIN	ADVISE
AFTER	ALL	ALL_ROWS
ALLOCATE	ALTER	ANALYZE
AND	ANY	ARCHIVE
ARCHIVELOG	ARRAY	AS
ASC	AT	AUDIT
AUTHENTICATED	AUTHORIZATION	AUTOEXTEND
AUTOMATIC	BACKUP	BECOME
BEFORE	BEGIN	BETWEEN
BFILE	BITMAP	BLOB
BLOCK	BODY	BY
CACHE	CACHE_INSTANCES	CANCEL
CASCADE	CAST	CFILE
CHAINED	CHANGE	CHAR
CHAR_CS	CHARACTER	CHECK
CHECKPOINT	CHOOSE	CHUNK
CLEAR	CLOB	CLONE
CLOSE	CLOSE_CACHED_OPEN_CURSORS	CLUSTER
COALESCE	COLUMN	COLUMNS
COMMENT	COMMIT	COMMITTED
COMPATIBILITY	COMPILE	COMPLETE
COMPOSITE_LIMIT	COMPRESS	COMPUTE
CONNECT	CONNECT_TIME	CONSTRAINT
CONSTRAINTS	CONTENTS	CONTINUE
CONTROLFILE	CONVERT	COST
CPU_PER_CALL	CPU_PER_SESSION	CREATE
CURRENT	CURRENT_SCHEMA	CURREN_USER
CURSOR	CYCLE	DATAFILE
DANGLING	DATABASE	DATE
DATAFILES	DATAOBJNO	DBLOW
DBA	DBHIGH	DEBUG

DBMAC	DEALLOCATE	DECLARE
DEC	DECIMAL	DEFERRED
DEFAULT	DEFERRABLE	DEREF
DEGREE	DELETE	DISABLE
DESC	DIRECTORY	DISTINCT
DISCONNECT	DISMOUNT	DOUBLE
DISTRIBUTED	DML	EACH
DROP	DUMP	END
ELSE	ENABLE	ESCAPE
ENFORCE	ENTRY	EXCHANGE
EXCEPT	EXCEPTIONS	EXECUTE
EXCLUDING	EXCLUSIVE	EXPLAIN
EXISTS	EXPIRE	EXTERNALLY
EXTENT	EXTENTS	FAST
FAILED_LOGIN_ ATTEMPTS	FALSE	FLAGGER
FILE	FIRST_ROWS	FLUSH
FLOAT	FLOB	FOREIGN
FOR	FORCE	FROM
FREELIST	FREELISTS	GLOBAL
FULL	FUNCTION	GRANT
GLOBALLY	GLOBAL_NAME	HASH
GROUP	GROUPS	HEADER
HASHKEYS	HAVING	IDGENERATORS
HEAP	IDENTIFIED	IMMEDIATE
IDLE_TIME	IF	INCREMENT
IN	INCLUDING	INDEXES
INDEX	INDEXED	INITIAL
INDICATOR	IND_PARTITION	INSERT
INITIALLY	INITTRANS	INSTEAD
INSTANCE	INSTANCES	INTERMEDIATE
INT	INTEGER	IS
INTERSECT	INTO	KEEP
ISOLATION	ISOLATION_LEVEL	LABEL
KEY	KILL	LEVEL
LAYER	LESS	LIMIT

LIBRARY	LIKE	LOB
LINK	LIST	LOCKED
LOCAL	LOCK	LOGGING
LOG	LOGFILE	LONG
LOGICAL_READS_PER_CALL	LOGICAL_READS_PER_SESSION	MAX
MANAGE	MASTER	MAXEXTENTS
MAXARCHLOGS	MAXDATAFILES	MAXLOGHISTORY
MAXINSTANCES	MAXLOGFILES	MAXTRANS
MAXLOGMEMBERS	MAXSIZE	MEMBER
MAXVALUE	MIN	MINUS
MINIMUM	MINEXTENTS	MLS_LABEL_FORMAT
MINVALUE	MLSLABEL	MOUNT
MODE	MODIFY	MULTISET
MOVE	MTS_DISPATCHERS	NCHAR_CS
NATIONAL	NCHAR	NESTED
NCLOB	NEEDED	NEXT
NETWORK	NEW	NOCACHE
NOARCHIVELOG	NOAUDIT	NOFORCE
NOCOMPRESS	NOCYCLE	NOMINVALUE
NOLOGGING	NOMAXVALUE	NOOVERRIDE
NONE	NOORDER	NOREVERSE
NOPARALLEL	NOSORT	NOT
NORMAL	NOWAIT	NULL
NOTHING	NUMERIC	NVARCHAR2
NUMBER	OBJNO	OBJNO_REUSE
OBJECT	OFF	OFFLINE
OF	OIDINDEX	OLD
OID	ONLINE	ONLY
ON	OPEN	OPTIMAL
OPCODE	OPTION	OR
OPTIMIZER_GOAL	ORGANIZATION	OSLABEL
ORDER	OWN	PACKAGE
OVERFLOW	PARTITION	PASSWORD
PARALLEL	PASSWORD_LIFE_TIME	PASSWORD_LOCK_TIME

PASSWORD_GRACE_TIME	PASSWORD_REUSE_TIME	PASSWORD_VERIFY_FUNCTION
PASSWORD_REUSE_MAX	PCTINCREASE	PCTTHRESHOLD
PCTFREE	PCTVERSION	PERCENT
PCTUSED	PLAN	PLSQL_DEBUG
PERMANENT	PRECISION	PRESERVE
POST_TRANSACTION	PRIOR	PRIVATE
PRIMARY	PRIVILEGE	PRIVILEGES
PRIVATE_SGA	PROFILE	PUBLIC
PROCEDURE	QUEUE	QUOTA
PURGE	RAW	RBA
RANGE	READUP	REAL
READ	RECOVER	RECOVERABLE
REBUILD	REF	REFERENCES
RECOVERY	REFRESH	RENAME
REFERENCING	RESET	RESETLOGS
REPLACE	RESOURCE	RESTRICTED
RESIZE	RETURNING	REUSE
RETURN	REVOKE	ROLE
REVERSE	ROLLBACK	ROW
ROLES	ROWNUM	ROWS
ROWID	SAMPLE	SAVEPOINT
RULE	SCAN_INSTANCES	SCHEMA
SB4	SCOPE	SD_ALL
SCN	SD_SHOW	SEGMENT
SD_INHIBIT	SEG_FILE	SELECT
SEG_BLOCK	SERIALIZABLE	SESSION
SEQUENCE	SESSIONS_PER_USER	SET
SESSION_CACHED_CURSORS	SHARED	SHARED_POOL
SHARE	SIZE	SKIP
SHRINK	SMALLINT	SNAPSHOT
SKIP_UNUSABLE_INDEXES	SORT	SPECIFICATION
SOME	SQL_TRACE	STANDBY

SPLIT	STATEMENT_ID	STATISTICS
START	STORAGE	STORE
STOP	SUCCESSFUL	SWITCH
STRUCTURE	SYS_OP_NTCIMG\$	SYNONYM
SYS_OP_ENFORCE_ NOT_NULL\$	SYSDBA	SYSOPER
SYSDATE	TABLE	TABLES
SYSTEM	TABLESPACE_NO	TABNO
TABLESPACE	THAN	THE
TEMPORARY	THREAD	TIMESTAMP
THEN	TO	TOPELVEL
TIME	TRACING	TRANSACTION
TRACE	TRIGGER	TRIGGERS
TRANSITIONAL	TRUNCATE	TX
TRUE	UB2	UBA
TYPE	UNARCHIVED	UNDO
UID	UNIQUE	UNLIMITED
UNION	UNRECOVERABLE	UNTIL
UNLOCK	UNUSED	UPDATABLE
UNUSABLE	USAGE	USE
UPDATE	USING	VALIDATE
USER	VALUE	VALUES
VALIDATION	VARCHAR2	VARYING
VARCHAR	WHEN	WHENEVER
VIEW	WITH	WITHOUT
WHERE	WRITE	WRITEDOWN
WORK	WRITEDOWN	WRITEUP
WRITE	YEAR	ZONE
XID		

O AUTOR

Marcos Alexandruk é professor (desde 2004) e coordenador (desde 2013) do curso de Tecnologia em Banco de Dados na Universidade Nove de Julho – UNINOVE. Ministra as seguintes disciplinas: Modelagem de Banco de Dados, Desenvolvimento de Banco de Dados, Programação para Banco de Dados (PL/SQL), Sistemas de Gerenciamento de Banco de Dados, dentre outras. Graduado em Sistemas de Informações, com especialização em Engenharia de Websites e mestrado em Engenharia Biomédica. Membro associado da Sociedade Brasileira de Engenharia Biomédica. Atuou durante dez anos como gerente de tecnologia e gerente comercial em empresas de T.I. atendendo clientes de grande porte, principalmente na área de telecom.

Esta obra está estruturada de forma diferente de muitas outras, que primeiro apresentam como realizar consultas em bases de dados já existentes para, mais adiante, apresentarem como estas bases são criadas. A parte teórica é apresentada de forma sucinta. As explicações sobre o que cada comando da linguagem SQL realiza são bem objetivas. Por outro lado, o livro apresenta exemplos em cada item para que o leitor possa, se assim desejar, colocar em prática seus conhecimentos. Esta é a melhor maneira de assimilar uma linguagem. Quando concluir a leitura e realizar os exercícios propostos, o leitor estará apto a executar as seguintes operações em bancos de dados relacionais:

- Criar uma base de dados relacional (criar, alterar e eliminar tabelas);
- Manipular dados na base criada (inserir, atualizar e excluir dados);
- Realizar consultas simples e complexas a partir de uma ou mais tabelas;
- Controlar as transações SQL;
- Administrar os privilégios dos usuários do banco de dados;
- Criar índices (para melhorar o desempenho de suas consultas);
- Criar visões de dados (consultas gravadas com base em uma ou mais tabelas).

Marcos Alexandruk

ISBN 978-85-69852-67-8

