PL/SQL

Domine a linguagem do banco de dados Oracle

Casa do Código

EDUARDO GONÇALVES

Prefácio

Sempre gostei de livros que "conversam" com o leitor, principalmente os técnicos. Como se fosse uma troca de ideias entre o autor e a pessoa que o está lendo. Procurei escrever desta forma, pois creio que com isso a leitura se torna mais clara e amigável, um bate papo entre amigos, conversando sobre qualquer assunto. A intenção foi colocar aqui tudo o que você vai precisar saber sobre a linguagem PL/SQL. O livro aborda conceitos que são utilizados no dia a dia do desenvolvimento e análise de sistemas para banco de dados Oracle. Demonstrei diversos exemplos, que também vão ajudá-lo como fonte de referência para consulta de comandos e como utilizá-los. Tenha uma otima leitura!

Público alvo

Este livro se destina a iniciantes e experientes na linguagem PL/SQL. Para os iniciantes, são abordados conceitos sobre a estrutura da linguagem PL/SQL e suas características. Ele ensina como criar programas, dos mais simples até os mais complexos, para atender às mais diversas necessidades. Você vai aprender como esta linguagem pode trazer um alto grau de produtividade e performance para as suas aplicações. Para os já experientes, ele ajudará como fonte de referência e para relembrar conceitos e técnicas da linguagem.

Como o livro está dividido?

Primeiramente, são abordados conceitos básicos da linguagem, como sua definição, como é estruturada e seus principais componentes. Logo após, entramos a fundo na sua estrutura, conhecendo cada comando e cada compo-

nente que ela utiliza, explorando suas funcionalidades e propriedades. O livro também mostra como incorporar a linguagem SQL dentro dos programas escritos em PL/SQL, para a recuperação e manipulação de dados. Tipos de dados, estruturas condicionais e de repetição, armazenamento de programas através de *procedures* (procedimentos) e *functions* (funções), e modularização através de *packages* (pacotes) são alguns dos itens mostrados.

Toda essa abordagem é realizada no âmbito prático, onde são detalhadas suas estruturas, seus comandos e características, sempre através de exemplos práticos e de fácil compreensão.

Os scripts de base (tabelas e dados) e fontes para a execução dos exemplos do livro estão disponíveis no endereco https://github.com/eduardogoncalvesbr/livroplsql-casadocodigo.

Confira outras obras do autor

SQL: Uma abordagem para bancos de dados Oracle http://www.casadocodigo.com.br/products/livro-sql-oracle

Contatos

Para falar com o autor, envie e-mail para eduardogoncalves.br@gmail.com Página no facebook: https://www.facebook.com/eduardogoncalvesescritor

Sobre o autor

Formado em Tecnologia da Informação, possui mais de 10 anos de experiência em análise e desenvolvimento de sistema voltados a tecnologias Oracle, trabalhando por grandes empresas como Lojas Renner, Mundial S.A, Tigre S.A, Pernambucanas, Tractebel Energia, Portobello Ceramica, Bematech, entre outros. Eduardo Gonçalvez é instrutor de cursos oficiais da Oracle – nas linguagens SQL e PL/SQL. Também atua no desenvolvimento de aplicações mobile para a plataforma iOS. Atualmente, atua como Coordenador de Operações na Supero Tecnologia.

Agradecimentos

Dedico este livro a todas os meus amigos e colegas da área de tecnologia, bem como meus professores e mentores, que tive o prazer de conhecer, e com os quais aprendi e até mesmo trabalhei. Durante esta minha trajetória aprendi muitas coisas com estas pessoas extraordinárias. Trocamos experiências e aprendemos muito, uns com os outros, programando e quebrando a cabeça para escrever a linha de código que ajudasse a solucionar um problema (regado a muito café, na maioria das vezes... risos), ou até mesmo, apenas pela paixão de programar. Saibam que cada um de vocês contribuiu imensamente para este trabalho! Sem vocês isto não seria possível! Ficam aqui meus sinceros agradecimentos!

Sumário

1 PL/SQL 1
1.1 O que é PL/SQL? . . . 1
1.2 Por que aprender PL/SQL? . . . 2
1.3 SQL, SQL*Plus, PL/SQL: Qual é diferença? . . . 3

2 Programação em bloco 7

3 Primeiros passos 13
3.1 Como iniciar no PL/SQL . . . 13

4 Pacote dbms_output 19
4.1 Exceções para o pacote dbms_output . . . 26

5 Variáveis bind e de substituição 27
5.1 Variáveis bind . . . 27
5.2 Variáveis de substituição . . . 31
5.3 Utilizando variáveis em arquivos . . . 35

6 Aspectos iniciais da programação PL/SQL 43
6.1 Caracteres e operadores . . . 44
6.2 Identificadores . . . 44
6.3 Transações . . . 47
6.4 Transações em PL/SQL . . . 51
6.5 Trabalhando com variáveis e constantes . . . 52
6.6 Tipos de dados em PL/SQL . . . 53

7 Exceções 59
7.1 Exceções predefinidas 60
7.2 Exceções definidas pelo usuário 79

8 Estruturas de condição: if 85
8.1 Estruturas do comando if-end if 86
8.2 Estruturas do comando if-else-end if 87
8.3 Estruturas do comando if-elsif(-else)-end if 89
8.4 Formatando as declarações if 93
8.5 Evitando erros comuns no uso de if 94

9 Comandos de repetição 95
9.1 for loop 95
9.2 while loop 101
9.3 loop 102
9.4 Qual loop deve-se usar? 105

10 Cursores 107
10.1 Cursores explícitos 108

10.3 Cursor for loop com definição interna 118
10.4 Cursores implícitos 119
10.5 Atributos de cursor explícito e implícito 121
10.6 Cursores encadeados 128
10.7 Cursor com for update 130

11 Funções de caracteres e operadores aritméticos 143
11.1 Funções de caracteres 144
11.2 Funções de cálculos 149
11.3 Operadores aritméticos 154

12 Funções de agregação (grupo) 159

13 Funções de data 177

14 Funções de conversão 183
14.1 to_date 184
14.2 to_number 192
14.3 to_char 204

15 Funções condicionais 211
15.1 decode vs. case 217

16 Programas armazenados 225
16.1 procedures e functions 226
16.2 Uso do comando replace 236
16.3 Recompilando programas armazenados 238
16.4 Recuperando informações 238
16.5 Recuperando códigos 239
16.6 Visualizando erros de compilação 240
16.7 Passando parâmetros 243
16.8 Dependência de objetos 249

17 packages 265
17.1 Estrutura de um package 266
17.2 Acesso a packages 270
17.3 Recompilando packages 278
17.4 Recuperando informações 278
17.5 Recuperando códigos 279
17.6 Visualizando erros de compilação 280

18 Transações autônomas 283

19 Triggers 293
19.1 Trigger de banco de dados 294
19.2 Trigger de tabela 294
19.3 Trigger de linha 301
19.4 Mutante table 316
19.5 Trigger de sistema 324
19.6 Trigger de view 330

20 PL/SQL Tables (estruturas homogêneas) 349

21 PL/SQL Records (estruturas heterogêneas) 359

22 Pacote utl_file 365

23 SQL dinâmico 385

23.1 Ref cursor 397

24 Apêndice: SQL – Primeiros passos 405

24.1 Como iniciar no SQL 405

25 Referências bibliográficas 413

26 Anexos 415

Capítulo 1

PL/SQL

1.1 O que é PL/SQL?

A sigla PL/SQL significa *Procedural Language / Structured Query Language*, ou seja, trata-se de uma linguagem procedural tendo como base a SQL (Linguagem de Consulta Estruturada). A PL/SQL é direcionada para o banco de dados Oracle e é escrita através de blocos de código que são executados diretamente no banco de dados. Ela permite o desenvolvimento de programas complexos, nos quais grandes volumes de informação são processados.

Como qualquer outra linguagem de programação, a PL/SQL é escrita através de códigos, sendo que os princípios de lógica de programa e programação estruturada podem ser implementados.

Os blocos PL/SQL podem ser escritos e guardados no próprio banco de dados, através de funções (`functions`), procedimentos (`procedures`),

gatilhos (`triggers`) ou pacotes (`packages`) para serem executados e reaproveitados quando necessário.

Os códigos podem ser feitos utilizando-se uma interface, como o *SQL*Plus*, que é totalmente integrada ao banco de dados. Outras ferramentas como Oracle Forms, Oracle Reports e Workflow Builder também possuem motores PL/SQL que permitem validar e compilar os códigos escritos em PL/SQL.

Através de todos esses atributos, a PL/SQL se mantém como uma linguagem robusta e muito utilizada pelos desenvolvedores e analistas do mundo todo. Muito provavelmente, isso se dá ao fato de sua crescente história, que vem desde as primeiras versões do banco de dados e sua evolução permanente desde então.

No mais, a PL/SQL dá suporte aos mais variados recursos de programação de sistemas como a utilização de variáveis, constantes, cursores, estruturas de repetição e condicional, o uso de pacotes, funções, procedimentos armazenados, tratamento de erros, chamadas à API, gatilhos, vetores, registros, execução direta de comandos SQL etc. Além do mais, ela condiciona em sua estrutura os padrões ANSI para linguagem SQL e suporte à codificação Java.

1.2 Por que aprender PL/SQL?

É importante conhecer a linguagem PL/SQL, independente da utilização ou não das ferramentas de desenvolvimento. Mesmo utilizando apenas o banco de dados, é fundamental conhecê-la tendo em vista que muitos processos dentro do servidor do banco são escritos ou executados por blocos nesta linguagem.

No que diz respeito ao desenvolvimento, a PL/SQL proporciona rapidez, eficiência e segurança para seus programas, pois seus códigos, ou melhor, programas, podem ser armazenados no banco de dados. Assim sendo, na próxima vez que precisar executá-lo novamente, é só chamá-lo, ele já está pronto. Outro ponto interessante é que os programas não precisam estar no cliente, já que a PL/SQL utiliza o servidor do banco de dados para a execução e armazenamentos dos seus processos.

Outra vantagem de se aprender PL/SQL é que no banco de dados Oracle,

embora seja implementado em diversas plataformas de hardware e software, a PL/SQL é igual para todas elas. Com isto, a portabilidade se mantém presente.

1.3 SQL, SQL*Plus, PL/SQL: Qual é diferença?

É tanto "SQL" nos títulos dos produtos Oracle que acabamos nos confundindo. Vamos entender o que é cada uma destas siglas.

- **SQL**: a sigla SQL significa *Structured Query Language*. É uma linguagem estruturada de acesso aos bancos de dados, declarativa, que usa comandos como `SELECT`, `INSERT`, `UPDATE` e `DELETE`. Ela sempre é executada no servidor através de uma interface conectada ao banco de dados. Apesar de não ser propriedade da Oracle, ela a incorpora em sua estrutura base e em suas linguagens.
- **PL/SQL**: é uma linguagem de procedimentos, propriedade da Oracle. Caracteriza-se por uma linguagem não declarativa, ou seja, não bastam apenas comandos SQL. É necessário o uso de codificação em blocos, chamados blocos PL/SQL. Os códigos podem ser executados tanto no cliente (Oracle Forms, Reports etc.) quanto diretamente no servidor do banco de dados.
- **SQL*Plus**: pode-se dizer que ele é a interface entre o usuário e o banco de dados. Quando executamos um comando SQL ou um bloco PL/SQL pelo SQL*Plus, ele os envia a um motor PL/SQL que executa os comandos PL/SQL e verifica a existência de comandos SQL. Caso existam, são enviados para um executor SQL, que os passa para o banco de dados. O SQL*Plus exibe o resultado na tela do seu computador.

Para visualizar de forma mais clara os conceitos empregados, observe a imagem a seguir onde é mostrado o papel de cada um dentro do contexto de suas aplicações.

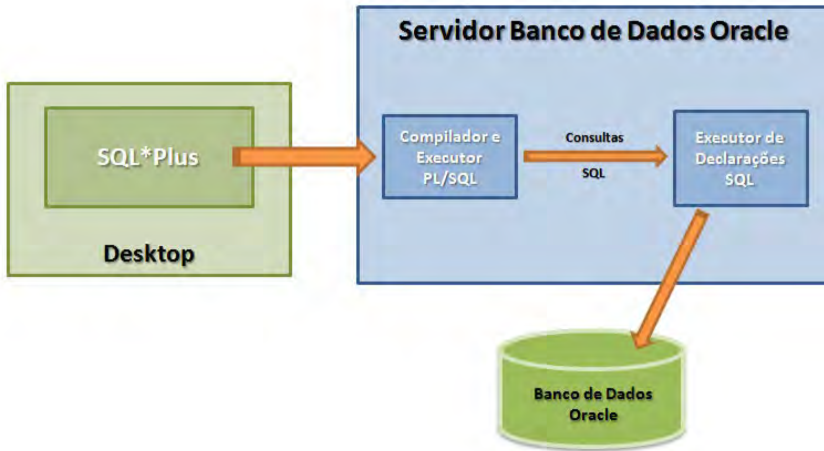

Fig. 1.1: Esquema com as funções do SQL*Plus, PLSQL e SQL

Conforme foi dito, através do SQL*Plus nós podemos entrar com comandos SQL ou blocos PL/SQL. Eles são encaminhados ao servidor do banco de dados, que pode direcionar para o motor PL/SQL, ou seja, o processador que vai validar e executar o bloco PL/SQL, e/ou para o executor de declarações de comandos SQL. Através desta visualização é possível entender a finalidade e importância de cada um desses produtos.

Um ponto importante a entender é como são processadas as instruções SQL e blocos PL/SQL dentro de aplicações desenvolvidas com Oracle Forms ou Reports. Para entendimento, o Oracle Forms e Reports são ferramentas *RAD* para o desenvolvimento de formulários e relatórios. Essas ferramentas se conectam nativamente ao banco de dados Oracle. O termo "nativamente" faz referência a como é realizada a conexão com o banco de dados. Na maioria das vezes, uma aplicação se conecta através de drivers disponíveis pela ferramenta ou por algum gerenciador de conexão, por exemplo, via ODBC (encontrado no Windows). Pois bem, dentro das ferramentas Oracle é possível inserir tanto comandos SQL, quanto blocos PL/SQL. Esses comandos ou blocos serão executados para algum fim e quando isso acontece a solicitação de execução pode ser feita de formas diferentes

Quando temos dentro da aplicação comandos SQL distintos, eles são enviados um a um para o servidor do bando de dados. Dessa forma, a aplicação envia um comando para o servidor, espera a resposta e depois envia outro.

Quando temos blocos PL/SQL, eles são enviados por completo ao servidor do banco de dados, não importando o seu teor. Dentro deste bloco podemos ter vários comandos SQL e demais estruturas em PL/SQL. Desse modo economizamos tempo, pois a aplicação envia de uma só vez todas as solicitações, e o número de respostas esperadas também reduz muito. Reduzindo o número de respostas e tráfego de informações entre aplicação e o servidor do banco de dados aumentamos a chance de ganho de desempenho, principalmente se esta comunicação depender de uma rede cliente *x* servidor. Veja a seguir a ilustração deste conceito.

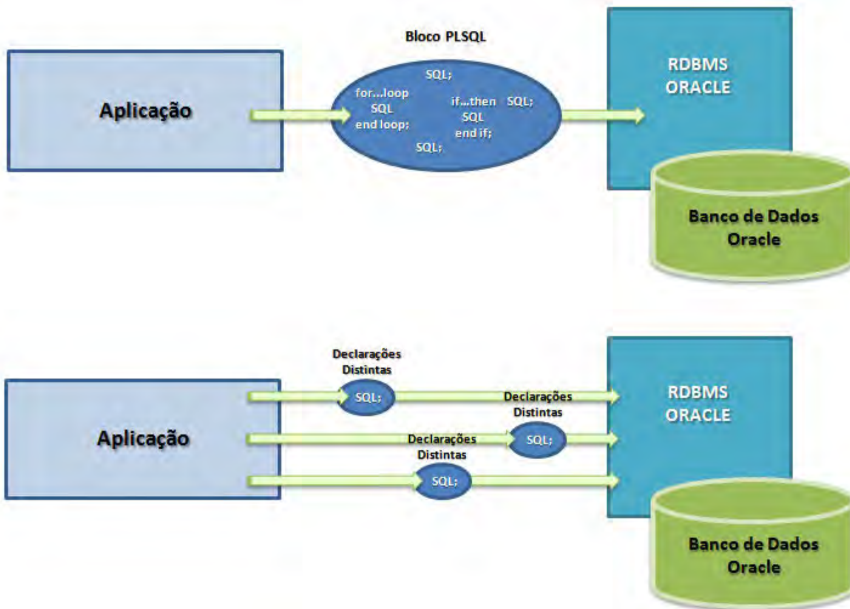

Fig. 1.2: Diferença entre comandos SQL e Blocos PLSQL

Capítulo 2

Programação em bloco

A linguagem PL/SQL trabalha em blocos de comando. Dentro de bloco podemos ter outros blocos, que neste caso são chamados de sub-blocos. Quando queremos escrever um programa para um determinado fim, utilizamos blocos para estruturar os comandos e a forma de como este programa vai se comportar.

Um bloco PL/SQL é iniciado pela expressão `begin` e é finalizada por `end`. Estas duas expressões determinam a área do nosso bloco. Veja um exemplo de bloco PL/SQL.

```
SQL> begin
  2
  3  end;
  4

SQL>
```

Como mencionado anteriormente, podemos ter blocos dentro de outros blocos, os quais são chamados de sub-blocos.

```
SQL> begin
  2
  3    begin
  4
  5    end;
  6
  7  end;
  8

SQL>
```

Além das expressões de delimitação do bloco, também podemos ter o `declare`, que é utilizado para delimitar uma área para declaração de variáveis que serão utilizadas pela aplicação e também a expressão `exception`, que delimita uma área para tratamento de erros. Basicamente, um bloco PL/SQL é composto por:

- Área de declaração de variáveis (`declare`);
- Área de escopo para inserção de comandos e demais sub-blocos (`begin-end`);
- Área de tratamento de erros.

A seguir um bloco PL/SQL contemplando estas premissas básicas.

```
SQL> declare
  2
  3  begin
  4
  5    begin
  6
  7    end;
  8
  9  exception
 10    when others then
```

```
 11
 12  end;
 13
 14

SQL>
```

Este tipo de bloco é chamado de bloco anônimo, pois não possuem um cabeçalho e também não podem ser gravados diretamente no banco de dados. Caso você queira executá-los novamente, é necessário salvar em um arquivo, pois ao fechar a ferramenta ele não é guardado.

A programação em bloco torna os programas mais estruturados e limpos. Principalmente, quando utilizamos vários sub-blocos para a realização de determinadas tarefas distintas, como a seleção de dados, uma atualização ou exclusão. Fazendo desta forma, é possível tratar cada bloco buscando uma programação mais lógica e que possa fornecer informações sobre os comandos contidos nele ou até mesmo identificar possíveis erros que possam surgir. Veja um exemplo de programa.

```
SQL> declare
  2    soma number;
  3  begin
  4    soma := 45+55;
  5    dbms_output.put_line('Soma :'||soma);
  6  exception
  7    when others then
  8     raise_application_error(-20001,'Erro ao somar valores!');
  9  end;
 10

SQL>
```

Neste programa, é possível verificar a existência de uma área de declaração de variáveis onde temos a variável `soma` declarada como `number` (veremos estes conceitos mais adiante); uma área onde são inseridos os comandos propriamente ditos, `soma` recebendo a soma de 45+45. Também temos o pacote `dbms_output`, utilizado quando queremos escrever algo na tela. Neste caso, estamos solicitando a escrita do resultado da soma. Por último, temos

uma área de declaração de erros onde podemos tomar alguma ação caso um erro aconteça. Neste exemplo, caso surja qualquer tipo de erro, chamamos o procedimento `raise_application_error`, que faz com que a aplicação pare e imprima uma mensagem na tela.

Vale lembrar que as áreas de declaração e tratamento de erros podem existir também dentro dos sub-blocos.

Ok. Nosso programa já está criado. Agora vamos ver como executá-lo. Para isso, podemos utilizar a ferramenta SQL*Plus. Através do comando barra / podemos executar um bloco PL/SQL.

```
SQL> declare
  2     soma number;
  3   begin
  4     soma := 45+55;
  5     dbms_output.put_line('Soma :'||soma);
  6   exception
  7     when others then
  8      raise_application_error(-20001,'Erro ao somar valores!');
  9   end;
 10   /
Soma :100

Procedimento PL/SQL concluído com sucesso.

SQL>
```

O comando / executa o bloco PL/SQL, mostrando a saída gerada pelo `dbms_output`. Blocos PL/SQL não necessariamente possuem uma saída impressa na tela. Uma mensagem indicando que o bloco foi executado com sucesso também é mostrada. Caso ocorra algum erro de sintaxe na construção do comando, ou seja, algum comando escrito incorretamente ou uma função utilizada indevidamente, ele também é mostrado. A seguir vamos simular um erro de sintaxe para ver como a ferramenta se comporta.

```
SQL> declare
  2     soma number;
  3   begin
  4     soma 45+55;
```

```
  5     dbms_output.put_line('Soma :'||soma);
  6   exception
  7     when others then
  8      raise_application_error(-20001,'Erro ao somar valores!');
  9   end;
 10   /
  soma 45+55;
       *
ERRO na linha 4:
ORA-06550: linha 4, coluna 8:
PLS-00103: Encontrado o símbolo "45" quando um dos seguintes
símbolos era esperado:
:= . ( @ % ;
O símbolo ":=" foi substituído por "45" para continuar.


SQL>
```

Quando o comando é executado, a ferramenta SQL*Plus faz várias verificações, inclusive a de sintaxe de comandos. Na linha 4 está faltando o comando de atribuição `:=` que atribui a soma de 45+55 à variável `soma`. Com isso, um erro é gerado e mostrado na tela. Ele geralmente vem acompanhado de algumas informações como a linha e a coluna onde o erro está ocorrendo. Além disso, a ferramenta costuma clarear o motivo pelo qual o erro está ocorrendo. Neste nosso exemplo, ele diz que foi encontrado o símbolo "45" quando na verdade era esperado algum outro, por exemplo, o símbolo de atribuição.

Note que este erro foi detectado antes mesmo de o Oracle executar o bloco. Agora vamos gerar um erro, mas não de sintaxe, mas sim, um erro referente a dados incorretos. Vamos tentar somar números com caracteres alfanuméricos.

```
SQL> declare
  2     soma number;
  3   begin
  4     soma := 45+'A';
  5     dbms_output.put_line('Soma :'||soma);
  6   exception
```

```
  7     when others then
  8      raise_application_error(-20001,'Erro ao somar valores!');
  9   end;
 10   /
declare
*
ERRO na linha 1:
ORA-20001: Erro ao somar valores!
ORA-06512: em line 8


SQL>
```

Agora temos outro erro, entretanto, ele ocorreu quando o bloco foi executado. Neste caso, o programa transferiu a ação para a área de tratamento de erros, gerando uma saída que informa a circunstância em que erro aconteceu.

Já vimos que para construir um programa em PL/SQL temos que trabalhar em nível de bloco. Assim sendo, a linguagem PL/SQL permite adicionar, dentro das estruturas destes blocos, todo e qualquer recurso para que este programa possa executar ações ou procedimentos servindo.

Dentro dos blocos é possível declarar variáveis e constantes, executar comandos DML (`select`, `delete`, `update` e `insert`), executar procedimentos armazenados, funções, utilizar estruturas de repetição, estruturas de condição, além do uso de operadores relacionais e numéricos.

Capítulo 3

Primeiros passos

3.1 Como iniciar no PL/SQL

Uma das dificuldades que encontramos quando estamos aprendendo uma linguagem de programação é saber por onde começar. No caso do aprendizado da PL/SQL, não seria diferente. É uma situação normal. Até sentirmos segurança e termos conhecimento suficiente, é interessante termos um roteiro contendo os primeiros passos para iniciar um programa PL/SQL.

Demonstro aqui uma técnica que utilizo bastante, mesmo tendo um bom conhecimento na linguagem. Na verdade, já está implícito na minha forma de pensar, tanto que acabo executando-a mentalmente, enquanto escrevo nesta linguagem. Este método não foi retirado de nenhum livro, foi algo que, entendendo a lógica, fui seguindo e deu certo. Espero que ajude vocês.

Vamos tomar como exemplo, o enunciado a seguir:

Escreva um programa PL/SQL que imprima na tela os nomes os empregados de um determinado gerente e de uma determinada localização.

Primeiro passo

Identifico no enunciado as fontes de dados, ou seja, as tabelas que farão parte do programa, caso seja necessário.

Caso tenha dúvida em identificar a fonte de dados, veja o documento: 24. Após identificar as tabelas, monto os `selects` e executo os comandos para trazer os dados que o programa solicita. Tudo isso sem escrever uma única linha em PL/SQL. Somente monto os `selects` e os executo para ver se os dados estão sendo retornados. Exemplo:

```
SQL> select ename, job, dname
  2  from   emp, dept
  3  where  emp.deptno = dept.deptno
  4  and    dept.loc    = 'CHICAGO' -- será o parâmetro referente
                                       a localização
  5  and    emp.mgr     = '7698'    -- será o parâmetro referente
                                       ao gerente
  6  /

ENAME      JOB       DNAME
---------- --------- --------------
ALLEN      SALESMAN  SALES
WARD       SALESMAN  SALES
MARTIN     SALESMAN  SALES
TURNER     SALESMAN  SALES
JAMES      CLERK     SALES
```

Tendo montado todos os `selects` de que o programa necessita, vamos para o próximo passo.

Segundo passo

Inicio a montagem da estrutura do programa PL/SQL.

Neste caso, analiso que tipo de objeto o programa pede. Geralmente, o enunciado traz esta informação, por exemplo, "faça um bloco PL/SQL anônimo", uma `procedure`, uma `function`, uma `package` etc. No nosso

exemplo, ele não menciona. Como ele não pede para retornar informações, apenas imprimir na tela, não criarei uma function, por exemplo. Como também não menciona nada sobre programas armazenados, também não criarei uma procedure, muito menos uma package. Como também não menciona disparos de triggers, não será preciso criar um. Vou criar um bloco PL/SQL anônimo, mesmo porque, se for o caso, mais adiante eu posso criar um cabeçalho para tornar este objeto um programa armazenado. É simples.

Começo desenhando o bloco PL/SQL, seguindo a seguinte estrutura:

```
SQL> declare
  2
  3  begin
  4
  5
  6  exception
  7    when others then
  8
  9  end;
 10  /
```

Esta não é a estrutura mínima de um PL/SQL, pois sabemos que a área de declaração de variáveis (declare) e a área de tratamento de erros (exception) não são obrigatórias, embora esta última seja imprescindível, pois o tratamento de erros é fundamental para o bom funcionamento do sistema. Contudo, na grande maioria dos programas teremos esta estrutura: a área de declaração de variáveis, a área onde ficarão a maioria dos comandos, propriamente ditos, (begin- end), e a área de tratamento de erros (exception).

Pronto. A estrutura inicial do seu programa PL/SQL está pronta para ser utilizada. Dentro da área de declaração de variáveis, você vai colocar todas as que você vai utilizar dentro do seu programa. Aqui também estarão declarados os tipos de dados definidos por você, os cursores, exceptions de usuário, funções e procedimentos etc. Caso seu programa não vá utilizar esta área, ela pode ser excluída.

Já dentro do corpo do programa, begin-end, é onde você vai escrever a parte principal dele. É nele onde se localizam e serão executados os comandos

da SQL, condições `if`s, laços de repetição, aberturas de cursor, chamadas a funções e outros procedimentos armazenados, e assim por diante.

A área de tratamento de erros é onde você vai tratar os possíveis problemas que poderão surgir durante a execução do seu programa. Quando estiver escrevendo um programa PL/SQL sempre trate os possíveis erros. Pelo menos a *exception* `others` deve ser tratada, para que ao sinal de um problema, o programa não aborte. Também é recomendada a utilização das funções `sqlerrm` e `sqlcode`, para que o motivo do erro seja mencionado na mensagem.

Um programa PL/SQL sempre seguirá esta estrutura. Vale ressaltar também que podemos ver esta estrutura *declare-BEGIN-exception-END*, de forma encadeada, ou seja, um bloco dentro de outro, como forma de isolar determinadas informações ou realizar verificações dentro do programa. Contudo, a estrutura lógica é a mesma.

Segue o programa completo:

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4              ,pmgr   number) is
  5      select ename, job, dname from emp, dept
  6      where  emp.deptno = dept.deptno
  7      and    dept.loc    = pdname
  8      and    emp.mgr     = pmgr;
  9    --
 10    r1 c1%rowtype;
 11  begin
 12    open c1( pmgr   => 7698, pdname => 'CHICAGO');
 13    loop
 14      fetch c1 into r1;
 15      --
 16      if c1%found then
 17        dbms_output.put_line('Nome: '||r1.ename||'
                                   Cargo: '||r1.job);
 18      else
 19        --
 20        exit;
```

```
21        end if;
22      end loop;
23      --
24      close c1;
25    exception
26      when others then
27        dbms_output.put_line('Erro: '||sqlerrm);
28    end;
29    /
Nome: ALLEN Cargo: SALESMAN
Nome: WARD Cargo: SALESMAN
Nome: MARTIN Cargo: SALESMAN
Nome: TURNER Cargo: SALESMAN
Nome: JAMES Cargo: CLERK

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Observação

Para garantir a impressão na tela, através do pacote `dbms_output`, execute o seguinte comando no SQL*Plus: `set serveroutput on`.

Capítulo 4

Pacote dbms_output

Este pacote possui funções e procedimentos que permitem a geração de mensagens a partir de blocos anônimos de PL/SQL, `procedures`, `packages` ou `triggers`. Ele utiliza-se de um buffer em memória para transferência destas mensagens na sessão onde o programa está sendo executado. Quando um programa envia mensagens através do pacote `dbms_output`, elas são armazenadas na área de buffer e são apresentadas apenas ao término do programa.

Se estivermos utilizando a ferramenta SQL*Plus, pode-se habilitar este recurso digitando o seguinte comando: `set serveroutput on`. Fazendo isso, todas as mensagens passarão a ser visualizadas no *prompt* da ferramenta. Na tabela a seguir, serão vistos os componentes mais usados deste pacote, todos do tipo `procedure`.

- `enable`: habilita a chamada das demais rotinas do pacote.

- `disable`: desabilita a chamada das demais rotinas do pacote.
- `put`: inclui uma informação na área de buffer.
- `put_line`: inclui uma informação na área de buffer e adiciona, simultaneamente, um caractere para quebra de linha (linha nova).
- `get_line`: recupera uma linha do buffer.
- `get_lines`: recupera várias linhas do buffer.

Agora, será abordado em detalhes, com scripts exemplos, como são utilizados estes componentes.

enable

Esta procedure habilita chamadas para `put`, `put_line`, `new_line`, `get_line` e `get_lines`. Deve ser especificado um tamanho para a área do buffer a ser usada, definido em bytes e podendo variar de 2.000 até 1.000.000. Se isso for ultrapassado, uma mensagem de erro será mostrada.

```
SQL> begin
  2    dbms_output.enable(2000);
  3    dbms_output.put_line ('TESTE');
  4  end;
  5  /
TESTE

Procedimento PL/SQL concluído com sucesso.

SQL>
```

disable

Esta procedure desabilita as chamadas para `put`, `put_line`, `new_line`, `get_line` e `get_lines` e limpa o buffer. É muito útil na depuração de programas, quando for indesejado o surgimento de mensagens informativas.

```
SQL> begin
  2    dbms_output.disable;
  3    dbms_output.put_line ('TESTE');
  4  end;
  5  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que a mensagem “TESTE” não apareceu na tela.

put

Esta procedure recebe um parâmetro cujo valor será armazenado na área do buffer imediatamente após a última informação. Não é incluído qualquer caractere indicativo de fim de linha (`enter`). É bom lembrar que estes valores de saída são transformados em strings (caractere), portanto, se desejar que eles tenham uma formatação diferente será preciso fazê-lo através do comando `to_char` ou `to_number`, por exemplo. Esta procedure por si só não imprime na tela. Por isso, podemos utilizá-la juntamente com a procedure `new_line`. Neste exemplo, como forma de armazenamento no buffer, utilizamos vários procedimentos `put`, um para cada letra. Veja o exemplo a seguir.

```
SQL> begin
  2    dbms_output.put('T');
  3    dbms_output.put('E');
  4    dbms_output.put('S');
  5    dbms_output.put('T');
  6    dbms_output.put('E');
  7    dbms_output.new_line;
  8  end;
  9  /
TESTE

Procedimento PL/SQL concluído com sucesso.

SQL>
```

put_line

Este procedimento envia o parâmetro informado para a área de buffer, acrescentando, automaticamente, um caractere indicativo de fim de linha após o texto enviado. Com isso, o resultado é impresso na tela, sem a necessidade da execução de qualquer outro procedimento. Note que em cada execução do `put_line` o buffer é limpo.

```
SQL> begin
  2     dbms_output.put_line('T');
  3     dbms_output.put_line('E');
  4     dbms_output.put_line('S');
  5     dbms_output.put_line('T');
  6     dbms_output.put_line('E');
  7  end;
  8  /
T
E
S
T
E

Procedimento PL/SQL concluído com sucesso.

SQL>
```

get_line

Este procedimento permite ler do buffer uma única linha de cada vez. Ele possui dois parâmetros de saída onde o primeiro retornará o conteúdo da linha (`line`) e o segundo retornará seu status (`status`). O status indica se a linha foi recuperada seguindo os critérios: `1` indica que foi recuperada uma linha do buffer;

```
SQL> set serveroutput off
SQL> begin
  2     dbms_output.enable(2000);
  3     --
  4     dbms_output.put('Como');
```

```
  5     dbms_output.new_line;
  6     dbms_output.put('aprender');
  7     dbms_output.new_line;
  8     dbms_output.put('PLSQL?');
  9     dbms_output.new_line;
 10   end;
 11   /

Procedimento PL/SQL concluído com sucesso.

SQL> set serveroutput on

SQL> declare
  2     var1     varchar2(100) default null;
  3     var2     varchar2(100) default null;
  4     var3     varchar2(100) default null;
  5     status   number        default null;
  6   begin
  7     --
  8     dbms_output.get_line(var1,status);
  9     dbms_output.get_line(var2,status);
 10     dbms_output.get_line(var3,status);
 11     --
 12     dbms_output.put_line('Pergunta: '||var1||' '||var2||'
                              '||var3);
 13   end;
 14   /
Pergunta: Como aprender PLSQL?

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Primeiramente, configuramos a sessão para não mostrar as saídas dos comandos `put` e `put_line`, através do comando `set serveroutput off`. Logo após, no primeiro bloco, habilitamos um buffer de 2000 bytes (linha 2). Nas linhas 4 a 9, temos comandos `put` inserindo caracteres no buffer que acabamos de habilitar. Estamos utilizando também o comando `new_line` para que os caracteres sejam guardados em linhas diferentes.

No segundo bloco, temos declaradas as variáveis `var1`, `var2` e `var3` (linha 2 a 4), que utilizaremos para atribuir os valores do buffer. Também declaramos a variável `status` (linha 5), que será utilizada para completar a chamada da procedure `get_line`. Nas linhas 8 a 10, temos as chamadas à procedure `get_line`, que recuperam as linhas do buffer e atribuem os valores às variáveis declaradas, anteriormente. Na linha 12, utilizamos o comando `put_line` para imprimir na tela o conteúdo das variáveis, ou seja, os mesmos recuperados do buffer. Note que, para que a impressão em tela funcione, executamos o comando `set serveroutput on`, antes da execução do segundo bloco.

get_lines

Este procedimento permite ler várias linhas do buffer utilizando um *array* de caracteres. Ele possui dois parâmetros, um de saída (`lines`) e outro de entrada e saída (`numlines`). O primeiro parâmetro, `lines`, se trata de uma tabela do tipo `varchar2(255)`, podendo ser declarada com tipo `dbms_output.chararr`. Já o parâmetro `numlines` serve tanto para informar a quantidade de linhas que se deseja recuperar, quanto para retornar a quantidade de linhas que realmente foram retornadas após a execução da procedure. Veja o exemplo a seguir.

```
SQL> set serveroutput off
SQL> begin
  2    dbms_output.enable(2000);
  3    --
  4    dbms_output.put('Como');
  5    dbms_output.new_line;
  6    dbms_output.put('aprender');
  7    dbms_output.new_line;
  8    dbms_output.put('PLSQL?');
  9    dbms_output.new_line;
 10  end;
 11  /

Procedimento PL/SQL concluído com sucesso.
```

```
SQL> set serveroutput on
SQL> declare
  2    tab       dbms_output.chararr;
  3    qtlines   number                default 3;
  4    res       varchar2(100)         default null;
  5  begin
  6    --
  7    dbms_output.get_lines(tab,qtlines);
  8    --
  9    dbms_output.put_line(
                    'Retornou: '||qtlines||' registros.');
 10    --
 11    for i in 1..qtlines loop
 12      res := res||' '||tab(i);
 13    end loop;
 14    --
 15    dbms_output.put_line('Pergunta: '||res);
 16    --
 17  end;
 18  /
Retornou: 3 registros.
Pergunta:  Como aprender PLSQL?

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Muito parecido com o exemplo anterior, neste exemplo, a alteração foi substituir as três variáveis `var1`, `var2` e `var3`, pela variável `tab`, do tipo `array` (linha 2). Também declaramos as variáveis `qtlines` e `res` (linhas 3 e 4), sendo que a primeira é usada para passar a quantidade de registros a serem retornados (e consequentemente fazer o retorno) e a segunda apenas para montar a string a ser impressa na tela. Na linha 7, temos a chamada à procedure `get_lines`, que retorna os dados do buffer para dentro de nossa variável `tab`. Já nas linhas 11 a 13, utilizamos uma estrutura `loop`, para ler os dados da variável `array` e concatená-los na variável `res`. Observe que utilizamos a variável `qtlines` (contendo a quantidade de linhas retornadas) para determinar o valor final da faixa para `loop`. Na linha 15, temos a

impressão do conteúdo de `res`.

4.1 Exceções para o pacote dbms_output

Existem duas exceções que podem ocorrer quando utilizamos o pacote `dbms_output`. Seguem informações sobre elas e como tratá-las.

- **ORU-10027** (Overflow de buffer). Solução: aumentar o tamanho do buffer se possível. Caso contrário, encontrar um modo de gravar menos dados.
- **ORU-10028** (Overflow de comprimento de linha, limite de 255 caracteres por linha). Solução: verificar se todas as chamadas feitas a caracteres por linha `put` e `put_line` têm menos de 255 caracteres por linha.

Neste capítulo falamos sobre o `dbms_output` e seus recursos. Durante o todo o livro de PL/SQL vamos utilizar este pacote para gerar as saídas das informações para nossos exemplos. Desta forma, é muito importante conhecê-lo e saber utilizá-lo.

Capítulo 5

Variáveis bind e de substituição

A ferramenta SQL*Plus permite o uso de variáveis com referências do tipo *bind* e de *substituição*.

5.1 Variáveis bind

As variáveis *bind* são declaradas dentro do SQL*Plus e podem ser utilizadas em todo seu ambiente, sendo em comandos SQL ou dentro de programas PL/SQL. A declaração deste tipo de variável é muito semelhante à declaração utilizada no PL/SQL, onde a nomeamos e definimos um tipo para ela. Contudo, não é necessária uma área específica para declaração, bastando apenas declará-la no prompt do SQL*Plus. Veja o exemplo:

```
SQL> variable mensagem varchar2(200)
SQL>
```

Estamos declarando uma variável chamada `mensagem` do tipo `varchar2` com 200 posições. Depois de declarada, é só utilizá-la nos programas.

```
SQL> begin
  2     :mensagem := 'Curso PLSQL';
  3   end;
  4   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que estamos utilizando-a dentro de um bloco PL/SQL, onde atribuímos um valor para ela. Para recuperar ou atribuir um valor para uma variável *bind*, em um programa PL/SQL ou comando SQL, é necessário referenciá-la através da utilização do caractere dois pontos (`:`), colocando-o antes do seu nome. No entanto, quando a definimos ou quando forçamos a impressão do seu valor em tela, não há a necessidade deste caractere especial.

Para visualizar o conteúdo de uma variável *bind* na tela do SQL*Plus, você deve habilitar a impressão através do comando a seguir.

```
SQL> set autoprint on
SQL>
```

Após habilitar a impressão, o conteúdo da variável é impresso logo após a execução do comando ou programa onde ela está sendo utilizada. Veja a execução do exemplo anterior, agora com o `autoprint` ligado.

```
SQL> begin
  2     :mensagem := 'Curso PLSQL';
  3   end;
  4   /

Procedimento PL/SQL concluído com sucesso.


MENSAGEM
```

```
--------------------------------------------
Curso PLSQL

SQL>
```

Veja mais um exemplo, agora utilizando a variável no comando `select`.

```
SQL>  select :mensagem from dual;

:MENSAGEM
----------------------------------------
Curso PLSQL

SQL>
```

Note que, como a variável já havia recebido um valor, quando executamos o bloco PL/SQL, este valor ainda persiste em execuções posteriores.

Para alterar um valor de uma variável *bind* diretamente pelo SQL*Plus, podemos utilizar o comando `exec`. Veja o exemplo a seguir, onde definimos uma variável chamada `gdepno`, e logo após atribuímos o valor 10 a ela.

```
SQL> variable gdepno number
SQL>
SQL> exec :gdepno := 10

Procedimento PL/SQL concluído com sucesso.


   gdepno
----------
        10

SQL>
```

Agora, selecionamos os empregados com base no código do departamento, vindo da variável *bind* `gdepno`.

```
SQL> select ename from emp where deptno = :gdepno;
```

```
ENAME
----------
CLARK
KING
MILLER

SQL>
```

Se quisermos visualizar o conteúdo de uma variável *bind*, utilizamos o comando `print`. Veja o exemplo a seguir:

```
SQL> print gdepno;

   gdepno
----------
        10

SQL>
```

Uma variável *bind* tem sua vida útil com base na sessão do SQL*Plus. Portanto, outras sessões não as enxergam, somente a que as criou. Quando o SQL*Plus é fechado, automaticamente, elas são excluídas da memória.

A utilização de variáveis *bind* tem algumas restrições, por exemplo seu uso na cláusula `from`, que não é permitido, e na substituição de palavras reservadas.

Para ver todas as variáveis *binds* declaradas em uma sessão do SQL*Plus utilize o comando `var`.

```
SQL> var
variável   wdname
Tipo de dados   NUMBER

variável   gnome
Tipo de dados   varCHAR2(100)
SQL>
```

5.2 Variáveis de substituição

Outro tipo de variável de usuário é a variável de substituição. Este tipo também pode ser utilizado em comandos DML ou PL/SQL. Seu objetivo é substituir tal variável, dentro de comandos SQL ou PL/SQL, por um conjunto de caracteres predefinidos ou definidos em tempo de execução. Ao contrário das variáveis *binds*, as de substituição podem ser utilizadas também como forma de completar comandos SQL, pois ela permite a utilização de palavras reservadas em seu teor. Veja a seguir como definir uma variável de substituição.

```
SQL> define wempno = 7369
SQL>
```

Para definir uma variável de substituição utilizamos o comando `define` seguido de um nome para a variável. Neste exemplo, além de definir um nome, atribuímos um valor para a variável através do operador de igualdade (=). Note que para as variáveis de substituição não definimos um tipo, pois são sempre do tipo alfanumérico. Veja um exemplo, utilizando a variável `wempno` que acabamos de definir.

```
SQL> select ename from emp where empno = &wempno;
antigo   1: select ename from emp where empno = &wempno
novo   1: select ename from emp where empno = 7369

ENAME
----------
SMITH

SQL>
```

Note que, para utilizarmos a variável de substituição, colocamos na frente de seu nome o `&`. Esta é a indicação de que estamos utilizando uma variável de substituição. Veja um exemplo utilizando-a em PL/SQL.

```
SQL> begin
  2   for i in (select ename from emp where empno = &wempno) loop
  3      dbms_output.put_line(i.ename);
  4   end loop;
  5  end;
```

```
  6  /
antigo   2:    for i in (
                 select ename from emp where empno = &wempno) loop
novo   2:    for i in (
                 select ename from emp where empno = 7369) loop
SMITH

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Para alterar um valor de uma variável de substituição também utilizamos o comando `define`. Veja a seguir:

```
SQL> define wempno = 10
SQL>
SQL> begin
  2    for i in (
          select ename from emp where empno = &wempno) loop
  3      dbms_output.put_line(i.ename);
  4    end loop;
  5  end;
  6  /
antigo   2:    for i in (
                  select ename from emp where empno = &wempno) loop
novo   2:    for i in (
                  select ename from emp where empno = 10) loop

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Já para sabermos qual o valor corrente de uma variável de substituição, utilizamos o comando `define` seguido do nome da variável.

```
SQL> define wempno
define wempno          = "10" (CHAR)
SQL>
```

Como mencionado anteriormente, as variáveis de substituição podem também ser utilizadas para substituição de palavras reservadas, ou seja, podemos não só substituir tal variável por um valor alfanumérico, como também por uma sentença SQL, como uma clausula `where` ou `order by`. Veja o exemplo:

```
SQL> define gfrom = 'from emp'
SQL> define gwhere = 'where empno = 7369'
SQL> define gorderby = 'order by 1'
SQL>
SQL> select ename &gfrom &gwhere &gorderby;
antigo   1: select ename &gfrom &gwhere &gorderby
novo   1: select ename from emp where empno = 7369 order by 1

ENAME
----------
SMITH

SQL>
```

Isso também funciona para PL/SQL.

```
SQL> begin
  2    for i in (select ename &gfrom &gwhere &gorderby) loop
  3      dbms_output.put_line(i.ename);
  4    end loop;
  5  end;
  6  /
antigo   2:    for i in (
                select ename &gfrom &gwhere &gorderby) loop
novo   2:    for i in (
                select ename from emp where
                empno = 7369 order by 1) loop
SMITH

Procedimento PL/SQL concluído com sucesso.

SQL>
```

A vida útil de uma variável de substituição também será limitada pela sessão do SQL*Plus. Enquanto a sessão estiver ativa, ela existirá. Outras sessões não podem vê-la, como também acontece com as variáveis do tipo *bind*. Uma variável de substituição pode ter sua definição excluída. Para tal ação, podemos utilizar o comando `undefine`. Veja o exemplo.

```
SQL> undefine gfrom
SQL>
```

Ao contrário de uma variável *bind*, uma variável de substituição não necessita ser obrigatoriamente definida ou declarada. Podemos simplesmente informá-la em nosso comando SQL ou PL/SQL, e deixar que o motor do SQL*Plus solicite o valor para ela, sem que seja necessária uma definição prévia. Observe o exemplo a seguir:

```
SQL> select job from emp where empno = &codigo;
Informe o valor para codigo: 7902
antigo   1: select job from emp where empno = &codigo
novo   1: select job from emp where empno = 7902

JOB
---------
ANALYST

SQL> define
define _CONNECT_IDENTIFIER = "xe" (CHAR)
define _SQLPLUS_RELEASE = "902000100" (CHAR)
define _EDITOR          = "Notepad" (CHAR)
define _O_VERSION       = "Oracle Database 10g Express Edition
                           Release 10.2.0.1.0 - Production" (CHAR)
define _O_RELEASE       = "1002000100" (CHAR)
define 1                = "1" (CHAR)
define _RC              = "1" (CHAR)
define wempno           = "1111" (CHAR)
define Gwhere           = "where empno = 7369" (CHAR)
define GORDERBY         = "order by 1" (CHAR)
SQL>
```

Veja que utilizamos uma variável como substituição, chamada `codigo`. Vale ressaltar que não a definimos em nenhum momento. Mesmo assim, o SQL*Plus a reconheceu como uma variável, pois utilizamos o `&` na frente do seu nome, e solicitou um valor para ela. Desta forma, se não tivermos a variável de substituição já definida, o SQL*Plus vai solicitar um valor. Note também que após a sua utilização o SQL*Plus não a deixa definida, ou seja, isso ocorre apenas em tempo de execução. Em PL/SQL também podemos utilizá-la desta forma.

```
SQL> declare
  2    wjob emp.job%type;
  3  begin
  4    select job into wjob from emp where empno = &cod_emp;
  5    --
  6    dbms_output.put_line(wjob);
  7  end;
  8  /
Informe o valor para cod_emp: 7902
antigo   4: select job into wjob from emp where empno = &cod_emp;
novo   4: select job into wjob from emp where empno = 7902;
ANALYST

Procedimento PL/SQL concluído com sucesso.

SQL>
```

5.3 Utilizando variáveis em arquivos

Também podemos utilizar variáveis *binds* ou de substituição dentro de arquivos e depois executá-los via SQL*Plus. Nesses casos, se forem encontradas variáveis do tipo *bind* ou de substituição, o SQL*Plus vai fazer o preenchimento dos valores correspondentes.

Nos casos de variáveis *bind*, o SQL*Plus não solicitará os valores, pois eles já devem estar definidos. No caso de variáveis de substituição, vai depender se já se encontram ou não definidas na sessão. Veja os exemplos:

A seguir estamos criando um arquivo de script chamado `S_EMP.sql` contendo o comando para retornar os empregados com base em um depar-

tamento informado via uma variável *bind*.

```
SQL> select ename, job from emp where deptno = :bdeptno
  2
SQL> save B_EMP.sql
Criado arquivo B_EMP.sql
SQL>
SQL> get B_EMP.sql
  1* select ename, job from emp where deptno = :bdeptno
SQL>
```

Definindo a variável *bind* dentro da sessão do SQL*Plus:

```
SQL> var bdeptno number
SQL>
```

Definindo um valor para a variável *bind* `bdeptno`:

```
SQL>  exec :bdeptno := 10

Procedimento PL/SQL concluído com sucesso.

SQL> print bdeptno

   BDEPTNO
----------
        10

SQL>
```

Executando o arquivo `S_EMP.sql`:

```
SQL> @B_EMP.sql

ENAME      JOB
---------- ---------
CLARK      MANAGER
KING       PRESIDENT
MILLER     CLERK

SQL>
```

Agora vamos ver outro exemplo utilizando variáveis de substituição:

```
SQL> select ename from emp where deptno = &sdeptno
  2
SQL>  save S_EMP.sql
Criado arquivo S_EMP.sql
SQL>
SQL>
```

No caso das variáveis de substituição, conforme já mencionado, mesmo não as definindo, o SQL*Plus faz a solicitação de valores. Vamos testar.

```
SQL> @S_EMP.sql
Informe o valor para sdeptno: 20
antigo   1: select ename from emp where deptno = &sdeptno
novo   1: select ename from emp where deptno = 20

ENAME
----------
SMITH
JONES
SCOTT
ADAMS
FORD

SQL>
```

Note que o SQL*Plus solicitou o valor através da linha "Informe o valor para `sdeptno`:". Neste caso, foi informado o valor 20. Contudo, como a variável não está definida, sempre que executamos este script o programa vai solicitar um valor.

Para que ele não fique solicitando um valor sempre o que script for executado, definimos a variável através do comando `define`. Com isso, o SQL*Plus não pedirá mais o valor. Veja o exemplo:

```
SQL> define sdeptno = 10
SQL> @S_EMP.sql
antigo   1: select ename from emp where deptno = &sdeptno
novo   1: select ename from emp where deptno = 10
```

```
ENAME
----------
CLARK
KING
MILLER

SQL>
```

Repare que agora ele não pede mais o valor, apenas faz a substituição da variável com base no valor guardado na sessão do SQL*Plus. Se excluirmos a definição da variável, ele volta a solicitar o valor.

```
SQL> undefine sdeptno
SQL> @S_EMP.sql
Informe o valor para sdeptno: 20
antigo   1: select ename from emp where deptno = &sdeptno
novo   1: select ename from emp where deptno = 20

ENAME
----------
SMITH
JONES
SCOTT
ADAMS
FORD

SQL>
```

Outra forma de definir uma variável de substituição é utilizando `&&` (duplo) em vez de um só `&`. Isso faz com que o SQL*Plus crie a definição da variável:

```
SQL> edit S_EMP.sql

SQL> get S_EMP.sql
  1* select ename from emp where deptno = &&sdeptno
SQL>
SQL> @S_EMP.sql
```

```
Informe o valor para sdeptno: 10
antigo   1: select ename from emp where deptno = &&sdeptno
novo   1: select ename from emp where deptno = 10

ENAME
----------
CLARK
KING
MILLER

SQL> @S_EMP.sql
antigo   1: select ename from emp where deptno = &&sdeptno
novo   1: select ename from emp where deptno = 10

ENAME
----------
CLARK
KING
MILLER

SQL>
```

Veja que editamos o arquivo `S_EMP` e acrescentamos mais um `&` ao já existente. Depois, salvamos o arquivo. Ao executá-lo, o SQL*Plus detectou os `&&`, contudo, como a variável não continha nenhum valor inicial definido, ele solicitou um valor. Já na segunda vez que executamos o arquivo, o SQL*Plus já não o solicitou. Se executarmos o comando `define` para ver as variáveis de substituição definidas na sessão, veremos que o SQL*Plus definiu automaticamente a nossa variável `SDEPTNO`.

```
SQL> define
define _CONNECT_IDENTIFIER = "xe" (CHAR)
define _SQLPLUS_RELEASE = "902000100" (CHAR)
define _EDITOR          = "Notepad" (CHAR)
define _O_VERSION       = "Oracle Database 10g Express Edition
                           Release 10.2.0.1.0 - Production" (CHAR)
define _O_RELEASE       = "1002000100" (CHAR)
define _RC              = "1" (CHAR)
define 1                = "30" (CHAR)
```

```
define SDEPTNO         = "10" (CHAR)
SQL>
```

Outro recurso que pode ser utilizado é a passagem de valores para as variáveis de substituição quando executamos um arquivo. Este recurso está habilitado somente para as variáveis deste tipo. Neste caso, devemos utilizar indexadores numéricos juntamente com o `&`, como por exemplo, `&1`, `&2` etc. nos comandos dentro do arquivo. Caso contrário, o SQL*Plus vai ignorar este valores, solicitando a entrada deles assim que for executado o arquivo. Veja o exemplo a seguir:

```
SQL> edit S_EMP.sql

SQL> get S_EMP.sql
  1* select ename from emp where deptno = &1
SQL>
```

Editamos o arquivo `S_EMP.sql` modificando o nome da variável de substituição de `sdeptno` para 1. Agora vamos executar o arquivo passando um valor como parâmetro na chamada da execução.

```
SQL> @S_EMP.sql 10
antigo   1: select ename from emp where deptno = &1
novo   1: select ename from emp where deptno = 10

ENAME
----------
CLARK
KING
MILLER

SQL>
```

Veja que o valor 10 informado na chamada da execução do arquivo foi passado para dentro dele, substituindo a variável `&1`. Com o uso de indexadores, o SQL*Plus define automaticamente a variável na sessão, e com isso, nas próximas execuções, não necessitamos informar o valor caso ele não seja diferente da execução anterior. Para trocar o valor da variável, basta informar um novo valor na chamada.

```
SQL> @S_EMP.sql
antigo   1: select ename from emp where deptno = &1
novo   1: select ename from emp where deptno = 10

ENAME
----------
CLARK
KING
MILLER

SQL>  @S_EMP.sql 20
antigo   1: select ename from emp where deptno = &1
novo   1: select ename from emp where deptno = 20

ENAME
----------
SMITH
JONES
SCOTT
ADAMS
FORD

SQL>
```

Note que, na primeira execução, apenas chamamos o arquivo sem a passagem de um valor. Como anteriormente já havíamos executado o arquivo passando o valor 10, ele assumiu este valor para as demais execuções. Na segunda execução, informamos um valor diferente, 20, fazendo com que ele seja passado como parâmetro a partir de então.

Outro comando que é utilizado para definir uma variável de substituição é o `accept`. A diferença dele para `define` é que podemos formatar uma mensagem para ser mostrada ao usuário no momento de solicitar a entrada de um valor. Sua utilização é muito interessante quando estamos executando scripts via arquivo. Veja o exemplo.

```
SQL> edit S_EMP.sql

SQL> get S_EMP.sql
```

```
  1  accept SDEPTNO number for 999 default 20 prompt "Informe o
  deptno: "
  2* select ename from emp where deptno = &SDEPTNO
SQL>
SQL> @S_EMP.sql
Informe o deptno: 20
antigo   1: select ename from emp where deptno = &SDEPTNO
novo   1: select ename from emp where deptno =         20

ENAME
----------
SMITH
JONES
SCOTT
ADAMS
FORD

SQL>
```

Em primeiro lugar, alteramos o arquivo `S_EMP` acrescentando a linha referente ao `accept`, que deve vir antes do comando SQL ao qual se quer associar a mensagem e a variável. Neste nosso exemplo, o comando está antes do `select`. Logo após o comando `accept`, informamos o nome para variável, seguido pelo tipo do dado. Escolhemos uma variável do tipo `number` que vai se chamar `SDEPTNO`. Logo após o tipo da variável, podemos informar também a máscara utilizando o comando `for` seguido do formato. No exemplo, colocamos o formato como 999 (máximo de 3 casas). Também definimos o valor 20 como padrão, através da expressão `default`, ou seja, a variável já será inicializada com este valor. E agora vem o suprassumo do recurso. Através do comando `prompt`, definimos uma mensagem para ser mostrada para o usuário no momento da solicitação dos valores. Esta mensagem deve ser informada sempre logo após o comando. Por fim, executamos o arquivo.

Capítulo 6

Aspectos iniciais da programação PL/SQL

Como em qualquer linguagem de programação, em PL/SQL também há regras que devem ser seguidas na hora de codificar programas. Cada linguagem de programação trabalha de uma forma, contudo certos conceitos prevalecem entre todas.

Aspectos relacionados ao desenvolvimento, como áreas predefinidas para declarações de variáveis, tratamentos de erros ou comentários em programas, por exemplo, são comuns em todas as linguagens. O que difere entre elas é a forma como cada uma trata estes aspectos.

Para programar em PL/SQL, você deve conhecer sua sintaxe e os elementos que podem ser utilizados na codificação dos programas. Veja agora alguns pontos importantes para você iniciar na escrita de códigos nesta linguagem.

6.1 Caracteres e operadores

Dentro da linguagem você pode utilizar caracteres e operadores para auxiliar na codificação. São eles:

- **Caracteres:** A a Z (maiúsculos e minúsculos), números de 0 a 9 e os caracteres especiais: ( ) + - * / < > = ! ; : . ' @ % , "$ & _ \{ } [ ] | #
- **Operadores Relacionais:** <, >, =, !=, >=, <=, IS NULL, IS NOT NULL
- **Operadores Lógicos:** AND, OR e NOT

6.2 Identificadores

Para nomear identificadores em PL/SQL, por exemplo, variáveis, constantes ou qualquer objeto, nós devemos seguir algumas regras. São elas:

- A quantidade de caracteres para nomear um identificador é de no máximo 30.
- Não podemos utilizar palavras reservadas como *begin*, *if*, *loop*, *end* etc.
- Para nomear os identificadores, podemos utilizar letras, números e alguns caracteres especiais, mas nem todos.
- Obrigatoriamente, o primeiro caractere em um nome de identificador deve ser uma letra.

Escopo de identificadores

O escopo de um identificador está limitado ao bloco onde foi declarado. Podemos ter vários blocos encadeados. Logo, cada bloco pode ter sua própria área de declaração de identificadores. Neste caso, um identificador, por exemplo, uma variável, declarada em um bloco mais interno, não poderá ser acessada em um bloco mais externo. Veja o exemplo a seguir.

```
SQL>
create or replace procedure folha_pagamento(pqt_dias number) is
```

```
2     --
3     wqt_dias     number;
4     wvl_bruto    number;
5     wvl_ir       number;
6     wvl_liquido  number;
7     --
8   begin
9     wvl_bruto   := (pqt_dias * 25);
10    --
11    declare
12       wtx_ir number;
13    begin
14      if wvl_bruto > 5400 then
15        --
16        wtx_ir := 27;
17        dbms_output.put_line('Taxa IR: '||wtx_ir);
18        --
19      else
20        --
21        wtx_ir := 8;
22        dbms_output.put_line('Taxa IR: '||wtx_ir);
23        --
24      end if;
25      --
26      wvl_ir      := (wvl_bruto * wtx_ir) / 100;
27      --
28      wvl_liquido := (wvl_bruto - wvl_ir);
29      --
30    end;
31    --
32    dbms_output.put_line(
         'Valor do salario bruto: '||wvl_bruto);
33    dbms_output.put_line(
         'Desconto do valor do IR: '||wvl_ir);
34    dbms_output.put_line(
         'Valor do salario liquido: '||wvl_liquido);
35    --
36  exception
37    when others then
```

```
 38      dbms_output.put_line(
          'Erro ao calcular pagamento. Erro: '||sqlerrm);
 39   end folha_pagamento;
 40   /

Procedimento criado.

SQL>
```

Nesse exemplo, temos um programa que calcula o valor a ser pago a um empregado, mediante a quantidade de horas passada por parâmetro a um procedimento. Uma das premissas para calcular o valor líquido a ser pago é o cálculo do imposto de renda (IR). Se analisarmos o programa, é possível identificar dois blocos. Um principal e outro mais interno. O bloco interno é utilizado para o cálculo do IR. Note que nele criamos uma segunda área de declaração de variáveis que é específica deste bloco. Ali declaramos a variável `wtx_ir` que receberá a taxa para o cálculo do imposto mediante uma condição. A partir da obtenção da taxa, é calculado o valor do imposto e também o valor líquido a ser pago.

Observações importantes: o escopo da variável `wtx_ir` está limitado ao bloco em que ela foi declarada. Caso a usássemos fora deste escopo, o Oracle geraria um erro indicando que ela não foi declarada. Em contrapartida, podemos observar que as variáveis `wvl_ir` e `wvl_liquido` estão sendo utilizadas dentro do bloco mais interno, sem que nenhum erro aconteça. Isso é possível devido ao fato de elas terem sido declaradas em um bloco mais externo. Logo, seus escopos abrangem todo o bloco onde foram declaradas, inclusive, blocos internos contidos neste. Segue o resultado.

```
SQL> begin folha_pagamento(pqt_dias => 300); end;
  2  /
Taxa IR: 27
Valor do salario bruto: 7500
Desconto do valor do IR: 2025
Valor do salario liquido: 5475

Procedimento PL/SQL concluído com sucesso.
```

```
SQL>
```

6.3 Transações

Primeiramente, antes de começarmos a trabalhar com PL/SQL, é preciso conhecer o conceito de transação. Uma das características mais bem-vindas dos bancos de dados Cliente/Servidor, em relação aos bancos de dados desktop, é o conceito de transações. Uma transação é uma unidade lógica de trabalho composta por uma ou mais declarações da *Data Manipulation Language* (DML) ou *Data Definition Language* (DDL). Os comandos para o controle da transação são aqueles necessários para que se possa controlar a efetivação ou não das modificações feitas no banco de dados.

Para explicar o que é uma transação, pode-se tomar como exemplo uma transferência bancária onde um determinado valor é transferido de uma conta para outra. O processo de transferência deste valor consiste em vários passos, que são agrupados de modo que, se não forem concluídos em sua totalidade, ou seja, se a transação não chegar até o final, todas as outras alterações já realizadas serão descartadas e a conta voltará ao seu estado anterior, como se nenhuma transferência tivesse sido realizada. Através deste controle, pode-se garantir que os dados permanecerão consistentes. Os comandos `commit` e `rollback` da DCL auxiliam neste trabalho.

Commit

Tornam permanentes todas as alterações feitas no banco de dados durante a sessão. Todas as alterações realizadas em uma determinada transação serão confirmadas caso este comando seja aplicado.

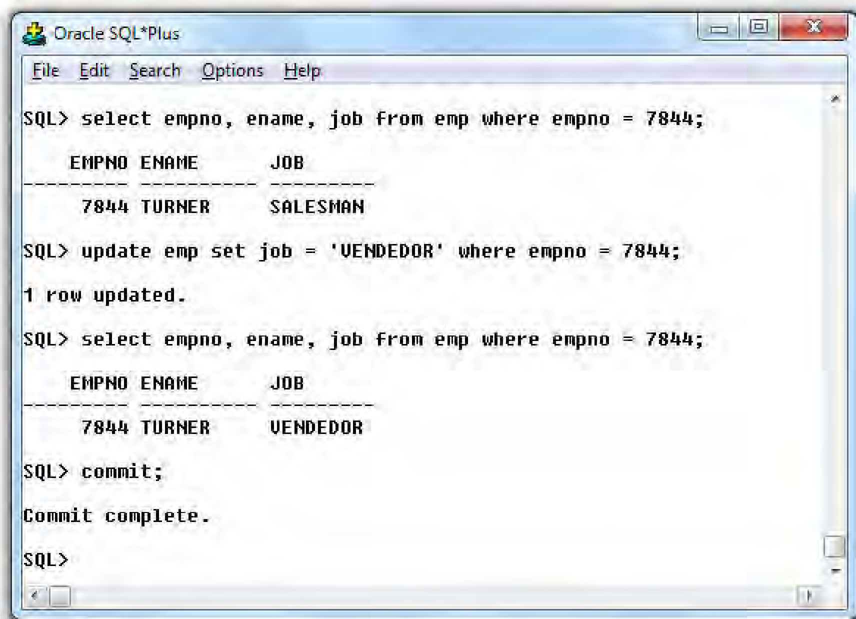

Fig. 6.1: Confirmando as alterações no banco de dados

Rollback

Usado para desfazer todas as alterações feitas desde o último *commit* durante a sessão. Esse comando vai restaurar os dados ao lugar onde eles estavam no último commit. Alterações serão desfeitas caso a transação seja encerrada pelo comando `rollback`.

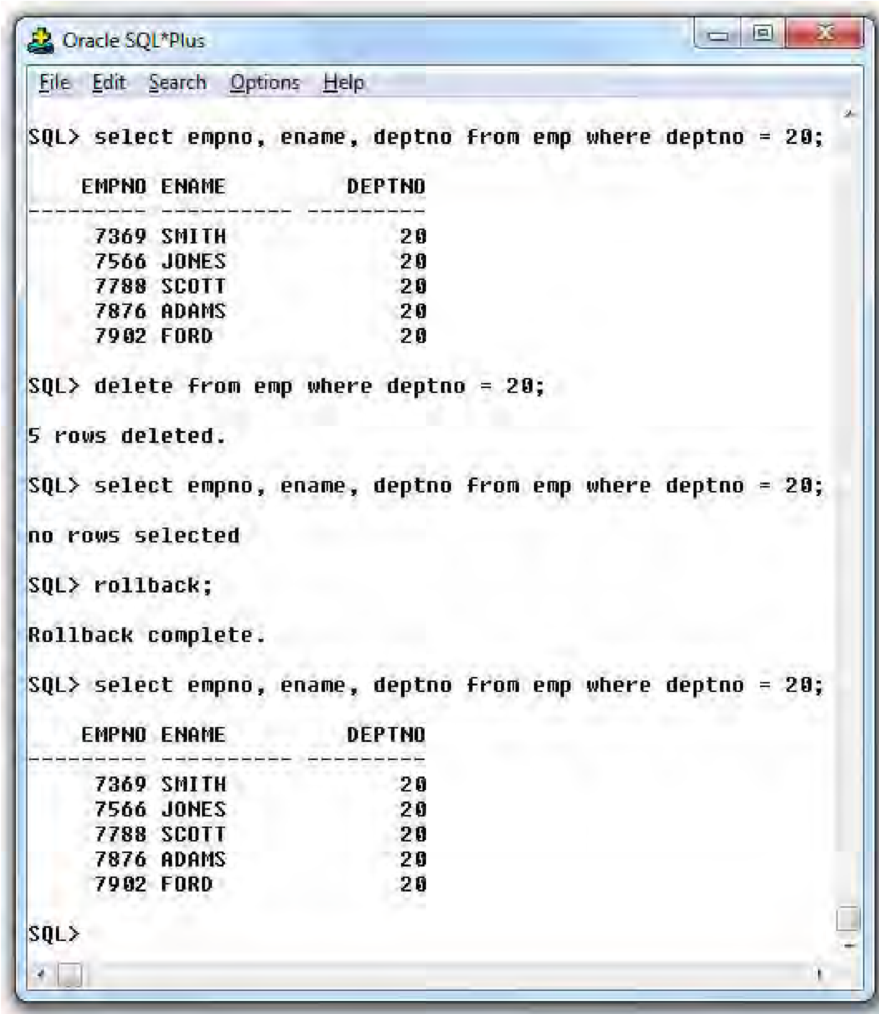

Fig. 6.2: Desfazendo as alterações no banco de dados

No Oracle, uma transação se inicia com a execução da primeira instrução SQL e termina quando as alterações são salvas ou descartadas. O comando `set transaction` também inicia uma transação – transação explícita. O uso do comando `set transaction` determina algumas regras, as quais são listadas a seguir.

- Deve ser o primeiro comando da transação (caso contrário, ocorrerá um erro);
- Somente consultas são permitidas na transação;
- Um `commit`, `rollback` ou qualquer outro comando de DDL (possuem commits implícitos) encerram o efeito do comando `set transaction`.

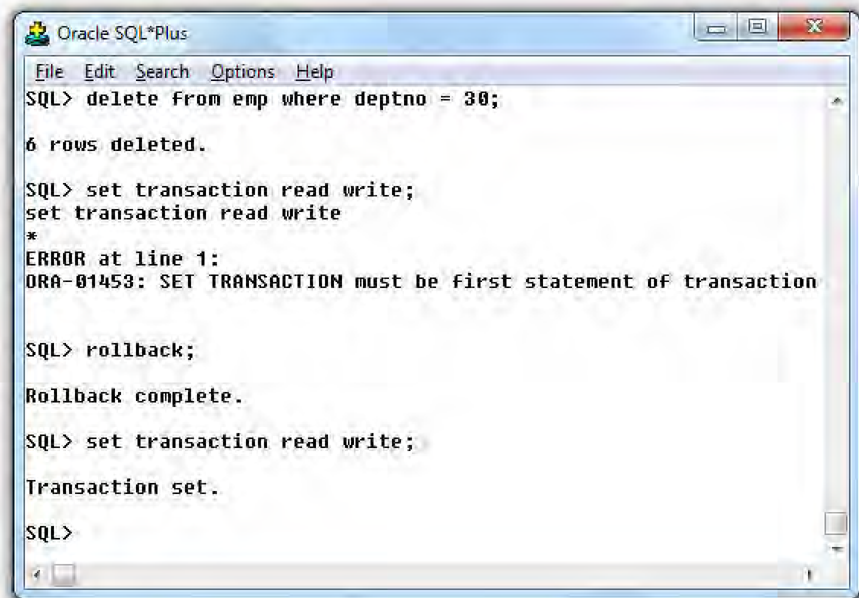

Fig. 6.3: Abrindo uma nova transação dentro do SQL*Plus

Nota: A figura mostra um comando DML sendo executado e, logo após, uma transação sendo aberta. Como uma das regras para se abrir uma transação é que ela seja um dos primeiros comandos da transação, o erro acontece quando executamos o comando. Note que após encerrarmos a transação (através do `rollback`) aberta pelo comando DML, nós conseguimos executar o comando e abrir uma nova transação.

6.4 Transações em PL/SQL

As transações em PL/SQL seguem os mesmo preceitos expostos anteriormente. Um ponto a acrescentar é o uso dos `savepoints`. Quando trabalhamos com PL/SQL temos este recurso que nos permite definir pontos de salvamento dentro do programa. Todavia, não é uma prática muito utilizada, principalmente pelo fato de geralmente trabalharmos com grandes volumes de código, pois pode dificultar o entendimento do programa e inclusive desestruturá-lo logicamente. É sempre bom ter muito cuidado quanto à efetivação de dados no momento de construirmos nossos programas. Devemos visar sempre à consistência e integridade das informações manipuladas a fim de atingirmos o objetivo esperado. Além de permitir que sejam salvas partes de ações em determinados pontos do programa, também é possível desfazer tais ações usando estes mesmos pontos de salvamento.

Veja um exemplo simples do uso deste recurso.

```
SQL> begin
  2    insert into dept values (41, 'GENERAL LEDGER', '');
  3    savepoint ponto_um;
  4    --
  5    insert into dept values (42, 'PURCHASING', '');
  6    savepoint ponto_dois;
  7    --
  8    insert into dept values (43, 'RECEIVABLES', '');
  9    savepoint ponto_tres;
 10    --
 11    insert into dept values (44, 'PAYABLES', '');
 12    rollback to savepoint ponto_dois;
 13    --
 14    commit;
 15  end;
 16  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo temos vários pontos de salvamento identificados por

`ponto_um`, `ponto_dois` e `ponto_tres`. Cada ponto está referenciando um comando `insert`. Logo após, temos um `rollback to savepoint` desfazendo todas as operações a partir do `ponto_dois`. Isso indica que os comandos referentes às linhas 2 e 5 permanecem intactos. Já os comandos referentes às linhas 8 e 11 serão descartados.

6.5 Trabalhando com variáveis e constantes

Identificadores de variáveis e constantes são muito comuns em programas PL/SQL, pois toda a informação geralmente é manipulada dentro deles e na maioria das vezes precisamos armazená-la em memória antes de, por exemplo, inseri-la em uma tabela, enviar para uma impressora ou para exportá-la para algum outro sistema.

Dessa forma, variáveis e constantes nada mais são que áreas em memória definidas e usadas dentro de programa que servem para guardar informações em tempo de execução. Como o nome sugere, uma variável pode ter seus valores atualizados várias vezes ao longo da execução de um programa. Contudo, uma constante nasce com um valor definido e o mantém até o término da sua utilização.

Uma variável ou constante deve ter um tipo definido, que não se altera ao longo da execução de um programa. Ao declará-las, recebem um nome que será sua identificação. Por regra, dentro do Oracle, este nome não pode ultrapassar 30 caracteres. No mais, ela deve obedecer às regras de definição mencionadas anteriormente quando falamos de identificadores. Esta definição deve ser feita dentro de uma área específica de declarações de variáveis que cada bloco PL/SQL pode ter, identificada como `declare`.

O escopo de uma variável ou constante se limita ao bloco em que foram declaradas, podendo sempre ser acessadas no seu bloco e em blocos mais internos, nunca em blocos externos à sua declaração. Entretanto, caso uma variável tenha sido declarada duas vezes (mesmo nome), uma em um bloco externo e outra em um bloco mais interno, a variável do bloco externo não poderá ser acessada dentro do interno, pois haveria um conflito. Todavia, pelas boas práticas de programação não devemos ter identificadores com o mesmo nome. Logo, casos como este não são comuns.

Além dos tipos predefinidos, como numérico, data, string, entre outros, as variáveis e constantes podem referenciar tipos de dados de determinadas colunas de uma tabela criada em um banco de dados (%TYPE), ou seu tipo pode referenciar uma tabela inteira (%ROWTYPE). Quando declaramos uma variável podemos definir um valor padrão, chamado de DEFAULT, para que ela já inicialize assim. Para as constantes, isso é obrigatório.

No mais, os valores de variáveis e constantes podem ser manipulados e convertidos para outros tipos a fim de serem utilizados dentro dos programas para satisfazer objetivos específicos. Seguem exemplos de declarações de variáveis.

```
DECLARE
  DT_ENTRADA            DATE       DEFAULT SYSDATE;
  DT_SAIDA              DATE;
  FORNECEDOR            TIPO_PESSOA; -- Tipo de dado definido pelo
                                        desenvolvedor
  QT_MAX                NUMBER(5)  DEFAULT    1000;
  QT_MIN      CONSTANT  NUMBER(50) DEFAULT     100;
  NM_PESSOA             CHAR(60);
  VL_SALARIO            NUMBER(11,2);
  CD_DEPTO              NUMBER(5);
  IN_NAO     CONSTANT   BOOLEAN DEFAULT FALSE;
  QTD                   NUMBER(10)  := 0;
  VL_PERC    CONSTANT   NUMBER(4,2) :=  55.00;
  CD_CARGO              EMPLOYEE.JOB%TYPE;
  REG_DEPT              DEPARTMENT%ROWTYPE;
```

6.6 Tipos de dados em PL/SQL

VARCHAR2

Tamanho máximo para campos de tabela: 4.000 bytes. Tamanho máximo para PL/SQL: 32.767 bytes. O VARCHAR2 é variável e somente usa o espaço que está ocupado. Diferentemente do CHAR.

VARCHAR é um subtipo (assim como STRING) que existe por questões de compatibilidade com outras marcas de banco de dados e também com o padrão SQL. Entretanto, a Oracle no momento não recomenda o uso do

tipo `VARCHAR` porque sua definição deve mudar à medida que o padrão SQL evoluir. Deve-se usar `VARCHAR2`.

CHAR

Tamanho máximo: 2.000 bytes

O tipo `CHAR` é usado para conter dados de string de comprimento fixo. Ao contrário das strings de `VARCHAR2`, uma string `CHAR` sempre contém o número máximo de caracteres.

Outros tipos:

- `NCHAR` Tamanho máximo: 2.000 bytes;
- `NCHAR VARYING` Tamanho máximo: 4.000 bytes;
- `CHAR VARYING` Tamanho máximo: 4.000 bytes.

NUMBER(p,s)

Numérico com sinal e ponto decimal, sendo que **p** é a precisão de 1 a 38 dígitos e **s** é a escala, de -84 a 127.

Este tipo também possui subtipos como:

- `DECIMAL`: igual a NUMBER
- `DEC`: igual a DECIMAL
- `DOUBLE PRECISION`: igual a NUMBER
- `NUMERIC`: igual a NUMBER
- `REAL`: igual a NUMBER
- `INTEGER`: equivalente a NUMBER(38)
- `INT`: igual a INTEGER
- `SMALLINT`: igual a NUMBER(38)
- `FLOAT`: igual a NUMBER

- `FLOAT(prec)`: igual a NUMBER(prec), mas a precisão é expressa em termos de bits binários, não de dígitos decimais. A precisão binária pode variar de 1 até 126.
- `BINARY_INTEGER`: semelhante a INTEGER. É usada para indexar tabela PL/SQL.

DATE

`1 JAN 4712 BC` até `31 DEC 4712 AD` (DATA com hora, minuto e segundo) O tipo `DATE` é usado para armazenar os valores de data e hora. Um nome melhor seria `DATETIME` porque o componente de hora está sempre lá, independente de você usá-lo ou não. Se não for especificada a hora ao atribuir um valor para uma variável deste tipo, o padrão de meia-noite (12:00:00 a.m.) será usado.

BOOLEAN

True e *False*.

LONG

Tamanho máximo para tabela: 2 GB. Tamanho máximo para PLSQL: 32.760. Somente pode existir uma coluna por tabela.

RAW

Tamanho máximo para campos de tabela: 2.000 bytes. Tamanho máximo para PL/SQL: 32.767. `LONG RAW` é outro tipo parecido com `RAW`, a diferença é que ele possui 7 bytes a menos quando utilizado em PL/SQL.

CLOB

Tamanho máximo: `(4 GB - 1) * DB_BLOCK_SIZE`. Parâmetro de inicialização: 8 TB a 128 TB. O número de colunas `CLOB` por tabela é limitado somente pelo número máximo de colunas por tabela.

Armazena textos, que são validados conforme o set de caracteres, ou seja, armazena acentuação etc.

Tipos LOB, surgiram em substituição aos tipos LONG e LONG RAW, pois eles só permitiam uma coluna por tabela. Já os tipos LOB permitem mais de uma coluna.

NCLOB

Tamanho máximo: (4 GB - 1) * DB_BLOCK_SIZE. Parâmetro de inicialização: 8 TB a 128 TB. O número de colunas NCLOB por tabela é limitado somente pelo número máximo de colunas por tabela.

Objeto de grande capacidade de caracteres nacionais – contém até 4 GB de caracteres de bytes simples ou caracteres multibyte que atendem o conjunto de caracteres nacional definido pelo banco de dados Oracle.

BLOB

Tamanho máximo: (4 GB - 1) * DB_BLOCK_SIZE. Parâmetro de inicialização: 8 TB a 128 TB. O número de colunas BLOB por tabela é limitado somente pelo número máximo de colunas por tabela.

Armazenam dados não estruturados como: som, imagem, dados binários.

BFILE

Tamanho máximo: 4 GB. Tamanho máximo para o nome do arquivo: 255 caracteres. Tamanho máximo para o nome do diretório: 30 caracteres. O valor máximo de BFILEs é limitado pelo valor do parâmetro de inicialização SESSION_MAX_OPEN_FILES, o qual é limitado pelo número máximo de arquivos abertos que o sistema operacional suporta.

ROWID

É um tipo especial usado para armazenar os ROWIDs (endereços físicos) das linhas armazenadas em uma tabela.

Campos LONG

Em resumo, os tipos comumente utilizados são: CHAR, VARCHAR2, NUMBER, DATE, BOOLEAN e os da família LOB.

No entanto, existem algumas restrições para campos `LONG` e `LONG RAW`.

- Não se pode criar um `OBJECT TYPE` com o atributo de `LONG`.
- Uma coluna `LONG` não pode estar dentro da cláusula `WHERE` ou com referência integral dos dados, exceto `NULL` ou `NOT NULL`.
- Uma função não pode retornar um campo `LONG`.
- Uma tabela poderá ter somente um campo `LONG`.
- `LONG` não pode ser indexada.
- `LONG` não pode usar cláusulas `WHERE`, conforme já mencionado, `GROUP BY`, `ORDER BY` e `CONNECT BY`.

Uma dica para você usar um campo `LONG` na cláusula `WHERE` é criar uma tabela temporária com os campos da tabela original, mas alterando um tipo `LONG` para `CLOB`. Também é possível alterar diretamente na tabela o campo `LONG` para `CLOB`, caso não tenha problema de alterar a estrutura da tabela original.

Capítulo 7

Exceções

Exceções são utilizadas dentro do Oracle quando algum erro acontece. Quando construímos programas em PL/SQL é fundamental tratarmos as exceções que podem ocorrer mediante a execução do sistema e seus diversos processos. Basicamente, existem dois tipos de exceções dentro do Oracle: as predefinidas e as definidas pelo usuário.

- **Exceções predefinidas**: são exceções existentes implicitamente dentro do Oracle e que são disparadas automaticamente por ele quando ocorre um erro no programa.
- **Exceções definidas pelo usuário**: são exceções que precisam ser declaradas e disparadas pelo usuário. O Oracle desconhece sua existência.

7.1 Exceções predefinidas

Como foi dito, uma exceção é acionada quando um erro acontece dentro do programa. Mesmo que não as tratemos, o Oracle intercepta o problema e mostra o erro. Todavia, se deixarmos que o Oracle faça isto nós podemos ter sérios problemas, pois erros que não são tratados causam a parada do sistema, ou seja, o programa é abortado.

Dentro da PL/SQL tratamos estes erros através das `exceptions`. Os mais variados tipos de erros podem ser tratados através deste recurso. Por exemplo, se temos dentro do nosso programa PL/SQL um cálculo onde possa acontecer uma divisão por zero, caso ocorra, um erro é disparado, pois sabemos que matematicamente não é possível resolvê-lo. A PL/SQL sabe disso e vai gerar uma exceção caso isto ocorra. Mediante isto, podemos nos antecipar e tratar este possível erro.

Um ponto bom de tratarmos os erros é a possibilidade de mostrarmos mensagens mais amigáveis aos usuários e lhes propor ações que possam ajudá-los a resolver sem a intervenção da área de suporte, por exemplo. Além do mais, evitamos que o programa ou todo o sistema seja abortado, o que geraria um grande incômodo. Através do tratamento de exceções também é possível determinar se o programa pode continuar ou não com a ação após o erro ter ocorrido.

Como este acionamento é feito?

Quando acontece um desses erros predefinidos pelo Oracle, automaticamente a PL/SQL percebe sua ocorrência e o transfere para uma área de tratamento de erro. No entanto, às vezes esses erros demonstram certa falta de clareza em suas descrições, envolvendo termos muito técnicos, geralmente em inglês, o que pode confundir o usuário. Com isso em mente, eles podem ser tratados pelo desenvolvedor de sistemas a fim de torná-los mais compreensíveis. Contudo, o mesmo não acontece com os erros causados pelos usuários. Este tipo de erro a PL/SQL não consegue detectar pelo fato de não estarem incluídos implicitamente no Oracle. O uso das exceções é bem flexível, podendo ser usadas dentro de blocos anônimos, `procedures`, `functions`, `packages` e `triggers`.

Primeiramente, vamos falar sobre as `exceptions` predefinidas. Segue uma lista de `exceptions` que podem ser utilizadas em PL/SQL.

- `no_data_found`: dispara quando um `select into` é executado e não retorna nenhum registro. Não ocorre se usarmos funções de grupo, pois estas retornam valores nulos quando não há registros. Também não ocorre em `fetch` (usado em cursores). Neste caso usamos atributos de cursor. Exemplo: `C1%NO* TFOUND` retorna verdadeiro se não encontrar mais linhas.
- `invalid_cursor`: tentamos usar um cursor que não está aberto.
- `invalid_number`: quando a conversão de um tipo para outro não é possível, ou quando um valor ultrapassa o número de casas definidas para o tipo de dado. Exemplo: uma variável `number(2)` recebe o valor 100.
- `login_denied`: tentativa de conectar ao banco de dados com usuário inválido.
- `cursor_already_open`: tentamos abrir um cursor que já se encontra aberto.
- `dup_val_on_index`: tentativa de inserir valor duplicado em um índice único. Usado na verificação de chave primária ou única.
- `not_logged_on`: tentamos usar algum recurso do banco sem estarmos conectados.
- `program_error`: erro interno ao sistema Oracle, chamado de `internal`.
- `rowtype_mismatch`: quando um `fetch` retorna uma determinada linha do banco de dados para uma variável do tipo registro onde os dados desta linha não são compatíveis com os tipos definidos na variável.
- `timeout_on_resource`: tempo expirou quando o banco esperava pelo recurso.

- `too_many_rows`: um `select into` retornou mais de uma linha.
- `value_error`: em algumas circunstâncias é disparado `invalid_number` ou `value_error`.
- `zero_divide`: tentamos dividir por zero.
- `others`: quando não sabemos qual erro pode ocorrer.

Esta é uma lista das `exceptions` mais comuns. Para maiores detalhes e demais exceções, consulte a documentação do banco de dados da versão que estiver utilizando.

Vamos abrir um parêntese para falar um pouco sobre a exceção `others` e sobre as exceções `invalid_number` e `value_error`.

A `exception others` é muito utilizada, pois mesmo que tratemos erros específicos sempre a utilizamos para garantir que demais erros não causem a parada do sistema. Qualquer `exception` quando não tratada em específico acaba sendo amparada pela exceção `others`. Portanto, quando tratamos diversas exceções devemos sempre colocá-la por último na seção de tratamento. Outro ponto altamente recomendado é que pelo menos a exceção `others` seja sempre tratada, independente se já estamos tratado de outras em específico.

Já com relação às exceções `invalid_number` e `value_error`, embora, possam parecer redundantes, existem lugares específicos onde elas podem ser usadas. `invalid_number` é acionada por erros em comandos SQL dentro dos blocos PL/SQL. Já a exceção `value_error` é disparada por erros em comandos PL/SQL.

Veja onde e como tratar as exceções. Analise o caso a seguir.

```
SQL> declare
  2    wempno number;
  3  begin
  4    --
  5    select empno
  6    into   wempno
  7    from   emp
  8    where  deptno = 9999;
```

```
  9     --
 10   end;
 11   /
declare
*
ERRO na linha 1:
ORA-01403: dados não encontrados
ORA-06512: em line 5


SQL>
```

Neste exemplo estamos tentando selecionar o empregado com o código 9999. Entretanto, este empregado não existe. Como estamos utilizando um `select into`, sabemos que quando um `select` deste tipo não retorna linhas o Oracle dispara uma exceção. Neste caso, foi o que aconteceu. Ao tentarmos selecionar a linha, o comando não a localizou fazendo com que o Oracle gerasse uma exceção. O resultado foi uma mensagem de erro informando que os dados não foram encontrados. Contudo, estamos falando de um exemplo simples. Se tivéssemos um programa com muitos comandos `select`, ficaria difícil saber onde estaria o erro. Sem falar que a mensagem não ajudaria muito o usuário a resolver o problema por si só. Agora vamos tratar esta exceção e ver o que acontece.

```
SQL> declare
  2     wempno number;
  3   begin
  4     --
  5     select empno
  6     into   wempno
  7     from   emp
  8     where  deptno = 9999;
  9     --
 10   exception
 11     when no_data_found then
 12       dbms_output.put_line('Empregado não encontrado.');
 13     when others then
 14       dbms_output.put_line('Erro ao selecionar empregado.');
```

```
 15  end;
 16  /
Empregado não encontrado.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Primeiramente, vamos analisar o programa. Como pode ser visto, entre o `begin` e o `end`, temos a expressão `exception`. É nesta área que definimos quais exceções vamos tratar. Perceba que logo após a expressão `exception` temos várias cláusulas `when` indicando quais exceções estamos tratando. Neste exemplo, estamos tratando as exceções `no_data_found` e `others`. O tratamento de exceções deve sempre acompanhar um bloco PL/SQL e seu escopo é válido somente dentro dele. Veja a sintaxe.

```
declare
  -- variáveis
begin
  --
  -- comandos
  begin
    -- comandos
  exception
    when <exception 1> then
      -- comandos ou mensagens.
    when <exception n> then
      -- comandos ou mensagens.
  end;
  -- mais comandos
  --
exception
  when <exception 1> then
    -- comandos ou mensagens.
  when <exception n> then
    -- comandos ou mensagens.
end;
```

Veja neste exemplo que temos um bloco dentro de outro, e cada um possui

sua área de tratamento de exceções. Note também que esta é definida sempre no fim do bloco. Muito bem, continuando nosso exemplo, podemos perceber que nele temos dois tratamentos. Um para verificar se o empregado existe e outro para o caso de acontecer qualquer outro erro de que não se tenha conhecimento.

De propósito, tentamos selecionar os empregados que são de um departamento que não existe na tabela EMP para que a exceção fosse acionada. Com o acionamento da exceção, a mensagem "Empregado não encontrado" foi impressa na tela, atingindo nosso objetivo. Com isso, foi possível mostrar uma mensagem muito mais inteligível e de quebra não permitimos que o Oracle abortasse o programa. Utilizamos o pacote `dbms_output` para imprimir na tela a nossa mensagem de erro.

Neste exemplo, vamos colocar um código de departamento que existe na tabela EMP.

```
SQL> declare
  2    wempno number;
  3  begin
  4    --
  5    select empno
  6    into   wempno
  7    from   emp
  8    where  deptno = 30;
  9    --
 10  exception
 11    when no_data_found then
 12      dbms_output.put_line('Empregado não encontrado.');
 13    when others then
 14      dbms_output.put_line('Erro ao selecionar empregado.');
 15  end;
 16  /
Erro ao selecionar empregado.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Analisando este segundo exemplo podemos observar que a exceção

no_data_found não foi acionada. Entretanto a exceção others foi. Também pudera, quando colocamos um código de departamento existente na tabela EMP, o select retornou várias linhas. Várias linhas em um select into geram uma exceção chamada too_many_rows. Desta forma vamos incluí-la em nosso programa.

```
SQL> declare
  2    wempno number;
  3  begin
  4    --
  5    select empno
  6    into   wempno
  7    from   emp
  8    where  deptno = 30;
  9    --
 10  exception
 11    when no_data_found then
 12      dbms_output.put_line('Empregado não encontrado.');
 13    when too_many_rows then
 14      dbms_output.put_line(
           'Erro: O código de departamento informado'||
 15        ' retornou mais de um registro.');
 16    when others then
 17      dbms_output.put_line('Erro ao selecionar empregado.');
 18  end;
 19  /
Erro: O código de departamento informado retornou mais de um
registro.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Incluindo a exceção too_many_rows, foi possível tratar o erro em específico e melhorar a mensagem de erro. Vale salientar que a exception foi incluída antes da exceção others. Isso sempre deve ser feito desta forma.

Voltando um pouco no exemplo anterior a este, foi possível observar que sem o tratamento da exceção too_many_rows, ficou difícil identificar qual

o problema ocorreu, pois como a exceção `others` é para todas as exceções que podem ocorrer dentro do programa, não conseguimos especificar uma mensagem de erro clara o suficiente. É óbvio que nesse exemplo, sabíamos qual era o problema e alteramos o programa para tratá-lo. Contudo, mesmo assim, outros tipos de problemas podem ocorrer fora os tratados especificamente. Caso isso aconteça, é necessário pelo menos uma mensagem mais detalhada sobre o erro, para que o problema possa ser identificado e resolvido.

Para identificar erros não conhecidos, aqueles que acabam caindo na exceção `others`, podemos contar com duas variáveis que são alimentadas cada vez que uma `exception` é disparada. As variáveis são `sqlcode` e `sqlerrm`. A primeira mostra somente o código e a segunda mostra o código e a descrição do erro, vindos do Oracle. Cada erro dentro Oracle possui um código associado a ele. Desta forma, quando uma `exception` é gerada, o código do erro corrente é gravado na variável `sqlcode`. A mesma coisa acontece com a variável `sqlerrm`. Cada erro possui uma descrição explicando seu motivo. Ao ser gerada uma exceção, esta variável recebe a descrição do erro corrente. Com isto, é possível obter o código e descrição do erro, o que possibilita identificar sua causa. Estas variáveis podem e devem ser usadas quando tratamos a exceção `others`.

Vale salientar, entretanto, que na maioria das vezes a descrição dos erros vindos do Oracle não são descrições muito amigáveis. Os usuários podem ter dificuldades em identificar o que exatamente ocorreu. Por isso a importância de tratarmos pelo menos as exceções conhecidas dentro de um programa, a fim de clarear ao máximo os erros que podem ocorrer. Vamos voltar ao exemplo anterior e usar as variáveis para identificar o problema ocorrido.

```
SQL> declare
  2    wempno number;
  3  begin
  4    --
  5    select empno
  6    into   wempno
  7    from   emp
  8    where  deptno = 30;
  9    --
 10  exception
```

```
11    when no_data_found then
12      dbms_output.put_line('Empregado não encontrado.');
13    when others then
14      dbms_output.put_line(
          'Erro ao selecionar empregado. Erro: '||sqlerrm
15        ||' - Código: ('||sqlcode||').');
16  end;
17  /
Erro ao selecionar empregado. Erro: ORA-01422: a extração exata
retorna mais do
que o número solicitado de linhas - Código: (-1422).

Procedimento PL/SQL concluído com sucesso.

SQL>
```

No exemplo a seguir concatenamos as variáveis `sqlcode` e `sqlerrm` com nossa mensagem de erro. Uma observação, a variável `sqlerrm` já traz o código do erro, não tendo necessidade de utilizar o `sqlcode`, a menos, é claro, que você queira usar este código para algo diferente de apenas mostrar a informação. Para mostrar apenas a descrição do erro use `sqlerrm(sqlcode)`. Note também que a informação vinda do banco de dados não é muito clara. Dependendo da linguagem instalada para banco de dados, o Oracle procura traduzir a mensagem de erro e às vezes torna mais difícil sua compreensão.

Uma dica importante: caso você não compreenda a mensagem de erro ou ela não exista, é possível, através do código do erro, pesquisar maiores detalhes na documentação do banco Oracle ou na internet. Estes códigos são padronizados e em qualquer versão ou linguagem eles se mantêm os mesmos.

Uma vez gerada a exceção, isso não garante que o programa vá parar o processo. Como foi visto, podemos ter vários blocos dentro do mesmo programa PL/SQL e cada um destes blocos podem ter tratamentos de erros. Estes blocos, por sua vez, podem conter dependências uns com os outros e para isto pode ser necessário parar o processo quando um erro acontece, não permitindo que o programa continue.

> **Nota:** o fato de nós tratarmos as `exceptions` nos remete a outro ponto a que devemos estar atentos. Quando não a tratamos, o Oracle realiza este tratamento e aborta o programa. Já quando o tratamento fica por nossa conta, ou melhor, por conta do nosso programa, somos nós quem define se haverá ou não uma parada no sistema ou no processo que está sendo executado, ou seja, quando tratamos um erro, o Oracle passa a não tratá-lo mais e as ações ficam por nossa conta.

Veja o exemplo a seguir.

```
SQL> declare
  2    wsal     number;
  3    wdeptno number;
  4  begin
  5    --
  6    begin
  7      select avg(sal), deptno
  8      into   wsal, wdeptno
  9      from   emp
 10      where  deptno = 99
 11      group  by deptno;
 12    exception
 13      when no_data_found then
 14        dbms_output.put_line(
             'Valores não encontrados para o departamento 99.');
 15      when others then
 16        dbms_output.put_line('Erro ao selecionar valores
             referentes ao depto. 99. '||
 17                    'Erro: '||sqlerrm||'.');
 18    end;
 19    --
 20    begin
 21      insert into emp (empno, ename, job, mgr, hiredate, sal,
                         comm, deptno)
 22         values (8002, 'ANGELINA', 'MANAGER', 7839,
                   TO_DATE('20/10/2011','DD/MM/RRRR'),
                   wsal, null, 20);
```

```
23        --
24        commit;
25        --
26      exception
27        when others then
28          dbms_output.put_line(
           'Erro ao inserir novo empregado. '||
29                        'Erro: '||sqlerrm||'.');
30      end;
31      --
32      dbms_output.put_line(
           'Empregado ANGELINA inserido com sucesso.');
33      --
34    exception
35      when no_data_found then
36        dbms_output.put_line('Empregado não encontrado.');
37      when too_many_rows then
38        dbms_output.put_line(
         'Erro: O código de departamento informado'||
39                             ' retornou mais de um registro.');
40      when others then
41        dbms_output.put_line(
         'Erro ao inserir empregado. Erro: '||sqlerrm
42                 ||' - Código: ('||sqlcode||').');
43    end;
44    /
Valores não encontrados para o departamento 99.
Empregado ANGELINA inserido com sucesso.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo criamos um bloco PL/SQL que insere um novo empregado. Dentro do nosso programa temos um `select` que busca a média dos salários dos empregados de um determinado departamento. Neste `select` estamos tratando as exceções `no_data_found` e `others`. Logo após, realizamos a inclusão do empregado através do comando `insert` informando como salário deste empregado o valor vindo do `select` anterior. Depois

informamos ao usuário que o empregado foi inserido com sucesso. Veja que para o comando `insert` também estamos tratando a exceção `others` para o caso de algum erro.

Entretanto, executando o programa podemos observar que algumas mensagens foram geradas, inclusive a de que os valores dos salários não foram encontrados para o departamento em questão. O resultado disso foi a inclusão do empregado com o salário igual a zero.

```
SQL> select * from emp order by empno;

     EMPNO ENAME      JOB              MGR HIREDATE        SAL
---------- ---------- --------- ---------- -------- ----------
      7499 ALLEN      SALESMAN        7698 20/02/81       1600
      7521 WARD       SALESMAN        7698 22/02/81       1250
      7654 MARTIN     SALESMAN        7698 28/09/81       1250
      7698 BLAKE      MANAGER         7839 01/05/81       2850
      7782 CLARK      MANAGER         7839 09/06/81       2450
      7839 KING       PRESIDENT            17/11/81       5000
      7844 TURNER     SALESMAN        7698 08/09/81       1500
      7900 JAMES      CLERK           7698 03/12/81        950
      7934 MILLER     CLERK           7782 23/01/82       1300
      7935 PAUL       SALESMAN        7698 15/03/80       1000
      8001 SILVESTER  MANAGER         7839                1000
      8002 ANGELINA   MANAGER         7839 20/10/11

      COMM     DEPTNO
---------- ----------
       306         30
       510         30
      1428         30
       285         30
       245         10
       500         10
       150         30
        95         30
       130         10
       100         30
                   80
                   20
```

```
12 linhas selecionadas.

SQL>
```

Podemos agravar a situação colocando a coluna `SAL` como `not null`, para que surja um erro no momento da inserção do registro.

```
SQL> alter table emp modify sal number(7,2) not null;

Tabela alterada.

SQL> desc emp
 Nome                                      Nulo?    Tipo
 ----------------------------------------- -------- ----------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                       NOT NULL NUMBER(7,2)
 COMM                                               NUMBER(7,2)
 DEPTNO                                             NUMBER(2)
 PC_COM_SAL                                         NUMBER

SQL>
```

Vamos excluir o registro existente e executar o programa novamente.

```
SQL> delete from emp where empno  = 8002;

1 linha deletada.

SQL> commit;

Validação completa.

SQL>

SQL> declare
  2     wsal    number;
```

```
3     wdeptno number;
4   begin
5     --
6     begin
7       select avg(sal), deptno
8       into   wsal, wdeptno
9       from   emp
10      where  deptno = 99
11      group  by deptno;
12    exception
13      when no_data_found then
14        dbms_output.put_line(
           'Valores não encontrados para o departamento 99.');
15      when others then
16        dbms_output.put_line('Erro ao selecionar valores
           referentes ao depto. 99. '||
17                      'Erro: '||sqlerrm||'.');
18    end;
19    --
20    begin
21      insert into emp (empno, ename, job, mgr, hiredate, sal,
                              comm, deptno)
22         values (8002, 'ANGELINA', 'MANAGER', 7839,
                   TO_DATE('20/10/2011','DD/MM/RRRR'),
 wsal, null, 20);
23      --
24      commit;
25      --
26    exception
27      when others then
28        dbms_output.put_line(
             'Erro ao inserir um novo empregado. '||
29                      'Erro: '||sqlerrm||'.');
30    end;
31    --
32    dbms_output.put_line(
       'Empregado ANGELINA inserido com sucesso.');
33    --
34  exception
```

```
35    when no_data_found then
36      dbms_output.put_line('Empregado não encontrado.');
37    when too_many_rows then
38      dbms_output.put_line(
        'Erro: O código de departamento informado'||
39                             ' retornou mais de um registro.');
40    when others then
41      dbms_output.put_line(
        'Erro ao inserir empregado. Erro: '||sqlerrm
42                ||' - Código: ('||sqlcode||').');
43  end;
44  /
```

```
Valores não encontrados para o departamento 99.
Erro ao inserir um novo empregado. Erro: ORA-01400: não é
possível inserir NULL em ("TSQL"."EMP"."SAL").
Empregado ANGELINA inserido com sucesso.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Veja que o programa nos deu várias mensagens. A primeira informa que não conseguiu recuperar o salário referente ao departamento encontrado. A segunda nos alerta que o empregado não pode ser inserido pelo fato de um erro ter ocorrido. Note que usamos a exceção `others` com as variáveis de erro, que nos apontou que não é possível inserir valores nulos para o campo `SAL` da tabela EMP. Isso aconteceu devido à alteração que realizamos. Entretanto, veja que a mensagem nos informando que o empregado foi inserido com sucesso também é mostrada. Isso é um erro, pois na verdade ele não foi inserido. Neste caso, temos que tratar estas situações no programa para que as ações e mensagens estejam condizentes com que realmente está acontecendo.

Se nós partimos do pressuposto de que nenhum dos empregados pudesse ser incluído sem um valor de salário definido, teríamos que alterar o programa para que ele não incluísse o empregado caso não conseguisse recuperar o salário. Pois bem, sem alterar a estrutura do nosso programa vamos utilizar um recurso da PL/SQL que permite parar a execução do programa quando for necessário.

Quando desejarmos que um programa ou processo seja interrompido após uma exceção, utilizamos a chamada `raise_applications_error`. Esta chamada faz com que o programa pare sua execução a partir de um determinado ponto. Em contrapartida, ela desvia o programa para o primeiro tratamento de erro existente, que se encaixe com o erro ocorrido, seguindo a hierarquia dos blocos. No geral, este desvio sempre é interceptado pela exceção `others`.

Esta chamada requer dois parâmetros, um código de erro e uma descrição. Como já foi dito, a Oracle mantém uma padronização de códigos de erros para que o mesmo erro seja identificado em qualquer banco de dados, em qualquer versão ou idioma, seja aqui ou do outro lado do mundo. Logo, estes códigos não podem ser usados pelos desenvolvedores para customizar erros. Em contrapartida ela disponibiliza uma faixa de código que vai a partir do número -20000 até -20999, para que utilizemos nestes casos. Seria uma faixa de códigos de erros customizável. A descrição do erro fica por nossa conta. Vamos testar a alteração do programa.

```
SQL> declare
  2    wsal    number;
  3    wdeptno number;
  4  begin
  5    --
  6    begin
  7      select avg(sal), deptno
  8      into   wsal, wdeptno
  9      from   emp
 10      where  deptno = 99
 11      group  by deptno;
 12    exception
 13      when no_data_found then
 14        raise_application_error(-20000,
          'Valores não encontrados para o departamento 99.');
 15      when others then
 16        dbms_output.put_line('Erro ao selecionar valores
                               referentes ao depto. 99. '||
 17                     'Erro: '||sqlerrm||'.');
 18    end;
```

```
19      --
20      begin
21        insert into emp (empno, ename, job, mgr, hiredate, sal,
                           comm, deptno)
22          values (8002, 'ANGELINA', 'MANAGER', 7839,
                    TO_DATE('20/10/2011','DD/MM/RRRR'),
wsal, null, 20);
23        --
24        commit;
25        --
26      exception
27        when others then
28          raise_application_error(-20000,'Erro ao inserir um
                                    novo empregado. '||
29                        'Erro: '||sqlerrm||'.');
30      end;
31      --
32      dbms_output.put_line(
         'Empregado ANGELINA inserido com sucesso.');
33      --
34    exception
35      when no_data_found then
36        dbms_output.put_line('Empregado não encontrado.');
37      when too_many_rows then
38        dbms_output.put_line(
           'Erro: O código de departamento informado'||
39                              ' retornou mais de um registro.');
40      when others then
41        dbms_output.put_line(
            'Erro ao inserir empregado. Erro: '||sqlerrm
42                ||' - Código: ('||sqlcode||').');
43    end;
44    /
Erro ao inserir empregado. Erro: ORA-20000: Valores não
encontrados para o departamento 99.
 -
Código: (-20000).

Procedimento PL/SQL concluído com sucesso.
```

```
SQL>
```

Analisando o resultado da execução, podemos perceber que ocorreu um erro na busca pelo valor do salário, onde não foi encontrado. Isso gerou a exceção `no_data_found`. Através do tratamento desta exceção foi chamado o `raise_application_error`, que gerou uma exceção fazendo com o programa fosse desviado para o tratamento de exceção mais externo ao seu bloco, no caso, o tratamento `others`, localizado na linha 40 no nosso programa. Isso, porque, embora, exista outro tratamento `others` logo a seguir, no bloco referente ao `insert`, eles estão em um mesmo nível. Por isso, a ação do programa foi desviada para o tratamento mais externo. Com o desvio, o programa não continua com as ações subsequentes atingindo assim nosso objetivo de tratamento.

> **Nota:** se estivermos utilizando uma estrutura `loop` e uma exceção for gerada dentro dela, e não havendo uma área de tratamento de exceções nesta estrutura, o `loop` é interrompido e o programa é desviado para o primeiro tratamento encontrado em um nível mais externo. Caso haja uma área de tratamento de exceções dentro `loop`, e ela não requerer a parada do sistema, o `loop` continuará normalmente até alcançar o término programado.

Agora vamos alterar o código do departamento para um existente, para realizar um novo teste.

```
SQL> declare
  2    wsal     number;
  3    wdeptno number;
  4  begin
  5    --
  6    begin
  7      select avg(sal), deptno
  8      into   wsal, wdeptno
  9      from   emp
 10      where  deptno = 10
```

```
11        group  by deptno;
12      exception
13        when no_data_found then
14          raise_application_error(-20000,
           'Valores não encontrados para o departamento 99.');
15        when others then
16          dbms_output.put_line('Erro ao selecionar valores
                                  referentes ao depto. 10. '||
17                       'Erro: '||sqlerrm||'.');
18      end;
19      --
20      begin
21        insert into emp (empno, ename, job, mgr, hiredate, sal,
                           comm, deptno)
22           values (8002, 'ANGELINA', 'MANAGER', 7839,
                     TO_DATE('20/10/2011','DD/MM/RRRR'),
                     wsal, null, 20);
23        --
24        commit;
25        --
26      exception
27        when others then
28          raise_application_error(-20000,
             'Erro ao inserir um novo empregado. '||
29                       'Erro: '||sqlerrm||'.');
30      end;
31      --
32      dbms_output.put_line(
           'Empregado ANGELINA inserido com sucesso.');
33      --
34    exception
35      when no_data_found then
36        dbms_output.put_line('Empregado não encontrado.');
37      when too_many_rows then
38        dbms_output.put_line(
           'Erro: O código de departamento informado'||
39                              ' retornou mais de um registro.');
40      when others then
41        dbms_output.put_line(
```

```
         'Erro ao inserir empregado. Erro: '||sqlerrm
 42                ||' - Código: ('||sqlcode||').');
 43   end;
 44   /
Empregado ANGELINA inserido com sucesso.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Como era o esperado, o programa funcionou normalmente. Pode-se ainda simular o erro referente ao comando `insert`, atribuindo `null` à variável `wsal` antes da inclusão.

7.2 Exceções definidas pelo usuário

Como foi dito anteriormente, além das exceções predefinidas existentes no Oracle, podemos criar novas exceções para atender a necessidades mais específicas do nosso programa. A diferença entre estes tipos está na definição e controle destas exceções. Este tipo, quando utilizado pelo usuário, deve ser declarado e manipulado pelo próprio programa. Para o Oracle, estas exceções não existem, por isso, todo o controle deve ser feito pela aplicação. Vamos ver o exemplo a seguir.

```
SQL> declare
  2    wsal           number;
  3    werro_salario exception;
  4   begin
  5    --
  6    begin
  7      select nvl(avg(sal),0)
  8      into   wsal
  9      from   emp
 10      where  deptno = 99;
 11      --
 12      if wsal = 0 then
 13        --
 14        raise werro_salario;
```

```
15          --
16        end if;
17        --
18      exception
19        when others then
20          raise_application_error(-20000,
           'Erro ao selecionar valores referentes ao depto.
 99. '||
21                        'Erro: '||sqlerrm||'.');
22      end;
23      --
24      begin
25        insert into emp (empno, ename, job, mgr, hiredate, sal,
                              comm, deptno)
26           values (8002, 'ANGELINA', 'MANAGER', 7839,
                        TO_DATE('20/10/2011','DD/MM/RRRR'),
 wsal, null, 20);
27        --
28        commit;
29        --
30      exception
31        when others then
32          raise_application_error(-20000,
           'Erro ao inserir um novo empregado. '||
33                        'Erro: '||sqlerrm||'.');
34      end;
35      --
36      dbms_output.put_line(
       'Empregado ANGELINA inserido com sucesso.');
37      --
38    exception
39      when no_data_found then
40        dbms_output.put_line('Empregado não encontrado.');
41      when too_many_rows then
42        dbms_output.put_line(
         'Erro: O código de departamento informado'||
43       ' retornou mais de um registro.');
44      when others then
45        dbms_output.put_line(
```

```
          'Erro ao inserir empregado. Erro: '||sqlerrm
 46       ||' - Código: ('||sqlcode||').');
 47   end;
 48   /
Erro ao inserir empregado. Erro: ORA-20000: Erro ao selecionar
valores referentes ao depto.
 99.
Erro: User-Defined exception. - Código: (-20000).

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Vamos entender o exemplo. Primeiramente, precisamos definir um nome para a exceção e declará-la na área de declaração de variáveis. No exemplo, chamamos nossa exceção de `werro_salario`. Quando declaramos uma variável do tipo exceção utilizamos o tipo `exception`. Feito isto, já podemos utilizá-la em nosso programa. Modificamos o programa utilizado nos exemplos anteriores, justamente para mostrar formas diferentes de tratar os mesmos problemas. Modificamos o `select` que busca o salário para que ele não gere a exceção nativa do Oracle.

Sabemos que o uso de funções de agrupamento, como `sum`, `min`, `max`, `avg` etc., não geram exceções mesmo não atendendo à cláusula `where`. Com isso, a exceção `no_data_found` não é gerada. Contudo, para realizar o controle e não permitir que valores nulos ou zerados sejam cadastrados no campo salário, nós utilizamos uma estrutura `if` para testar o resultado vindo do `select`, e dependendo como for, acionar ou não a exceção que definimos.

Nosso teste é bem simples, verificamos se o valor é igual ou diferente de zero. Caso seja igual a zero chamamos nossa exceção através do comando `raise`. Este comando aciona nossa exceção e faz com que a ação do programa seja desviada para a área de tratamento de erros dentro da hierarquia de níveis, ou seja, a partir daí as regras e sequências são as mesmas utilizadas nas exceções predefinidas.

Uma observação importante. Se houver comandos após a chamada `raise`, eles nunca serão executados. Portanto, fique atento quanto a isto. Note também que duas mensagens de erro foram geradas, inclusive, uma que

diz que a exceção gerada foi uma exceção definida pelo usuário: *User-Defined exception*. Ainda podemos melhorar esta mensagem inibindo os códigos de erro e fazendo com que apareça apenas uma mensagem. Veja a seguir.

```
SQL> declare
  2    wsal           number;
  3    werro_salario exception;
  4  begin
  5    --
  6    begin
  7      select nvl(avg(sal),0)
  8      into   wsal
  9      from   emp
 10      where  deptno = 99;
 11      --
 12      if wsal = 0 then
 13        --
 14        raise werro_salario;
 15        --
 16      end if;
 17      --
 18    exception
 19      when werro_salario then
 20        raise werro_salario;
 21      when others then
 22        raise_application_error(-20000,'Erro ao selecionar
              valores referentes ao depto. 99. '||
 23                       'Erro: '||sqlerrm||'.');
 24    end;
 25    --
 26    begin
 27      insert into emp (empno, ename, job, mgr, hiredate, sal,
                          comm, deptno)
 28         values (8002, 'ANGELINA', 'MANAGER', 7839,
            TO_DATE('20/10/2011','DD/MM/RRRR'),
            wsal, null, 20);
 29      --
 30      commit;
 31      --
```

```
32     exception
33       when others then
34         raise_application_error(-20000,
          'Erro ao inserir um novo empregado. '||
35          'Erro: '||sqlerrm||'.');
36     end;
37     --
38     dbms_output.put_line(
        'Empregado ANGELINA inserido com sucesso.');
39     --
40   exception
41     when werro_salario then
42       dbms_output.put_line(
          'O salário necessita ser maior que zero.');
43     when no_data_found then
44       dbms_output.put_line('Empregado não encontrado.');
45     when too_many_rows then
46       dbms_output.put_line(
           'Erro: O código de departamento informado'||
47         ' retornou mais de um registro.');
48     when others then
49       dbms_output.put_line(
          'Erro ao inserir empregado. Erro: '||sqlerrm
50        ||' - Código: ('||sqlcode||').');
51   end;
52   /
O salário necessita ser maior que zero.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Com isso, finalizamos a parte sobre exceções. Vimos que através delas é possível realizar todos os tratamentos de erros de uma forma organizada e sem prejudicar o funcionamento do sistema. Outro fator que deve ser comentado é que as exceções não servem apenas para geração de mensagens de erro. Dentro das áreas de tratamento podemos usar chamadas a `functions`, `procedures` ou `packages` para solucionar problemas sem a intervenção ou

até mesmo sem conhecimento do usuário.

Capítulo 8

Estruturas de condição: if

As estruturas de condição são utilizadas quando precisamos alterar o fluxo de caminho de um programa. Através deste tipo de estrutura conseguimos criar condições que podem levar o programa a executar tarefas diferentes dependendo de cada situação. Portanto, utilizamos a declaração `if` para avaliar uma ou mais condições, executando ou não determinadas linhas de instruções.

No PL/SQL utilizamos o comando `if` (se) para montar estas condições. Podemos ter condições simples, com uma condição apenas, até o uso de condições aninhadas onde podemos ter comandos `if` dentro de outros comandos `if`, em uma mesma estrutura.

Juntamente com o comando `if` temos o comando `else` (se não), que é utilizado para direcionar o programa para outros caminhos caso o `if` não satisfaça a condição desejada. Dentro da estrutura de um comando `if` podemos ter desde um `else` até vários `else`s (`elsif`), seguindo sua sintaxe.

O que deve ficar claro é que esta estrutura gera um resultado mediante uma condição que satisfaça o critério imposto.

No mais, podemos dizer que será comum achar este tipo de estrutura dentro dos programas. Portanto, você vai utilizá-la com frequência, pois como na grande maioria das vezes escrevemos códigos que refletem e abstraem situações do dia a dia, precisamos testar *n* variáveis para fazer com que o programa chegue ao objetivo proposto em cada situação.

8.1 Estruturas do comando if-end if

```
SQL> begin
  2    --
  3    if <condicao> then
  4      <instruções>
  5    end if;
  6  end;
  7

SQL>
```

Nesta estrutura, temos apenas uma condição. Caso o resultado de `<condicao>` (linha 3) seja verdadeiro, ele estará satisfazendo a condição e executará os comandos que estiverem dentro do escopo do `if`. Caso contrário, não. O `end if` indica o fim do comando `if`. Veja as execuções a seguir, onde o programa é executado duas vezes, mas com diferentes valores:

```
SQL> declare
  2    x   number := 10;
  3    res number;
  4  begin
  5    res := mod(x,2);
  6    if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8    end if;
  9    --
 10    dbms_output.put_line('Resultado do cálculo: '||res);
 11  end;
 12  /
```

```
O resto da divisão é zero!
Resultado do cálculo: 0

Procedimento PL/SQL concluído com sucesso.

SQL>
SQL> declare
  2    x   number := 7;
  3    res number;
  4  begin
  5    res := mod(x,2);
  6    if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8    end if;
  9    --
 10    dbms_output.put_line('Resultado do cálculo: '||res);
 11  end;
 12  /
Resultado do cálculo: 1

Procedimento PL/SQL concluído com sucesso.

SQL>
```

8.2 Estruturas do comando IF-ELSE-END IF

```
SQL> begin
  2    --
  3    if <condicao> then
  4      <instruções>
  5    else
  6      <instruções>
  7    end if;
  8  end;
  9

SQL>
```

Esta estrutura é formada por um `if` e um `else`. Caso a condição do `if`

não seja atendida o fluxo será desviado para o `else`. Note que o `else` não faz restrição ou checagem de condição. Em vias gerais o comando quer dizer: se a condição for verdadeira faça isso, se não faça aquilo.

Vamos ver o mesmo exemplo, agora com duas condições, `if` e `else`, onde dependendo do resultado pode-se determinar qual fluxo o programa deve percorrer.

```
SQL> declare
  2    x   number := 10;
  3    res number;
  4  begin
  5    res := mod(x,2);
  6    if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8    else
  9      dbms_output.put_line('O resto da divisão não é zero!');
 10    end if;
 11    --
 12    dbms_output.put_line('Resultado do cálculo: '||res);
 13  end;
 14  /
O resto da divisão é zero!
Resultado do cálculo: 0

Procedimento PL/SQL concluído com sucesso.

SQL> declare
  2    x   number := 7;
  3    res number;
  4  begin
  5    res := mod(x,2);
  6    if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8    else
  9      dbms_output.put_line('O resto da divisão não é zero!');
 10    end if;
 11    --
 12    dbms_output.put_line('Resultado do cálculo: '||res);
 13  end;
```

```
 14  /
O resto da divisão não é zero!
Resultado do cálculo: 1

Procedimento PL/SQL concluído com sucesso.

SQL>
```

8.3 Estruturas do comando if-elsif(-else)-end if

```
SQL> begin
  2    --
  3    if <condicao> then
  4      <instruções>
  5    elsif <condicao> then
  6      <instruções>
  7    end if;
  8  end;
  9

SQL>
```

```
SQL> begin
  2    --
  3    if <condicao> then
  4      <instruções>
  5    elsif <condicao> then
  6      <instruções>
  7    else
  8      <instruções>
  9    end if;
 10  end;
 11

SQL>
```

Esta estrutura permite testar mais de uma condição dentro de uma estrutura `if`. Além do `if` e `else` podemos ter o `elsif`. Podemos ter uma

estrutura `if` com vários `elsif`, dependendo da necessidade e quantidade de condições que precisarmos testar. A seguir, um exemplo:

Entrando na primeira condição:

```
SQL> declare
  2     x   number := 10;
  3     res number;
  4   begin
  5     res := mod(x,5);
  6     if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8     elsif res > 0 then
  9      dbms_output.put_line('O resto da divisão não é zero!');
 10     else
 11      dbms_output.put_line('O resto da divisão é menor zero!');
 12     end if;
 13     --
 14     dbms_output.put_line('Resultado do cálculo: '||res);
 15   end;
 16   /
O resto da divisão é zero!
Resultado do cálculo: 0

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Entrando na segunda condição:

```
SQL> declare
  2     x   number := 11;
  3     res number;
  4   begin
  5     res := mod(x,5);
  6     if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8     elsif res > 0 then
  9      dbms_output.put_line('O resto da divisão não é zero!');
 10     else
```

```
 11      dbms_output.put_line('O resto da divisão é menor zero!');
 12     end if;
 13     --
 14     dbms_output.put_line('Resultado do cálculo: '||res);
 15   end;
 16   /
O resto da divisão não é zero!
Resultado do cálculo: 1

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Entrando na terceira condição:

```
SQL> declare
  2     x   number := -6;
  3     res number;
  4   begin
  5     res := mod(x,5);
  6     if res = 0 then
  7      dbms_output.put_line('O resto da divisão é zero!');
  8     elsif res > 0 then
  9      dbms_output.put_line('O resto da divisão não é zero!');
 10     else
 11      dbms_output.put_line('O resto da divisão é menor zero!');
 12     end if;
 13     --
 14     dbms_output.put_line('Resultado do cálculo: '||res);
 15   end;
 16   /
O resto da divisão é menor zero!
Resultado do cálculo: -1

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Vimos que a estrutura de condição nos dá várias possibilidades para fazer com que os programas tomem rumos diferentes para cada situação. Também

foi mencionado anteriormente que é possível montar estruturas `if` aninhadas, ou seja, uma estrutura `if` dentro de outra estrutura `if`, por exemplo. Portanto, pode-se usar um aninhamento de declarações `if`, quando se tem a necessidade de filtrar uma série de dados. Mas tome cuidado, pois muitos níveis de declarações `if` podem causar problemas no momento da depuração do programa. O número máximo de níveis de declarações `if` recomendado são quatro. Veja a seguir o exemplo da sintaxe de declarações aninhadas `if`:

```
SQL> begin
  2    if <condição> then
  3      if <condição> then
  4        <instruções>
  5      else
  6        <instruções>
  7        if <condição> then
  8          <instruções>
  9        else
 10          <instruções>
 11        end if;
 12      end if;
 13    end if;
 14  end;
 15

SQL>
```

Vamos ver o exemplo:

```
SQL> declare
  2    x1   number       := 10;
  3    x2   number       := 5;
  4    op   varchar2(1) := '+';
  5    res number;
  6  begin
  7    if (x1 + x2) = 0 then
  8      dbms_output.put_line('Resultado: 0');
  9    elsif op = '*' then
 10      res := x1 * x2;
 11    elsif op = '/' then
```

```
12        if x2 = 0 then
13          dbms_output.put_line('Erro de divisão por zero!');
14        else
15          res := x1 / x2;
16        end if;
17      elsif op = '-' then
18        res := x1 - x2;
19        if res = 0 then
20          dbms_output.put_line('Resultado igual a zero!');
21        elsif res < 0 then
22          dbms_output.put_line('Resultado menor que zero!');
23        elsif res > 0 then
24          dbms_output.put_line('Resultado maiorl a zero!');
25        end if;
26      elsif op = '+' then
27        res := x1 + x2;
28      else
29        dbms_output.put_line('Operador inválido!');
30      end if;
31      dbms_output.put_line('Resultado do cálculo: '||res);
32    end;
33    /
Resultado do cálculo: 15
Procedimento PL/SQL concluído com sucesso.
SQL>
```

8.4 Formatando as declarações if

Para que o código que contenha a declaração `if` fique mais legível e mais fácil de entender, é necessário o emprego de algumas regras de alinhamento tais como:

- Usando-se várias declarações `if`, recua-se a próxima declaração `if` alguns espaços para dentro;
- Os comentários devem ficar após a declaração `end if`;
- Os blocos de declarações devem ficar recuados alguns espaços para dentro, contando a partir da declaração `if`;

- Se uma condição for muito grande e mudar de linha, recua-se esta linha alguns espaços para dentro;
- Deve-se colocar o `else` sempre abaixo da declaração `if` ao qual ele é correspondente, assim como o `end if` do mesmo;

8.5 Evitando erros comuns no uso de if

É recomendado que se tomem algumas precauções para que não ocorram erros nas declarações `if`. A tabela a seguir demonstra alguns cuidados necessários:

- Verifique se toda declaração `if` tem uma declaração `end if` correspondente, e se você digitou `elsif` sem o *e* extra (como "elseif").
- Não faça loops aninhados muito complexos. A complexidade dificulta o acompanhamento e a depuração quando ocorrerem problemas ou alterações. Avalie sua lógica para ver se uma função realiza a mesma tarefa.
- Verifique se você colocou um espaço na declaração `end if` em vez de não usar espaço ou usar um traço.
- Não se esqueça da sua pontuação. Você precisa de pontos e vírgulas depois de `end if` e depois de cada uma das declarações, mas não depois da palavra-chave `then`.

Nota: a função `mod` tem o objetivo de retornar o resto da divisão entre dois números passados por parâmetro.

`Nota`: caso não esteja visualizando as saída do comando `dbms_output`, no SQL*Plus, execute o seguinte comando: `set serveroutput on`.

Capítulo 9

Comandos de repetição

Os comandos de repetição são utilizados para que possamos repetir uma determinada ação ou ações dentro de um programa quantas vezes sejam necessárias. As repetições podem ser iniciadas a partir de uma condição e sua finalização também deve acontecer através deste critério. Em PL/SQL temos basicamente três tipos de estruturas de repetição. São elas: `loop`, `while loop` e `for loop`. Cada estrutura possui características que se adequam às mais variadas situações, conforme as necessidades de cada desenvolvimento. Vamos conhecer os comandos que fazem parte destas estruturas:

9.1 FOR LOOP

Usa-se o `for loop` para repetir diversas vezes o mesmo bloco de código, até que a condição predefinida seja atendida e impeça a execução do looping. Veja a seguir um exemplo de `looping for`:

```
SQL> begin
  2     for i in 1..10 loop
  3       --
  4       dbms_output.put_line('5 X '||i||' = '||(5*i));
  5       --
  6     end loop;
  7   end;
  8   /
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Nota: caso não esteja conseguindo visualizar resultado na tela execute o comando:

```
SQL>
SQL> set serveroutput on
SQL>
```

Este programa imprime na tela a tabuada de cinco. Criamos uma estrutura `for loop` que repetirá o bloco PL/SQL dez vezes, começando por 1. Cada vez que uma volta acontece, a variável `i`, que neste caso não necessitou ser declarada, pois o `for loop` a declara dentro do seu escopo, é incrementada. Quando esta variável chegar a 10, o bloco PL/SQL será executado uma última vez e então o comando é finalizado. Note que utilizamos a variável `i` como base para nosso cálculo, e assim montamos a saída proposta.

Juntamente com o comando `for` podemos utilizar o comando `REVERSE`. Este comando faz com que a contagem aconteça de forma contrária. Neste

exemplo, `i` receberá 10 inicialmente, e será decrementado a cada volta. Quando `i` chegar a 1, o bloco PL/SQL será executado uma última vez e o comando é finalizado. Veja o exemplo:

```
SQL> begin
  2    for i in REVERSE 1..10 loop
  3      --
  4      dbms_output.put_line('5 X '||i||' = '||(5*i));
  5      --
  6    end loop;
  7  end;
  8  /
5 X 10 = 50
5 X 9 = 45
5 X 8 = 40
5 X 7 = 35
5 X 6 = 30
5 X 5 = 25
5 X 4 = 20
5 X 3 = 15
5 X 2 = 10
5 X 1 = 5

Procedimento PL/SQL concluído com sucesso.

SQL>
```

O `loop for` também pode ser executado aninhadamente. Quando se aninha os `loops for`, o `loop` externo é executado primeiro e posteriormente os internos.

```
SQL> begin
  2    for x in 5..6 loop
  3      --
  4      dbms_output.put_line('Tabuada de '||x);
  5      dbms_output.put_line(' ');
  6      --
  7      for y in 1..10 loop
  8        --
```

```
 9         dbms_output.put_line(x||' X '||y||' = '||(x*y));
10         --
11       end loop;
12       --
13       dbms_output.put_line(' ');
14       --
15     end loop;
16   end;
17   /
Tabuada de 5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
Tabuada de 6
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo, utilizamos os `for loops` aninhados para mostrar na tela os cálculos das tabuadas de 5 e 6. Pode-se utilizar o incremento de um loop como parte da lógica de um programa, fazendo com que determinadas

ações só sejam executadas tendo como base esta informação. Veja o exemplo a seguir.

```
SQL> begin
  2    for x in 1..15 loop
  3      --
  4      if mod(x,2) = 0 then
  5        dbms_output.put_line('Número divisível por 2: '||x);
  6      end if;
  7      --
  8    end loop;
  9  end;
 10  /
Número divisível por 2: 2
Número divisível por 2: 4
Número divisível por 2: 6
Número divisível por 2: 8
Número divisível por 2: 10
Número divisível por 2: 12
Número divisível por 2: 14

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Aqui utilizamos uma repetição onde o bloco PL/SQL será executado 15 vezes. Contudo, a impressão em tela será realizada apenas se os critérios do comando `if` forem verdadeiros. Neste exemplo, estamos solicitando que o programa imprima em tela somente os números divisíveis por 2. Note que estamos utilizando a variável de incremento (neste caso, X) como parte da nossa lógica.

Outra opção interessante no uso do `for loop` é a possibilidade de substituir os números fixos do intervalo por variáveis. Veja o mesmo exemplo utilizando variáveis para definir o intervalo.

```
SQL> declare
  2     inter1 number default 1;
  3     inter2 number default 15;
```

```
 4  begin
 5    for x in inter1..inter2 loop
 6      --
 7      if mod(x,2) = 0 then
 8        dbms_output.put_line('Número divisível por 2: '||x);
 9      end if;
10      --
11    end loop;
12  end;
13  /
Número divisível por 2: 2
Número divisível por 2: 4
Número divisível por 2: 6
Número divisível por 2: 8
Número divisível por 2: 10
Número divisível por 2: 12
Número divisível por 2: 14

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Cuidados ao utilizar o comando for loop

- Não se esquecer de colocar um espaço em `end loop`;
- Não se esquecer do ponto e vírgula depois de `end loop`;
- Não inserir o contador do mais alto para o mais baixo ao usar `reserve` ou definir o intervalo do mais alto para o mais baixo e se esquecer de usar `reserve`;
- Não definir as variáveis de um `loop` de modo que o limite inferior tenha um valor maior do que o limite superior;
- Não permitir que as variáveis dos limites acabem em valores `null`;
- Ao aninhar os `loops`, verifique se as declarações seguem a lógica pretendida.

9.2 WHILE LOOP

`while loop` é usado para avaliar uma condição antes de uma sequência de códigos seja executado. A diferença entre o `loop while` e o `loop for`, é que o `loop while` permite a execução de código apenas se a condição for verdadeira, e o `loop for` faz com que o código seja executado pelo menos uma vez, independente da condição. Veja a seguir um exemplo do `loop while`:

```
SQL> declare
  2    x           number        default 0;
  3    label_vert varchar2(240) default '&label';
  4    tam_label  number        default 0;
  5  begin
  6    --
  7    tam_label := length(label_vert);
  8    --
  9    while (x < tam_label) loop
 10      --
 11      x := x + 1;
 12      --
 13      dbms_output.put_line(substr(label_vert,x,1));
 14      --
 15    end loop;
 16  end;
 17  /
Informe o valor para label: CURSO PLSQL
antigo   3:   label_vert varchar2(240) default '&label';
novo   3:   label_vert varchar2(240) default 'CURSO PLSQL';
C
U
R
S
O
P
L
S
Q
L
```

```
Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que neste exemplo não temos um número determinado de repetições. O número vai depender do tamanho da string informada através da variável `label`. Este programa imprime na tela em vertical uma string informada via parâmetro. A lógica foi escrita de forma que as repetições ocorram conforme a quantidade de caracteres existente dentro da variável. O controle é realizado pela existência da condição contida no comando `while`.

9.3 LOOP

O `loop` é o comando de repetição mais simples e fácil de entender. Seu objetivo é igual aos dos demais comandos de repetição. Entretanto, possui características diferentes de funcionamento. Neste comando não é permitido definir um intervalo de repetição, muito menos uma condição que o faça iniciar e parar sua execução. Em tese, seu funcionamento é infinito, pois ao entrar em uma estrutura como esta não há como sair. Por isso, juntamente com ele utilizamos o comando `exit`. Este comando é quem vai determinar a finalização das repetições, impedindo que o programa entre em um loop eterno.

Declarações exit e exit when

Quando uma declaração `exit` é encontrada, o `loop` é imediatamente encerrado e o controle é passado para a declaração seguinte. A declaração `exit when` permite que você especifique a condição requerida para sair da execução do `loop`. Se o resultado da condição for verdadeiro, o `loop` é encerrado, e se o resultado for falso, o looping continua.

Ao usar `exit` ou `exit when` coloque sempre esses comandos no início ou no final do bloco `loop`. Dessa forma você pode evitar muitos erros lógicos.

Seguem exemplos de loop utilizando `exit` e `exit when`:

No exemplo a seguir utilizamos a estrutura `if` para criar uma condição que possa determinar a finalização do comando. Caso a condição contida no `if` seja verdadeira, chamamos o comando `exit` para finalizar a sequência de repetições. Caso contrário, o comando continua sua execução.

```
SQL> declare
  2    x           number        default 0;
  3    label_vert varchar2(240) default '&label';
  4    tam_label  number        default 0;
  5  begin
  6    --
  7    tam_label := length(label_vert);
  8    --
  9    loop
 10      --
 11      x := x + 1;
 12      --
 13      dbms_output.put_line(substr(label_vert,x,1));
 14      --
 15      if x = tam_label then
 16        exit;
 17      end if;
 18      --
 19    end loop;
 20  end;
 21  /
Informe o valor para label: ORACLE PLSQL
antigo   3:   label_vert varchar2(240) default '&label';
novo   3:   label_vert varchar2(240) default 'ORACLE PLSQL';
O
R
A
C
L
E
P
L
S
Q
```

```
L

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Já no exemplo a seguir utilizamos `exit when`, que nos permite incluir uma condição que possa finalizar o comando. Note que com isso não precisamos utilizar estruturas auxiliares, como o `if`, como foi utilizado no exemplo anterior.

```
SQL> declare
  2    x          number        default 0;
  3    label_vert varchar2(240) default '&label';
  4    tam_label  number        default 0;
  5  begin
  6    --
  7    tam_label := length(label_vert);
  8    --
  9    loop
 10      --
 11      x := x + 1;
 12      --
 13      dbms_output.put_line(substr(label_vert,x,1));
 14      --
 15      exit when x = tam_label;
 16      --
 17    end loop;
 18  end;
 19  /
Informe o valor para label: CURSO PLSQL
antigo   3:   label_vert varchar2(240) default '&label';
novo   3:   label_vert varchar2(240) default 'CURSO PLSQL';
C
U
R
S
O
P
```

```
L
S
Q
L

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Em outras linguagens de programação é comum encontrar o comando `repeat until` (repita até). Na PL/SQL não existe este comando. Contudo, utilizamos os comandos `loop` e `exit when` para sanar esta necessidade.

9.4 Qual loop deve-se usar?

Conforme visto até aqui, temos disponíveis três opções de loops. Mas como saber se estamos fazendo uso do tipo correto de loop para determinada situação? A tabela a seguir ajuda a esclarecer esta dúvida.

- **for**: use sempre o loop for se você souber especificamente quantas vezes o loop deve ser executado. Se tiver de codificar uma declaração `exit` ou `exit when` em um `loop for`, você pode reconsiderar seu código e usar um loop ou uma abordagem diferente.
- **while**: use este loop quando não há certeza se ele será executado. Embora seja possível conseguir esse resultado em um `loop for` usando `exit` ou `exit when`, essa situação é mais adequada para o `loop while`. O `loop while` é o loop mais usado porque ele fornece mais flexibilidade.
- **loop**: você pode usar o loop simples se quiser criar um loop do tipo `repeat until`. O loop simples é perfeito para executar essa tarefa.

Orientações sobre os loopS

A lista a seguir apresentará algumas orientações sobre os loops.

- Verifique ao usar um loop com uma declaração `exit` ou `exit when` se a condição será atendida pelo menos uma vez; caso contrário, você terá um loop infinito.
- Nunca crie um loop infinito.
- Use sempre os nomes de rótulos nos loops. Isso torna o código muito mais fácil de acompanhar, além de lhe dar flexibilidade.
- Não use uma declaração `return` dentro de um loop ao usar os loops em uma função. Embora isso funcione, essa é uma prática ruim de programação que tem alguns resultados indesejados, e é o modo errado de encerrar um loop.
- Use `exit when` em vez de `exit`. `exit when` é muito mais fácil de acompanhar e requer menos código.
- Verifique se a pontuação dos seus loops está adequada. Selecione o tipo de loop que vai usar com os incrementos. Você pode lidar com qualquer tipo de incremento em qualquer loop.
- Crie variáveis de limite superior e inferior nos loops for se um dos limites puder ser alterado no futuro. Você pode atribuir esses limites imediatamente ao seu código. Na verdade, muito provavelmente você nem terá um limite fixo, de modo que você deve seguir este conselho automaticamente.

Capítulo 10

Cursores

O cursor é um comando do PL/SQL que permite a construção de uma estrutura de repetição, ou seja, pode-se varrer uma tabela, linha por linha, coluna por coluna através da utilização deste comando, e assim, podemos manipular todos os dados de uma determinada tabela.

O cursor deve ser declarado no início do bloco, onde será utilizado. Depois de declará-lo, ele deve ser criado dentro da mesma estrutura. As informações contidas no cursor são baseadas nos dados pelos quais sua estrutura foi definida. Por exemplo, se definirmos um cursor com base na tabela de empregados, será com os dados referentes aos empregados desta tabela em que vamos trabalhar quando estivermos manipulando tal cursor. Portanto, os retornos dos dados apresentados pelo cursor provem de comandos `selects` feitos em determinadas tabelas e que trazem dados de linhas e colunas específicas.

Existem dois tipos de cursores no PL/SQL, os explícitos e os implícitos. O

cursor explícito é o que definimos nos programas, e o implícito, como o nome já diz é declarado implicitamente pelo Oracle, ou seja, quando usado o cursor explícito, o desenvolvedor, no caso quem está escrevendo a aplicação, além de declará-lo deve inserir comandos que especifiquem a sua inicialização e finalização, bem como a manipulação dos dados. Quando não existir um cursor explícito associado para o comando SQL, o Oracle o cria implicitamente. Um exemplo disto é quando temos um comando `update` ou `delete`, dentro de uma aplicação. Embora, não seja visível explicitamente, o Oracle cria um cursor para executar tal comando. Os cursores implícitos e explícitos podem existir na mesma aplicação em um mesmo bloco.

Neste livro daremos ênfase nos cursores explícitos. Através deles podemos atender a situações bem específicas no desenvolvimento de nossos programas. Contudo, serão mostradas as características e o funcionamento destes dois tipos.

10.1 Cursores explícitos

Para utilizarmos um cursor explícito, algumas regras devem ser obedecidas. Primeiramente, temos que declará-lo. Fazemos isto da mesma forma que com as variáveis e constantes. Veja a seguir algumas formas de declaração:

```
cursor c1 is
  select ename
        ,job
  from   emp
  where  deptno = 30;
```

É na declaração que definimos a estrutura do cursor. Esta estrutura inicia com a expressão `cursor <nome cursor> is`. O `<nome cursor>` segue as mesmas regras da definição de variáveis. Com isto estamos definindo o cabeçalho do cursor, rotulando-o. Logo após a expressão `is` vem o comando `select` que será a base do nosso cursor. Este `select` pode ser escrito normalmente como se fôssemos executá-lo pelo SQL*Plus, por exemplo. Os comandos `select` podem conter várias tabelas e colunas. Também pode ser usada a expressão asterisco `*` para se referir a todas as colunas de uma ou mais tabelas. Vale ressaltar que, quando utilizamos cursor, não é necessário usar

o comando `into` no `select`. Também podemos passar parâmetros para o `select` através dos cursores. Isso torna nossa estrutura mais organizada e possibilita utilizá-la para execução de diferentes critérios. Olhe o próximo exemplo.

```
cursor c1( pdeptno number
          ,pjob    varchar2) is
  select empno
        ,ename
  from   emp
  where  deptno = pdeptno
  and    job    > pjob;
```

O que difere nesta declaração é a definição de parâmetros. Note que, após ser informado o nome do nosso cursor, são definidos parâmetros os quais serão utilizados dentro comando `select`. A definição de parâmetros é idêntica a definição de variáveis. A única diferença é que não é necessário informar a quantidade de caracteres para o tipo do parâmetro. Por exemplo, o parâmetro `pjob` foi definido apenas como `varchar2`, não importando a quantidade de caracteres. Nestes casos, o Oracle considera o máximo de caracteres que o tipo suporta. Neste exemplo foram definidos dois parâmetros e os mesmos foram utilizados na cláusula `where` do comando select.

Depois que definimos o cursor, precisamos declarar uma variável que receberá esta definição. Esta variável funcionará como um `array`, ou seja, ao abrirmos o cursor os registros serão jogados para este `array` para trabalharmos com eles. Vejamos a declaração a seguir.

```
declare
  --
  cursor c1 is -- lista todos os empregados do departamento 30.
    select ename
          ,job
    from   emp
    where  deptno = 30;
  --
  r1 c1%rowtype;
  --
begin
```

Neste exemplo, é declarado o cursor `C1` que seleciona todos os empregados do departamento 30. Depois de definido nosso cursor, foi declarada uma variável chamada `R1` que é do tipo `c1%rowtype`. `C1` vem do nosso cursor. Já o `%rowtype`, indica que nossa variável será um array de linhas vindas do cursor. Com isto, nossa declaração está completa e já podemos usar nosso cursor.

Quando utilizamos no `select` do cursor apenas uma tabela, podemos optar por usar a própria estrutura da tabela como sendo o array para nossa variável. Como a PL/SQL mantém uma conexão direta (chamamos de nativa) com o banco de dados e seus objetos, ela consegue reconhecer a compatibilidade entre as duas estruturas, a do array e a do `cursor`. Veja o exemplo a seguir.

```
declare
  --
  cursor c1 is -- lista todos os empregados do departamento 30.
    select *
    from   emp
    where  deptno = 30;
  --
  r1 emp%rowtype;
  --
begin
```

Veja que neste exemplo, declaramos a nossa variável `R1` do tipo EMP em vez do tipo `C1` referente ao nosso cursor. Quando informamos que `R1` é do tipo `emp%rowtype`, estamos dizendo que nossa variável array é do mesmo tipo, ou melhor, terá a mesma definição baseada na estrutura da tabela EMP. Logo, em nosso cursor, trazemos todas as colunas desta tabela. Assim sendo, o Oracle compara a estrutura do cursor com a estrutura da tabela EMP e valida esta definição.

Note que, neste caso, não poderíamos suprimir colunas no `select` do cursor, pois na comparação entre as estruturas o Oracle acusaria uma diferença. Portanto, resumindo, os nomes, as quantidades e os tipos das colunas devem ser iguais entre as estruturas.

Usando o cursor explícito

O uso de cursores basicamente envolve três passos. Abertura, recuperação e manipulação dos registros e fechamento do cursor. Segue o exemplo mostrando o uso do cursor que declaramos anteriormente.

```
SQL> declare
  2    --
  3    cursor c1 is -- lista todos os empregados do
                       departamento 30.
  4      select ename
  5            ,job
  6      from   emp
  7      where  deptno = 30;
  8    --
  9    r1 c1%rowtype;
 10    --
 11  begin
 12    --
 13    open c1;
 14    loop
 15      fetch c1 into r1;
 16      exit when c1%notfound;
 17      --
 18      dbms_output.put_line('Nome: '||r1.ename||'
                                Cargo: '||r1.job);
 19      --
 20    end loop;
 21    --
 22    close c1;
 23    --
 24  end;
 25  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

As linhas 3 a 7 mostram a declaração do cursor visto anteriormente. Na linha 9 está declarada a variável array `R1` do tipo `C1`, que é nosso cursor. Na

linha 13 está presente o comando `open c1;`. Este comando abre o cursor. Já na linha 15, temos o retorno das linhas referentes ao `select` através da linha de comando `fetch c1 into r1;`. Em linhas gerais este comando significa: retorne a primeira linha de registro do `select`. Na linha 18 temos o comando `put_line` do pacote `dbms_output` que imprime na tela o nome e cargo do empregado.

Note que o programa recupera estas informações através do array `R1`. Note também que as mesmas colunas definidas no `select` do cursor agora podem ser usadas através de `R1`. Para encerrar, na linha 22 temos o fechamento do cursor. Vale lembrar que estas delimitações não servem apenas para tornar mais fácil a visualização do código, mas principalmente para demarcar o uso dos dados vindos do cursor através do array. Por exemplo, não podemos utilizar `r1.ename` antes da abertura e `fetch` do cursor, nem após o fechamento.

Vamos a uma observação interessante. Você deve estar se perguntando sobre o porquê do `loop` presente dentro da estrutura, certo? Pois bem, anteriormente, para recuperar um registro utilizamos o comando `fetch`. Este comando dentro do nosso programa retornará apenas um registro, entretanto, nosso comando `select` pode retornar mais de um registro. Por isso, em conjunto com cursor precisamos sempre solicitar o retorno dos demais registros. O cursor por si só não retorna todos os registros resultante do comando `select`. Por isso utilizamos uma estrutura de repetição para nos auxiliar.

Só para vocês entenderem, cada vez que abrimos um cursor o Oracle aponta-o para o primeiro registro do `select`. Todavia, ele só é recuperado quando chamamos o `fetch`. Este por sua vez traz apenas o primeiro registro. Para que os próximos registros sejam recuperados, temos que executar o `fetch` novamente. É aí que entra a estrutura de repetição. Repetimos o comando `fetch` até que não haja mais registros retornados. Para que o programa saiba quando não há mais registros para retornar, podemos utilizar a linha de comando `exit when c1%notfound`. Esta linha indica quando se deve sair da estrutura loop, ou seja, quando o cursor `C1` não tiver mais registros para retornar.

`%notfound` é um atributo de cursor que indica a não existência de mais registros a serem lidos dentro de um cursor. Mais adiante veremos mais sobre

este assunto. Note que a estrutura de repetição deve obedecer às delimitações de abertura e fechamento do cursor. A ausência de uma estrutura de repetição não caracteriza um erro. Contudo, apenas um registro seria retornado.

No uso de cursor é mais comum vermos a estrutura do tipo loop simples, mas podemos utilizar também as estruturas `while` e `for loop`, sendo que este último torna a manipulação do cursor bem mais fácil, pois ele já faz alguns controles automaticamente. Mais à frente nós veremos um exemplo. Vamos ver outro exemplo utilizando passagem de valores por parâmetro.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4               ,pmgr   number) is
  5      select ename, job, dname
  6      from   emp, dept
  7      where  emp.deptno = dept.deptno
  8      and    dept.loc    = pdname
  9      and    emp.mgr     = pmgr;
 10    --
 11    r1 c1%rowtype;
 12    --
 13  begin
 14    --
 15    open c1('CHICAGO',7698);
 16    loop
 17      fetch c1 into r1;
 18      exit when c1%notfound;
 19      --
 20      dbms_output.put_line('Nome: '||r1.ename||'
                           Cargo: '||r1.job);
 21      --
 22    end loop;
 23    --
 24    close c1;
 25    --
 26  end;
 27  /
```

```
Procedimento PL/SQL concluído com sucesso.

SQL>
```

A declaração de cursores com passagem de parâmetro, nós já havíamos visto anteriormente. O que devemos prestar atenção é como e quando devemos passar os valores para o cursor. Observando a linha 15 do programa anterior é possível verificar a abertura do cursor. É nesta abertura que devemos passar os valores para os parâmetros. Ao abrir o cursor, o Oracle já leva os valores de entrada para serem utilizados dentro do `select`. Os valores informados devem seguir a mesma ordem definida para os parâmetros. Uma alternativa a isso é a chamada passagem de parâmetro nomeada, onde ao informar os valores também é informado qual parâmetro deve recebê-los. Nestes casos, a ordem não precisa ser respeitada. Veja o mesmo programa, agora utilizando passagem de parâmetros nomeada.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4              ,pmgr   number) is
  5      select ename, job, dname
  6      from   emp, dept
  7      where  emp.deptno = dept.deptno
  8      and    dept.loc   = pdname
  9      and    emp.mgr    = pmgr;
 10    --
 11    r1 c1%rowtype;
 12    --
 13  begin
 14    --
 15    open c1( pmgr   => 7698
 16            ,pdname => 'CHICAGO');
 17    loop
 18      fetch c1 into r1;
 19      exit when c1%notfound;
 20      --
 21      dbms_output.put_line('Nome: '||r1.ename||'
                              Cargo: '||r1.job);
```

```
 22       --
 23     end loop;
 24     --
 25     close c1;
 26     --
 27   end;
 28   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note as linhas 15 e 16. A passagem nomeada consiste em informar o nome do parâmetro e o valor que ele deve receber. Deve ser informado o nome do parâmetro seguido dos sinais =>, seguido do valor a ser passado como parâmetro. Veja também que a ordem foi invertida com relação ao que foi definido na declaração do cursor.

Contudo, como estamos informando qual parâmetro recebe tal valor, não teremos problema se os parâmetros estiverem em uma ordem contrária. O uso deste artifício é bem interessante quando temos uma grande quantidade de parâmetros. Se a quantidade for grande corre-se o risco de colocar algum parâmetro fora da ordem correta, que certamente ocasionará um erro na abertura do cursor ou na lógica do programa.

10.2 Cursor for loop

Anteriormente, mencionei algo sobre o cursor com uso do `for loop`. Pois bem. Conforme eu já havia falado, o uso de cursores com `for loop` torna o trabalho mais cômodo, pois não precisamos nos preocupar em abrir e fechar o cursor, nem realizar o `fetch`. Ele faz tudo sozinho. Para melhor entendimento vamos ver o programa a seguir.

```
SQL> declare
  2     --
  3     cursor c1( pdname varchar2
  4                ,pmgr   number) is
  5       select ename, job, dname
```

```
 6       from    emp, dept
 7       where   emp.deptno = dept.deptno
 8       and     dept.loc    = pdname
 9       and     emp.mgr     = pmgr;
10     --
11   begin
12     --
13     for r1 in c1( pmgr    => 7698
14                  ,pdname => 'CHICAGO') loop
15      --
16      dbms_output.put_line('Nome: '||r1.ename||'
                             Cargo: '||r1.job);
17      --
18     end loop;
19     --
20   end;
21   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

É nítida a diferença entre um método e outro. Veja que eliminamos algumas linhas de código. O próprio `for` se encarrega de abrir o cursor, realizar a busca dos registros, verificar quando não há mais registros e fechar o cursor. Mais um detalhe: utilizando o `for loop`, não é necessário declarar a variável do tipo `rowtype` com base no cursor, o próprio `for loop` faz isto mediante um nome informado. Veja que no exemplo, só foi preciso informar um nome, no caso `R1`, após o comando for, que o resto ele se encarregou de fazer. Assim como no uso do `loop`, o `for` também delimita uma área que deve respeitada.

Agora você deve estar se questionando: por que usar `loop` se `for loop` traz mais facilidade sem falar que não é necessário que se controle uma série de coisas? Na verdade você tem razão, nada impede de você utilizar sempre o cursor com `for loop`. Contudo, à medida que você for trabalhando com cursores, vai sentir a necessidade de usar um ou outro. Por exemplo, quando temos um cursor onde não necessariamente temos que ler todas as linhas

vindas do `select`, podemos utilizar o `loop`, pois neste método somos nós que controlamos a busca das linhas.

Neste caso, é bem mais prático utilizar um `loop`. Vale ressaltar que com o uso de `for loop`, devemos informar quando ele deve parar sua execução, caso não for de nossa vontade que ele leia todos os registros resultantes do `select`. Já com o uso do `loop` é o contrário, se quisermos que ele leia os registros, devemos comandá-lo através do `fetch`. Quando quiser parar uma execução `for loop`, utilizamos o comando `exit` ou `exit when`. Veja um exemplo.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4               ,pmgr   number) is
  5      select ename, job, dname
  6      from   emp, dept
  7      where  emp.deptno = dept.deptno
  8      and    dept.loc   = pdname
  9      and    emp.mgr    = pmgr;
 10    --
 11    --
 12    --
 13  begin
 14    --
 15    for r1 in c1( pmgr   => 7698
 16                  ,pdname => 'CHICAGO') loop
 17     --
 18     dbms_output.put_line('Nome: '||r1.ename||'
                              Cargo: '||r1.job);
 19     --
 20     /*
 21     if r1.ename = 'MARTIN' then
 22       exit;
 23     end if;
 24     */
 25     --
 26     exit when r1.ename = 'MARTIN';
 27     --
```

```
28      end loop;
29      --
30    end;
31    /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Mais à frente nós também veremos que às vezes pode ser mais performático utilizar um cursor para retornar um único registro em vez de utilizar `into`. Neste caso não utilizamos estruturas de repetição.

10.3 Cursor for loop com definição interna

10.3. CURSOR FOR LOOP COM DEFINIÇÃO INTERNA

Com o uso de cursores com `for loop` temos disponível mais um recurso que é a possibilidade de declararmos o comando `select` do cursor dentro do próprio comando `for loop`. Desta forma, não precisamos declarar o cursor no escopo de declarações, conforme visto nos exemplos anteriores, e sim, na própria definição do `for loop`. O único inconveniente é que se nós tivermos que chamar este cursor `for loop` várias vezes dentro do programa, teremos o comando `select` escrito repetidamente em todas as aberturas. No caso de uma alteração nele, teremos que modificar em todas as chamadas, enquanto na declaração fora do `for loop` (no bloco `declare`) a alteração será em um único lugar.

```
SQL> declare
  2     --
  3   begin
  4     --
  5     for r1 in (select empno, ename
  6                from   emp
  7                where  job = 'MANAGER') loop
  8       --
  9       dbms_output.put_line(
            'Gerente: '||r1.empno||' - '||r1.ename);
```

```
10      --
11      for r2 in (select empno, ename, dname
12                 from   emp, dept
13                 where  emp.deptno = dept.deptno
14                 and    mgr        = r1.empno) loop
15        --
16        dbms_output.put_line(
             '  Subordinado: '||r2.empno||' - '||r2.ename);
17        --
18      end loop;
19      --
20      dbms_output.put_line('');
21      --
22    end loop;
23    --
24  end;
25  /

SQL>
```

10.4 Cursores implícitos

10.4. CURSORES IMPLÍCITOS

Em vias gerais quando não criamos um cursor explicito o Oracle cria um cursor automaticamente. A este cursor se dá o nome de **cursor implícito**. Implícito porque é criado, aberto, manipulado e fechado pelo próprio Oracle. Mas como sabemos, ou melhor, como vemos o uso de tal cursor? Na verdade não vemos, pelo menos não na sua totalidade.

O cursor implícito é criado quando dentro do nosso programa PL/SQL utilizamos algum dos comandos: `delete`, `update`, `insert` ou `select into`.

Quando executamos os comandos `delete`, `insert` ou `update`, o Oracle cria um cursor implícito, abre este cursor, executa o comando e depois o fecha. Quando executamos um `select into` dentro do nosso programa, por exemplo, o Oracle vai além e abre um cursor implicitamente e executa vários passos e verificações adicionais. Veja os passos a seguir.

Cursor implícito: passos realizados pelo Oracle

1) Cria um cursor implícito.
2) Realiza um primeiro `fetch` para retornar uma linha.
3) Realiza um segundo `fetch` para verificar se o comando não retorna mais de uma linha.
4) Fecha o cursor.

Note que no caso do `select into` o Oracle precisa tratar determinadas situações, pois o uso do comando `select` dentro dos programas PL/SQL podem gerar `exceptions`. Quando não é retornada nenhuma linha uma `exception` de `no_data_found` é disparada. Já quando várias linhas são retornadas outra exception chamada `too_many_rows` precisa ser acionada também.

Desta forma, quando se há certeza de que o `select` em questão vai retornar apenas uma linha, pode ser usado um cursor explícito. Neste caso, menos passos serão utilizados, pois quem tratará estas situações somos nós dentro do programa. Entretanto, esta forma parece ser mais trabalhosa, e com certeza é! Contudo, quando estamos falando de performance, os ganhos podem ser bem consideráveis. É óbvio que estamos falando aqui para os casos onde os comandos `selects` são complexos com muitas tabelas e grandes quantidades de dados. Para um cursor explícito os passos ficariam assim:

Cursor explícito: passos realizados pelo programa

1) Cria um cursor implícito.
2) Realiza um primeiro `fetch` para retornar a linha.
3) Fecha o cursor.

Com isto, fazemos com que pelo menos um passo não seja executado. Para um `select` que trabalhe com milhões de registros isso pode nos dar um ótimo ganho de tempo de execução.

10.5 Atributos de cursor explícito e implícito

10.5. ATRIBUTOS DE CURSOR EXPLÍCITO E IMPLÍCITO

Nos exemplos anteriores vimos alguns atributos que auxiliam na hora de se utilizar cursores. Estes atributos trazem informações que podem ajudar o desenvolvedor a tomar decisões e a fazer com que o programa tome certos caminhos. Estes atributos estão disponíveis tanto para cursores explícitos como para os implícitos, com exceção do atributo `%isopen` que só pode ser utilizado em cursores explícitos. Estes atributos são utilizados da seguinte forma:

Quando estamos trabalhando com os cursores explícitos usamos o nome do cursor seguido do atributo, por exemplo, `c1%found`. Para os cursores implícitos, o Oracle disponibiliza o cursor "SQL" que aponta sempre para o comando que está sendo executado no momento. Desta forma, usa-se o nome deste cursor seguido do atributo, exemplo, `sql%found`. Observe que o Oracle utiliza sempre o mesmo cursor para uma ou mais execuções. Portanto, ele deve ser utilizado sempre após a execução do comando, pois as informações de um cursor recém-executado sobrescreverão as informações da execução anterior. A seguir os atributos que podemos utilizar.

%found

Este atributo é utilizado para indicar se a última operação realizada pelo `fetch` foi concluída com êxito ou se uma determinada linha foi alterada através de algum comando `insert`, `update` ou `delete`, ou se, no caso de um `select into` ouve algum retorno de uma ou mais linhas. Para cursores explícitos:

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4              ,pmgr   number) is
  5      select ename, job, dname
  6      from   emp, dept
  7      where  emp.deptno = dept.deptno
  8      and    dept.loc    = pdname
  9      and    emp.mgr     = pmgr;
 10    --
```

```
11     r1 c1%rowtype;
12     --
13   begin
14     --
15     open c1( pmgr   => 7698, pdname => 'CHICAGO');
16     loop
17       fetch c1 into r1;
18       --
19       if c1%found then
20         --
21         dbms_output.put_line('Nome: '||r1.ename||'
                                Cargo: '||r1.job);
22         --
23       else
24         --
25         exit;
26         --
27       end if;
28     end loop;
29     --
30     close c1;
31     --
32   end;
33   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Para cursores implícitos:

```
SQL> declare
  2     --
  3     wename emp.ename%type;
  4     wjob   emp.job%type;
  5     wdname dept.dname%type;
  6     --
  7   begin
  8     --
  9     select ename, job, dname
```

```
 10    into    wename, wjob, wdname
 11    from    emp, dept
 12    where   emp.deptno = dept.deptno
 13    and     dept.loc    = 'CHICAGO'
 14    and     emp.mgr     = 7698
 15    and     job         = 'CLERK';
 16    --
 17    if SQL%found then
 18      --
 19      dbms_output.put_line(
          'O select retornou o seguinte registro: Nome: '||
 20                 wename||' Cargo: '||wjob);
 21      --
 22    end if;
 23    --
 24  end;
 25  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

%notfound

Este outro atributo mostra exatamente o contrário ao atributo `%found`, pois o comando `%notfound` retorna sempre *false* se o último comando `fetch` não retornar uma linha no caso de um cursor explícito, ou se o último comando de `update`, `insert`, `delete` não alterar nenhuma linha. O mesmo não serve para o não retorno de uma linha em um comando `select into`, pois neste caso a exception `no_data_found` é disparada. Para cursores explícitos:

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2
  4              ,pmgr   number) is
  5      select ename, job, dname
  6      from   emp, dept
  7      where  emp.deptno = dept.deptno
```

```
  8       and     dept.loc    = pdname
  9       and     emp.mgr     = pmgr;
 10     --
 11     r1 c1%rowtype;
 12     --
 13   begin
 14     --
 15     open c1( pmgr    => 7698, pdname => 'CHICAGO');
 16     loop
 17       fetch c1 into r1;
 18       --
 19       exit when c1%notfound;
 20       --
 21       dbms_output.put_line(
           'Nome: '||r1.ename||' Cargo: '||r1.job);
 22       --
 23     end loop;
 24     --
 25     close c1;
 26     --
 27   end;
 28   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Para cursores implícitos:

```
SQL> declare
  2     --
  3     wename emp.ename%type;
  4     wjob   emp.job%type;
  5     wdname dept.dname%type;
  6     --
  7   begin
  8     --
  9     update emp
 10     set    deptno = 100
 11     where  job    = 'ANALISTA_1';
```

```
 12     --
 13     if SQL%notfound then
 14       --
 15       dbms_output.put_line('Nenhum registro foi atualizado.');
 16       --
 17     end if;
 18     --
 19   end;
 20   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

%rowcount

Este atributo é utilizado para fazer a contagem do número de linhas lidas, ou para linhas que foram afetadas por algum comando `insert`, `update` ou `delete`, ou, por algum retorno ocasionado por um comando `select into`. Vale salientar que a utilização deles segue as mesmas regras para os tipos de cursores mencionados nos outros atributos anteriores. Para cursores explícitos:

```
SQL> declare
  2     --
  3     cursor c1( pdname varchar2
  4               ,pmgr   number) is
  5       select ename, job, dname
  6       from   emp, dept
  7       where  emp.deptno = dept.deptno
  8       and    dept.loc    = pdname
  9       and    emp.mgr     = pmgr;
 10     --
 11     r1 c1%rowtype;
 12     --
 13   begin
 14     --
 15     open c1( pmgr   => 7698, pdname => 'CHICAGO');
 16     --
```

```
 17      loop
 18        fetch c1 into r1;
 19        --
 20        exit when c1%notfound;
 21        --
 22        dbms_output.put_line(
            'Nome: '||r1.ename||' Cargo: '||r1.job);
 23        --
 24      end loop;
 25      --
 26      dbms_output.put_line('');
 27      dbms_output.put_line(
          'Registros recuperados: '||c1%rowcount||'.');
 28      --
 29      close c1;
 30      --
 31    end;
 32    /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Para cursores implícitos:

```
SQL> declare
  2      --
  3      wename emp.ename%type;
  4      wjob   emp.job%type;
  5      wdname dept.dname%type;
  6      --
  7    begin
  8      --
  9      delete emp
 10      where  deptno = (select deptno
 11                       from   dept
 12                       where  dname = 'SALES');
 13      --
 14      dbms_output.put_line(
          SQL%rowcount||' registro(s) foram excluídos.');
```

```
 15     --
 16     commit;
 17     --
 18   end;
 19   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

%isopen

Este atributo é utilizado para verificar se um cursor está ou não aberto. Ele estará como verdadeiro se o cursor estiver aberto, e falso se estiver fechado. Este atributo só pode ser usado em cursores explícitos. Para cursores explícitos:

```
SQL> declare
  2     --
  3     cursor c1( pdname varchar2
  4               ,pmgr   number) is
  5       select ename, job, dname
  6       from   emp, dept
  7       where  emp.deptno = dept.deptno
  8       and    dept.loc    = pdname
  9       and    emp.mgr     = pmgr;
 10     --
 11     r1 c1%rowtype;
 12     --
 13   begin
 14     --
 15     loop
 16       if c1%isopen then
 17         --
 18         fetch c1 into r1;
 19         --
 20         if c1%notfound then
 21           --
 22           close c1;
```

```
23          exit;
24          --
25        else
26          --
27          dbms_output.put_line(
             'Nome: '||r1.ename||' Cargo: '||r1.job);
28          --
29        end if;
30        --
31      else
32        --
33        dbms_output.put_line('O Cursor não foi aberto!');
34        dbms_output.put_line('Abrindo cursor...');
35        open c1( pmgr   => 7698, pdname => 'CHICAGO');
36        --
37      end if;
38      --
39    end loop;
40    --
41  end;
42  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

10.6 Cursores encadeados

10.6. CURSORES ENCADEADOS

Outro recurso dos cursores é a possibilidade de serem usados de forma encadeada, ou seja, podemos ter cursores sendo manipulados dentro de outros cursores. Veja um exemplo.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select empno, ename
  5      from   emp
```

```
 6       where  job = 'MANAGER';
 7     --
 8     cursor c2(pmgr number) is
 9       select empno, ename, dname
10       from   emp, dept
11       where  emp.deptno = dept.deptno
12       and    mgr        = pmgr;
13     --
14   begin
15     --
16     for r1 in c1 loop
17       --
18       dbms_output.put_line(
          'Gerente: '||r1.empno||' - '||r1.ename);
19       --
20       for r2 in c2(r1.empno) loop
21         --
22         dbms_output.put_line(
            '  Subordinado: '||r2.empno||' - '||r2.ename);
23         --
24       end loop;
25       --
26       dbms_output.put_line('');
27       --
28     end loop;
29     --
30   end;
31   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo estamos selecionando todos os empregados gerentes com seus respectivos subordinados. O primeiro cursor `C1` localiza os gerentes. Para cada gerente encontrado é listado todos os seus subordinados pelo cursor `C2`. Note que o cursor `C2` recebe como parâmetro o código do gerente vindo de `C1`. Para cada linha retornada de `C1` são recuperadas todas as linhas de `C2`. O `for loop` se encarrega de abrir e fechar os cursores quantas vezes for

necessário até que a leitura de todos os registros em ambos os cursores chegue ao fim.

10.7 Cursor com for update

10.7. CURSOR COM FOR UPDATE

Quando queremos atualizar ou excluir linhas de uma tabela, podemos fazer isto com o auxílio de um cursor `for update`. Podemos criar um cursor baseado na tabela em questão e utilizar a instrução `for update` para garantir que enquanto ele estiver varrendo as linhas da tabela, nenhuma outra sessão possa estar manipulando os mesmos dados. Isso também acontece mesmo que você não venha a atualizá-los. O simples fato de estar abrindo um cursor `for update` já faz com que o acesso seja exclusivo. Veja a seguir um exemplo de cursor `for update`.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2) is
  4      select ename, job, dname
  5      from   emp, dept
  6      where  emp.deptno = dept.deptno
  7      and    dept.loc    = pdname
  8      for update;
  9    --
 10    r1 c1%rowtype;
 11    --
 12  begin
 13    open c1( pdname => 'DALLAS');
 14    loop
 15      if c1%isopen then
 16        fetch c1 into r1;
 17        if c1%notfound then
 18          close c1;
 19          exit;
 20        else
 21          dbms_output.put_line(
              'Nome: '||r1.ename||' Cargo: '||r1.job);
 22        end if;
```

```
23      else
24        dbms_output.put_line('O Cursor não foi aberto!');
25        exit;
26      end if;
27    end loop;
28  end;
29  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Utilizando `for update`, o Oracle realiza um `lock` nas linhas da tabela garantindo exclusividade para manipulá-las. Após um `rollback` ou `commit`, as linhas voltam a ser liberadas. Este comando pode ser utilizado de duas formas, sendo acompanhado ou não por nomes de colunas específicas. Quando temos um `select` com várias tabelas na cláusula `from` nós podemos determinar quais, ou qual, tabelas devem ser locadas informando suas colunas. Caso queira que todas as tabelas sejam locadas use somente `for update`. Agora veja o mesmo exemplo, informando as colunas para a locação da tabela.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2) is
  4      select ename, job, dname
  5      from   emp, dept
  6      where  emp.deptno = dept.deptno
  7      and    dept.loc    = pdname
  8      for update of ename, dname;
  9    --
 10    r1 c1%rowtype;
 11    --
 12  begin
 13    open c1( pdname => 'DALLAS');
 14    loop
 15      if c1%isopen then
 16        fetch c1 into r1;
```

```
17          if c1%notfound then
18            close c1;
19            exit;
20          else
21            dbms_output.put_line(
               'Nome: '||r1.ename||' Cargo: '||r1.job);
22          end if;
23        else
24          dbms_output.put_line('O Cursor não foi aberto!');
25          exit;
26        end if;
27      end loop;
28    end;
29    /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo estamos locando as tabelas `EMP` e `DEPT`, pois estamos informando colunas de ambas às tabelas. Contudo, podemos locar uma única tabela informando uma ou mais colunas que fazem parte dela. Segue um exemplo.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2) is
  4      select ename, job, dname
  5      from   emp, dept
  6      where  emp.deptno = dept.deptno
  7      and    dept.loc   = pdname
  8      for update of ename;
  9    --
 10    r1 c1%rowtype;
 11    --
 12  begin
 13    open c1( pdname => 'DALLAS');
 14    loop
 15      if c1%isopen then
```

```
16          fetch c1 into r1;
17          if c1%notfound then
18            close c1;
19            exit;
20          else
21            dbms_output.put_line(
               'Nome: '||r1.ename||' Cargo: '||r1.job);
22          end if;
23        else
24          dbms_output.put_line('O Cursor não foi aberto!');
25          exit;
26        end if;
27      end loop;
28    end;
29    /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Enquanto a tabela estiver em `lock` qualquer sessão que tentar manipular seus dados, exceto `select`, não terá êxito. A sessão ficará esperando a liberação até que o programa que lockou as linhas as libere. Feito isso, qualquer outra sessão poderá efetuar alterações.

Embora o programa que contém um cursor `for update` tenha exclusividade de acesso, ele só conseguirá realizar um `lock` caso as tabelas que fazem parte deste `lock` não estejam lockadas por outras sessões. Se isso estiver ocorrendo, o programa também sofrerá uma espera até que outra sessão libere tais tabelas.

Quando executamos alguma operação em uma determinada tabela, e a mesma está lockada, acontece uma "espera" pelo recurso. Isso quer dizer que sua sessão ficará tentando executar os comandos até que o recurso esteja disponível ou caso você cancele a operação.

Quando você estiver utilizando um cursor com `for update` e não quiser que, ao executá-lo, ele fique esperando por um recurso, por exemplo, uma tabela que esteja sendo locada por outra sessão, use a diretriz `nowait`. Feito isto, o programa não ficará em espera e será disparado uma `exception` in-

formando que o recurso em questão está ocupado.

```
SQL> declare
  2    --
  3    cursor c1( pdname varchar2) is
  4      select ename, job, dname
  5      from   emp, dept
  6      where  emp.deptno = dept.deptno
  7      and    dept.loc    = pdname
  8      for update of ename nowait;
  9    --
 10    r1 c1%rowtype;
 11    --
 12  begin
 13    open c1( pdname => 'DALLAS');
 14    loop
 15      if c1%isopen then
 16        fetch c1 into r1;
 17        if c1%notfound then
 18          close c1;
 19          exit;
 20        else
 21          dbms_output.put_line(
              'Nome: '||r1.ename||' Cargo: '||r1.job);
 22        end if;
 23      else
 24        dbms_output.put_line('O Cursor não foi aberto!');
 25        exit;
 26      end if;
 27    end loop;
 28  end;
 29  /
declare
*
ERRO na linha 1:
ORA-00054: o recurso está ocupado e é obtido com o NOWAIT
especificado
ORA-06512: em line 4
ORA-06512: em line 13
```

```
SQL>
```

Com o `for update` para auxiliar na exclusão e atualização das linhas de uma tabela, é possível utilizar um recurso muito poderoso que traz simplicidade e ótimos ganhos de performance. Quando você estiver executando um comando `update` ou `delete` para atualizar linhas de uma tabela existente no próprio cursor, use `current of`. Esta cláusula faz com que o Oracle utilize o `rowid` da linha, que é o acesso mais direto e rápido ao registro. Observe o exemplo.

```
SQL> declare
  2    --
  3    cursor c1( pdeptno number) is
  4      select *
  5      from   emp
  6      where  deptno = pdeptno
  7      for update of sal nowait;
  8    --
  9    r1 c1%rowtype;
 10    --
 11    wreg_excluidos number default 0;
 12    --
 13  begin
 14    open c1( pdeptno => 10);
 15    loop
 16      fetch c1 into r1;
 17      exit when c1%notfound;
 18      --
 19      update emp set sal = sal + 100.00
 20      where current of c1;
 21      --
 22      wreg_excluidos := wreg_excluidos + sql%rowcount;
 23      --
 24    end loop;
 25    --
 26    dbms_output.put_line(wreg_excluidos||'
        registros excluídos!');
```

```
 27    --
 28   end;
 29   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note no código que, ao utilizar o comando `update`, não foi preciso informar uma cláusula `where` baseada em alguma chave da tabela, como por exemplo, `empno`. Como os registros estão reservados para o programa, através do `for update`, podemos atualizar as linhas usando como critério a referencia da linha do cursor. Dessa forma, o Oracle garante a integridade dos dados. O mesmo pode ser usado para o comando `delete`.

Este recurso também funciona para quando estivermos usando mais de uma tabela na cláusula `from`. O único ponto de observação é que não podemos deixar de definir as colunas para o `for update`, tão pouco definir colunas de diferentes tabelas se quisermos utilizar o `current of`. Isso porque, nestes casos, o Oracle só pode trabalhar com `rowids` de uma única tabela. Veja alguns exemplos:

```
SQL> declare
  2    --
  3    cursor c1( pdloc varchar2) is
  4      select ename, dname
  5      from   emp, dept
  6      where  emp.deptno  = dept.deptno
  7      and    dept.loc    = pdloc
  8      for update of  emp.ename, dept.loc;
  9    --
 10    r1 c1%rowtype;
 11    --
 12    wreg_atua_dep number default 0;
 13    wreg_excl_emp number default 0;
 14    --
 15   begin
 16    open c1( pdloc => 'NEW YORK' );
 17    loop
```

```
18        fetch c1 into r1;
19        exit when c1%notfound;
20        --
21        update dept set loc = 'FLORIDA' where current of c1;
22        --
23        wreg_atua_dep := wreg_atua_dep + sql%rowcount;
24        --
25        delete emp where current of c1;
26        --
27        wreg_excl_emp := wreg_excl_emp + sql%rowcount;
28        --
29      end loop;
30      --
31      dbms_output.put_line(wreg_atua_dep||' registros de
         departamentos atualizados!');
32      dbms_output.put_line(wreg_excl_emp||' registros de
         empregados excluídos!');
33      --
34    end;
35    /
0 registros de departamentos atualizados!
0 registros de empregados excluídos!

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo estamos tentando atualizar a tabela `DEPT` e também excluir registros da tabela `EMP`. Entretanto, mesmo informando uma coluna de cada tabela no `for update`, e o Oracle não apresentando qualquer problema na execução, os registros não sofreram qualquer modificação. Vamos tentar algo diferente no exemplo a seguir.

```
SQL> declare
  2      --
  3      cursor c1( pdloc varchar2) is
  4        select ename, dname
  5        from   emp, dept
  6        where  emp.deptno  = dept.deptno
```

```
  7      and    dept.loc    = pdloc
  8      for update of  emp.ename, dept.loc;
  9    --
 10    r1 c1%rowtype;
 11    --
 12    wreg_atua_dep number default 0;
 13    wreg_excl_emp number default 0;
 14    --
 15  begin
 16    open c1( pdloc => 'NEW YORK' );
 17    loop
 18      fetch c1 into r1;
 19      exit when c1%notfound;
 20      --
 21      delete emp where current of c1;
 22      --
 23      wreg_excl_emp := wreg_excl_emp + sql%rowcount;
 24      --
 25    end loop;
 26    --
 27    dbms_output.put_line(wreg_excl_emp||' registros de
        empregados excluídos!');
 28    --
 29  end;
 30  /
0 registros de empregados excluídos!

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Mesmo retirando um dos comandos, o Oracle não consegue concretizar as alterações. Portanto, o problema não está na quantidade ou distinção de comandos dentro do bloco do cursor e, sim, na sua definição. Vamos deixar na cláusula `for update` somente a coluna referente à tabela `EMP`.

```
SQL> declare
  2    --
  3    cursor c1( pdloc varchar2) is
```

```
 4       select ename, dname
 5       from   emp, dept
 6       where  emp.deptno  = dept.deptno
 7       and    dept.loc    = pdloc
 8       for update of  emp.ename;
 9     --
10     r1 c1%rowtype;
11     --
12     wreg_atua_dep number default 0;
13     wreg_excl_emp number default 0;
14     --
15   begin
16     open c1( pdloc => 'NEW YORK' );
17     loop
18       fetch c1 into r1;
19       exit when c1%notfound;
20       --
21       delete emp where current of c1;
22       --
23       wreg_excl_emp := wreg_excl_emp + sql%rowcount;
24       --
25     end loop;
26     --
27     dbms_output.put_line(wreg_excl_emp||' registros de
        empregados excluídos!');
28     --
29   end;
30   /
3 registros de empregados excluídos!

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Agora nosso comando funcionou com êxito. Foi utilizada mais de uma tabela na cláusula `from`, todavia, na cláusula `for update` definimos apenas a coluna relacionada à tabela `EMP`. Lembre-se que nos casos onde o `select` possua mais de uma tabela na cláusula `from`, se não relacionarmos a coluna na cláusula `for update`, lockando a tabela, não poderemos utilizar a cláu-

sula `current of`. Se isso acontecer, o Oracle dispara uma `exception` gerando um erro. Nos casos onde temos apenas uma tabela na cláusula `from` podemos utilizar somente `for update`. Veja os exemplos.

```
SQL> declare
  2    --
  3    cursor c1( pdloc varchar2) is
  4      select ename, dname
  5      from   emp, dept
  6      where  emp.deptno  = dept.deptno
  7      and    dept.loc    = pdloc
  8      for update of dept.loc;
  9    --
 10    r1 c1%rowtype;
 11    --
 12    wreg_atua_dep number default 0;
 13    wreg_excl_emp number default 0;
 14    --
 15  begin
 16    open c1( pdloc => 'NEW YORK' );
 17    loop
 18      fetch c1 into r1;
 19      exit when c1%notfound;
 20      --
 21      delete emp where current of c1;
 22      --
 23      wreg_excl_emp := wreg_excl_emp + sql%rowcount;
 24      --
 25    end loop;
 26    --
 27    dbms_output.put_line(wreg_excl_emp||' registros de
        empregados excluídos!');
 28    --
 29  end;
 30  /
 declare
*
ERRO na linha 1:
ORA-01410: ROWID inválido
```

```
ORA-06512: em line 21


SQL>
```

Aqui tentamos utilizar `current of` em um `delete` na tabela `EMP`, sendo que em nosso `for update` definimos uma coluna da tabela `DEPT`.

```
SQL> declare
  2    --
  3    cursor c1( pdloc varchar2) is
  4      select dname
  5      from   dept
  6      where  dept.loc = pdloc
  7      for update;
  8    --
  9    r1 c1%rowtype;
 10    --
 11    wreg_atua_dep number default 0;
 12    --
 13  begin
 14    open c1( pdloc => 'NEW YORK' );
 15    loop
 16      fetch c1 into r1;
 17      exit when c1%notfound;
 18      --
 19      update dept set loc = 'FLORIDA' where current of c1;
 20      --
 21      wreg_atua_dep := wreg_atua_dep + sql%rowcount;
 22      --
 23    end loop;
 24    --
 25    dbms_output.put_line(wreg_atua_dep||' registros de
        departamentos atualizados!');
 26    --
 27  end;
 28  /
1 registros de departamentos atualizados!
```

```
Procedimento PL/SQL concluído com sucesso.

SQL>
```

Aqui o comando foi executado de forma correta e com sucesso. Como o `select` do nosso cursor é formado apenas por uma tabela, não foi preciso especificar colunas na cláusula `for update`.

Capítulo 11

Funções de caracteres e operadores aritméticos

Funções de caracteres e de cálculos também podem ser usadas nas expressões SQL. Através delas podem-se modificar os dados, tanto no que diz respeito aos valores selecionados, como também na forma que são apresentados, por exemplo, separar informações dentro de uma determinada String, concatenar caracteres, definir a caixa das letras (maiúsculas, minúsculas e intercaladas) etc.

Já os operadores aritméticos podem ser utilizados para a inserção de cálculos dentro dos comandos SQL. Cálculos estes referentes à soma, subtração, divisão e multiplicação. Vale salientar que estas funções e operadores podem ser utilizados em qualquer cláusula SQL exceto na cláusula `from`.

11.1 Funções de caracteres

- `INITCAP`: retorna o primeiro caractere de cada palavra em maiúscula.
- `LOWER`: força caracteres maiúsculos aparecerem em minúsculos.
- `UPPER`: força caracteres minúsculos aparecerem em maiúsculos.
- `SUBSTR`: extrai um trecho de uma string, começando por uma posição inicial e a partir desta posição conta com base na quantidade solicitada.
- `to_char`: converte um valor numérico para uma string de caracteres. Também é utilizada para inserir máscara em campos numéricos e de data.
- `INSTR`: retorna a posição do primeiro caractere encontrado, passado como parâmetro.
- `LENGTH`: traz o tamanho dos caracteres em bytes.
- `RPAD`: faz alinhamento à esquerda e preenche com caracteres à direita, até uma determinada posição. Ambos os valores são passados como parâmetro.
- `LPAD`: faz alinhamento à direita e preenche com caracteres à esquerda, até uma determinada posição. Ambos os valores são passados como parâmetro.

Seguem exemplos do uso destas funções:

```
SQL> declare
  2    --
  3    wnome1 varchar2(100) default 'analista de sistemas';
  4    wnome2 varchar2(100) default 'PEDREIRO';
  5    wnome3 varchar2(100) default 'padeiro';
  6    --
  7  begin
  8    --
  9    wnome1 := initcap(wnome1);
 10    wnome2 := lower(wnome2);
```

```
 11     wnome3 := upper(wnome3);
 12     --
 13     dbms_output.put_line(wnome1);
 14     dbms_output.put_line(wnome2);
 15     dbms_output.put_line(wnome3);
 16     --
 17   end;
 18   /
Analista De Sistemas
pedreiro
PADEIRO

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções `initcap`, `lower` e `upper`. Note que os valores iniciais das variáveis `wnome1`, `wnome2` e `wnome3` são alterados conforme a ação de cada função.

```
SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa

SQL> declare
  2     wregion_name_short varchar2(500);
  3   begin
  4     --
  5     for r1 in (select region_name from regions) loop
  6     --
  7     wregion_name_short := upper(substr(r1.region_name,1,2));
  8     --
  9     dbms_output.put_line(wregion_name_short);
 10     --
```

```
 11     end loop;
 12     --
 13   end;
 14   /
EU
AM
AS
MI

PL/SQL procedure successfully completed.
SQL>
```

Esse exemplo mostra o uso das funções `substr` e `upper`. O exemplo retorna os nomes das regiões mostrando apenas parte da string referente a estes nomes.

```
SQL> declare
  2      wsalario varchar2(200);
  3   begin
  4     --
  5     for r1 in (select ename, sal from emp where comm is
                   null) loop
  6       --
  7       wsalario := 'R$ '||to_char(r1.sal,'fm999G990D00');
  8       --
  9       dbms_output.put_line('Nome: '||r1.ename||'
                                Salário: '||wsalario);
 10       --
 11     end loop;
 12     --
 13   end;
 14   /
Nome: SMITH Salário: R$ 800.00
Nome: JONES Salário: R$ 2,975.00
Nome: BLAKE Salário: R$ 2,850.00
Nome: CLARK Salário: R$ 2,450.00
Nome: SCOTT Salário: R$ 3,000.00
Nome: KING Salário: R$ 5,000.00
Nome: ADAMS Salário: R$ 1,100.00
```

```
Nome: JAMES Salário: R$ 950.00
Nome: FORD Salário: R$ 3,000.00
Nome: MILLER Salário: R$ 1,300.00

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função `to_char`. A função foi utilizada para formatar os valores da coluna `SAL` (salários).

```
SQL> declare
  2    valor number;
  3  begin
  4    --
  5    valor := instr(37462.12,'62');
  6    --
  7    dbms_output.put_line('Posição: '||valor);
  8    --
  9  end;
 10  /
Posição: 4

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função `instr`.

```
SQL> begin
  2    --
  3    for r1 in (select first_name from employees where
                   length(first_name) > 10) loop
  4      --
  5      dbms_output.put_line('Nome: '||r1.first_name);
  6      --
  7    end loop;
  8    --
  9  end;
 10  /
```

```
Nome: Christopher
Nome: Jose Manuel

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função `length`.

```
SQL> declare
  2    wlast_name varchar2(200);
  3    wsalary    varchar2(50);
  4  begin
  5    --
  6    for r1 in (select last_name, salary
  7               from employees where department_id = 30) loop
  8      --
  9      wlast_name := rpad(r1.last_name,12,'++++');
 10      wsalary    := lpad(r1.salary,7,'0');
 11      --
 12      dbms_output.put_line('Último Nome: '||wlast_name||'
                                Salário: '||wsalary);
 13      --
 14    end loop;
 15    --
 16  end;
 17  /
Último Nome: Raphaely++++ Salário: 0011000
Último Nome: Khoo++++++++ Salário: 0003100
Último Nome: Baida+++++++ Salário: 0002900
Último Nome: Tobias++++++ Salário: 0002800
Último Nome: Himuro++++++ Salário: 0002600
Último Nome: Colmenares++ Salário: 0002500

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções `rpad` e `lpad`.

11.2 Funções de cálculos

- ROUND: arredonda valores com casas decimais.
- TRUNC: trunca valores com casas decimais.
- MOD: mostra o resto da divisão de dois valores.
- SQRT: retorna a raiz quadrada de um valor.
- POWER: retorna um valor elevado a outro valor.
- ABS: retorna o valor absoluto.
- CEIL: retorna o menor inteiro, maior ou igual a valor.
- FLOOR: retorna o maior inteiro, menor ou igual a valor.
- SIGN: se valor maior que o retornar +1. Se valor menor que o retornar -1. Se valor igual a o retorna o.

```
SQL> declare
  2    wsal_calc number;
  3    wcomm     number;
  4  begin
  5    --
  6    for r1 in (select sal, ename from emp where
                  deptno = 20) loop
  7      --
  8      wsal_calc := (r1.sal / 2.7);
  9      --
 10      dbms_output.put_line('Nome: '||r1.ename||'
                                  Salário: '||wsal_calc);
 11      --
 12    end loop;
 13    --
 14    dbms_output.put_line('-');
 15    --
 16    for r1 in (select comm, ename from emp where comm is
                  not null) loop
 17      --
```

```
18        wcomm := round((r1.comm / 2.7));
19        --
20        dbms_output.put_line('Nome: '||r1.ename||'
                                     Comissão: '||wcomm);
21        --
22      end loop;
23      --
24      dbms_output.put_line('-');
25      --
26      for r1 in (select sal, ename
27                 from emp where empno between 7500 and 7700) loop
28        --
29        wsal_calc := round((r1.sal / 2.7),2);
30        --
31        dbms_output.put_line('Nome: '||r1.ename||' Salário: '||
32                                  r1.sal||' Salário Calc: '||wsal_calc);
33        --
34      end loop;
35      --
36    end;
37    /
Nome: SMITH Salário: 296.2962962962962962962962962962962963
Nome: JONES Salário: 1101.851851851851851851851851851851852
Nome: SCOTT Salário: 1111.111111111111111111111111111111111
Nome: ADAMS Salário: 407.4074074074074074074074074074074074
Nome: FORD Salário: 1111.111111111111111111111111111111111
-
Nome: ALLEN Comissão: 111
Nome: WARD Comissão: 185
Nome: MARTIN Comissão: 519
Nome: TURNER Comissão: 0
-
Nome: WARD Salário: 1250 Salário Calc: 462.96
Nome: JONES Salário: 2975 Salário Calc: 1101.85
Nome: MARTIN Salário: 1250 Salário Calc: 462.96
Nome: BLAKE Salário: 2850 Salário Calc: 1055.56

PL/SQL procedure successfully completed.
```

```
SQL>
```

Esse exemplo mostra o uso da função `round`. No exemplo, são mostradas diferentes formas de chamada a esta função.

```
SQL> declare
  2    wsal_calc number;
  3  begin
  4    --
  5    for r1 in (select first_name, salary, job_id
  6                 from employees where job_id = 'MK_MAN') loop
  7      --
  8      wsal_calc := (r1.salary / 2.7);
  9      --
 10      dbms_output.put_line('Nome: '||r1.first_name||'
                              Salário Calc.: '||
 11                           wsal_calc||' Salário: '||r1.salary
                              ||' Job: '||r1.job_id);
 12      --
 13    end loop;
 14    --
 15    dbms_output.put_line('-');
 16    --
 17    for r1 in (select last_name, salary, email
 18                 from employees where email = 'NSARCHAN') loop
 19      --
 20      wsal_calc := trunc((r1.salary / 2.7));
 21      --
 22      dbms_output.put_line('Nome: '||r1.last_name||'
                              Sal. Calc: '||wsal_calc||
 23                           ' Sal. : '||r1.salary||'
                              Email. : '||r1.email);
 24      --
 25    end loop;
 26    --
 27    dbms_output.put_line('-');
 28    --
 29    for r1 in (select last_name, salary
 30                 from employees where employee_id between 100
```

```
                  and 105) loop
 31       --
 32       wsal_calc := trunc((r1.salary / 2.7),2);
 33       --
 34       dbms_output.put_line('Nome: '||r1.last_name||'
                                Sal. Calc: '||wsal_calc||
 35                             ' Sal. : '||r1.salary);
 36       --
 37     end loop;
 38     --
 39   end;
 40   /
Nome: Michael Salário Calc.:
4814.814814814814814814814814814815 Salário:
13000 Job: MK_MAN
-
Nome: Sarchand Sal. Calc: 1555 Sal. : 4200 Email. : NSARCHAN
-
Nome: King Sal. Calc: 8888.88 Sal. : 24000
Nome: Kochhar Sal. Calc: 6296.29 Sal. : 17000
Nome: De Haan Sal. Calc: 6296.29 Sal. : 17000
Nome: Hunold Sal. Calc: 3333.33 Sal. : 9000
Nome: Ernst Sal. Calc: 2222.22 Sal. : 6000
Nome: Austin Sal. Calc: 1777.77 Sal. : 4800

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função `trunc`. No exemplo, são mostradas diferentes formas de chamada a esta função.

```
SQL> declare
  2     wres number;
  3   begin
  4     --
  5     wres := mod(10,2);
  6     dbms_output.put_line('Resultado: '||wres);
  7     --
```

```
  8     wres := sqrt(64);
  9     dbms_output.put_line('Resultado: '||wres);
 10     --
 11     wres := power(8,2);
 12     dbms_output.put_line('Resultado: '||wres);
 13     --
 14   end;
 15   /
Resultado: 0
Resultado: 8
Resultado: 64

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções `mod`, `sqrt` e `power`.

```
SQL> declare
  2     wres number;
  3   begin
  4     --
  5     wres := abs(-20);
  6     dbms_output.put_line('Resultado: '||wres);
  7     --
  8     wres := ceil(10.2);
  9     dbms_output.put_line('Resultado: '||wres);
 10     --
 11     wres := floor(10.2);
 12     dbms_output.put_line('Resultado: '||wres);
 13     --
 14   end;
 15   /
Resultado: 20
Resultado: 11
Resultado: 10

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo é mostrado o uso das funções `abs`, `ceil` e `floor`.

```
SQL> declare
  2     wres number;
  3   begin
  4     --
  5     wres := sign(-2000);
  6     dbms_output.put_line('Resultado: '||wres);
  7     --
  8     wres := sign(2000);
  9     dbms_output.put_line('Resultado: '||wres);
 10     --
 11     wres := sign(0);
 12     dbms_output.put_line('Resultado: '||wres);
 13     --
 14   end;
 15   /
Resultado: -1
Resultado: 1
Resultado: 0

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função `sign`. No exemplo, são mostrados diferentes parâmetros na chamada desta função.

11.3 Operadores aritméticos

- * Multiplicação
- / Divisão
- + Adição
- – Subtração

```
SQL> declare
  2     wsal_calc number;
```

```
  3  begin
  4   --
  5   for r1 in (select sal from emp where comm is not null) loop
  6     --
  7     wsal_calc := (r1.sal*2/3);
  8     --
  9     dbms_output.put_line('Salário Calc.: '||wsal_calc||'
                                Salário: '||r1.sal);
 10     --
 11   end loop;
 12   --
 13  end;
 14  /
Salário Calc.: 1066.666666666666666666666666666666666667
Salário: 1600
Salário Calc.: 833.3333333333333333333333333333333333333
Salário: 1250
Salário Calc.: 833.3333333333333333333333333333333333333
Salário: 1250
Salário Calc.: 1000 Salário: 1500

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções aritméticas de multiplicação e divisão.

```
SQL> declare
  2    --
  3    wsal_calc number;
  4    --
  5  begin
  6    --
  7    for r1 in (select ename, sal
  8                 from emp where deptno = 30) loop
  9      --
 10      wsal_calc := round((r1.sal*2)/3+100.00,2);
 11      --
```

```
 12       dbms_output.put_line('Nome: '||r1.ename||'
                               Salário Calc.: '||
 13                            wsal_calc||' Salário: '||r1.sal);
 14       --
 15     end loop;
 16     --
 17   end;
 18   /
Nome: ALLEN Salário Calc.: 1166.67 Salário: 1600
Nome: WARD Salário Calc.: 933.33 Salário: 1250
Nome: MARTIN Salário Calc.: 933.33 Salário: 1250
Nome: BLAKE Salário Calc.: 2000 Salário: 2850
Nome: TURNER Salário Calc.: 1100 Salário: 1500
Nome: JAMES Salário Calc.: 733.33 Salário: 950

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções aritméticas de multiplicação, divisão e soma.

```
SQL> declare
  2     --
  3     wdt_emissao number;
  4     wpremiacao  number;
  5     --
  6     cursor c1 is
  7       select ename
  8             ,dname
  9             ,hiredate
 10             ,sal
 11       from   emp   e
 12             ,dept d
 13       where  e.deptno = d.deptno
 14       and    trunc((sysdate - hiredate) / 365) = 30;
 15     --
 16   begin
 17     --
```

```
18    for r1 in c1 loop
19      --
20      wdt_emissao := trunc((sysdate - r1.hiredate) / 365);
21      wpremiacao  :=
           (r1.sal/10*trunc((sysdate - r1.hiredate) / 365));
22      --
23      dbms_output.put_line('Nome: '||r1.ename||'
                             Dt. Emissão: '||
24                           wdt_emissao||' Premiação:
                             '||wpremiacao);
25      --
26    end loop;
27    --
28  end;
29  /
Nome: KING Dt. Emissão: 30 Premiação: 15000
Nome: CLARK Dt. Emissão: 30 Premiação: 7350
Nome: FORD Dt. Emissão: 30 Premiação: 9000
Nome: JONES Dt. Emissão: 30 Premiação: 8925
Nome: SMITH Dt. Emissão: 30 Premiação: 2400
Nome: JAMES Dt. Emissão: 30 Premiação: 2850
Nome: TURNER Dt. Emissão: 30 Premiação: 4500
Nome: BLAKE Dt. Emissão: 30 Premiação: 8550
Nome: MARTIN Dt. Emissão: 30 Premiação: 3750
Nome: WARD Dt. Emissão: 30 Premiação: 3750
Nome: ALLEN Dt. Emissão: 30 Premiação: 4800

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso das funções aritméticas de multiplicação, divisão e subtração.

Capítulo 12

Funções de agregação (grupo)

As funções de agregação são responsáveis por agrupar vários valores e retornar somente um único valor para um determinado grupo. As funções de agregação, também chamadas de funções de grupo, são especificadas no comando `select` de uma coluna e são seguidas pela coluna à qual se aplicam. A utilização das funções de agregação pode implicar no uso da cláusula `group by`. Isso acontece porque, ao informarmos colunas com funções e colunas sem funções em um mesmo `select`, precisamos agrupar as colunas que não estão sendo afetadas pelo agrupamento causado pelas funções. Veja a ilustração:

Empregado	Departamento	Salário
SMITH	RESEARCH	800
ALLEN	SALES	1600
WARD	SALES	1250
JONES	RESEARCH	2975
MARTIN	SALES	1250
BLAKE	SALES	2850
CLARK	ACCOUNTING	2450

Fig. 12.1: Dados de empregados

Nesta ilustração, temos alguns empregados, departamentos e valores de salário. Nosso objetivo aqui é tentar de alguma forma somar todos os salários por departamento, ou seja, ver quanto de salário temos para os empregados referentes aos departamentos `RESEARCH`, `SALES` e `ACCOUNTING`. Da forma como os dados estão dispostos, não conseguimos visualizar isto, pois se tentarmos agrupar departamento, não conseguiremos, visto que o agrupamento consiste em selecionar dados que possuem o mesmo valor e torná-lo único para cada conjunto de dados.

Por exemplo, temos os departamentos `RESEARCH`, `SALES` e `ACCOUNTING` aparecendo diversas vezes. Se agruparmos, teremos um único registro para o departamento `RESEARCH`, outro para `SALES`, e outro para `ACCOUNTING`. Entretanto, também estamos selecionando os nomes dos empregados e, na maioria dos casos, cada um possui um nome deferente, impossibilitando que os agrupemos. Se não conseguimos agrupar os empregados, logo não conseguimos agrupar os departamentos. É como se fosse uma sequência. Quando usamos funções de grupo nas colunas de um `select`, temos que agrupar todas as outras, sendo através de uma função de agregação ou sendo pelo uso do `group by`.

Já vimos que se nós quisermos a somatória de todos os salários por departamento não podemos selecionar os empregados, ou melhor, os nomes deles.

Logo, a coluna `Empregado` não poderá aparecer no nosso `select`. Caso contrário, estaríamos incluindo também por empregados, o que nos daria um agrupamento inútil, tendo em vista que cada nome de empregado é diferente.

Sempre quando trabalhamos com agrupamentos, temos que ter em mente a seguinte situação: vai haver colunas que estarão sobre o efeito das funções de agregação, por exemplo, funções de somatória ou de média, e colunas que não estarão sobre o efeito destas funções, mas que precisarão ser agrupadas para que juntas possam formar um conjunto de dados.

Veja a próxima ilustração:

Agrupamento por Group By

Agrupamento por Somatória

Empregado	Departamento	Salário
SMITH	RESEARCH	800
ALLEN	SALES	1600
WARD	SALES	1250
JONES	RESEARCH	2975
MARTIN	SALES	1250
BLAKE	SALES	2850
CLARK	ACCOUNTING	2450

Resultado Final: Valores Não agrupados por Departamento

Fig. 12.2: Dados agrupados parcialmente

Nessa outra ilustração, temos dois grupos. Um grupo formado pelas colunas `Empregado` e `Departamento`, que sofrerão a ação do `group by`, e outro grupo formado apenas pela coluna `Salário`, que sofrerá a ação da nossa função de agregação. Vale ressaltar que nosso objetivo aqui é agrupar os salários por departamento.

Pois bem, como pode ser visto na ilustração, neste caso, não conseguimos montar o agrupamento. Note que na coluna de departamento é possível

agrupar os valores, mas na coluna de empregados isso não é possível. Como a coluna de empregados faz parte do `select`, ela acaba comprometendo todo nosso agrupamento. Atenção a um detalhe – o fato de a coluna `Empregado` estar sendo visualizada primeiro não quer dizer que seja a causa de não conseguirmos agrupar por departamento. A ordem das colunas não altera o resultado. Vamos retirá-la do nosso `select`.

Empregado
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK

Departamento	Salário
RESEARCH	800
SALES	1600
SALES	1250
RESEARCH	2975
SALES	1250
SALES	2850
ACCOUNTING	2450

Fig. 12.3: Processo de agrupamento de salários por departamento

Agora sim. Tiramos a coluna de empregados e ficamos apenas com as colunas `Departamento` e `Salário`.

Veja como ficou nosso agrupamento:

Fig. 12.4: Dados agrupados - Salários x Departamento

Fazendo desta forma conseguiremos alcançar nosso objetivo. Resumindo:

- Devemos saber que para obter sucesso em nossos agrupamentos, as colunas que não estão sendo agrupadas pelas funções de agrupamento devem ser agrupadas pelo `group by`;
- Também podemos agrupar determinadas colunas, mesmo que elas não estejam presentes na cláusula `select`;
- Somente vamos precisar agrupar colunas através do `group by` quando desejarmos mostrar um resultado com base em outro. Exemplo: valores de salário por departamento, quantidades de empregados por departamento e assim por diante. Se quisermos apenas saber a somatória de todos os valores de salário independente do departamento ou de qualquer outra informação, não precisaremos utilizar o `group by`;
- Funções de agregação, no geral, ignoram valores nulos;
- Para realizar o agrupamento de informações o Oracle poderá ordenar ou não as colunas. Caso a coluna que está sobre a ação da função for uma coluna com índice, o banco poderá utilizar este índice. Como os

índices são ordenados, não será necessário ordenar os dados para o agrupamento. Caso contrário, ele vai realizar a ordenação, primeiro, e depois agrupa. Nem todas as funções permitem usar índices.

Agora vamos ver estes conceitos na prática.

Primeiramente, visualizamos o nome de todos os empregados, os nomes dos seus departamentos e seus respectivos salários.

```
SQL> select ename, dname, sal
  2  from emp e, dept d
  3  where e.deptno = d.deptno
  4  order by ename;

ENAME      DNAME                SAL
---------- -------------- ---------
ADAMS      RESEARCH            1100
ALLEN      SALES               1600
BLAKE      SALES               2850
CLARK      ACCOUNTING          2450
FORD       RESEARCH            3000
JAMES      SALES                950
JOHN       RESEARCH            1000
JONES      RESEARCH            2975
KING       ACCOUNTING          5000
MARTIN     SALES               1250
MILLER     ACCOUNTING          1300
SCOTT      RESEARCH            3000
SMITH      RESEARCH             800
TURNER     SALES               1500
WARD       SALES               1250

15 rows selected.

SQL>
```

Através de um programa PL/SQL selecionamos os mesmos dados do `select` anterior, mas agora sumarizando os salários. Note que continuamos selecionando as colunas nome do empregado e departamento do empregado. O objetivo do programa é mostrar a soma dos salários por departamento.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select ename, dname, sum(sal) soma_sal
  5      from emp e, dept d
  6      where e.deptno = d.deptno
  7      order by ename;
  8    --
  9  begin
 10    --
 11    for r1 in c1 loop
 12      --
 13      dbms_output.put_line('Nome: '||r1.ename||'
                             Departamento: '||
 14                          r1.dname||'
                             Soma Sal.: '||r1.soma_sal);
 15      --
 16    end loop;
 17    --
 18  end;
 19  /
declare
*
ERROR at line 1:
ORA-00937: not a single-group group function
ORA-06512: at line 4
ORA-06512: at line 11

SQL>
```

Ao executar o programa, surgiu um erro que, em linhas gerais, quer dizer que o comando `select`, contido no cursor, está tentando utilizar uma função de grupo, juntamente com outras colunas não agrupadas, sem utilizar a cláusula de agrupamento. Como visto nos conceitos apresentados anteriormente, isso não é permitido. Dessa forma, devemos agrupar as colunas que não estão associadas a funções de agrupamento. Veja a seguir como ficou.

```
SQL> declare
  2    --
```

```
 3    cursor c1 is
 4      select ename, dname, sum(sal) soma_sal
 5      from emp e, dept d
 6      where e.deptno = d.deptno
 7      group by ename, dname
 8      order by ename;
 9    --
10  begin
11    --
12    for r1 in c1 loop
13      --
14      dbms_output.put_line('Nome: '||r1.ename||'
                           Departamento: '||
15                         r1.dname||'
                           Soma Sal.: '||r1.soma_sal);
16      --
17    end loop;
18    --
19  end;
20  /
Nome: ADAMS Departamento: RESEARCH Soma Sal.: 1100
Nome: ALLEN Departamento: SALES Soma Sal.: 1600
Nome: BLAKE Departamento: SALES Soma Sal.: 2850
Nome: CLARK Departamento: ACCOUNTING Soma Sal.: 2450
Nome: FORD Departamento: RESEARCH Soma Sal.: 3000
Nome: JAMES Departamento: SALES Soma Sal.: 950
Nome: JONES Departamento: RESEARCH Soma Sal.: 2975
Nome: KING Departamento: ACCOUNTING Soma Sal.: 5000
Nome: MARTIN Departamento: SALES Soma Sal.: 1250
Nome: MILLER Departamento: ACCOUNTING Soma Sal.: 1300
Nome: SCOTT Departamento: RESEARCH Soma Sal.: 3000
Nome: SMITH Departamento: RESEARCH Soma Sal.: 800
Nome: TURNER Departamento: SALES Soma Sal.: 1500
Nome: WARD Departamento: SALES Soma Sal.: 1250

PL/SQL procedure successfully completed.

SQL>
```

Feitos os agrupamentos necessários, voltamos a executar o programa. O resultado foi apresentado logo em seguida. No entanto, veja que algo não saiu como deveria. Os salários não foram sumarizados por departamento e, sim, por empregado. Seria a mesma coisa que não sumarizar. Vamos alterar o comando retirando a coluna nome do empregado do comando SQL.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select dname, sum(sal) soma_sal
  5      from emp e, dept d
  6      where e.deptno = d.deptno
  7      group by ename, dname
  8      order by ename;
  9    --
 10  begin
 11    --
 12    for r1 in c1 loop
 13      --
 14      dbms_output.put_line('Departamento: '||
 15                          r1.dname||' Soma Sal.: '||r1.soma_sal);
 16      --
 17    end loop;
 18    --
 19  end;
 20  /
Departamento: RESEARCH Soma Sal.: 1100
Departamento: SALES Soma Sal.: 1600
Departamento: SALES Soma Sal.: 2850
Departamento: ACCOUNTING Soma Sal.: 2450
Departamento: RESEARCH Soma Sal.: 3000
Departamento: SALES Soma Sal.: 950
Departamento: RESEARCH Soma Sal.: 2975
Departamento: ACCOUNTING Soma Sal.: 5000
Departamento: SALES Soma Sal.: 1250
Departamento: ACCOUNTING Soma Sal.: 1300
Departamento: RESEARCH Soma Sal.: 3000
Departamento: RESEARCH Soma Sal.: 800
Departamento: SALES Soma Sal.: 1500
```

```
Departamento: SALES Soma Sal.: 1250

PL/SQL procedure successfully completed.

SQL>
```

Ao retirar a coluna, o erro persiste. Isso acontece pois não adianta retirar apenas da seleção, mas também é necessário retirar do agrupamento. Veja que na linha 7 ainda consta a coluna `ename`. Veja a seguir, como deve ficar o `select`, para que o programa consiga atingir o objetivo proposto.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select dname, sum(sal) soma_sal
  5      from emp e, dept d
  6      where e.deptno = d.deptno
  7      group by dname;
  8    --
  9  begin
 10    --
 11    for r1 in c1 loop
 12      --
 13      dbms_output.put_line('Departamento: '||
 14                           r1.dname||' Soma Sal.: '||r1.soma_sal);
 15      --
 16    end loop;
 17    --
 18  end;
 19  /
Departamento: ACCOUNTING Soma Sal.: 8750
Departamento: RESEARCH Soma Sal.: 10875
Departamento: SALES Soma Sal.: 9400

PL/SQL procedure successfully completed.

SQL>
```

Selecionando apenas a coluna referente ao nome do departamento e su-

marizando os salários, através da função de agregação `sum`, temos como resultado a soma dos salários por departamento.

Segue as funções de agregação mais utilizadas:

- **count**: retorna a quantidade de incidências de registros.
- **sum**: exibe a soma dos valores dos registros.
- **avg**: exibe a média dos valores de uma determinada coluna.
- **min**: exibe o menor valor de uma coluna.
- **max**: retorna o maior valor de uma coluna.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select count(employee_id) cont_emp, country_name
  5      from   employees e
  6            ,departments d
  7            ,locations l
  8            ,countries c
  9      where  e.department_id = d.department_id
 10      and    d.location_id = l.location_id
 11      and    l.country_id = c.country_id
 12      group  by country_name
 13      order  by country_name;
 14    --
 15  begin
 16    --
 17    for r1 in c1 loop
 18      --
 19      dbms_output.put_line('Qtde. Empregados: '||
 20                           r1.cont_emp||' Cidade: '||r1.country_name);
 21      --
 22    end loop;
 23    --
 24  end;
 25  /
Qtde. Empregados: 2 Cidade: Canada
```

```
Qtde. Empregados: 1 Cidade: Germany
Qtde. Empregados: 35 Cidade: United Kingdom
Qtde. Empregados: 68 Cidade: United States of America

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função: `count`. No exemplo, o objetivo é selecionar a quantidade de empregados por país.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select count(*) cont_emp, country_name
  5      from   employees e
  6            ,departments d
  7            ,locations l
  8            ,countries c
  9      where  e.department_id = d.department_id
 10      and    d.location_id = l.location_id
 11      and    l.country_id = c.country_id
 12      group  by country_name
 13     order  by country_name;
 14    --
 15  begin
 16    --
 17    for r1 in c1 loop
 18      --
 19      dbms_output.put_line('Qtde. Empregados: '||
 20                           r1.cont_emp||' Cidade: '||r1.country_name);
 21      --
 22    end loop;
 23    --
 24  end;
 25  /
Qtde. Empregados: 2 Cidade: Canada
Qtde. Empregados: 1 Cidade: Germany
Qtde. Empregados: 35 Cidade: United Kingdom
```

```
Qtde. Empregados: 68 Cidade: United States of America

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo é quase igual ao anterior, apenas por um detalhe. Note que como conhecemos a tabela `employees` e sabemos que existe apenas um registro para cada empregado, podemos usar a função `count` de outra forma, utilizando asterisco (`*`) no lugar da coluna `employee_id`.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select dname, max(hiredate) dt_admissao
  5      from emp e, dept d
  6      where e.deptno = d.deptno
  7      group by dname
  8      order by 2 desc;
  9    --
 10  begin
 11    --
 12    for r1 in c1 loop
 13      --
 14      dbms_output.put_line('Departamento: '||
 15                 r1.dname||' Dt. Admissão: '||r1.dt_admissao);
 16      --
 17    end loop;
 18    --
 19  end;
 20  /
Departamento: RESEARCH Dt. Admissão: 12-JAN-83
Departamento: ACCOUNTING Dt. Admissão: 23-JAN-82
Departamento: SALES Dt. Admissão: 03-DEC-81

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função: `max`. No exemplo, o objetivo é

selecionar a maior data de admissão de cada departamento.

```
SQL> declare
  2    wres date;
  3  begin
  4    select min(hire_date) hire_date_min into wres
        from employees;
  5    --
  6    dbms_output.put_line('Menor Dt. Emissão: '||wres);
  7    --
  8  end;
  9  /
Menor Dt. Emissão: 17-JUN-87

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostra o uso da função: `min`. O objetivo é selecionar a menor data de admissão entre todos os empregados.

Além das funções de agregação e do uso do `group by`, também podemos contar com o `having`, para nos ajudar a restringir registros com base nos valores retornados pelas funções de agregação. O `having` existe pois não podemos utilizar funções de agregação na cláusula `where`.

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select count(employee_id) cont_emp, sum(salary)
          soma_salario, department_name
  5      from   employees e
  6             ,departments d
  7      where  e.department_id = d.department_id
  8      having count(employee_id) > 5
  9      group  by department_name
 10      order  by department_name;
 11    --
 12  begin
 13    --
```

```
14     for r1 in c1 loop
15       --
16       dbms_output.put_line('Qtde. Empregado: '||
17                 r1.cont_emp||' Soma Sal.: '||r1.soma_salario||
18                 ' Depto.: '||r1.department_name);
19       --
20     end loop;
21     --
22   end;
23   /
Qtde. Empregado: 6 Soma Sal.: 51600 Depto.: Finance
Qtde. Empregado: 6 Soma Sal.: 24900 Depto.: Purchasing
Qtde. Empregado: 34 Soma Sal.: 304500 Depto.: Sales
Qtde. Empregado: 45 Soma Sal.: 156400 Depto.: Shipping

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, é mostrado o uso da cláusula `having`. O objetivo é selecionar a quantidade de empregados e a soma dos salários, agrupados por departamento, onde a quantidade de empregados é maior que 5. Observe que a cláusula `having` atua somente após o agrupamento das linhas. Por isso, não seria possível utilizar a cláusula `where`, pois ela atua no momento em que as linhas estão sendo selecionadas, ou seja, antes do agrupamento.

```
SQL> declare
  2     --
  3     cursor c1 is
  4       select department_name, sum(salary) soma_sal,
         country_name
  5       from   employees e
  6             ,departments d
  7             ,locations l
  8             ,countries c
  9       where  e.department_id = d.department_id
 10       and    d.location_id = l.location_id
 11       and    l.country_id = c.country_id
 12       having sum(salary) > (select avg(em.salary)
```

```
13                     from   employees em
14                           ,departments dm
15                           ,locations lm
16                           ,countries cm
17                     where  em.department_id = dm.department_id
18                     and    dm.location_id = lm.location_id
19                     and    lm.country_id = cm.country_id
20                     and    cm.country_id = c.country_id)
21       group by c.country_id
22               ,department_name
23               ,country_name
24       order by country_name
25               ,department_name;
26      --
27    begin
28      --
29      for r1 in c1 loop
30        --
31        dbms_output.put_line('Departamento: '||
32                             r1.department_name||'
                               Soma Sal.: '||r1.soma_sal||
33                             ' Cidade: '||r1.country_name);
34        --
35      end loop;
36      --
37    end;
38    /
Departamento: Marketing Soma Sal.: 19000 Cidade: Canada
Departamento: Sales Soma Sal.: 304500 Cidade: United Kingdom
Departamento: Accounting Soma Sal.: 20300 Cidade: United States
of America
Departamento: Executive Soma Sal.: 58000 Cidade: United States
of America
Departamento: Finance Soma Sal.: 51600 Cidade: United States
of America
Departamento: IT Soma Sal.: 28800 Cidade: United States
of America
Departamento: Purchasing Soma Sal.: 24900 Cidade: United States
of America
```

```
Departamento: Shipping Soma Sal.: 156400 Cidade: United States
of America

PL/SQL procedure successfully completed.

SQL>
```

Já nesse outro exemplo, o objetivo é selecionar a soma dos salários, agrupados por departamento e país, cuja seja maior que a média dos salários por país.

Capítulo 13

Funções de data

Funções de data são utilizadas para manipularmos valores do tipo `date`, como aplicar formatações para uma visualização mais refinada, ou extrair partes de uma data, como as horas, dia do mês ou somente o ano. Seguem algumas delas:

- `add_months`: adiciona meses em uma determinada data.
- `months_between`: retorna a quantidade de meses entre duas datas.
- `next_day`: procura o próximo dia após uma data informada.
- `last_day`: retorna o último dia do mês com base em uma data informada.
- `trunc`: trunca uma data passada por parâmetro. O `trunc` pode ser feito por dia e mês, utilizando o parâmetro `FMT` (formato).

- `sysdate`: retorna a data corrente com base no servidor do banco de dados.
- `sessiontimezone`: mostra o fuso horário com base na sessão aberta no banco de dados, mediante sua localização. Vale lembrar que os fusos horários são calculados com base no meridiano de Greenwich.
- `current_date`: mostra a data corrente com base na zona de tempo da sessão do usuário. A zona de tempo é afetada em relação ao Meridiano. Caso não haja mudanças de zona, esta função terá o mesmo valor que `sysdate`. `sysdate` busca a hora do servidor do banco de dados; mesmo que a sessão tenha sido aberta em uma zona diferente da qual o servidor encontra-se, ele refletirá o horário da zona onde está o servidor. Já o `current_date` refletirá a zona onde foi aberta a sessão.

```
SQL> declare
  2    --
  3    wdt_admissao date;
  4    wsexta       date;
  5    --
  6    cursor c1 is
  7      select first_name
  8             ,hire_date
  9      from   employees
 10      where  to_char(hire_date,'mm') = to_char(sysdate,'mm')
 11      order  by 2;
 12    --
 13  begin
 14    --
 15    for r1 in c1 loop
 16      --
 17      wdt_admissao := to_date(to_char(
                         r1.hire_date,'dd/mm')||'/'||
 18                      to_char(sysdate,'rrrr'),'dd/mm/rrrr');
 19      wsexta       := next_day(to_date(
                         to_char(r1.hire_date,'dd/mm')||'/'
 20                      ||to_char(sysdate,'rrrr'),'dd/mm/rrrr')
                         ,'FRIDAY');
```

```
 21        --
 22        dbms_output.put_line('Nome: '||
 23                             r1.first_name||'
                                Dt. Admissão: '||wdt_admissao||
 24                             ' Sexta de Folga: '||wsexta);
 25        --
 26      end loop;
 27      --
 28    end;
 29    /
Nome: Clara Dt. Admissão: 11-NOV-11 Sexta de Folga: 18-NOV-11
Nome: Sarath Dt. Admissão: 03-NOV-11 Sexta de Folga: 04-NOV-11
Nome: Guy Dt. Admissão: 15-NOV-11 Sexta de Folga: 18-NOV-11
Nome: Kevin Dt. Admissão: 16-NOV-11 Sexta de Folga: 18-NOV-11
Nome: Oliver Dt. Admissão: 23-NOV-11 Sexta de Folga: 25-NOV-11

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo foi mostrado o uso da função `next_day`, onde selecionamos os empregados que têm como mês de admissão o mês corrente. Através dessa função é calculada qual a primeira sexta-feira logo em seguida ao dia da admissão de cada empregado. Este dia será o dia da folga dele.

```
SQL> declare
  2      --
  3      wdt_termino_exp    date;
  4      wqt_meses_trabalho number;
  5      --
  6      cursor c1 is
  7        select ename, dname
  8               ,hiredate
  9        from emp e, dept d
 10        where e.deptno = d.deptno
 11        and add_months(hiredate,350) >= sysdate;
 12
 13      --
 14    begin
```

```
 15     --
 16     for r1 in c1 loop
 17       --
 18       wdt_termino_exp    := add_months(r1.hiredate,3);
 19       wqt_meses_trabalho := to_char(months_between(
                                  sysdate,r1.hiredate),'990D00');
 20       --
 21       dbms_output.put_line('Nome: '||r1.ename||' '||
 22                            'Depto: '||r1.dname||' '||
 23                            'Dt. Admissão: '||r1.hiredate||' '||
 24                            'Término Exp.: '||wdt_termino_exp||' '||
 25                            'Qtde. Meses Trab.: '||wqt_meses_trabalho
 26                            );
 27       --
 28     end loop;
 29     --
 30   end;
 31   /
Nome: ADAMS Depto: RESEARCH Dt. Admissão: 12-JAN-83 Término
Exp.: 12-APR-83
Qtde. Meses Trab.: 346.61
Nome: SCOTT Depto: RESEARCH Dt. Admissão: 09-DEC-82 Término
Exp.: 09-MAR-83
Qtde. Meses Trab.: 347.71

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo foi mostrado o uso das funções `add_months` e `months_between`. O exemplo seleciona os empregados e suas respectivas datas de término de experiência do cargo. Note que limitamos o número de empregados no `select`, através da função `add_months`, para evitar o retorno de todas as linhas da tabela.

```
SQL> declare
  2    wsessiontimezone varchar(10);
  3    wcurrent_date    date;
  4    wsysdate         date;
```

```
  5  begin
  6    wsessiontimezone := sessiontimezone;
  7    wcurrent_date    := current_date;
  8    wsysdate         := sysdate;
  9    --
 10    dbms_output.put_line(
         'Fuso Horário: '||wsessiontimezone||' '||
 11      'Data Corrente: '||wcurrent_date||' '||
 12      'Data Atual: '||wsysdate);
 13    --
 14  end;
 15  /
Fuso Horário: -03:00 Data Corrente: 30-NOV-11 Data Atual:
30-NOV-11

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo foi mostrado o uso das funções `sessiontimezone`, `current_date` e `sysdate`.Exibimos o fuso horário, a data corrente local e data atual do servidor.

```
SQL> declare
  2    --
  3    wlast  date;
  4    wround date;
  5    wtrunc date;
  6    --
  7    cursor c1 is
  8      select ename
  9            ,hiredate
 10      from   emp
 11      where  deptno = 20;
 12    --
 13  begin
 14    --
 15    for r1 in c1 loop
 16      --
```

```
17        wlast  := last_day(r1.hiredate);
18        wround := round(r1.hiredate, 'YEAR');
19        wtrunc := trunc(r1.hiredate, 'YEAR');
20        --
21        dbms_output.put_line('Nome: '||r1.ename||' '||
22                       'Dt. Admissão: '||r1.hiredate||' '||
23                       'Último dia Mês Admissão: '||wlast||' '||
24                       'Arredonda Ano Admissão.: '||wround||' '||
25                       'trunc Ano Admissão: '||wtrunc
26                       );
27        --
28      end loop;
29      --
30    end;
31    /
Nome: SMITH Dt. Admissão: 17-DEC-80 Último dia Mês Admissão:
31-DEC-80 Arredonda
Ano Admissão.: 01-JAN-81 trunc Ano Admissão: 01-JAN-80
Nome: JONES Dt. Admissão: 02-APR-81 Último dia Mês Admissão:
30-APR-81 Arredonda
Ano Admissão.: 01-JAN-81 trunc Ano Admissão: 01-JAN-81
Nome: SCOTT Dt. Admissão: 09-DEC-82 Último dia Mês Admissão:
31-DEC-82 Arredonda
Ano Admissão.: 01-JAN-83 trunc Ano Admissão: 01-JAN-82
Nome: ADAMS Dt. Admissão: 12-JAN-83 Último dia Mês Admissão:
31-JAN-83 Arredonda
Ano Admissão.: 01-JAN-83 trunc Ano Admissão: 01-JAN-83
Nome: FORD Dt. Admissão: 03-DEC-81 Último dia Mês Admissão:
31-DEC-81 Arredonda
Ano Admissão.: 01-JAN-82 trunc Ano Admissão: 01-JAN-81

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo foi mostrado o uso das funções `last_day`, `round` e `trunc`. Perceba várias formas de usar as funções para extrair ou manipular determinadas informações, por exemplo arredondar e truncar datas.

Capítulo 14

Funções de conversão

Em muitos casos, precisamos converter um determinado dado de um tipo para outro. A Oracle disponibiliza funções de conversão para este trabalho. Essas funções, embora sejam simples de serem usadas, ajudam em muito no momento de converter ou formatar dados provenientes dos seus comandos SQL ou programas PL/SQL.

- `to_date`: converte uma string (`char` ou `varchar2`) de caractere para uma data;
- `to_number`: converte uma string (`char` ou `varchar2`) de caractere para um número;
- `to_char`: converte um número ou uma data para uma string de caractere.

Cada função possui suas características e é utilizada para cumprir um objetivo diferente. O que elas possuem em comum é o número de parâmetros. O primeiro parâmetro está relacionado ao valor que deve ser convertido, o segundo corresponde ao formato que você deseja aplicar e, por último e opcional, o parâmetro de linguagem. Trata-se de funções muito utilizadas no dia a dia, e é de fundamental importância conhecê-las e entender como se comportam. Além das mencionadas na lista, a Oracle disponibiliza várias outras funções para as mais diversas situações. Contudo, essas três são as que mais comumente utilizamos na escrita de nossos programas. Para saber mais sobre outras funções de conversão, consulte a documentação disponível no site da Oracle.

14.1 TO_DATE

Esta função é bem interessante. Com um pouco de treino e criatividade, podemos realizar várias conversões que podem ajudar muito no momento de formatar, selecionar ou criticar os dados retornados de um comando SQL. Ela funciona basicamente da seguinte forma: você vai passar um valor caractere para função, juntamente com um formato, que deve ser compatível com o conjunto de caracteres que você passou. Qualquer incompatibilidade, o Oracle gera um erro de conversão.

Exemplo: `to_date('21/05/2009','dd/mm')`. Esta conversão vai gerar um erro, pois você está informando dia, mês e ano, como caractere, mas na máscara só mencionou dia e mês. Quando o Oracle vai fazer a conversão, ele analisa o formato que você está passando como parâmetro e verifica o que é elemento de função e o que é caractere. Neste caso, ele sabe que `dd` e `mm` são elementos conhecidos de dia e mês, e que `/` é um caractere que serve como uma espécie de separador. Depois desta identificação, ele pega cada caractere informado e vai convertendo conforme o formato. 2=d, 1=d,/=/,0=m,5=m etc. Mas quando ele chega à segunda `/` vê que não há formato para o caractere, pois terminou no elemento `m`. Logo, é gerado um erro de conversão. Veja a execução:

```
SQL> declare
  2    wdata date;
```

```
  3  begin
  4    wdata := to_date('21/05/2009','dd/mm');
  5  end;
  6  /
declare
*
ERROR at line 1:
ORA-01830: date format picture ends before converting entire
input string
ORA-06512: at line 4


SQL>
```

O contrário também gera outro erro:

```
SQL> declare
  2    wdata date;
  3  begin
  4    wdata := to_date('21/05','dd/mm/yyyy');
  5  end;
  6  /
declare
*
ERROR at line 1:
ORA-01840: input value not long enough for date format
ORA-06512: at line 4


SQL>
```

Seguem exemplos do uso do `to_date`:

```
SQL> begin
  2    for r1 in (select ename
  3                     ,hiredate
  4               from   emp
  5               where  hiredate > to_date('010182','ddmmrr')) loop
  6      --
  7      dbms_output.put_line('Empregado: '||r1.ename||
```

```
  8                          ' - Data de Admissão: '||r1.hiredate);
  9        --
 10      end loop;
 11    end;
 12    /
Empregado: SCOTT - Data de Admissão: 09-DEC-82
Empregado: ADAMS - Data de Admissão: 12-JAN-83
Empregado: MILLER - Data de Admissão: 23-JAN-82

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo mostrou como é possível converter strings em datas válidas. Veja que na linha 5 do programa temos a função convertendo a string `010182` para uma data utilizando o formato `DDMMRR`.

```
SQL> declare
  2     wdate date;
  3   begin
  4     wdate := to_date('21.05.2009','dd.mm.yyyy');
  5     --
  6     dbms_output.put_line('Data: '||wdate);
  7   end;
  8   /
Data: 21-MAY-09

PL/SQL procedure successfully completed.

SQL>
```

Já nesse outro exemplo, também utilizando `to_date`, note que na linha 4 do programa, temos a função convertendo outra string em uma data, utilizando um formato diferente.

```
SQL> declare
  2     wdate date;
  3   begin
  4     wdate := to_date('April 21','month dd',
```

```
                        'nls_date_language=american');
  5     --
  6     dbms_output.put_line('Data: '||wdate);
  7   end;
  8   /
Data: 21-APR-11

PL/SQL procedure successfully completed.

SQL>
```

Podemos também adicionar à chamada da função aspectos referentes à linguagem. Nesse último exemplo, estamos convertendo uma string por extenso em data, utilizando o formato americano.

```
SQL> declare
  2     wdate date;
  3   begin
  4     wdate := to_date('Abril 21','month dd',
  5                 'nls_date_language=''BRAZILIAN PORTUGUESE''');
  6     --
  7     dbms_output.put_line('Data: '||wdate);
  8   end;
  9   /
Data: 21-APR-11

PL/SQL procedure successfully completed.

SQL>
```

Contudo, como pôde ser visto no exemplo anterior, a visualização continua sendo no formato americano, embora estejamos convertendo a string para data utilizando o formato brasileiro. Lembre-se, conversão não necessariamente, tem a ver com a forma com que o dado será impresso na tela.

```
SQL> declare
  2     wdate date;
  3   begin
  4     wdate := to_date('Abril 21','month XX',
```

```
5                  'nls_date_language=''BRAZILIAN PORTUGUESE''');
6     --
7     dbms_output.put_line('Data: '||wdate);
8   end;
9   /
declare
*
ERROR at line 1:
ORA-01821: date format not recognized
ORA-06512: at line 4


SQL>
```

Esse exemplo, mostra que devemos informar formatos válidos, ou melhor, formatos conhecidos da linguagem. Caso contrário, a conversão não é realizada e erros ocorrerão.

A seguir, algumas limitações com relação a função `to_date`:

- A `String` a ser passada para a conversão não pode conter mais de 220 caracteres;
- Existem vários formatos de máscara disponíveis para a utilização. Qualquer máscara diferente das permitidas pela Oracle gerará um erro de conversão;
- Não pode haver confronto de máscaras. Exemplo: caso você queira utilizar a máscara `HH24` e também solicitar que seja mostrado AM (indicador de antemeridiano para manhã) ou PM (indicador de pós-meridiano para noite).
- Não é permitido especificar elementos de conversão duplicados. Exemplo: 'DD-MM-MM'. Neste caso, o formato para mês aparece duas vezes.

Veja alguns elementos de formatação que podem ser usados:

- `CC`: adiciona 1 aos dois primeiros dígitos do ano (YYYY).
- `SCC`: igual `CC`, prefixando datas `BC` com um sinal negativo.

- `YY`: representa o ano com duas casas.
- `YYYY`: representa o ano com quatro casas.
- `RR`: representa os dois últimos dígitos do ano, mas obedecendo à seguinte regra: soma 1 aos dois primeiros dígitos de `CC` se ano for `< 50` e os últimos 2 dígitos do ano corrente forem `>= 50`. Subtrai 1 de `CC` se ano `>= 50` e os últimos dois dígitos do ano corrente forem `< 50`.
- `RRRR`: representa o ano. Aceita 2 ou 4 dígitos. Se ano informado com 2 dígitos, segue as mesmas regras de RR.
- `YEAR`: escreve o ano por extenso.
- `MM`: número do mês de 01 a 12. 01 = Janeiro, 02 = Fevereiro etc.
- `MONTH`: nome do mês.
- `MON`: representa o nome do mês abreviado com três caracteres.
- `DD`: dia do mês de 1 a 31.
- `DDD`: representa o dia do ano de 1 a 366.
- `DAY`: representa o nome do dia por extenso.
- `HH`, `HH12`, `HH24`: HH e HH12, horas de 1 a 12. HH24, horas de 0 a 23.
- `MI`: equivale aos minutos de 0 a 59.
- `SS`: equivale aos segundos de 0 a 59.
- `SP`: converte o número para seu formato escrito. Disponível apenas para a escrita no idioma inglês.
- `SPTH`: mostra os números de maneira ordinal. 1 = First, 2 = Second etc.
- `FM`: retira espaços em branco proveniente da ausência de caracteres em um formato.

> **Nota:** estes elementos também são utilizados na conversão do tipo `date` para `String`.

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date('2008','yyyy');
  5    --
  6    dbms_output.put_line('Data: '||wdate);
  7  end;
  8  /
Data: 01-NOV-08

PL/SQL procedure successfully completed.

SQL>
```

A função `to_date`, pode converter não só strings representando datas completas, como também strings representando partes de uma data. Esse exemplo mostra a função realizando a conversão da string `2008` em data. Note que, ao ser impresso o valor da variável, a qual recebeu o dado convertido, ele recebeu a atribuição da data atual, modificado apenas pelo ano convertido.

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date(200,'ddd');
  5    --
  6    dbms_output.put_line('Data: '||wdate);
  7  end;
  8  /
Data: 19-JUL-11

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo é similar ao anterior. Aqui está sendo realizada a conversão com base na representação numérica do dia referente ao total de dias do ano.

Uso de YYYY e RRRR

Sabemos que a máscara YYYY representa os quatro dígitos do ano. Opcionalmente, pode-se utilizar YY para mostrar apenas os dois últimos dígitos. Contudo, para corrigir problemas de compatibilidade com a virada do século, a Oracle criou as máscaras RR e RRRR. Veja a aplicação a seguir:

```
SQL> declare
  2    wdate varchar2(50);
  3  begin
  4    wdate := to_char(sysdate,'dd/mm/yyyy');
  5    dbms_output.put_line('Data: '||wdate);
  6    --
  7    wdate := to_char(
        to_date('01/01/49','dd/mm/yy'),'dd/mm/yyyy');
  8    dbms_output.put_line('Data: '||wdate);
  9    --
 10    wdate := to_char(to_date(
        '01/01/50','dd/mm/yy'),'dd/mm/yyyy');
 11    dbms_output.put_line('Data: '||wdate);
 12    --
 13    wdate := to_char(
        to_date('01/01/49','dd/mm/rr'),'dd/mm/rrrr');
 14    dbms_output.put_line('Data: '||wdate);
 15    --
 16    wdate := to_char(
        to_date('01/01/50','dd/mm/rr'),'dd/mm/rrrr');
 17    dbms_output.put_line('Data: '||wdate);
 18    --
 19  end;
 20  /
Data: 30/11/2011
Data: 01/01/2049
Data: 01/01/2050
Data: 01/01/2049
Data: 01/01/1950
```

```
PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, vemos formatações de datas utilizando a função `to_char`, juntamente com os elementos de formatação YY e RR. Note que podemos ter datas diferentes ao converter utilizando RR e YY, dependendo do ano da data informada. Para mais detalhes e compreensão desse exemplo, consulte as regras do uso do formato RR na lista de elementos de formatação vista anteriormente.

14.2 TO_NUMBER

Muito semelhante à função `to_date`, esta função também tem o papel de converter determinados valores. Seu objetivo, no entanto, é fazer a conversão de caracteres para numéricos.

Quando falamos na função `to_date`, foi mencionado que o valor do caractere que está sendo informado como parâmetro deve ser compatível com o formato. Pois bem, quando trabalhamos com `to_number`, o mesmo também acontece. Ao informarmos um valor caractere para a função `to_number`, ele deve ser compatível com o formato que estamos passando para a função. Todavia, existe uma particularidade quanto ao formato para casas decimais e de grupo (milhar, por exemplo).

Para cálculos internos do Oracle, sempre será usado ponto (`.`) como separador decimal e vírgula (`,`) para separador de grupo, como padrão americano. Para atribuições de variáveis do tipo caractere ou para visualização, o Oracle pegará a formatação conforme estiver configurado na variável `nls_numeric_characters`.

Uma das razões de o Oracle utilizar esta premissa talvez seja pelo fato de a vírgula ser utilizada para a separação de valores ou colunas em um comando SQL. Veja o exemplo:

Fazendo um `select` do número 111,1 utilizando vírgula como separador decimal.

```
SQL> select 111,1 from dual;

       111          1
---------- ----------
       111          1

SQL>
```

Agora, executando o mesmo `select`, mas utilizando o ponto como separador decimal.

```
SQL> select 111.1 from dual;

     111.1
----------
     111.1

SQL>
```

No primeiro exemplo, o comando SQL acabou entendendo que 111 era um dado e 1 era outro. O mesmo não aconteceu quando, em vez de separarmos por vírgula, separamos por ponto.

Vejamos o exemplo a seguir. Nele estamos tentando converter um valor onde temos como separador de decimais a vírgula (`,`), e para milhar, o ponto (`.`). Pois bem, já sabemos que para cálculos de conversões internas o Oracle utiliza o ponto como decimal. Logo, o comando SQL a seguir gera um erro.

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4.569.900,87');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number
```

```
conversion error
ORA-06512: at line 4


SQL>
```

Ok! O erro acontece porque onde há vírgula deveria ser ponto e onde há ponto deveria ser vírgula. Pois bem, vejamos o `select` a seguir:

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4,569,900.87') ;
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number
conversion error
ORA-06512: at line 4


SQL>
```

Agora você deve estar se perguntando: "Mas por que o erro, sendo que tudo indica que agora o formato foi informado corretamente, ponto para decimais e vírgula para milhares?". Pois bem, veja o `select` a seguir:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
```

```
     7654 MARTIN     SALESMAN       7698 28-SEP-81        1250
     7698 BLAKE      MANAGER        7839 01-MAY-81        2850
     7782 CLARK      MANAGER        7839 09-JUN-81        2450
     7788 SCOTT      ANALYST        7566 09-DEC-82        3000
     7839 KING       PRESIDENT           17-NOV-81        5000
     7844 TURNER     VENDEDOR       7698 08-SEP-81        1500
     7876 ADAMS      CLERK          7788 12-JAN-83        1100
     7900 JAMES      CLERK          7698 03-DEC-81         950
     7902 FORD       ANALYST        7566 03-DEC-81        3000
     7934 MILLER     CLERK          7782 23-JAN-82        1300
     8000 JOHN       CLERK          7902 30-MAR-11        1000

     COMM    DEPTNO
--------- ---------
                20
      300       30
      500       30
                20
     1400       30
                30
                10
                20
                10
        0       30
                20
                30
                20
                10
      200       20

15 rows selected.

SQL>
```

Observe a coluna SAL, mais precisamente o registro EMPNO=7566. Veja que o valor do salário está aparecendo como 3272.5. Este número possui como separador de decimais o caractere ponto, mas não possui separador de grupo de milhar.

Isso nos leva à conclusão de que, além de sabermos as regras para pontos

e vírgulas, também devemos estar cientes de como está definida a sessão do Oracle. No Oracle, a formatação de milhar não aparece, a menos que seja aplicada para tal. Vamos voltar ao exemplo anterior, agora com a formatação:

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4.569.900,87','9G999G999D00');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 4


SQL>
```

Ainda sim o erro persiste. Também pudera, vimos no registro `EMPNO=7566` que, na sessão do Oracle, está definido como casa decimal o ponto, e não a vírgula. Vamos trocar:

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4,569,900.87','9G999G999D00');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
Valor: 4569900.87

PL/SQL procedure successfully completed.

SQL>
```

Bingo! Este comando select funcionou por duas razões. Primeira: o valor passado no formato caractere possui um número válido que contém como separador de casas decimais o mesmo definido na sessão do Oracle. Segundo: foi definido um formato onde estipulamos que neste conjunto de caracteres há separadores de grupo o qual representamos com o elemento de função G. Utilizando desta forma, o Oracle entende que o ponto é o separador de decimal e a vírgula é o de milhar. Para provar nossa tese, vamos mudar os separadores na sessão e vamos executar o mesmo select anterior:

```
SQL> alter session set nls_numeric_characters='.,';

Session altered.

SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4.569.900,87','9G999G999D00');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 4


SQL>
```

Nota: além das alterações referentes aos separadores numéricos, também podemos alterar as sessões para definir um novo formato de data e também para a definição de linguagens. Exemplos:

```
alter session set nls_language = 'BRAZILIAN PORTUGUESE';
alter session set nls_date_language = 'PORTUGUESE';
alter session set nls_date_format    = 'DD/MM/RRRR';
```

Agora vamos trocar os caracteres de lugar:

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4,569,900.87','9G999G999D00');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
Valor: 4569900.87

PL/SQL procedure successfully completed.

SQL>
```

Funcionou. Mas espere um momento. Além de converter, preciso saber como está definido na sessão do Oracle? Não. Não é necessário se você especificar, além do formato, quais caracteres devem ser utilizados para a separação de decimais e grupos. Vejamos novamente o penúltimo exemplo. Nele, definimos que o ponto seria a decimal, e vírgula o grupo. Contudo, isso gerou um erro ao executar nosso `select` onde estava justamente definido ao contrário da parametrização feita.

```
SQL> alter session set nls_numeric_characters='.,';

Session altered.
```

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4.569.900,87','9G999G999D00');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 4


SQL>
```

Utilizando o parâmetro `nls`, aquele terceiro parâmetro da função que é opcional, nós conseguimos, em vez de alterar a sessão do Oracle, validar estes caracteres apenas no nosso comando. Vamos executar novamente o `select` anterior, mas agora incorporando o terceiro parâmetro:

```
SQL> declare
  2    wvalor number;
  3  begin
  4    wvalor := to_number('4.569.900,87','9G999G999D00',
                          'nls_numeric_characters=,.');
  5    dbms_output.put_line('Valor: '||wvalor);
  6    --
  7  end;
  8  /
Valor: 4569900.87

PL/SQL procedure successfully completed.

SQL>
```

Funcionou perfeitamente. Embora pareça complicado no início, você vai se acostumando com as características e logo pega a lógica. Vale salientar que

esta questão da sessão também é válida para o `to_date`. Veja o `select`, agora observando a coluna `HIREDATE`.

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300
     8000 JOHN       CLERK           7902 30-MAR-11       1000

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10
```

```
      200          20

15 rows selected.

SQL>
```

Para o registro `EMPNO=7566`, o valor da coluna `HIREDATE` é 02-APR-81. Logo, este é o formato definido na sessão. Vamos olhar o próximo `select`.

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date('02-APR-81');
  5    dbms_output.put_line('Data: '||wdate);
  6    --
  7  end;
  8  /
Data: 02-APR-81

PL/SQL procedure successfully completed.

SQL>
```

Nota-se que no `select` anterior não foi gerado erro, mesmo não sendo informado o formato. Isso aconteceu porque o valor de caractere informado como string está no mesmo formato da sessão do Oracle. Veja o que acontece quando colocamos a string em outro formato:

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date('05/21/2009');
  5    dbms_output.put_line('Data: '||wdate);
  6    --
  7  end;
  8  /
declare
*
ERROR at line 1:
```

```
ORA-01843: not a valid month
ORA-06512: at line 4


SQL>
```

Para consertar isso devemos informar um formato compatível com o conjunto de caracteres passado ou trocar o formato da sessão do Oracle. Informando um formato compatível:

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date('05/21/2009','mm/dd/yyyy');
  5    dbms_output.put_line('Data: '||wdate);
  6    --
  7  end;
  8  /
Data: 21-MAY-09

PL/SQL procedure successfully completed.

SQL>
```

Alterando a sessão do Oracle:

```
SQL> declare
  2    wdate date;
  3  begin
  4    wdate := to_date('05/21/2009');
  5    dbms_output.put_line('Data: '||wdate);
  6    --
  7  end;
  8  /
Data: 05/21/2009

PL/SQL procedure successfully completed.

SQL>
```

Veja alguns elementos de formatação que podem ser usados:

- `9`: cada nove representa um caractere que será substituído pelo caractere referente ao valor numérico passado como parâmetro. Os zeros na frente são tratados como espaços em branco.
- `0`: adicionando o como um prefixo ou sufixo ao número, todos os zeros iniciais ou finais são tratados e exibidos como zeros em vez de um espaço em branco.
- `$`: prefixo do símbolo de moeda impresso na primeira posição.
- `S`: exibe um sinal de + inicial ou final quando o valor for positivo e um sinal de - inicial ou final quando o valor for negativo.
- `D`: localização do ponto decimal. Os noves de ambos os lados refletem o número máximo de dígitos permitidos.
- `G`: especifica um separador de grupo (milhar, por exemplo) como uma vírgula.
- `L`: especifica a localização do símbolo de moeda local (tal como `$`).
- `,`: coloca uma vírgula na posição especificada, independentemente do separador de grupo.
- `.`: especifica a localização do ponto decimal, independentemente do separador decimal.
- `FM`: remove os espaços em branco inicial e final.

Há uma confusão muito comum com o uso do `to_date` e do `to_number` no que diz respeito ao resultado mostrado pelo `select` quando são utilizadas estas duas funções. Embora estejamos informando um formato, o Oracle não apresenta o resultado do SQL baseado nele. Isso acontece porque o formato no uso destas funções é apenas para a conversão e não para a visualização. Par visualizarmos o resultado com base no formato que queremos, utilizamos o `to_char`.

14.3 TO_CHAR

Função utilizada para converter tipos de dados numéricos e datas para caracteres. Além da conversão, ele é muito utilizado para formatação visual de dados. Seguem exemplos:

```
SQL> declare
  2    wsal varchar2(50);
  3  begin
  4    for r1 in (select ename
  5                     ,sal
  6                from   emp) loop
  7      --
  8      wsal := to_char(r1.sal,'9G999G999D00',
                        'nls_numeric_characters=''.,''');
  9      --
 10      dbms_output.put_line('Nome: '||r1.ename||'
                            Salário: '||wsal);
 11      --
 12    end loop;
 13  end;
 14  /
Nome: SMITH Salário: 800.00
Nome: ALLEN Salário: 1,600.00
Nome: WARD Salário: 1,250.00
Nome: JONES Salário: 2,975.00
Nome: MARTIN Salário: 1,250.00
Nome: BLAKE Salário: 2,850.00
Nome: CLARK Salário: 2,450.00
Nome: SCOTT Salário: 3,000.00
Nome: KING Salário: 5,000.00
Nome: TURNER Salário: 1,500.00
Nome: ADAMS Salário: 1,100.00
Nome: JAMES Salário: 950.00
Nome: FORD Salário: 3,000.00
Nome: MILLER Salário: 1,300.00

PL/SQL procedure successfully completed.
```

```
SQL>
```

Neste exemplo são selecionados todos os nomes dos empregados e seus respectivos salários. Foi utilizada a função `to_char` para formatar os valores do salário, com o elemento de formatação para casas decimais e milhar. Note que aqui usamos o parâmetro `nls_numeric_characters`, para definir quais caracteres devem ser utilizados para cada separador do elemento.

```
SQL> begin
  2    for r1 in (select count(*) qt_admitidos
  3                     ,to_char(hiredate,'mm') MES
  4                from   emp
  5                group  by to_char(hiredate,'mm')) loop
  6      --
  7      dbms_output.put_line('Admitidos: '||r1.qt_admitidos||'
                               Mês: '||r1.mes);
  8      --
  9    end loop;
 10  end;
 11  /
Admitidos: 1 Mês: 04
Admitidos: 2 Mês: 09
Admitidos: 4 Mês: 12
Admitidos: 1 Mês: 11
Admitidos: 2 Mês: 01
Admitidos: 2 Mês: 02
Admitidos: 1 Mês: 05
Admitidos: 1 Mês: 06

PL/SQL procedure successfully completed.

SQL>
```

Aqui a função `to_char` foi utilizada para formatar a data de admissão do empregado, mostrando apenas o mês referente esta data. O `select` agrupa e apresenta quantos empregados foram admitidos em cada mês.

```
SQL> declare
  2    wdata_extenso varchar2(100);
```

```
  3  begin
  4    wdata_extenso := '22 de agosto de 2009 será o dia '||
  5           to_char(to_date('22/08/2009','dd/mm/yyyy'),'ddd')||
  6           ' do ano';
  7    --
  8    dbms_output.put_line(wdata_extenso);
  9    --
 10  end;
 11  /
22 de agosto de 2009 será o dia 234 do ano

PL/SQL procedure successfully completed.

SQL>
```

Outro exemplo com `to_char`, usado na formatação de datas. Nesse programa, são impressos uma data e o número do dia que ela representa no ano.

```
SQL> declare
  2    wdia_semana varchar2(50);
  3  begin
  4    for r1 in (select ename, hiredate from emp) loop
  5      --
  6      wdia_semana := to_char(r1.hiredate,'day');
  7      --
  8      dbms_output.put_line('Nome: '||r1.ename||' Admissão: '||
  9                 r1.hiredate||' Dia da Semana: '||wdia_semana);
 10      --
 11    end loop;
 12  end;
 13  /
Nome: SMITH Admissão: 12/17/1980 Dia da Semana: wednesday
Nome: ALLEN Admissão: 02/20/1981 Dia da Semana: friday
Nome: WARD Admissão: 02/22/1981 Dia da Semana: sunday
Nome: JONES Admissão: 04/02/1981 Dia da Semana: thursday
Nome: MARTIN Admissão: 09/28/1981 Dia da Semana: monday
Nome: BLAKE Admissão: 05/01/1981 Dia da Semana: friday
Nome: CLARK Admissão: 06/09/1981 Dia da Semana: tuesday
Nome: SCOTT Admissão: 12/09/1982 Dia da Semana: thursday
```

```
Nome: KING Admissão: 11/17/1981 Dia da Semana: tuesday
Nome: TURNER Admissão: 09/08/1981 Dia da Semana: tuesday
Nome: ADAMS Admissão: 01/12/1983 Dia da Semana: wednesday
Nome: JAMES Admissão: 12/03/1981 Dia da Semana: thursday
Nome: FORD Admissão: 12/03/1981 Dia da Semana: thursday
Nome: MILLER Admissão: 01/23/1982 Dia da Semana: saturday

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, são mostrados dados referentes à admissão do empregado: nome do empregado, data de admissão e dia da semana. Este último, formatado através da função `to_char`.

```
SQL> declare
  2    wdata_extenso varchar2(100);
  3  begin
  4    wdata_extenso :=
          'Joinville, '||to_char(sysdate,'dd')||' de '||
  5        initcap(to_char(sysdate, 'fmmonth'))||' de '
  6        ||to_char(sysdate,'yyyy')||'.';
  7    --
  8    dbms_output.put_line(wdata_extenso);
  9    --
 10  end;
 11  /
Joinville, 30 de November de 2011.

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, usamos `to_char` para imprimir em tela a data por extenso.

```
SQL> begin
  2    for r1 in (select ename
  3                     ,hiredate
```

```
  4                from   emp
  5                where  to_char(hiredate,'yyyy') = '1982') loop
  6        --
  7        dbms_output.put_line(
            'Nome: '||r1.ename||' Admissão: '||r1.hiredate);
  8        --
  9      end loop;
 10    end;
 11    /
Nome: SCOTT Admissão: 12/09/1982
Nome: MILLER Admissão: 01/23/1982

PL/SQL procedure successfully completed.

SQL>
```

Aqui utilizamos a função `to_char` para restringir dados em um `select`.

```
SQL> declare
  2     wsal_formatado varchar2(50);
  3  begin
  4     for r1 in (select sal from emp) loop
  5       --
  6       wsal_formatado :=
             'R$ '||to_char(r1.sal,'fm9G999G990D00');
  7       --
  8       dbms_output.put_line('Salário: '||r1.sal||'
                                Salário Formatado: '||wsal_formatado);
  9       --
 10     end loop;
 11  end;
 12  /
Salário: 800 Salário Formatado: R$ 800.00
Salário: 1600 Salário Formatado: R$ 1,600.00
Salário: 1250 Salário Formatado: R$ 1,250.00
Salário: 2975 Salário Formatado: R$ 2,975.00
Salário: 1250 Salário Formatado: R$ 1,250.00
Salário: 2850 Salário Formatado: R$ 2,850.00
```

```
Salário: 2450 Salário Formatado: R$ 2,450.00
Salário: 3000 Salário Formatado: R$ 3,000.00
Salário: 5000 Salário Formatado: R$ 5,000.00
Salário: 1500 Salário Formatado: R$ 1,500.00
Salário: 1100 Salário Formatado: R$ 1,100.00
Salário: 950 Salário Formatado: R$ 950.00
Salário: 3000 Salário Formatado: R$ 3,000.00
Salário: 1300 Salário Formatado: R$ 1,300.00

PL/SQL procedure successfully completed.

SQL>

SQL> declare
  2    wsal_formatado varchar2(50);
  3  begin
  4    for r1 in (select sal from emp) loop
  5      --
  6      wsal_formatado := to_char(r1.sal,'fmL9G999G990D00');
  7      --
  8      dbms_output.put_line('Salário: '||r1.sal||'
           Salário Formatado: '||wsal_formatado);
  9      --
 10    end loop;
 11  end;
 12  /
Salário: 800 Salário Formatado: $800.00
Salário: 1600 Salário Formatado: $1,600.00
Salário: 1250 Salário Formatado: $1,250.00
Salário: 2975 Salário Formatado: $2,975.00
Salário: 1250 Salário Formatado: $1,250.00
Salário: 2850 Salário Formatado: $2,850.00
Salário: 2450 Salário Formatado: $2,450.00
Salário: 3000 Salário Formatado: $3,000.00
Salário: 5000 Salário Formatado: $5,000.00
Salário: 1500 Salário Formatado: $1,500.00
Salário: 1100 Salário Formatado: $1,100.00
Salário: 950 Salário Formatado: $950.00
Salário: 3000 Salário Formatado: $3,000.00
```

```
Salário: 1300 Salário Formatado: $1,300.00

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo se parece com o anterior. Contudo, aqui estamos utilizando os elementos de formatação `FM` e `L`. O primeiro retira os espaços em branco, onde algum elemento de formatação não tenha sido preenchido, por ausência de valores. O segundo mostra a moeda configurada na sessão do usuário, onde o programa está sendo executado.

```
SQL> declare
  2    wpositivo varchar2(100);
  3    wnegativo varchar2(100);
  4  begin
  5    wpositivo := to_char(174984283.75,'fm999,999,999.00S');
  6    wnegativo := to_char(100-1000,'fm999,999,999.00S');
  7    --
  8    dbms_output.put_line('Positivo: '||wpositivo||'  -
                            Negativo: '||wnegativo);
  9    --
 10  end;
 11  /
Positivo: 174,984,283.75+  - Negativo: 900.00-

PL/SQL procedure successfully completed.

SQL>
```

Já nesse exemplo, utilizamos o elemento de formatação `S`, que indica o sinal referente ao valor formatado. Note que no valor positivo foi impresso, atrás do número, o sinal de positivo (+), quanto para o valor negativo é impresso o sinal de negativo (-).

Capítulo 15

Funções condicionais

O Oracle disponibiliza outros tipos de funções, dentre as quais as funções condicionais. Elas são utilizadas tanto na seleção de dados pela cláusula `select` como também no uso de cláusulas `where`. Seu uso é bastante difundido e bem flexível.

- `decode`: esta estrutura funciona como uma estrutura `if-else` dentro de uma cláusula `select`. É muito utilizada principalmente para visualização de dados onde é preciso realizar algum teste para saber se estes dados podem ou não aparecer.

- `nullif`: são passados dois parâmetros para esta função. Ela compara os dois, se forem iguais é retornado `Null`. Caso contrário, ela retorna o primeiro parâmetro.

- `nvl`: para esta função são passados dois parâmetros. Se o primeiro for nulo, ele retorna o segundo, caso contrário, retorna o primeiro.
- `case`: muito parecido com o `decode`. Seu objetivo também é permitir a utilização de uma estrutura tipo `if-else` dentro do comando SQL. Ao contrário do `decode` sua aplicação e visualização são mais inteligíveis (padrão ANSI).
- `greatest`: retorna a maior expressão de uma lista de valores passada como parâmetro. Todas as expressões após a primeira são convertidas para o tipo de dado da primeira antes da comparação ser realizada.
- `least`: funciona o inverso da `greatest`. Esta traz a menor expressão.

Veja alguns exemplos:

```
SQL> declare
  2    cursor c1 is
  3      select job,
  4             sum(case
  5                  when deptno = 10 then sal
  6                 else
  7                   0
  8                 end) depart_10,
  9             sum(case
 10                   when deptno = 20 then sal
 11                 else
 12                   0
 13                 end) depart_20,
 14             sum(case
 15                   when deptno = 30 then sal
 16                 else
 17                   0
 18                 end) depart_30,
 19             sum(sal) total_job
 20         from emp
 21         group by job;
 22    --
 23  begin
```

```
 24     for r1 in c1 loop
 25       --
 26       dbms_output.put_line(r1.job||' - Depart. 10: '
                                ||r1.depart_10||' - Depart. 20: '||
 27                             r1.depart_20||' - Depart. 30: '||
                                r1.depart_30||' - Total: '||
 28                             r1.total_job);
 29       --
 30     end loop;
 31   end;
 32   /
CLERK - Depart. 10: 1300 - Depart. 20: 1900 - Depart. 30: 950 -
Total: 4150
SALESMAN - Depart. 10: 0 - Depart. 20: 0 - Depart. 30: 5600 -
Total: 5600
PRESIDENT - Depart. 10: 5000 - Depart. 20: 0 - Depart. 30: 0 -
Total: 5000
MANAGER - Depart. 10: 2450 - Depart. 20: 2975 - Depart. 30:
2850 - Total: 8275
ANALYST - Depart. 10: 0 - Depart. 20: 6000 - Depart. 30: 0 -
Total: 6000

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, utilizamos a função `case`, para somar todos os salários por departamento, agrupando por cargo.

```
SQL> declare
  2     cursor c1 is
  3       select job
  4             ,sum(decode(deptno, 10, sal, 0)) depart_10
  5             ,sum(decode(deptno, 20, sal, 0)) depart_20
  6             ,sum(decode(deptno, 30, sal, 0)) depart_30
  7             ,sum(sal) total_job
  8       from   emp
  9       group  by job;
 10     --
```

```
11  begin
12    for r1 in c1 loop
13      --
14      dbms_output.put_line(r1.job||' - Depart. 10: '||
                           r1.depart_10||' - Depart. 20: '||
15                         r1.depart_20||' - Depart. 30: '||
                           r1.depart_30||' - Total: '||
16                         r1.total_job);
17      --
18    end loop;
19  end;
20  /
CLERK - Depart. 10: 1300 - Depart. 20: 1900 - Depart. 30: 950 -
Total: 4150
SALESMAN - Depart. 10: 0 - Depart. 20: 0 - Depart. 30: 5600 -
Total: 5600
PRESIDENT - Depart. 10: 5000 - Depart. 20: 0 - Depart. 30: 0 -
Total: 5000
MANAGER - Depart. 10: 2450 - Depart. 20: 2975 - Depart. 30: 2850
- Total: 8275
ANALYST - Depart. 10: 0 - Depart. 20: 6000 - Depart. 30: 0 -
Total: 6000

PL/SQL procedure successfully completed.

SQL>
```

Esse exemplo é idêntico ao anterior. Apenas trocamos a função `case` pela função `decode`. Vale lembrar que `case` é do padrão SQL ANSI e `decode` é do padrão Oracle.

```
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE          SAL
---------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK           7902 12/17/1980        800
      7499 ALLEN      SALESMAN        7698 02/20/1981       1600
      7521 WARD       SALESMAN        7698 02/22/1981       1250
      7566 JONES      MANAGER         7839 04/02/1981       2975
```

```
      7654 MARTIN     SALESMAN        7698 09/28/1981       1250
      7698 BLAKE      MANAGER         7839 05/01/1981       2850
      7782 CLARK      MANAGER         7839 06/09/1981       2450
      7788 SCOTT      ANALYST         7566 12/09/1982       3000
      7839 KING       PRESIDENT            11/17/1981       5000
      7844 TURNER     SALESMAN        7698 09/08/1981       1500
      7876 ADAMS      CLERK           7788 01/12/1983       1100
      7900 JAMES      CLERK           7698 12/03/1981        950
      7902 FORD       ANALYST         7566 12/03/1981       3000
      7934 MILLER     CLERK           7782 01/23/1982       1300

      COMM     DEPTNO
---------- ----------
                   20
       300         30
       500         30
                   20
      1400         30
                   30
                   10
                   20
                   10
         0         30
                   20
                   30
                   20
                   10

14 rows selected.

SQL> declare
  2    wsal_comm1 number;
  3    wsal_comm2 number;
  4  begin
  5    --
  6    select sum(sal+comm) into wsal_comm1 from emp;
  7    --
  8    select sum(sal+nvl(comm,0)) into wsal_comm2 from emp;
  9    --
```

```
 10    dbms_output.put_line('Sal. Comm1: '||wsal_comm1||'  -
                             Sal. Comm2: '||wsal_comm2);
 11    --
 12  end;
 13  /
Sal. Comm1: 7800  - Sal. Comm2: 31225

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo estamos utilizando a função `nvl`. Note que quando realizamos cálculos com valores nulos (linha 6) o Oracle não executa tal operação. O Oracle sempre considera como falso quando existem valores `NULL` (nulo) em cálculos aritméticos ou no uso de restrição de dados, como, por exemplo, em cláusulas `where`. Quando isso ocorre, ele ignora a ação, mas não gera erros.

```
SQL> declare
  2    wmaior_letra varchar2(1);
  3    wmenor_letra varchar2(1);
  4  begin
  5    --
  6    select greatest('b','x','t','u','a') into wmaior_letra
          from dual;
  7    --
  8    select least('b','x','t','u','a') into wmenor_letra
          from dual;
  9    --
 10    dbms_output.put_line('Maior Letra: '||wmaior_letra||'  -
 Menor Letra: '||wmenor_letra);
 11    --
 12  end;
 13  /
Maior Letra: x  - Menor Letra: a

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, temos o uso das funções `greatest` e `least`.

```
SQL> declare
  2    comparacao1 varchar2(100);
  3    comparacao2 varchar2(100);
  4  begin
  5    --
  6    select decode(nullif('abacaxi','abacaxi'),null,
                             'são iguais','são diferentes')
  7    into   comparacao1 from dual;
  8    --
  9    select decode(nullif('abacaxi','morango'),null,
                             'são iguais','são diferentes')
 10    into   comparacao2 from dual;
 11    --
 12    dbms_output.put_line('Comparação 1: '||comparacao1||'  -
 Comparação 2: '||comparacao2);
 13    --
 14  end;
 15  /
Comparação 1: são iguais  - Comparação 2: são diferentes

PL/SQL procedure successfully completed.

SQL>
```

Nesse exemplo, temos o uso da função `nullif`.

15.1 DECODE VS. CASE

Dos comandos condicionais vistos anteriormente, o `decode` é o mais utilizado. Ele funciona como uma espécie de condição SE (`if`) para a linguagem SQL. Este comando é exclusivo do Oracle, entretanto, no padrão ANSI da linguagem SQL existe um comando similar, o `case`. Você pode usar qualquer um deles. Porém, o uso do `case` só é permitido nas versões mais novas do banco de dados Oracle. A Oracle, nas versões mais recentes, vem inserindo comandos padrões ANSI enquanto seus comandos específicos mantêm suas

características para questões de compatibilidade. Veja comparações entre estes dois comandos.

Exemplo 1

Padrão ANSI (também suportado pelo Oracle nas versões mais novas do banco de dados):

```
SQL> declare
  2    cursor c1 is
  3      select job,
  4             sum(case
  5                   when deptno = 10 then sal
  6                 else
  7                   0
  8                 end) depart_10,
  9             sum(case
 10                   when deptno = 20 then sal
 11                 else
 12                   0
 13                 end) depart_20,
 14             sum(case
 15                   when deptno = 30 then sal
 16                 else
 17                   0
 18                 end) depart_30,
 19             sum(sal) total_job
 20            from emp
 21            group by job;
 22    --
 23  begin
 24    for r1 in c1 loop
 25      --
 26      dbms_output.put_line(r1.job||' - Depart. 10: '||
                             r1.depart_10||' - Depart. 20: '||
 27                          r1.depart_20||' - Depart. 30: '||
                             r1.depart_30||' - Total: '||
 28                          r1.total_job);
 29      --
```

```
 30     end loop;
 31   end;
 32   /
CLERK - Depart. 10: 1300 - Depart. 20: 1900 - Depart. 30: 950 -
Total: 4150
SALESMAN - Depart. 10: 0 - Depart. 20: 0 - Depart. 30: 5600 -
Total: 5600
PRESIDENT - Depart. 10: 5000 - Depart. 20: 0 - Depart. 30: 0 -
Total: 5000
MANAGER - Depart. 10: 2450 - Depart. 20: 2975 - Depart. 30: 2850
- Total: 8275
ANALYST - Depart. 10: 0 - Depart. 20: 6000 - Depart. 30: 0 -
Total: 6000

PL/SQL procedure successfully completed.

SQL>
```

Padrão Oracle:

```
SQL> declare
  2     cursor c1 is
  3       select job,
  4              sum(decode(deptno, 10, sal, 0)) depart_10,
  5              sum(decode(deptno, 20, sal, 0)) depart_20,
  6              sum(decode(deptno, 30, sal, 0)) depart_30,
  7              sum(sal) total_job
  8              from emp
  9              group by job;
 10     --
 11   begin
 12     for r1 in c1 loop
 13       --
 14       dbms_output.put_line(r1.job||' - Depart. 10: '||
                              r1.depart_10||' - Depart. 20: '||
 15                           r1.depart_20||' - Depart. 30: '||
                              r1.depart_30||' - Total: '||
 16                           r1.total_job);
 17       --
 18     end loop;
```

```
 19   end;
 20   /
CLERK - Depart. 10: 1300 - Depart. 20: 1900 - Depart. 30: 950 -
Total: 4150
SALESMAN - Depart. 10: 0 - Depart. 20: 0 - Depart. 30: 5600 -
Total: 5600
PRESIDENT - Depart. 10: 5000 - Depart. 20: 0 - Depart. 30: 0 -
Total: 5000
MANAGER - Depart. 10: 2450 - Depart. 20: 2975 - Depart. 30: 2850
- Total: 8275
ANALYST - Depart. 10: 0 - Depart. 20: 6000 - Depart. 30: 0 -
Total: 6000

PL/SQL procedure successfully completed.

SQL>
```

Exemplo 2

Padrão ANSI (também suportado pelo Oracle nas versões mais novas do banco de dados):

```
SQL> declare
  2     cursor c1 is
  3       select ename
  4             , job
  5             , mgr
  6             ,
  7       case
  8         when mgr = 7902 then 'MENSALISTA'
  9         when mgr = 7902 then 'MENSALISTA'
 10         when mgr = 7839 then 'COMISSIONADO'
 11         when mgr = 7566 then 'MENSAL/HORISTA'
 12       else
 13         'OUTROS'
 14       end tipo
 15       from emp;
 16
 17     --
```

```
 18  begin
 19    for r1 in c1 loop
 20      --
 21      dbms_output.put_line('Nome: '||r1.ename||' - Cargo:
                           '||r1.job||' - Gerente: '||
 22                        r1.mgr||' - Tipo: '||r1.tipo);
 23      --
 24    end loop;
 25  end;
 26  /
Nome: SMITH - Cargo: CLERK - Gerente: 7902 - Tipo: MENSALISTA
Nome: ALLEN - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: WARD - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: JONES - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: MARTIN - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: BLAKE - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: CLARK - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: SCOTT - Cargo: ANALYST - Gerente: 7566 - Tipo:
MENSAL/HORISTA
Nome: KING - Cargo: PRESIDENT - Gerente:  - Tipo: OUTROS
Nome: TURNER - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: ADAMS - Cargo: CLERK - Gerente: 7788 - Tipo: OUTROS
Nome: JAMES - Cargo: CLERK - Gerente: 7698 - Tipo: OUTROS
Nome: FORD - Cargo: ANALYST - Gerente: 7566 - Tipo:
MENSAL/HORISTA
Nome: MILLER - Cargo: CLERK - Gerente: 7782 - Tipo: OUTROS

PL/SQL procedure successfully completed.

SQL>
```

Padrão Oracle:

```
SQL> declare
  2    cursor c1 is
  3      select ename
  4            ,job
  5            ,mgr
  6            ,decode(mgr,7902,'MENSALISTA'
  7                       ,7839,'COMISSIONADO'
```

```
 8                         ,7566,'MENSAL/HORISTA'
 9                         ,'OUTROS') tipo
10      from emp;
11    --
12  begin
13    for r1 in c1 loop
14      --
15      dbms_output.put_line('Nome: '||r1.ename||' -
                             Cargo: '||r1.job||' -
                             Gerente: '||
16                           r1.mgr||' - Tipo: '||r1.tipo);
17      --
18    end loop;
19  end;
20  /

Nome: SMITH - Cargo: CLERK - Gerente: 7902 - Tipo: MENSALISTA
Nome: ALLEN - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: WARD - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: JONES - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: MARTIN - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: BLAKE - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: CLARK - Cargo: MANAGER - Gerente: 7839 - Tipo: COMISSIONADO
Nome: SCOTT - Cargo: ANALYST - Gerente: 7566 - Tipo:
MENSAL/HORISTA
Nome: KING - Cargo: PRESIDENT - Gerente:  - Tipo: OUTROS
Nome: TURNER - Cargo: SALESMAN - Gerente: 7698 - Tipo: OUTROS
Nome: ADAMS - Cargo: CLERK - Gerente: 7788 - Tipo: OUTROS
Nome: JAMES - Cargo: CLERK - Gerente: 7698 - Tipo: OUTROS
Nome: FORD - Cargo: ANALYST - Gerente: 7566 - Tipo:
MENSAL/HORISTA
Nome: MILLER - Cargo: CLERK - Gerente: 7782 - Tipo: OUTROS

PL/SQL procedure successfully completed.

SQL>
```

A opção entre usar um ou outro vai depender da abrangência dos seus programas. Se eles forem específicos para o uso em Oracle, o `decode` pode

ser usado sem problemas. Inclusive, como pôde ser visto nos exemplos, ele pode se tornar visualmente mais claro para o entendimento do código. Já, se suas aplicações forem abrangentes no que diz respeito a operar em vários bancos de dados, você terá que usar o padrão ANSI, ou seja, usar o `case` para que eles funcionem em qualquer banco de dados, ou pelo menos naqueles que seguem este padrão.

Capítulo 16

Programas armazenados

Até aqui, construímos programas em blocos chamados blocos anônimos. Caso quiséssemos guardá-los, teríamos que salvá-los em um ou mais arquivos em algum diretório no computador para não perdê-los. Quando quiséssemos executá-los novamente, teríamos que resgatá-los do computador para então executá-los em uma ferramenta, por exemplo, no SQL*Plus. Pois bem, agora vamos aprender como gravar estes programas no banco de dados. isso nos traz muitos benefícios e abre muitas possibilidades. A seguir vemos algumas delas:

- **Reaproveitamento de códigos**: podemos escrever procedimentos e funções que podem servir como base para as demais partes de um sistema. Por exemplo, podemos criar uma função para validação de números de CPF que possa ser utilizada em vários módulos do sistema,

como no módulo de RH ou de compras, ou seja, podemos ter um programa gravado no banco de dados ao qual todos podem ter acesso.

- **Rapidez**: tendo o programa armazenado no banco de dados, podemos acessá-lo rapidamente, sem ter que utilizar chamadas externas a arquivos. outro ponto interessante é que as ferramentas de desenvolvimento e análise do Oracle enxergam de forma nativa estes objetos armazenados.
- **Controle de alterações**: um programa armazenado no banco de dados é muito mais fácil de ser alterado. É possível abri-lo e alterá-lo de forma mais rápida, compilando e em seguida efetivando as alterações. Como o programa está em um único lugar, os demais sistemas que o utilizam enxergarão todas as alterações realizadas.
- **Controle de acesso**: através de concessões é possível limitar os acessos a estes programas, permitindo-os a apenas alguns usuários.
- **Modularização**: veremos mais adiante que, pelo fato de estarem armazenados em um banco de dados, os programas podem ser agrupados dentro de pacotes, o que permite que nós os organizemos e estruturemos de acordo com seus escopos, viabilizando a modularização do sistema.

Os programas armazenados podem ser de três tipos: procedimentos (`procedures`), funções (`functions`) e pacotes (`packages`). A escolha de um ou de outro vai depender da necessidade ou da característica do programa. Trataremos neste capítulo de `procedures` e `functions`. Falaremos sobre `packages` no capítulo a seguir.

16.1 PROCEDURES E FUNCTIONS

Para que um programa seja armazenado em um banco de dados, ele precisa receber um nome único, chamado de identificador, que será sua identificação dentro do banco de dados. É por ele que vamos manipulá-lo, localizá-lo no banco de dados, ou executá-lo. A nomenclatura utilizada para nomear este

identificador segue as mesmas regras dos identificadores de variáveis e constantes vistos em capítulos anteriores: ele pode ter um tamanho máximo de 30 caracteres, que se restringem a alguns caracteres especiais, na sua maioria, caracteres alfanuméricos; e deve iniciar por uma letra.

Depois de criados, os programas armazenados podem ser executados através de ferramentas tais como SQL*Plus, Oracle Forms, Oracle Reports etc., bem como podem ser chamados por outros programas armazenados ou por blocos PL/SQL anônimos.

Também temos a opção de criar `procedures` e `functions` dentro de blocos PL/SQL anônimos ou dentro de outras `procedures`, `functions`, `triggers` ou `packages`. Contudo, suas definições não ficam armazenadas dentro do dicionário de objetos do Oracle, apenas dentro do objeto onde estão sendo criadas.

Conforme dito anteriormente, a escolha do tipo no momento de criar um programa armazenado depende da necessidade, ou melhor, do escopo em que se encaixa. Por exemplo, quando queremos que um programa retorne uma informação, podemos criá-lo como `function`. Este tipo de programa armazenado, além de executar alguma ação, pode ainda retornar um tipo de informação para o programa chamador ou para a ferramenta que o executou. Um exemplo seria uma função que retornasse se o número de CPF é válido ou não.

Caso não seja necessário retornar uma informação, podemos criar o programa como sendo uma `procedure`. Neste caso, ela executa suas ações como um programa qualquer, mas não retorna nada (veremos mais adiante que é possível, sim, ter retorno através de procedures). O que deve ficar claro é que funções sempre devem retornar um valor, já procedimentos não possuem esta obrigatoriedade. Veja a seguir um exemplo de `procedure`.

```
SQL> create procedure calc is
  2    --
  3    x1   number      := 10;
  4    x2   number      := 5;
  5    op   varchar2(1) := '+';
  6    res number;
  7    --
  8  begin
```

```
 9     if (x1 + x2) = 0 then
10       dbms_output.put_line('Resultado: 0');
11     elsif op = '*' then
12       res := x1 * x2;
13     elsif op = '/' then
14       if x2 = 0 then
15         dbms_output.put_line('Erro de divisão por zero!');
16       else
17         res := x1 / x2;
18       end if;
19     elsif op = '-' then
20       res := x1 - x2;
21       if res = 0 then
22         dbms_output.put_line('Resultado igual a zero!');
23       elsif res < 0 then
24         dbms_output.put_line('Resultado menor que zero!');
25       elsif res > 0 then
26         dbms_output.put_line('Resultado maiorl a zero!');
27       end if;
28     elsif op = '+' then
29       res := x1 + x2;
30     else
31       dbms_output.put_line('Operador inválido!');
32     end if;
33     dbms_output.put_line('Resultado do cálculo: '||res);
34   end;
35   /

Procedimento criado.

SQL>
```

O procedimento anterior tem o objetivo de realizar cálculos numéricos. Note que basicamente, o que muda de uma `procedure` para um bloco anônimo é que nela temos um cabeçalho onde informamos um identificador, neste caso, `calc`, que precede o comando `create procedure` e antecede o comando `is` que indica o início do procedimento. Após isso, temos basicamente um bloco PL/SQL comum ao que já vimos até aqui. O mesmo acontece

com a criação de funções.

Agora vejamos um exemplo de `function`.

```
SQL> create function valida_cpf return varchar2 is
  2    --
  3    m_total  number      default 0;
  4    m_digito number      default 0;
  5    cpf      varchar2(50) default '02411848430';
  6    --
  7  begin
  8    for i in 1..9 loop
  9      m_total := m_total + substr (cpf, i, 1) * (11 - i);
 10    end loop;
 11    --
 12    m_digito := 11 - mod (m_total, 11);
 13    --
 14    if m_digito > 9 then
 15      m_digito := 0;
 16    end if;
 17    --
 18    if m_digito != substr (cpf, 10, 1) then
 19      return 'I';
 20    end if;
 21    --
 22    m_digito := 0;
 23    m_total := 0;
 24    --
 25    for i in 1..10 loop
 26      m_total := m_total + substr (cpf, i, 1) * (12 - i);
 27    end loop;
 28    --
 29    m_digito := 11 - mod (m_total, 11);
 30    --
 31    if m_digito > 9 then
 32      m_digito := 0;
 33    end if;
 34    --
 35    if m_digito != substr (cpf, 11, 1) then
 36      return 'I';
```

```
 37     end if;
 38     --
 39     return 'V';
 40     --
 41   end valida_cpf;
 42   /

Função criada.

SQL>
```

Essa função tem o objetivo de validar um número de CPF e retornar falso ou verdadeiro, dependendo da validação. Igualmente às procedures, têm-se um cabeçalho onde definimos um identificador para a função, neste exemplo, `valida_cpf`, que precede o comando `create function` e antecede os comandos `return`, seguido pelo tipo de dado a ser retornado, e `is` indicando o início da função. Logo após, temos a codificação da função dentro de um bloco PL/SQL.

Como comentamos anteriormente, `procedures` e `functions` também podem ser criadas dentro de blocos PL/SQL anônimos ou dentro de objetos. A seguir, utilizamos os mesmos objetos dos exemplos anteriores para mostrar como isso é feito.

`Procedure` criada no bloco:

```
SQL> declare
  2     --
  3     procedure calc is
  4       --
  5       x1   number       := 10;
  6       x2   number       := 5;
  7       op   varchar2(1)  := '+';
  8       res number;
  9       --
 10     begin
 11       if (x1 + x2) = 0 then
 12         dbms_output.put_line('Resultado: 0');
 13       elsif op = '*' then
 14         res := x1 * x2;
```

```
15        elsif op = '/' then
16          if x2 = 0 then
17            dbms_output.put_line('Erro de divisão por zero!');
18          else
19            res := x1 / x2;
20          end if;
21        elsif op = '-' then
22          res := x1 - x2;
23          if res = 0 then
24            dbms_output.put_line('Resultado igual a zero!');
25          elsif res < 0 then
26            dbms_output.put_line('Resultado menor que zero!');
27          elsif res > 0 then
28            dbms_output.put_line('Resultado maiorl a zero!');
29          end if;
30        elsif op = '+' then
31          res := x1 + x2;
32        else
33          dbms_output.put_line('Operador inválido!');
34        end if;
35        dbms_output.put_line('Resultado do cálculo: '||res);
36      end;
37      --
38    begin
39      calc;
40    end;
41    /
Resultado do cálculo: 15

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Nesse exemplo, é possível verificar que a `procedure` foi criada no nível do bloco, ou seja, ele existe apenas dentro do bloco e não está armazenada no banco de dados. Logo, caso seja de nossa vontade utilizá-la novamente, teremos que salvar todo o código do bloco em um arquivo externo. Desta forma, também não é possível que outros programas usem-na. Note também que nestes casos não usamos o comando `create`. Começamos direto pelo

tipo do programa, a `procedure`.

`Function` criada em uma `procedure`.

```
SQL> declare
  2    --
  3    res varchar2(1) default null;
  4    --
  5    function valida_cpf return varchar2 is
  6      --
  7      m_total  number      default 0;
  8      m_digito number      default 0;
  9      cpf      varchar2(50) default '02411848430';
 10      --
 11    begin
 12      for i in 1..9 loop
 13        m_total := m_total + substr (cpf, i, 1) * (11 - i);
 14      end loop;
 15      --
 16      m_digito := 11 - mod (m_total, 11);
 17      --
 18      if m_digito > 9 then
 19        m_digito := 0;
 20      end if;
 21      --
 22      if m_digito != substr (cpf, 10, 1) then
 23        return 'I';
 24      end if;
 25      --
 26      m_digito := 0;
 27      m_total := 0;
 28      --
 29      for i in 1..10 loop
 30        m_total := m_total + substr (cpf, i, 1) * (12 - i);
 31      end loop;
 32      --
 33      m_digito := 11 - mod (m_total, 11);
 34      --
 35      if m_digito > 9 then
 36        m_digito := 0;
```

```
37        end if;
38        --
39        if m_digito != substr (cpf, 11, 1) then
40          return 'I';
41        end if;
42        --
43        return 'V';
44        --
45      end valida_cpf;
46      --
47    begin
48      res := valida_cpf;
49      --
50      if res = 'V' then
51        dbms_output.put_line('CPF válido');
52      else
53        dbms_output.put_line('CPF inválido');
54      end if;
55      --
56    end;
57    /
CPF inválido

Procedimento PL/SQL concluído com sucesso.

SQL>
```

O mesmo acontece aqui com o exemplo desta função. Como ela está criada no nível do bloco, ela não se encontra armazenada no banco de dados. Note também que nestes casos não usamos o comando `create`. Começamos direto pelo tipo do programa, a `function`.

Uma `procedure` pode ser chamada de dentro de um bloco PL/SQL, de dentro de programas em Oracle Forms, Oracle Reports etc. Entretanto, elas não podem ser chamadas através de um comando SQL. Já as `functions` podem ser chamadas também de dentro de comandos SQL, pelo fato de elas retornarem um valor. Contudo, há algumas restrições, por exemplo, ela não pode ter em sua composição comandos DML, DDL e DCL, apenas `selects`. outra característica das `functions` é a recursividade, com a qual podemos

criar uma função que chama ela mesma.

Os procedimentos podem ser chamados de duas formas. Se estivermos trabalhando com SQL*Plus podemos chamá-la através do comando `execute`, e através de um bloco PL/SQL anônimo ou programa armazenado. Veja os dois exemplos de chamada a seguir.

```
SQL> execute calc;
Resultado do cálculo: 15

Procedimento PL/SQL concluído com sucesso.

SQL>

SQL> begin
  2    calc;
  3  end;
  4  /
Resultado do cálculo: 15

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Assim como nos procedimentos, as funções também podem ser chamadas de duas formas. Uma através de blocos PL/SQL anônimos ou programas armazenados, e através de comandos SQL (`select`, `insert`, `delete` ou `update`). Vale ressaltar que para chamadas via comando SQL existem algumas restrições. Veja os dois exemplos de chamada a seguir.

```
SQL> declare
  2    --
  3    res varchar2(1) default null;
  4    --
  5  begin
  6    res := valida_cpf;
  7    --
  8    if res = 'V' then
  9      dbms_output.put_line('CPF válido');
 10    else
```

```
 11       dbms_output.put_line('CPF inválido');
 12     end if;
 13     --
 14   end;
 15   /
CPF inválido

Procedimento PL/SQL concluído com sucesso.

SQL>

SQL> select decode(valida_cpf,'V','Válido','inválido') CPF
from dual;

CPF
---------
inválido

SQL>
```

Concedendo acesso a procedures e functions

Uma vantagem no uso de `procedures` e `functions` está relacionada à restrição de acesso. Objetos criados no banco de dados precisam de permissão para acesso, caso quem os queira acessar não seja o dono ou DBA do sistema. Para dar acesso a `procedures` e `functions` utilizamos o comando `grant execute`.

```
SQL> grant execute on calc to public;

Concessão bem-sucedida.

SQL> grant execute on valida_cpf to TSQL2;

Concessão bem-sucedida.

SQL>
```

Também é possível criarmos sinônimos para facilitar o acesso. Os sinônimos podem ser específicos a um usuário ou público. Vale ressaltar que eles

não dão acesso, apenas permitem criar um *alias* para o objeto.

```
SQL> create public synonym calc_valores for tsql.calc;

Sinônimo criado.

SQL> create synonym tsql2.valida_cpf for tsql.valida_cpf;

Sinônimo criado.

SQL>
```

Note que podemos dar `grants` e criar sinônimos para o usuário `public`. Quando fazemos desta forma, todos os usuários do banco de dados terão acesso ao referido objeto.

16.2 Uso do comando replace

Quando precisamos dar manutenção em `procedures` e `functions`, podemos utilizar a cláusula `replace` para garantir que algumas definições sejam preservadas. Como esses objetos possuem um identificador único, não é possível criar outro com o mesmo nome. Logo, para recriarmos um objeto como este nós podemos utilizar a cláusula `replace`. Caso contrário, seria necessário excluir o objeto existente e criá-lo novamente. Com isto, informações referentes a permissões de acesso seriam excluídas também. Já com o comando `replace`, este tipo de informação permanece, assim como a marcação de objetos dependentes para recompilação.

Com relação à dependência de objetos, da qual falaremos mais à frente, só para um entendimento prévio, é comum termos programas que chamam outros programas, acabando por criar dependências entre eles, o que pode invalidar todo um conjunto de objetos caso um seja excluído ou invalidado. Em alguns casos, o Oracle poderá recompilar objetos, automaticamente (dependendo do nível de dependência), quando são recriados, e não excluídos e criados logo em seguida.

outra vantagem do comando `replace` é a possibilidade de criar objetos mesmo com erros de sintaxe em seus códigos. Nesse caso, eles perma-

necem inválidos enquanto tais erros existirem. A seguir o uso do comando `replace`.

```
SQL> create or replace procedure calc is
  2    --
  3    x1   number      := 10;
  4    x2   number      := 5;
  5    op   varchar2(1) := '+';
  6    res number;
  7    --
  8  begin
  9    if (x1 + x2) = 0 then
 10      dbms_output.put_line('Resultado: 0');
 11    elsif op = '*' then
 12      res := x1 * x2;
 13    elsif op = '/' then
 14      if x2 = 0 then
 15        dbms_output.put_line('Erro de divisão por zero!');
 16      else
 17        res := x1 / x2;
 18      end if;
 19    elsif op = '-' then
 20      res := x1 - x2;
 21      if res = 0 then
 22        dbms_output.put_line('Resultado igual a zero!');
 23      elsif res < 0 then
 24        dbms_output.put_line('Resultado menor que zero!');
 25      elsif res > 0 then
 26        dbms_output.put_line('Resultado maiorl a zero!');
 27      end if;
 28    elsif op = '+' then
 29      res := x1 + x2;
 30    else
 31      dbms_output.put_line('Operador inválido!');
 32    end if;
 33    dbms_output.put_line('Resultado do cálculo: '||res);
 34  end;
 35  /
```

```
Procedimento criado.

SQL>
```

16.3 Recompilando programas armazenados

Quando alteramos uma `procedure` ou `function` no banco de dados, pode acontecer de termos que compilá-la novamente. Para isso, utilizamos o comando `alter`. Veja os exemplos:

```
SQL> alter procedure calc compile;

Procedimento alterado.

SQL> alter function valida_cpf compile;

Função alterada.

SQL>
```

16.4 Recuperando informações

Para visualizar informações referentes a `procedures` e `functions` utilize as *views* `user_objects`, `all_objects` ou `dba_objects`. Nesta view constam informações como `STATUS` e data de criação do objeto.

```
SQL> select owner, object_type, status, created, last_ddl_time
  2  from   all_objects where object_name = 'CALC';

OWNER                          OBJECT_TYPE         STATUS
------------------------------ ------------------- -------
TSQL                           procedure           VALID

createD  LAST_DDL
-------- --------
25/11/11 25/11/11

SQL>
```

Outra forma de recuperar dados referentes a procedures e functions é utilizando o comando describe. Este comando mostra informações referentes a estes objetos, como dados do cabeçalho e parâmetros existentes.

```
SQL> desc calc
procedure calc
 Nome do Argumento                   Tipo
 ------------------------------ -----------------------
 X1                             NUMBER
 X2                             NUMBER
 OP                             VARCHAR2
 RES                            VARCHAR2

      in/out Default?
 ------ --------
 in
 in
 in
 out


SQL>
```

16.5 Recuperando códigos

Já para visualizar o código dos objetos armazenados no banco de dados, utilize as views user_source, all_source ou dba_source.

```
SQL> column text format a100
SQL> set pages 1000
SQL> select line, text
  2  from   all_source where name = 'CALC';

   LinE TEXT
-------- -----------------------------------------------------
       1 procedure calc is
       2   --
       3   x1    number      := 10;
       4   x2    number      := 5;
```

```
 5   op   varchar2(1) := '+';
 6   res number;
 7   --
 8 begin
 9   if (x1 + x2) = 0 then
10     dbms_output.put_line('Resultado: 0');
11   elsif op = '*' then
12     res := x1 * x2;
13   elsif op = '/' then
14     if x2 = 0 then
15       dbms_output.put_line('Erro de divisão por zero!');
16     else
17       res := x1 / x2;
18     end if;
19   elsif op = '-' then
20     res := x1 - x2;
21     if res = 0 then
22       dbms_output.put_line('Resultado igual a zero!');
23     elsif res < 0 then
24       dbms_output.put_line('Resultado menor que zero!');
25     elsif res > 0 then
26       dbms_output.put_line('Resultado maiorl a zero!');
27     end if;
28   elsif op = '+' then
29     res := x1 + x2;
30   else
31     dbms_output.put_line('Operador inválido!');
32   end if;
33   dbms_output.put_line('Resultado do cálculo: '||res);
34 end;

34 linhas selecionadas.

SQL>
```

16.6 Visualizando erros de compilação

Para visualizar erros de compilação, use o comando `show error`.

```
SQL> create or replace procedure calc is
  2    --
  3    x1   number      := 10;
  4    x2   number      := 5;
  5    op   varchar2(1) := '+';
  6    res number;
  7    --
  8  begin
  9    if (x1 + x2) = 0 then
 10      dbms_output.put_line('Resultado: 0');
 11    elsif op = '*' then
 12      res := x1 * x2;
 13    elsif op = '/' then
 14      if x2 = 0 then
 15        dbms_output.put_line('Erro de divisão por zero!');
 16      else
 17        res := x1 / x2;
 18      end if;
 19    elsif op = '-' then
 20      res := x1 - x2;
 21      if res = 0 then
 22        dbms_output.put_line('Resultado igual a zero!');
 23      elsif res < 0 then
 24        dbms_output.put_line('Resultado menor que zero!');
 25      elsif res > 0 then
 26        dbms_output.put_line('Resultado maiorl a zero!');
 27      end if;
 28    elsif op = '+' then
 29      res := x1 + x2;
 30    else
 31      dbms_output.put_line('Operador inválido!');
 32    end if;
 33    dbms_output.put_line('Resultado do cálculo: '||res);
 34    ; -- provocando um erro de sintaxe.
 35  end;
 36  /

Advertência: Procedimento criado com erros de compilação.
```

```
SQL> show error
Erros para procedure CALC:

LinE/COL ERROR
-------- -------------------------------------------------------
34/3     PLS-00103: Encontrado o símbolo ";" quando um dos
         seguintes símbolos era esperado:
         begin case declare end exception exit for goto if loop
         mod null pragma raise return select update while with
         <um identificador>
         <um identificador delimitado por aspas duplas>
         <uma variável de ligação> << close current delete fetch
         lock insert open rollback savepoint set sql execute
         commit forall merge pipe
         O símbolo "exit" foi substituído por ";" para continuar.

SQL>
```

O comando basicamente mostra duas colunas. A primeira com a linha e a coluna onde ocorreu o erro, e a segunda, a coluna com a descrição. Este erro também pode ser encontrado através das views `user_errors`, `all_errors` e `dba_errors`.

```
SQL> select line
  2        ,position
  3        ,text
  4  from   user_errors
  5  where  name = 'CALC';

 LinE  POSITION TEXT
----- --------- -----------------------------------------------
   34         3 PLS-00103: Encontrado o símbolo ";" quando um
                           dos seguintes símbolos era esperado:

                         begin case declare end exception exit
                         for goto if loop mod null pragma raise
                         return select update while with
                         <um identificador> <um identificador
                         delimitado por aspas duplas>
```

```
                                   <uma variável de ligação> << close
                                   current delete fetch
                                   lock insert open rollback savepoint
                                   setsql execute commit
                                   forall merge pipe
                                O símbolo "exit" foi substituído por ";"
                                para continuar.


SQL>
```

16.7 Passando parâmetros

Programas armazenados, como procedures e functions, permitem trabalhar com passagem de parâmetros. Quando criamos um procedimento ou uma função, podemos especificar parâmetros de entrada e saída para que valores possam ser levados para dentro destes programas ou recuperados deles.

Os parâmetros podem ser do tipo in, out ou in out. in define que se trata de um parâmetro de entrada. Já um parâmetro do tipo out indica que ele é de saída. Logo, in out indica um parâmetro de entrada e saída.

Quando estamos falando de parâmetros de entrada (in), isso quer dizer que os valores contidos neles podem ser atribuídos a outras variáveis ou parâmetros existentes dentro da procedure ou function. Contudo, não podemos atribuir valores a eles. Já quando estamos falando dos parâmetros de saída (out), estamos dizendo que os valores destes parâmetros podem ser alterados, mas não atribuídos a outras variáveis ou parâmetros existentes dentro destes objetos.

Logo, quando temos parâmetros do tipo in out sendo usados, isso indica que tanto podemos atribuir valores a ele como também atribuir seus valores a outras variáveis ou parâmetros existentes dentro dos objetos. Veja isso na prática.

```
procedure exemplo( param1 in     number
                  ,param2 out    number
                  ,param3 in out number) is
  --
```

```
  x number;
  y number;
  z number;
  --
begin
  --
  x := param1; -- uso correto
  param1 := x; -- uso incorreto
  --
  y := param2; -- uso incorreto
  param2 := y; -- uso correto
  --
  z := param3; -- uso correto
  param3 := z; -- uso correto
  --
end;
/
```

> **Nota:** quando não informamos o tipo do parâmetro, por padrão, ele será do tipo `in`.

O uso de parâmetros em programas armazenados pode tornar nossos programas muito mais flexíveis, possibilitando o reaproveitamento de código. A seguir, são mostrados os mesmos exemplos de `procedures` e `functions` anteriores, mas agora recriados utilizando passagem de parâmetros.

```
SQL> create or replace procedure calc ( x1  in  number
  2                                   ,x2  in  number
  3                                   ,op  in  varchar2
  4                                   ,res out varchar2) is
  5    --
  6  begin
  7    --
  8    if (x1 + x2) = 0 then
  9      res := 0;
 10    elsif op = '*' then
 11      res := x1 * x2;
```

```
 12    elsif op = '/' then
 13      if x2 = 0 then
 14        res := 'Erro de divisão por zero!';
 15      else
 16        res := x1 / x2;
 17      end if;
 18    elsif op = '-' then
 19      res := x1 - x2;
 20      if res = 0 then
 21        res := 'Resultado igual a zero: '||res;
 22      elsif res < 0 then
 23        res := 'Resultado menor que zero: '||res;
 24      elsif res > 0 then
 25        res := 'Resultado maior que zero: '||res;
 26      end if;
 27    elsif op = '+' then
 28      res := x1 + x2;
 29    else
 30      res := 'Operador inválido!';
 31    end if;
 32    --
 33  end;
 34  /

Procedimento criado.

SQL>
```

Executando `calc`:

```
SQL> declare
  2    wres varchar2(100);
  3  begin
  4    calc ( x1  => 10
  5          ,x2  => 5
  6          ,op  => '*'
  7          ,res => wres);
  8    --
  9    dbms_output.put_line('Resultado Calc 1: '||wres);
 10    --
```

```
 11     calc ( 10
12            ,5
13            ,'/'
14            ,wres);
15     --
16     dbms_output.put_line('Resultado Calc 2: '||wres);
17     --
18   end;
19   /
Resultado Calc 1: 50
Resultado Calc 2: 2

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Perceba que nesta execução chamamos duas vezes a `procedure`. A primeira usando a passagem de parâmetros nomeada, e a segunda sem nomeação. Quando informamos os parâmetros no cabeçalho do programa, o Oracle toma como ordenação a ordem em que os dispomos. Todavia, quando executamos tal programa precisamos respeitar esta ordem. Portanto, quando nomeamos a passagem de parâmetros, no momento da execução do programa, não precisamos necessariamente informá-los na ordem com a qual foram definidos, basta apenas informar seus nomes e os respectivos valores a serem atribuídos.

```
SQL> create or replace function valida_cpf (cpf in char) return
varchar2 is
  2     --
  3     m_total  number     default 0;
  4     m_digito number     default 0;
  5     --
  6   begin
  7     for i in 1..9 loop
  8       m_total := m_total + substr (cpf, i, 1) * (11 - i);
  9     end loop;
 10     --
 11     m_digito := 11 - mod (m_total, 11);
```

```
12     --
13     if m_digito > 9 then
14       m_digito := 0;
15     end if;
16     --
17     if m_digito != substr (cpf, 10, 1) then
18       return 'I';
19     end if;
20     --
21     m_digito := 0;
22     m_total := 0;
23     --
24     for i in 1..10 loop
25       m_total := m_total + substr (cpf, i, 1) * (12 - i);
26     end loop;
27     --
28     m_digito := 11 - mod (m_total, 11);
29     --
30     if m_digito > 9 then
31       m_digito := 0;
32     end if;
33     --
34     if m_digito != substr (cpf, 11, 1) then
35       return 'I';
36     end if;
37     --
38     return 'V';
39     --
40   end valida_cpf;
41   /

Função criada.

SQL>
```

Executando `valida_cpf`:

```
SQL> declare
  2     --
  3     res varchar2(1) default null;
```

```
 4     --
 5   begin
 6     res := valida_cpf(cpf => '02091678520');
 7     --
 8     if res = 'V' then
 9       dbms_output.put_line('CPF válido: 02091678520');
10     else
11       dbms_output.put_line('CPF inválido: 02091678520');
12     end if;
13     --
14     res := valida_cpf('02011648920');
15     --
16     if res = 'V' then
17       dbms_output.put_line('CPF válido: 02011648920');
18     else
19       dbms_output.put_line('CPF inválido: 02011648920');
20     end if;
21     --
22   end;
23   /
CPF inválido: 02091678520
CPF válido: 02011648920

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Anteriormente, mencionamos que embora `procedures` não retornem valores, vimos que através de parâmetros do tipo `out` isto é possível. Contudo, as `procedures` não retornam valores implicitamente como as `functions`. Vale lembrar que, embora as `functions` retornem valor, nós podemos utilizar parâmetros do tipo `out` em sua definição, mesmo porque uma `function` só pode retornar um único valor por chamada, enquanto através dos parâmetros do tipo `out` é possível ter vários retornos.

16.8 Dependência de objetos

É de fundamental importância que o desenvolvedor conheça como o Oracle trata a questão da dependência entre os objetos armazenados no banco de dados. Existem casos em que o próprio desenvolvedor é quem vai liberar os objetos na base de dados, sendo ela uma base teste ou até em uma base de produção. Diante disto, é sempre bom saber o que pode acontecer quando liberamos objetos que possuem dependências entre si.

O Oracle trabalha em cima de dois conceitos com relação a este tipo de dependência. Os conceitos de **dependência direta** e **dependência indireta**. Conforme está na documentação da Oracle, quando se trata de uma dependência direta o Oracle consegue restabelecer o programa que está inválido. Já quando não se trata de uma dependência direta, o Oracle não garante que este restabelecimento seja feito de forma automática. Vamos verificar as características referentes às dependências diretas e indiretas entre objetos.

Para evidenciar as diferenças entre os tipos de dependências, criaremos exemplos práticos. Para começar, vamos partir do ponto onde temos quatro procedures chamadas: `proc1`, `proc2`, `proc3` e `proc4`.

```
SQL> create or replace procedure proc4 is
  2  begin
  3    dbms_output.put_line('proc. 4!!!');
  4  end;
  5  /

Procedimento criado.

SQL> create or replace procedure proc3 is
  2  begin
  3    dbms_output.put_line('proc. 3!!!');
  4    proc4;
  5  end;
  6  /

Procedimento criado.

SQL> create or replace procedure proc2 is
```

```
  2  begin
  3    dbms_output.put_line('proc. 2!!!');
  4    proc3;
  5  end;
  6  /

Procedimento criado.

SQL> create or replace procedure proc1 is
  2  begin
  3    dbms_output.put_line('proc. 1!!!');
  4    proc2;
  5  end;
  6  /

Procedimento criado.

SQL>
```

Observando os scripts anteriores, vemos que a `procedure proc1` chama a `proc2` e a `proc2` chama a `proc3` até chegarmos à `procedure proc4`. Neste caso, podemos identificar os dois casos de dependência os quais estamos abordando.

A dependência direta está representada nas seguintes situações: `proc1` com `proc2`, `proc2` com `proc3` e `proc3` com `proc4`. Ou seja, a procedure `proc2` mantém uma dependência com a `procedure proc1`, pois para a `proc2` ser executada é preciso que a `proc1` seja executada também. Logo, a `proc2` precisa estar válida para que a `proc1` esteja validada. Assim acontece com a `procedure proc2` que mantém uma dependência com a `proc3` e a `procedure proc3` com a `proc4`.

A dependência indireta está representada nas situações seguintes: `proc1` com `proc3`, `proc2` com `proc4`, `proc1` com `proc4`. Da mesma forma que na dependência direta, a procedure `proc1` precisa ser executada para que a `proc3` também seja (indiretamente). Logo, a validação da `procedure proc3` precisa existir para que a `proc1` também esteja válida. Veja o desenho.

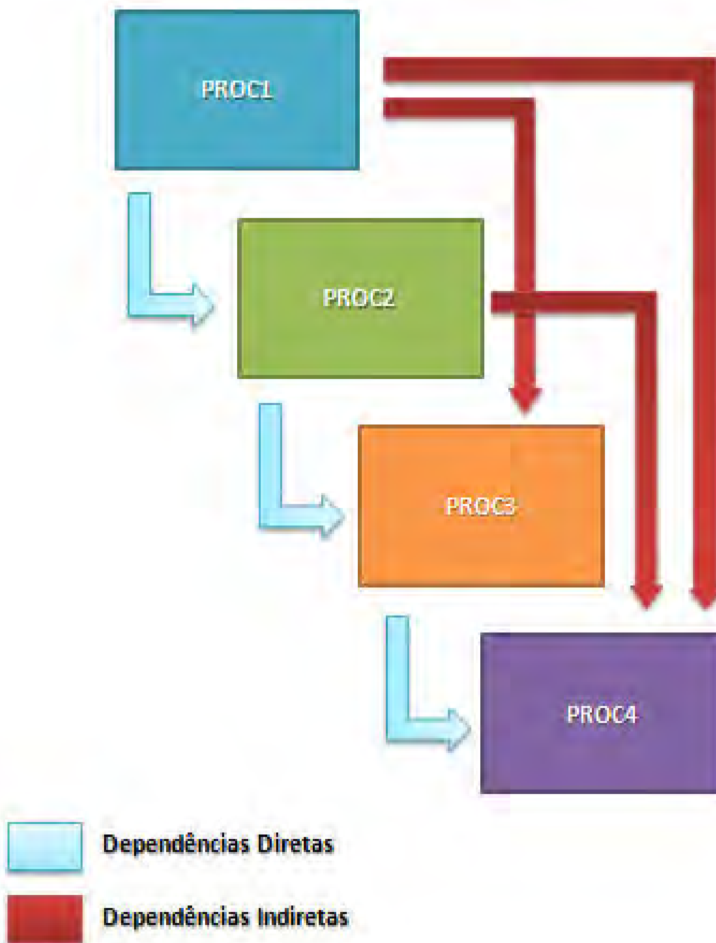

Fig. 16.1: Esquema mostrando dependências diretas e indiretas

Para verificar as dependências entre os objetos, podemos fazer um `select` na view `all_dependencies`, onde constam todas as dependências existentes entres os objetos criados na base de dados. Para limitarmos a pesquisa, vamos selecionar as dependências referentes aos objetos que criamos, informando os nomes dos mesmos na cláusula `where` do `select`.

```
SQL> select name || ' => ' ||referenced_name "Referências"
  2  from   all_dependencies
  3  where  owner           = 'TSQL'
  4  and    name            in ( 'PROC1'
  5                             ,'PROC2'
  6                             ,'PROC3'
  7                             ,'PROC4'
  8                            )
  9  and    referenced_type = 'procedure'
 10  order  by name
 11  /

Referências
------------------------------------------------------------
PROC1 => PROC2
PROC2 => PROC3
PROC3 => PROC4

SQL>
```

Já para verificar os status dos objetos, devemos fazer um `select` na view `all_objects`, selecionando o campo `STATUS`.

```
SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                         ,'PROC2'
  7                         ,'PROC3'
  8                         ,'PROC4'
  9                        )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          VALID
PROC2                          VALID
```

```
PROC3                          VALID
PROC4                          VALID

SQL>
```

Para tornar mais clara a explanação sobre as dependências entre objetos, na sequência estão alguns exemplos relacionados aos conceitos expostos anteriormente:

Criando objetos fora da ordem de dependência. Como já havíamos criado os objetos anteriormente, vamos primeiramente excluí-los.

```
SQL> drop procedure proc1;

Procedimento eliminado.

SQL> drop procedure proc2;

Procedimento eliminado.

SQL> drop procedure proc3;

Procedimento eliminado.

SQL> drop procedure proc4;

Procedimento eliminado.

SQL>
```

Após excluir cada `procedure`, vamos criá-las novamente.

```
SQL> create or replace procedure proc1 is
  2  begin
  3    dbms_output.put_line('proc. 1!!!');
  4    proc2;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.
```

```
SQL>
SQL> create or replace procedure proc2 is
  2  begin
  3    dbms_output.put_line('proc. 2!!!');
  4    proc3;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.

SQL>

SQL> create or replace procedure proc3 is
  2  begin
  3    dbms_output.put_line('proc. 3!!!');
  4    proc4;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.

SQL>
SQL> create or replace procedure proc4 is
  2  begin
  3    dbms_output.put_line('proc. 4!!!');
  4  end;
  5  /

Procedimento criado.

SQL>
```

Veja que `proc1`, `proc2` e `proc3` foram criadas com advertência. Contudo, a `proc4` foi criada com sucesso. Agora vamos verificar os status dos objetos.

```
SQL> select object_name
  2         ,status
```

```
  3  from    all_objects
  4  where   owner        = 'TSQL'
  5  and     object_name in ( 'PROC1'
  6                          ,'PROC2'
  7                          ,'PROC3'
  8                          ,'PROC4'
  9                         )
 10  and     object_type = 'procedure';

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          INVALID
PROC4                          VALID
SQL>
```

Como era previsto, as procedures PROC1, PROC2 e PROC3 estão inválidas, enquanto a PROC4 está válida. Isto ocorreu pelo fato de que na ordem em que os objetos foram criados sempre faltava a criação do objeto dependente, com exceção, é claro, do objeto PROC4.

Criando os objetos na ordem de dependência:

```
SQL> create or replace procedure proc4 is
  2  begin
  3    dbms_output.put_line('proc. 4!!!');
  4  end;
  5  /
Procedimento criado.

SQL> create or replace procedure proc3 is
  2  begin
  3    dbms_output.put_line('proc. 3!!!');
  4    proc4;
  5  end;
  6  /
Procedimento criado.

SQL> create or replace procedure proc2 is
```

```
  2  begin
  3    dbms_output.put_line('proc. 2!!!');
  4    proc3;
  5  end;
  6  /
Procedimento criado.

SQL> create or replace procedure proc1 is
  2  begin
  3    dbms_output.put_line('proc. 1!!!');
  4    proc2;
  5  end;
  6  /
Procedimento criado.
SQL>
```

Verificando Objetos.

```
SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                         ,'PROC2'
  7                         ,'PROC3'
  8                         ,'PROC4'
  9                        )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          VALID
PROC2                          VALID
PROC3                          VALID
PROC4                          VALID
SQL>
```

Invalidando o objeto `PROC1`:

```
SQL> create or replace procedure proc1 is
  2  begin
  3    dbms_output.put_line('proc. 1!!!');
  4    ; -- provocando um erro de sintaxe
  5    proc2;
  6  end;
  7  /

Advertência: Procedimento criado com erros de compilação.
SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                         ,'PROC2'
  7                         ,'PROC3'
  8                         ,'PROC4'
  9                        )
 10  and    object_type = 'procedure';

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          VALID
PROC3                          VALID
PROC4                          VALID
SQL>
```

Invalidando o objeto PROC2:

```
SQL> create or replace procedure proc2 is
  2  begin
  3    dbms_output.put_line('proc. 2!!!');
  4    ; -- provocando um erro de sintaxe
  5    proc3;
  6  end;
  7  /

Advertência: Procedimento criado com erros de compilação.
SQL> select object_name
```

```
  2         ,status
  3  from    all_objects
  4  where   owner        = 'TSQL'
  5  and     object_name in ( 'PROC1'
  6                          ,'PROC2'
  7                          ,'PROC3'
  8                          ,'PROC4'
  9                         )
 10  and     object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          VALID
PROC4                          VALID
SQL>
```

Invalidando o objeto `PROC3`:

```
SQL> create or replace procedure proc3 is
  2  begin
  3    dbms_output.put_line('proc. 3!!!');
  4    ; -- provocando um erro de sintaxe
  5    proc4;
  6  end;
  7  /
Advertência: Procedimento criado com erros de compilação.

SQL> select object_name
  2         ,status
  3  from    all_objects
  4  where   owner        = 'TSQL'
  5  and     object_name in ( 'PROC1'
  6                          ,'PROC2'
  7                          ,'PROC3'
  8                          ,'PROC4'
  9                         )
 10  and     object_type = 'procedure'
```

```
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          INVALID
PROC4                          VALID
SQL>
```

Invalidando o objeto `PROC4`:

```
SQL> create or replace procedure proc4 is
  2  begin
  3    dbms_output.put_line('proc. 4!!!');
  4    ; -- provocando um erro de sintaxe
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.
SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                         ,'PROC2'
  7                         ,'PROC3'
  8                         ,'PROC4'
  9                        )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          INVALID
PROC4                          INVALID
SQL>
```

Corrigindo e recriando o objeto `PROC1`:

```
SQL> create or replace procedure proc1 is
  2  begin
  3    dbms_output.put_line('proc. 1!!!');
  4    proc2;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.
SQL> show error
Erros para procedure PROC1:

LinE/COL ERROR
-------- -----------------------------------------------------------------
4/3      PL/SQL: Statement ignored
4/3      PLS-00905: o objeto TSQL.PROC2 é inválido
SQL>
```

Corrigindo e recriando o objeto `PROC2`:

```
SQL> create or replace procedure proc2 is
  2  begin
  3    dbms_output.put_line('proc. 2!!!');
  4    proc3;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.
SQL> show error
Erros para procedure PROC2:

LinE/COL ERROR
-------- -----------------------------------------------------------------
4/3      PL/SQL: Statement ignored
4/3      PLS-00905: o objeto TSQL.PROC3 é inválido
SQL>
```

Corrigindo e recriando o objeto `PROC3`:

```
SQL> create or replace procedure proc3 is
  2  begin
  3    dbms_output.put_line('proc. 3!!!');
  4    proc4;
  5  end;
  6  /

Advertência: Procedimento criado com erros de compilação.
SQL> show error
Erros para procedure PROC3:

LinE/COL ERROR
-------- -----------------------------------------------------
4/3      PL/SQL: Statement ignored
4/3      PLS-00905: o objeto TSQL.PROC4 é inválido
SQL>
```

Verificando objetos:

```
SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                        ,'PROC2'
  7                        ,'PROC3'
  8                        ,'PROC4'
  9                       )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          INVALID
PROC4                          INVALID

SQL>
```

Corrigindo e validando o objeto `PROC4`:

```
SQL> create or replace procedure proc4 is
  2  begin
  3    dbms_output.put_line('proc. 4!!!');
  4  end;
  5  /

Procedimento criado.

SQL> select object_name
  2         ,status
  3  from   all_objects
  4  where  owner       = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                         ,'PROC2'
  7                         ,'PROC3'
  8                         ,'PROC4'
  9                        )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          INVALID
PROC3                          INVALID
PROC4                          VALID

SQL>
```

Recompilando `PROC2`:

```
SQL> alter procedure proc2 compile;

Procedimento alterado.

SQL> select object_name
  2         ,status
  3  from   all_objects
```

```
  4  where  owner        = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                          ,'PROC2'
  7                          ,'PROC3'
  8                          ,'PROC4'
  9                         )
 10  and    object_type = 'procedure'
 11  /

OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          INVALID
PROC2                          VALID
PROC3                          VALID
PROC4                          VALID

SQL>
```

Executando a `PROC1`:

```
SQL> execute proc1;
proc. 1!!!
proc. 2!!!
proc. 3!!!
proc. 4!!!

Procedimento PL/SQL concluído com sucesso.

SQL> select object_name
  2        ,status
  3  from   all_objects
  4  where  owner        = 'TSQL'
  5  and    object_name in ( 'PROC1'
  6                          ,'PROC2'
  7                          ,'PROC3'
  8                          ,'PROC4'
  9                         )
 10  and    object_type = 'procedure'
 11  /
```

```
OBJECT_NAME                    STATUS
------------------------------ -------
PROC1                          VALID
PROC2                          VALID
PROC3                          VALID
PROC4                          VALID

SQL>
```

Vimos que, embora existam dependências indiretas entre os objetos, o Oracle conseguiu identificar as chamadas e validar os objetos subsequentes. Embora isdo tenha ocorrido, a Oracle não garante que em alguns casos isto venha a acontecer. Diante disto, podemos dizer que o Oracle está preparado para restabelecer as validações em caso de dependência direta e indireta, contudo, não é garantido que ele conseguirá detectar certas ligações.

Todavia, é sempre adequado verificar os status dos objetos após a liberação de objetos novos ou novas versões no banco de dados. Dependendo da quantidade de objetos com dependência, invalidá-los poderá nos custar algum tempo até deixar o sistema estável novamente. Vale lembrar que, mesmo existindo objetos inválidos no banco de dados, podemos não ter problemas no funcionamento, caso o Oracle, ao executar alguma operação com um destes objetos, detecte a existência de dependências diretas e as restabeleça.

Capítulo 17

packages

No capítulo anterior, falamos sobre `procedures` e `functions` e como elas são armazenadas e executadas através de um banco de dados. Pois bem, `packages`, também chamados de pacotes, são programas armazenados, tendo como diferencial a possibilidade de funcionar como repositório para agrupar vários objetos do tipo `procedure` e `function`, bem como conter código PL/SQL ou servir de área para definições de variáveis, cursores, exceções, procedimentos e funções.

Contudo, talvez sua maior utilização esteja no agrupamento de programas que possuem uma mesma finalidade, como aqueles que façam parte de uma área específica como RH, Financeira ou Comercial. Através de `packages` conseguimos organizar estes programas dentro de um único objeto e definir como o acesso a estes objetos será realizado.

17.1 Estrutura de um package

Um `package` pode ser constituído por até duas partes, uma chamada `especification` e outra chamada `body`. Podemos ter um `package especification` sem `package body`, contudo, não podemos ter o contrário. Obrigatoriamente, para criamos um `package body` é necessário criar também um `package especification`. Já vamos entender por quê.

Dentro do `especification`, podemos declarar variáveis, `types`, cabeçalhos de `procedures` e `functions`, `exceptions`, cursores etc., que podem fazer ou não referência a um `package body`. Quando estiver fazendo referência a um `body`, podemos dizer que o `especification`, como nome sugere, funciona como uma especificação do `body`.

Por exemplo, se temos um `body` dentro do qual há um código de uma `procedure`, podemos ter declarado no `especification` o cabeçalho deste programa, mesmo porque, por definição deste tipo de objeto, só conseguiremos acessar um programa que está armazenado dentro de um `body` caso sua especificação esteja declarada em um `especification`. Por isto, um `body` não pode existir sem um `especification`.

Caso o `especification` não faça referência a um `body`, ele pode estar sendo utilizado para declaração de variáveis ou outros tipos de objetos que não necessitariam de um `body`. Vale salientar que, caso tenhamos cabeçalhos de `procedures` e `functions` em um `especification`, se faz necessário que exista um `body` relacionado.

Já dentro do `body` é onde colocamos toda a codificação dos nossos programas, como codificações de `procedures` e `functions`. Além destes objetos, também podemos declarar variáveis, cursores, exceções, types etc., que podem ser utilizados pelo restante dos objetos criados no `body`. No entanto, podemos ter objetos dentro do `body` que não estão declarados no `especification`, os quais não poderão ser acessados diretamente, somente através de outros objetos contidos no próprio `body`. Os objetos declarados no `especification`, por exemplo, variáveis e cursores, podem ser acessados de dentro do `body`. Além disso, dentro do `body` podemos ter codificação PL/SQL correspondente ao próprio `package`, pois o `body` pode conter sua própria área `begin` e `end`.

Podemos ter no especification

- Especificação de *procedures* e *functions*;
- Declaração de variáveis e constantes;
- Declaração de cursores;
- Declaração de *exceptions*;
- Declaração de *types*.

Podemos ter no body

- Códigos PL/SQL;
- Códigos de *procedures* e *functions*;
- Declaração de variáveis e constantes;
- Declaração de cursores;
- Declaração de *exceptions*;
- Declaração de *types*.

Exemplo de `especification`:

```
SQL> create package listagem is
  2     --
  3     cursor c1 is
  4       select  d.department_id
  5              ,department_name
  6              ,first_name
  7              ,hire_date
  8              ,salary
  9       from   departments d
 10             ,employees e
 11       where  d.manager_id = e.employee_id
 12       order  by department_name;
```

```
13      --
14      type tab is table of c1%rowtype index by binary_integer;
15      --
16      tbgerente tab;
17      n  number;
18      --
19      procedure lista_gerente_por_depto;
20      --
21   end listagem;
22   /

Pacote criado.

SQL>
```

Para criarmos um `package especification`, utilizamos o comando `create`. Seguindo nosso exemplo, na primeira linha temos o comando `create` e logo em seguida o tipo de objeto que estamos criando, no caso, `package`. Quando não informamos o tipo `body`, automaticamente o Oracle cria o `package` como sendo do tipo `especification`.

Logo depois do tipo de objeto, informamos o nome (`listagem`) seguido da expressão `is` (linha 01). `is` pode ser substituído por `as`. Sempre que for criar um objeto `package especification`, você deve utilizar esta sintaxe.

Depois da expressão `is`, são declarados os objetos que poderão ou não ser usados em um `package body`. Neste exemplo, temos um `especification` com a declaração de alguns objetos como o cursor `c1`, o `type tab`, as variáveis `n` e `tbgerente`, e a especificação da procedure `lista_gerente_por_depto`. Note que só informamos o cabeçalho da `procedure`. Logo terá que existir um `body` onde seu código estará criado. Os cabeçalhos dos objetos informados devem ser idênticos aos descritos no corpo do `package`.

Note que, nestes casos, todos os objetos, por estarem no `especification`, têm seu escopo público, ou seja, qualquer usuário que tenha acesso a este objeto poderá referenciá-los em sua sessão, utilizando-os. Toda a declaração é finalizada por ponto e vírgula. Por fim,

temos o comando end (linha 21), que finaliza o package. Colocar ou não o nome do package logo após o comando end é opcional. Contudo, não podemos ter nomes diferentes no cabeçalho e rodapé do objeto.

Exemplo de body:

```
SQL> create package body listagem is
  2    --
  3    procedure lista_gerente_por_depto is
  4      --
  5    begin
  6      for r1 in c1 loop
  7        tbgerente(r1.department_id) := r1;
  8      end loop;
  9      --
 10      n := tbgerente.first;
 11      --
 12      while n <= tbgerente.last loop
 13        dbms_output.put_line(
                'Depto: '||tbgerente(n).department_name||' '||
 14                'Gerente: '||tbgerente(n).first_name||' '||
 15                'Dt. Admi.: '||tbgerente(n).hire_date||' '||
 16             'Sal.: '||to_char(tbgerente(n).salary,
                                  'fm$999g999g990d00'));
 17        n := tbgerente.next(n);
 18      end loop;
 19      --
 20    end lista_gerente_por_depto;
 21    --
 22  end listagem;
 23  /

Corpo de Pacote criado.

SQL>
```

Igualmente ao utilizado na criação do package especification, para criarmos um package body, nós utilizamos também o comando create. Na primeira linha deste exemplo, temos o comando create e, em seguida, o tipo de objeto que estamos criando, no caso, package body.

Logo depois do tipo de objeto, informamos um nome para ele (`listagem`), seguido da expressão `is` (linha 01), que pode ser substituído por `as`. Sempre que for criar um objeto `package body`, você deve utilizar esta sintaxe.

Este objeto `body` faz referência ao `especification` criado no exemplo anterior. Nele está o código da `procedure lista_gerente_por_depto` (linhas 3 a 20), declarada no `especification`. Note que, neste exemplo, não utilizamos a área `begin end` do `body` e também não declaramos qualquer objeto dentro do corpo do `package`. Contudo, dentro da `procedure`, estamos utilizando os objetos que foram declarados no `especification`, por exemplo, o cursor (linha 6 a 8), o type (linha 7, 13 a 17) e as variáveis (linhas 10, 12 e 17).

Assim como esta, outras `procedures`, caso existissem mais dentro do `body`, poderiam também acessar estes objetos. Todavia, se eles estivessem declarados no `body`, em vez de no `especification`, o efeito seria o mesmo. Entretanto, o escopo destes objetos seria interno ao `body`, não podendo ser acessados externamente.

Por fim, temos o comando `end` (linha 22), que finaliza o `package`. Como no `especification`, colocar ou não o nome do `package` logo após o comando `end` é opcional. Contudo, não podemos ter nomes diferentes no cabeçalho e rodapé do objeto. Vale ressaltar que toda a declaração deve ser finalizada por ponto e vírgula.

17.2 Acesso a packages

Como mencionado anteriormente, só podemos acessar um objeto que se encontra em um `body` se ele estiver declarado no `especification`. Logo, o `especification` tem a característica de ser uma área pública onde todos os usuários que possuam concessão de acesso a este objeto podem referenciar ou executar os objetos do `body` por intermédio do que está especificado nesta área.

Tudo o que estiver declarado no `especification` e no `body` tem seu escopo no nível de usuário, ou melhor, na sessão do usuário que executou tal `package`. Os objetos `package` só começam a ocupar recursos na memória quando algum objeto é referenciado. A área de `begin end` do `body` é exe-

cutada uma única vez na sessão do usuário, na primeira vez que o `package` é referenciado.

Por isso, quando utilizada, é comum vê-la servir como local para inicialização de variáveis ou para execução de procedimentos que inicializem algum objeto que será utilizado por outros objetos ao longo da vida útil da sessão. Quanto aos objetos definidos dentro das `procedures` e `functions` internas ao `body`, estes têm seu escopo em nível local e os recursos só são ocupados mediante a execução de tais `procedures` e `functions`.

Às vezes, `packages especifications` são utilizadas como área para declaração e inicialização de variáveis ou cursores dentro de uma sessão, não existindo um `body`. Como o `especification` tem seu escopo em nível de sessão, ele é muito útil para ser usado pelos programas para guardar informações que serão utilizadas em algum processamento ou para determinar padrões em um sistema. Como estas informações ficam na sessão enquanto ela estiver aberta, todos os programas que estiverem sendo executados poderão compartilhar desta área e, consequentemente, dos dados e objetos que estão declarados nela.

Resumindo, quando trabalhamos com `packages` precisamos saber de alguns detalhes:

- `Packages` podem ser constituídas de duas partes: `especification` e `body`.
- Um `especification` pode existir sem um `body`, mas não o contrário.
- Quando existirem um `especification` e um `body`, os dois objetos devem ter exatamente o mesmo nome.
- Para acessar objetos de dentro de um `body`, é preciso que eles estejam declarados em um `especification` (acesso direto), ou que, pelo menos, possam ser acessados através de algum objeto que esteja no `especification` (acesso indireto).
- Os objetos pertencentes ao `package` têm seu escopo em nível de sessão do usuário o qual executou tal `package`.

- Os objetos de um `package` ocuparão recursos somente quando forem referenciados.
- Áreas como `especification` e `begin end` do `body` serão executadas uma única vez na sessão do usuário, na primeira vez em que são referenciadas.
- Igualmente a `procedures` e `functions`, é necessário ter concessão de acesso para executar `packages`.
- Com o uso de `especification` e `body`, é possível disponibilizar informações em nível global e em nível de programa, mantendo a integridade e consistência através de encapsulamento.

Veja na sequência um exemplo de mais uma `package` e sua aplicação. O `package` a seguir é responsável por efetuar cálculos como somatória, subtração, divisão e multiplicação. Para cada operação, existe uma `function` onde são passados dois parâmetros referentes aos números que se quer calcular. Após o cálculo, a função retorna o resultado.

```
SQL> create package calculo as
  2    --
  3    function soma (x1 number, x2 number) return number;
  4    function subtrai (x1 number, x2 number) return number;
  5    function multiplica (x1 number, x2 number) return number;
  6    function divide (x1 number, x2 number) return number;
  7    --
  8  end calculo;
  9  /

Pacote criado.

SQL>
```

Na primeira linha deste exemplo, temos o comando `create` e em seguida o tipo de objeto que estamos criando, no caso, `package` (`especification`). Logo depois do tipo de objeto, informamos o seu nome (`calculo`) seguido da expressão `as` (linha 01). No corpo do

especification, declaramos as funções soma, subtrai, multiplica e divide (linhas 3 a 6). Note que aqui só são informados os cabeçalhos das funções. Por fim, temos o comando end (linha 8) que finaliza o package.

Detalhes sobre o package especification calculo:

- Foi criado o package especification chamado calculo;
- No package especification foram declaradas quatro funções, cada uma responsável por uma operação matemática (soma, subtração, multiplicação e divisão).

```
SQL> create package body calculo as
  2    --
  3    res number;
  4    --
  5    procedure imprime_msg(msg varchar2) is
  6    begin
  7      dbms_output.put_line(msg);
  8    end;
  9    --
 10    function soma (x1 number, x2 number) return number is
 11    begin
 12      res := x1 + x2;
 13      return res;
 14    end;
 15    --
 16    function subtrai (x1 number, x2 number) return number is
 17    begin
 18      res := x1 - x2;
 19      --
 20      if res = 0 then
 21        imprime_msg('Resultado igual a zero: '||res);
 22      elsif res < 0 then
 23        imprime_msg('Resultado menor que zero: '||res);
 24      elsif res > 0 then
 25        imprime_msg('Resultado maior que zero: '||res);
 26      end if;
 27      --
 28      return res;
```

```
29        --
30      end;
31      --
32      function multiplica (x1 number, x2 number) return
         number is
33      begin
34        res := x1 * x2;
35        return res;
36      end;
37      --
38      function divide (x1 number, x2 number) return number is
39      begin
40        if x2 = 0 then
41          res := null;
42          imprime_msg('Erro de divisão por zero!');
43        else
44          res := x1 / x2;
45        end if;
46        --
47        return res;
48      end;
49      --
50    end calculo;
51    /

Corpo de Pacote criado.

SQL>
```

Agora vamos criar o segundo objeto, que é a segunda parte do nosso programa. Na primeira linha temos o comando `create` e em seguida o tipo do objeto, `package body`, com seu nome, `calculo`, e a expressão `as` (linha 1). Declaramos uma variável chamada `res` no corpo do `package` (linha 3). Esta variável está sendo utilizada em todo o programa. Entre as linhas 10 a 46, estão os códigos das funções declaradas no objeto `especification`. Além destes, também temos um objeto `procedure` (linhas 5 a 8) utilizado para imprimir mensagens na tela. Se observarmos este objeto, veremos que ele não foi declarado no `especification`, logo, não poderá ser acessado

de fora do `body`.

Em nosso programa, a `procedure imprime_msg` está sendo utilizada nas linhas 21,23, 25 e 42, pelas funções `subtrai` e `divide`. Vale lembrar mais uma vez que toda a declaração ou codificação de objeto deve ser finalizada por ponto e vírgula. Como no `especification`, o `body também` é finalizado com o comando `end` (linha 48), seguindo as mesmas regras. É importante frisar que os nomes dos objetos `especification` e `body` devem ser iguais.

Detalhes sobre o `package body calculo`:

- Foi criado o `package body` chamado `calculo`;
- No `package body` foi declarada uma variável chamada `res` que será usada internamente pelos programas existentes (escopo privado);
- Foram codificados os objetos referentes às functions `soma`, `subtrai`, `multiplica` e `divide`, declaradas no `especification`; estas, de escopo público;
- Também foi codificado um objeto `procedure` chamado `imprime_msg`, de escopo privado, e que imprime mensagens na tela, vindas das `functions`.

Executando o `package calculo`:

```
SQL> declare
  2    res number;
  3  begin
  4    res := calculo.soma(450, 550);
  5    --
  6    dbms_output.put_line('450 + 550 = '||res);
  7  end;
  8  /
450 + 550 = 1000

Procedimento PL/SQL concluído com sucesso.

SQL> declare
  2    res number;
```

```
  3  begin
  4    res := calculo.subtrai(350, 650);
  5    --
  6    dbms_output.put_line('350 - 650 = '||res);
  7  end;
  8  /
Resultado menor que zero: -300
350 - 650 = -300

Procedimento PL/SQL concluído com sucesso.

SQL> declare
  2    res number;
  3  begin
  4    res := calculo.multiplica(20, 10);
  5    --
  6    dbms_output.put_line('20 * 10 = '||res);
  7  end;
  8  /
20 * 10 = 200

Procedimento PL/SQL concluído com sucesso.
SQL> declare
  2    res number;
  3  begin
  4    res := calculo.divide(50, 5);
  5    --
  6    dbms_output.put_line('50 / 5 = '||res);
  7  end;
  8  /
50 / 5 = 10

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

Observe o que acontece quando tentamos acessar algum objeto que não tem escopo no nível público, ou seja, que não se encontra declarado no `especification`:

```
SQL> declare
```

```
  2    res number;
  3  begin
  4    res := calculo.divide(50, 5);
  5    --
  6    calculo.imprime_msg('50 / 5 = '||res);
  7  end;
  8  /
  calculo.imprime_msg('50 / 5 = '||res);
          *
ERRO na linha 6:
ORA-06550: linha 6, coluna 11:
PLS-00302: o componente 'imprime_msg' deve ser declarado
ORA-06550: linha 6, coluna 3:
PL/SQL: Statement ignored

SQL> begin
  2    calculo.res := calculo.divide(50, 5);
  3    --
  4    dbms_output.put_line('50 / 5 = '||calculo.res);
  5  end;
  6  /
  calculo.res := calculo.divide(50, 5);
          *
ERRO na linha 2:
ORA-06550: linha 2, coluna 11:
PLS-00302: o componente 'RES' deve ser declarado
ORA-06550: linha 2, coluna 3:
PL/SQL: Statement ignored
ORA-06550: linha 4, coluna 45:
PLS-00302: o componente 'RES' deve ser declarado
ORA-06550: linha 4, coluna 3:
PL/SQL: Statement ignored
SQL>
```

Nos casos em que tentarmos acessar objetos não públicos, o Oracle informa que ele deve ser declarado.

17.3 Recompilando packages

Quando alteramos um `package` no banco de dados, pode acontecer de termos que compilá-la, novamente. Para isso, utilizamos o comando `alter`. Veja os exemplos na sequência:

```
SQL> alter package listagem compile;

Pacote alterado.

SQL>
```

17.4 Recuperando informações

Para visualizar informações referentes às `packages`, utilize as views `user_objects`, `all_objects` ou `dba_objects`. Nesta view constam informações como `STATUS` e data de criação do objeto.

```
SQL> select owner, object_type, status, created, last_ddl_time
  2  from   all_objects where object_name = 'listagem';

OWNER                          OBJECT_TYPE         STATUS
------------------------------ ------------------- -------
TSQL                           package             VALID
TSQL                           package body        VALID

createD  LAST_DDL
-------- --------
02/12/11 02/12/11
02/12/11 02/12/11

SQL>
```

Outra forma de recuperar dados referentes a `packages` é utilizando o comando `describe`. Ele mostra a definição de todas as `procedures` e `functions` contidas no `package`.

```
SQL> desc listagem
procedure LISTA_GERENTE_POR_DEPTO
```

```
SQL>
```

17.5 Recuperando códigos

Já para visualizar o código dos objetos `package` no banco de dados utilize as views `user_source`, `all_source` ou `dba_source`.

```
SQL> column text format a300
SQL> set lines 1000
SQL> set pages 1000
SQL> select line, text
  2  from   all_source where name = 'listagem';

      LINE TEXT
---------- --------------------------------------------------------
         1 package listagem is
         2    --
         3    cursor c1 is
         4      select  d.department_id
         5             ,department_name
         6             ,first_name
         7             ,hire_date
         8             ,salary
         9      from   departments d
        10            ,employees e
        11      where  d.manager_id = e.employee_id
        12      order  by department_name;
        13    --
        14    type tab is table of c1%rowtype index by
           binary_integer;
        15    --
        16    tbgerente tab;
        17    n  number;
        18    --
        19    procedure lista_gerente_por_depto;
        20    --
        21 end listagem;
         1 package body listagem is
```

```
 2    --
 3    procedure lista_gerente_por_depto is
 4      --
 5    begin
 6      for r1 in c1 loop
 7        tbgerente(r1.department_id) := r1;
 8      end loop;
 9      --
10      n := tbgerente.first;
11      --
12      while n <= tbgerente.last loop
13        dbms_output.put_line(
           'Depto: '||tbgerente(n).department_name||' '||
14            'Gerente: '||tbgerente(n).first_name||' '||
15            'Dt. Admi.: '||tbgerente(n).hire_date||' '||
16         'Sal.: '||to_char(tbgerente(n).salary,
                                 'fm$999g999g990d00'));
17        n := tbgerente.next(n);
18      end loop;
19      --
20    end lista_gerente_por_depto;
21    --
22 end listagem;

43 linhas selecionadas.

SQL>
```

17.6 Visualizando erros de compilação

Para visualizar erros de compilação, use o comando `show error`.

```
SQL> create or replace package listagem is
  2     --
  3     cursor c1 is
  4       select  d.department_id
  5              ,department_name
  6              ,first_name
  7              ,hire_date
```

```
 8              ,salary
 9        from   departments d
10              ,employees e
11        where  d.manager_id = e.employee_id
12        order  by department_name;
13      --
14      type tab is table of c1%rowtype index by binary_integer;
15      --
16      tbgerente tab;
17      n  number;
18      --
19      procedure lista_gerente_por_depto;
20      ; -- provocando um erro de sintaxe.
21      --
22   end listagem;
23   /

Advertência: Pacote criado com erros de compilação.

SQL> show error
Erros para package listagem:

LINE/COL ERROR
-------- -------------------------------------------------------
20/4     PLS-00103: Encontrado o símbolo ";" quando um dos
         seguintes
         símbolos era esperado:
         end function package pragma private procedure subtype
         type
         use <um identificador>
         <um identificador delimitado por aspas duplas> form
         current
         cursor

SQL>
```

O comando basicamente mostra duas colunas. A primeira com a linha e a coluna onde ocorreu o erro, e a segunda coluna com a descrição. Estes erros também podem ser visualizados através das views `user_errors`,

all_errors e dba_errors.

```
SQL> select line
  2         ,position
  3         ,text
  4  from   user_errors
  5  where  name = 'listagem';

 LINE  POSITION TEXT
----- --------- -------------------------------------------------
   20         4 PLS-00103: Encontrado o símbolo ";" quando um dos
                           seguintes símbolos era esperado:

                         end function package pragma private
                         procedure subtype type use
                         <um identificador> <um identificador
                         delimitado por aspas duplas> form
                         current cursor


SQL>
```

Capítulo 18

Transações autônomas

Quando buscamos informações em um banco de dados, esperamos que elas estejam íntegras. Para garantir a integridade e consistências dos dados, o Oracle mantém controles de transações. Quando uma operação DML é disparada, isso indica que os dados de uma tabela, por exemplo, podem estar sendo alterados. Nesse momento, podemos concluir que enquanto as alterações não forem confirmadas os dados que estão sendo manipulados na transação em questão não estão consistentes. Apenas quando forem confirmados, através de um `rollback` ou de um `commit`, os dados estarão íntegros.

Há situações em que podemos ter várias chamadas a outros objetos que estejam executando comandos DML e efetivando-os logo em seguida. Entretanto, ao fazer isso, corremos o risco de efetivar ações de outros comandos DML, executados anteriormente, que não deveriam ser efetivados, ou pelo menos não naquele momento. Isso acontece pois sabemos que, ao executar um comando de efetivação, como um `commit` ou um `rollback`, o Oracle

efetiva tudo o que estiver pendente na sessão.

Para esses casos, podemos utilizar transações autônomas para isolar ações de determinados programas. Utilizamos este recurso informando-o na declaração dos objetos, como em `procedures` e `functions` (dentro ou fora de `packages`), blocos anônimos (menos em sub-blocos) ou `triggers`. Sua função é fazer com que o Oracle abra uma nova sessão somente para executar tal objeto, ou seja, neste momento existirão pelo menos duas transações em aberto: a transação original e outra transação autônoma. Veja o exemplo a seguir.

```
SQL> declare
  2    --
  3    procedure lista_dept is
  4      --
  5      pragma autonomous_transaction;
  6    begin
  7      --
  8      dbms_output.new_line;
  9      dbms_output.put_line(
         '----------- Lista Departamentos -----------');
 10      dbms_output.new_line;
 11      --
 12      for i in (select * from dept order by deptno) loop
 13        dbms_output.put_line(i.deptno||' - '||i.dname);
 14      end loop;
 15      --
 16      commit;
 17      --
 18    end;
 19    --
 20  begin
 21    insert into dept (deptno, dname, loc)
        values (43, 'ORDER MANAGER','BRASIL');
 22    --
 23    lista_dept;
 24    --
 25    commit;
 26    --
```

```
27    lista_dept;
28    --
29  end;
30  /
```

Nesse exemplo, criamos um bloco PL/SQL anônimo e dentro dele declaramos uma procedure chamada `lista_dept`. Esta `procedure` lista todos os departamentos existentes. Ela foi declarada como `autonomous_transaction`, ou seja, quando ela for executada será aberta uma transação autônoma.

No corpo do bloco PL/SQL, temos um `insert` que inclui um departamento novo. Logo depois deste comando, chamamos a procedure `lista_dept` para listar todos os departamentos que já se encontram cadastrados. Depois disso, efetivamos a alteração realizada pelo `insert` e mandamos listar os departamentos, novamente. Agora vamos analisar o resultado.

```
----------- Lista Departamentos -----------
10 - ACCOUNTING
30 - SALES
40 - OPERATIONS
41 - GENERAL LEDGER
42 - PURCHASING
----------- Lista Departamentos -----------
10 - ACCOUNTING
30 - SALES
40 - OPERATIONS
41 - GENERAL LEDGER
42 - PURCHASING
43 - ORDER MANAGER

Procedimento PL/SQL concluído com sucesso.

SQL>
```

No resultado, temos duas listagens. Uma impressa após o comando `insert`, e outra após a efetivação do `insert` através do comando `commit`. Note que a impressão realizada antes do `commit` não mostra o departamento

43, embora, ele já tenha sido inserido neste ponto. Já a impressão após o comando `commit` mostra o novo departamento.

Desta forma, temos que nos atentar para o seguinte cenário. Sabemos que, quando é aberta uma transação no Oracle, ela monta uma imagem, digamos assim, dos objetos e dados disponíveis para o usuário nesta sessão. Com relação aos dados, a visão permite enxergar somente aquilo que está efetivado no banco de dados, ou seja, dados consistentes e íntegros. Isso quer dizer que dados pendentes de efetivação não poderão ser visualizados, isso é, dados excluídos, inseridos ou atualizados que ainda não tenham sido commitados.

Logo, quando isto acontece, não conseguimos visualizar as alterações que estão sendo realizadas. Quando temos um objeto utilizando transação autônoma, conseguimos manipular somente os dados efetivados. Dessa forma, se nós realizarmos um comando `select` a fim de obter os dados alterados na sessão original, mas não efetivados, não conseguiremos enxergá-los. Apenas para ressaltar, isso se dá ao fato de que dentro deste objeto estamos em outro escopo, em uma nova transação.

Outro ponto muito importante é que, quando utilizamos objetos com transação autônoma, precisamos efetivar a transação através de um `commit` ou de um `rollback`. Como este tipo de ação faz com que uma nova transação seja aberta, temos que finalizá-la. Caso contrário, ela ficaria pendente e isso poderia gerar um erro.

Conforme visto no exemplo anterior, utilizamos a seguinte linha de comando `pragma autonomous_transaction`, na área de declaração, para abrir uma nova transação. Veja outro exemplo a seguir.

Para este exemplo, criamos vários objetos. Primeiramente, criamos uma `procedure` chamada `lista_pais`, que listará na tela todos os países já cadastrados na tabela `countries`. Cada lista impressa terá um número indicando o número da impressão.

```
SQL> create procedure lista_pais(num_lista number) is
  2    --
  3  begin
  4    --
  5    dbms_output.put_line('Executando procedure: LISTA_PAIS');
  6    --
```

```
  7     dbms_output.new_line;
  8     dbms_output.put_line(
         '----------- Lista País - '||num_lista||'-----------');
  9     dbms_output.new_line;
 10     --
 11     for i in (select * from countries where region_id = 1
                  order by country_name) loop
 12       dbms_output.put_line(i.country_id||' -
           '||i.country_name);
 13     end loop;
 14     --
 15   end;
 16   /

Procedimento criado.

SQL>
```

A procedure a seguir tem o objetivo de inserir um registro na tabela `countries` referente ao país Portugal. Note que nesta procedure temos declarada uma transação autônoma. No corpo, além do comando `insert`, temos a chamada à procedure `lista_pais`, que vai listar os países cadastrados. Esta listagem será a segunda lista a ser impressa. Logo após, temos um `commit` que efetivará a ação do comando `insert` e encerrará a transação autônoma.

```
SQL> create procedure insere_pais_portugal is
  2     --
  3     pragma autonomous_transaction;
  4     --
  5   begin
  6     --
  7     dbms_output.put_line('Executando procedure:
         INSERE_PAIS_PORTUGAL');
  8     --
  9     insert into countries (country_id, country_name,
                               region_id) 10 values ('PT',
                               'Portugal',1);
 11     --
```

```
 12     lista_pais(2);
 13     --
 14     commit;
 15     --
 16   end;
 17   /

Procedimento criado.

SQL>
```

A próxima procedure tem a finalidade de apenas chamar a procedure insere_pais_portugal. Ela serve apenas para, digamos assim, estabelecer uma ligação indireta entre as procedures insere_pais_espanha e insere_pais_portugal.

```
SQL> create procedure chama_insere_pais_portugal is
  2     --
  3   begin
  4     --
  5     dbms_output.put_line('Executando procedure:
         CHAMA_INSERE_PAIS_PORTUGAL');
  6     --
  7     insere_pais_portugal;
  8     --
  9   end;
 10   /

Procedimento criado.

SQL>
```

Já a procedure insere_pais_espanha tem o objetivo de inserir um registro na tabela countries referente ao país Espanha. No corpo da procedure, além do comando insert, temos a chamada à procedure lista_pais, que irá listar os países cadastrados. Esta listagem será a primeira lista a ser impressa. Logo após, temos um rollback que desfará a ação do comando insert. Depois do rollback, temos uma nova cha-

mada à procedure `lista_pais`, a segunda neste objeto, que imprime a terceira listagem.

```
SQL> create procedure insere_pais_espanha is
  2    --
  3  begin
  4    --
  5    dbms_output.put_line('Executando procedure:
        INSERE_PAIS_ESPANHA');
  6    --
  7    insert into countries (country_id, country_name,
                               region_id)
  8                     values ('ES','Espanha',1);
  9    --
 10    lista_pais(1);
 11    --
 12    chama_insere_pais_portugal;
 13    --
 14    rollback;
 15    --
 16    lista_pais(3);
 17    --
 18  end;
 19  /

Procedimento criado.

SQL>
```

Note que em todos os objetos temos chamadas à procedure `dbms_output`, indicando a execução do objeto corrente. Usaremos esta identificação para saber a ordem cronológica da execução dos programas. Agora vamos executar a procedure `insere_pais_espanha`, que é a que desencadeará a execução dos demais objetos criados.

```
SQL> begin
  2    insere_pais_espanha;
  3  end;
  4  /
```

Esse código PL/SQL executa a procedure `insere_pais_espanha`, que chama as demais procedures onde obtivemos os resultados a seguir.

Neste nosso exemplo, ao executarmos essa procedure, primeiramente, o país Espanha foi inserido na tabela `countries`. Logo após a inserção, mandamos listar os países. O resultado da listagem 1 é mostrado a seguir.

```
Executando procedure: INSERE_PAIS_ESPANHA
Executando procedure: LISTA_PAIS
----------- Lista País - 1-----------
BE - Belgium
DK - Denmark
ES - Espanha
FR - France
DE - Germany
IT - Italy
NL - Netherlands
CH - Switzerland
UK - United Kingdom
```

Note que, na terceira linha da listagem, temos o país Espanha inserido. Logo após a execução do procedimento de listagem, continuando a execução da procedure `insere_pais_espanha`, foi chamada a procedure `chama_insere_pais_portugal`, que por sua vez, chama a procedure `insere_pais_portugal`. A procedure `insere_pais_portugal` será executada em outra transação, pois nela declaramos um `pragma autonomous_transaction`. Ela executa o `insert` referente ao país Portugal e logo em seguida executa a impressão da listagem 2. Para encerrar, um `commit` é executado, efetivando as operações. Veja o resultado.

```
Executando procedure: CHAMA_INSERE_PAIS_PORTUGAL
Executando procedure: INSERE_PAIS_PORTUGAL
Executando procedure: LISTA_PAIS
----------- Lista País - 2-----------
BE - Belgium
DK - Denmark
FR - France
DE - Germany
IT - Italy
```

```
NL - Netherlands
PT - Portugal
CH - Switzerland
UK - United Kingdom
```

Veja que aqui não apareceu o país Espanha. Isso porque a procedure `insere_pais_portugal` foi executada dentro de uma transação autônoma, logo, quando ela chama a listagem, apenas os dados efetivados e os da sessão corrente são visualizados.

Após o término da execução da procedure `insere_pais_portugal`, voltamos para a procedure `insere_pais_espanha`, onde, na sequência do programa, temos um `rollback`, desfazendo tudo que ficou pendente na sessão. Após desfazer as ações, executamos a listagem de número 3. A seguir, o resultado.

```
Executando procedure: LISTA_PAIS
---------- Lista País - 3-----------
BE - Belgium
DK - Denmark
FR - France
DE - Germany
IT - Italy
NL - Netherlands
PT - Portugal
CH - Switzerland
UK - United Kingdom

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que na saída da listagem final (listagem 3) temos cadastrado o país Portugal, mas não o país Espanha. O `rollback` acabou desfazendo a inserção referente ao país Espanha, mas não a inserção referente ao país Portugal, pois esta inserção foi realizada e efetivada via outra transação, a transação autônoma.

Não é comum ver transações autônomas sendo utilizadas em programas PL/SQL, contudo, em alguns casos elas se tornam imprescindíveis. Quando

estivermos trabalhando com `triggers` (capítulo mais a seguir), será visto que, para o disparo de determinados tipos de `triggers` em situações específicas, somos obrigados a usar transações autônomas para poder atender o objetivo.

Capítulo 19

Triggers

O objeto `trigger` nada mais é do que um bloco de código PL/SQL armazenado no banco de dados e que é disparado automaticamente mediante uma ação. Este disparo pode ocorrer de duas formas, por intermédio de alterações feitas em registros de uma determinada tabela, neste caso se tratando do disparo de um `trigger` de banco de dados ou por ações em nível de sistema por um `trigger` de sistema.

Vamos aprender sobre estes dois tipos, mostrando as características dos `triggers` de banco de dados que são muito utilizados no desenvolvimento de programas e dos *triggers de sistema* que são mais utilizados pelos administradores de banco de dados.

19.1 Trigger de banco de dados

Um trigger de banco de dados sempre está associado a uma tabela. Não existindo tabela, não existe trigger de bando de dados. Quando queremos que uma ação aconteça automaticamente mediante uma alteração em alguma tabela, ou melhor, nos registros que ela possui, podemos utilizá-los para isto.

Um trigger pode disparar quando alguma ação ocorre em uma tabela e também em um determinado momento. Por exemplo, podemos criar um trigger que dispare quando fizermos um update em determinada tabela, e que ocorra após o update dos registros. Além disso, podemos definir que este trigger disparará para cada linha afetada pelo comando dml, chamado de *trigger de linha*, ou se executará uma única vez independente de quantas linhas sofrerem alterações. Este último, nós chamamos de *trigger de tabela* ou de comando. Também podemos definir cláusulas where para o trigger para que o disparo só aconteça se alguns critérios forem obedecidos.

19.2 Trigger de tabela

Vamos analisar a definição do trigger.

```
create trigger tipo_tabela
  before delete or insert or update of sal on emp
begin

end;
```

Para criar um trigger, primeiramente damos um nome. Utilizamos o comando create trigger para criar o objeto. Temos obrigatoriamente que definir quando será o disparo. Neste exemplo, informamos que seu disparo ocorrerá antes da ação, neste caso before. Caso tenhamos necessidade que o disparo ocorra depois, informamos after.

Vamos abrir um parênteses aqui para explicar melhor esta questão de after e before na especificação do trigger. A expressão before definida no trigger pode ser um pouco confusa, pois dá a impressão que o trigger vai detectar que vamos executar um insert, por exemplo, e antes de acontecer tal ação ele dispara o trigger.

Mas não é bem assim. Na verdade, o `before` quer dizer antes que as alterações sejam feitas nos registros da tabela em questão, ou seja, podemos dizer que o Oracle faz as alterações primeiramente em memória, para depois efetivá-las no banco de dados. Com isso, dependendo do momento tratado, `before` ou `after`, podemos ou não alterar estes dados.

Quando o `trigger` é disparado no momento `before`, os dados ainda não foram efetivados, neste caso podemos alterar os valores antes que eles sejam gravados no banco de dados. Quando é disparado no `after` não há mais como alterá-los, pois já foram efetivados no banco de dados.

Depois de definir quando ocorrerá, precisamos informar em quais situações, ou melhor, ações, o `trigger` deve disparar. Note que para esta definição foi determinado que o disparo sempre ocorrerá quando a tabela `emp` sofrer um `delete`, `insert` ou `update` específico na coluna `sal`. Nos casos de `update`, podemos ou não informar colunas específicas para determinar o disparo. Neste exemplo, qualquer atualização que não seja na coluna `sal` da tabela `emp` o `trigger` não vai disparar. Caso não queira definir colunas específicas apenas não use a expressão `of`.

Este `trigger` pode ser caracterizado como um *trigger de tabela*, pois não informamos que ele vai disparar para cada linha afetada. Portanto, não importa quantas linhas serão excluídas, inseridas ou atualizadas, ele sempre disparará uma única vez.

Na sequência, segue um exemplo de *trigger de tabela* e sua aplicação. Vamos criar um `trigger` que fará a auditoria da tabela `emp`. Cada vez que um ou mais registros forem inseridos ou atualizados, este programa vai calcular a quantidade de registros contidos na tabela `emp`, a soma de todos os salários e comissões. Esses dados serão inseridos em uma tabela chamada `tab_audit_emp`. Vamos criar a tabela:

```
SQL> create table tab_audit_emp
  2  (
  3    nr_registros      number
  4   ,vl_total_salario  number
  5   ,vl_total_comissao number
  6  )
  7  /
```

```
Tabela criada.

SQL>
```

Segue o código da `trigger`:

```
SQL> create trigger tig_audit_emp
  2    after insert or delete or update of sal, comm on emp
  3  declare
  4     wnr_registros       number default 0;
  5     wvl_total_salario   number default 0;
  6     wvl_total_comissao  number default 0;
  7     wnr_registros_audit number default 0;
  8  begin
  9    select count(*)
 10    into   wnr_registros
 11    from   emp;
 12    --
 13    select sum(sal)
 14    into   wvl_total_salario
 15    from   emp;
 16    --
 17    select sum(comm)
 18    into   wvl_total_comissao
 19    from   emp;
 20    --
 21    select count(*)
 22    into   wnr_registros_audit
 23    from   tab_audit_emp;
 24    --
 25    if wnr_registros_audit = 0 then
 26      --
 27      insert into tab_audit_emp (
          nr_registros, vl_total_salario, vl_total_comissao)
 28       values (wnr_registros, wvl_total_salario,
                  wvl_total_comissao);
 29      --
 30    else
 31      --
 32      update tab_audit_emp
```

```
33      set    nr_registros      = wnr_registros
34             ,vl_total_salario  = wvl_total_salario
35             ,vl_total_comissao = wvl_total_comissao;
36      --
37    end if;
38  end;
39  /

Gatilho criado.

SQL>
```

Vamos entender o que este `trigger` faz. Primeiramente, chamamos este de `tig_audit_emp`. Ele vai disparar após a tabela `emp` sofrer um comando `insert`, `delete` ou `update`, sendo que para este último, somente quando as colunas `sal` ou `comm` forem alteradas. Note que a linha `for each row` não aparece. Isso indica que nosso `trigger` é um de tabela. Portanto, não importa a quantidade de linhas afetadas pelo comando, ele só vai disparar uma única vez. O `trigger` também só vai ser disparado após (`after`) os comandos `insert`, `delete` ou `update` forem efetivados.

Dentro destas condições o `trigger` dispara e executa comandos SQL para buscar as informações que serão utilizadas para gerar nossa auditoria, como a quantidade de registros na tabela, a soma de todos os salários e comissões. Isso justifica o fato de o `trigger` ser disparado uma única vez, pois neste disparo ele varre todos os registros da tabela, não necessitando trabalhar registro por registro afetado.

Depois de buscar as informações, o programa faz um `select` para verificar se já existe um registro na tabela de auditoria. Caso exista, ele somente atualiza as informações existentes. Caso contrário, insere um registro. Vamos testá-lo. Primeiramente, verificamos a tabela `tab_audit_emp`.

```
SQL> select * from tab_audit_emp;

não há linhas selecionadas

SQL>
```

Agora vamos pegar uma massa de teste para realizar uma atualização nos dados da tabela emp.

```
SQL> select * from emp where deptno = 20
  2  /

     EMPNO ENAME      JOB              MGR HIREDATE        SAL
---------- ---------- --------- ---------- -------- ----------
      7369 SMITH      CLERK           7902 17/12/80        880
      7566 JONES      MANAGER         7839 02/04/81     3272,5
      7788 SCOTT      ANALYST         7566 09/12/82       3300
      7876 ADAMS      CLERK           7788 12/01/83       1210
      7902 FORD       ANALYST         7566 03/12/81       3300

      COMM     DEPTNO
---------- ----------
                   20
                   20
                   20
                   20
                   20



SQL>
```

Vamos atualizar os salários de todos os empregados que estão alocados no departamento 20.

```
SQL> update emp
  2  set    sal = sal + (sal * 10 / 100)
  3  where  deptno = 20
  4  /

5 linhas atualizadas.

SQL> commit;

Validação completa.
```

```
SQL>
```

Vamos verificar as atualizações na tabela emp.

```
SQL> select * from emp where deptno = 20;

     EMPNO ENAME      JOB              MGR HIREDATE        SAL
---------- ---------- --------- ---------- -------- ----------
      7369 SMITH      CLERK           7902 17/12/80        968
      7566 JONES      MANAGER         7839 02/04/81    3599,75
      7788 SCOTT      ANALYST         7566 09/12/82       3630
      7876 ADAMS      CLERK           7788 12/01/83       1331
      7902 FORD       ANALYST         7566 03/12/81       3630

      COMM     DEPTNO
---------- ----------
                   20
                   20
                   20
                   20
                   20

SQL>
```

Ok. Os dados foram atualizados. Agora vamos verificar nossa tabela de auditoria.

```
SQL> select * from tab_audit_emp;

NR_REGISTROS VL_TOTAL_SALARIO VL_TOTAL_COMISSAO
------------ ---------------- -----------------
          14         31308,75              2244

SQL>
```

Note que o trigger realizou o cálculo considerando não só as linhas afetadas, mas sim todas as linhas da tabela, como era o esperado, ou seja, foi criado um registro nesta tabela contendo informações referentes a todos os

registros de funcionários cadastrados. Agora vamos excluir alguns registros da tabela `emp`.

```
SQL> delete from emp
  2  where deptno = 10
  3  /

3 linhas deletadas.

SQL> commit;

Validação completa.

SQL>
```

Ok. Excluímos os dados com sucesso. Agora vamos verificar nossa tabela de auditoria.

```
SQL> select * from tab_audit_emp;

NR_REGISTROS VL_TOTAL_SALARIO VL_TOTAL_COMISSAO
------------ ---------------- -----------------
          11         22558,75              2244

SQL>
```

Por último, vamos testar a inserção de dados.

```
SQL> insert into emp (empno, ename, job, mgr, hiredate, sal,
                      comm, deptno)
  2    values (7935, 'PAULO', 'CLERK', 7902,
              to_date('17/09/2011','dd/mm/yyyy'),
   1000, null, 20);

1 linha criada.

SQL>
```

Novo resultado.

```
SQL> select * from tab_audit_emp;

NR_REGISTROS VL_TOTAL_SALARIO VL_TOTAL_COMISSAO
------------ ---------------- -----------------
          12         23558,75              2244

SQL>
```

Apresentamos aqui apenas uma forma de como utilizar os `triggers` de tabela. Contudo, as possibilidades de utilização são muitas. Tudo vai depender da necessidade.

19.3 Trigger de linha

Muito parecido com o *trigger de tabela*, o de linha tem praticamente as mesmas características, sendo por um detalhe: o *trigger de linha* dispara para cada linha afetada pelo comando `dml`. Veja o exemplo na sequência.

```
create or replace trigger tipo_linha
  after update on func
  referencing old as V
              new as N
  for each row
begin

end;
```

Neste exemplo temos algo mais do que a definição de quando e como o `trigger` será disparado. Estamos definindo que ele será do tipo linha, ou seja, ele vai disparar para cada linha afetada pelo `update`. Para isso foi colocado em sua definição a seguinte linha de comando: `for each row`.

Quando trabalhamos com *trigger de linha*, podemos manipular os valores das linhas afetadas. Em alguns casos, podemos até alterar os valores de certas colunas antes de as mesmas serem salvas. Quando definimos que um `trigger` é de linha, o Oracle cria duas referências chamadas `old` e `new`. Estas referências apontam sempre para os dados anteriores às alterações e para os dados posteriores às alterações, respectivamente.

Por exemplo, se alteramos o valor da coluna `sal` de 100 para 500, `old.sal` conterá o valor 100 e `new.sal` conterá o valor 500. E assim será para todos os valores das colunas da tabela que estão sendo afetadas ou não pelo comando.

Quando temos um `trigger` sendo disparado no `before`, ou seja, antes do comando `dml`, nós podemos alterar o valor de `new`. O valor de `old` nunca pode ser alterado. Caso a alteração seja no `after`, nem `old` nem `new` podem ser alterados. Nem sempre teremos os valores de `old` e `new`. Quando estamos atualizando os registros em uma tabela teremos o `new` e o `old`, pois neste caso estamos atualizando um registro que já existe.

Quando estamos inserindo um registro, não temos os valores de `old`, pois se trata de um registro novo, não havendo assim valores antigos. Quando estamos excluindo um registro temos o `old` e não temos o `new`, pois estamos excluindo um registro, não informando dados novos.

As referências `old` e `new` podem receber apelidos. Neste caso utilizamos a diretriz `referencing` para definir esta informação. Neste exemplo, estamos criando os seguintes *alias*: `V` para `old` e `N` para `new`. Logo, quando formos trabalhar com as colunas devemos usar, por exemplo, `v.sal` para pegar o valor antigo e `n.sal` para o novo.

Veremos a seguir uma aplicação de *trigger de linha* onde criaremos um histórico para a troca de cargos dos empregados. Cada vez que um empregado trocar de cargo, um histórico contendo algumas informações será guardado. Neste histórico teremos o código do empregado, o cargo anterior, o novo cargo, a data da alteração e uma descrição. Para isso, vamos criar uma tabela que será o repositório destas informações históricas.

```
SQL> create table tab_hist_cargo_emp
  2  (
  3    empno              number
  4   ,job_anterior       varchar2(9)
  5   ,job_atual          varchar2(9)
  6   ,dt_alteracao_cargo date
  7   ,ds_historico       varchar2(2000)
  8  )
  9  /
```

```
Tabela criada.

SQL>
```

Criamos a `tab_hist_cargo_emp` que abrigará as informações que compõem o nosso histórico. Agora vamos para o `trigger` que fará este controle.

```
SQL> create trigger tig_hist_cargo_emp
  2    after update of job on emp
  3    referencing old as v
  4                new as n
  5    for each row
  6  begin
  7    insert into tab_hist_cargo_emp ( empno
  8                                    ,job_anterior
  9                                    ,job_atual
 10                                    ,dt_alteracao_cargo
 11                                    ,ds_historico)
 12                        values ( :n.empno
 13                                ,:v.job
 14                                ,:n.job
 15                                ,sysdate
 16                                ,'O empregado '||:n.ename||
 17                                 ' passou para o cargo '||:n.job||
 18                                 ' em carater de promoção.');
 19
 20  end;
 21  /

Gatilho criado.

SQL>
```

Cada vez que o `trigger tig_hist_cargo_emp` disparar, depois da atualização de um cargo, será gravado um histórico, ou seja, uma linha, na tabela `tab_hist_cargo_emp` contendo as informações pertinentes à alteração do cargo. O `trigger` vai disparar para cada linha alterada. Vamos testá-lo.

```
SQL> update emp
  2  set    job = 'MANAGER'
  3  where  empno = 7499
  4  /

1 linha atualizada.

SQL>
```

Verificando o histórico:

```
SQL> select * from tab_hist_cargo_emp
  2  /

     EMPNO JOB_ANTER JOB_ATUAL DT_ALTER
---------- --------- --------- --------
      7499 SALESMAN  MANAGER   05/09/11
DS_HISTORICO
-----------------------------------------------------------------
O empregado ALLEN passou para o cargo MANAGER em carater de
promoção.


SQL>
```

Muito bem. O histórico foi criado informando que o empregado `allen` mudou de cargo. Agora vamos a outro exemplo. Criaremos um `trigger` que irá calcular e guardar na tabela `emp` o percentual referente à comissão em relação ao salário do empregado. Para isso, vamos incluir uma coluna na tabela `emp` chamada `pc_com_sal`. Neste `trigger`, faremos uso na cláusula `when` para determinar quais os registros poderão sofre este cálculo. Vale salientar que a cláusula `when` pode ser usada tanto para os triggers de linhas como para os de tabela.

Alterando a tabela `emp`:

```
SQL> alter table emp add pc_com_sal number;

Tabela alterada.
```

```
SQL>
```

Criando o `trigger`:

```
SQL> create trigger tig_pc_com_sal_emp
  2    before insert or update of sal, comm on emp
  3    referencing old as v
  4                new as n
  5    for each row
  6    when (n.job = 'SALESMAN')
  7  begin
  8    :n.pc_com_sal := nvl(:n.comm,0) * 100 / :n.sal;
  9  end;
 10  /

Gatilho criado.

SQL>
```

Note que agora criamos um `trigger` que vai disparar no `before`, ou seja, antes das informações serem efetivadas no banco de dados. Usamos o `before`, pois o programa estará alterando um dado, ou melhor, o valor de uma coluna, referente à mesma linha que vai disparar o `trigger`. Portanto, isso deve acontecer antes que as informações sejam efetivadas. Se usássemos `after`, não conseguiríamos fazer tal alteração. Utilizamos a cláusula `when` para determinar que o `trigger` só vá disparar quando a linha em questão for de algum empregado que tenha como cargo (`job`) igual a `salesman`. Vamos testá-lo.

```
SQL> select empno, ename, job, sal, comm, pc_com_sal from emp;

     EMPNO ENAME      JOB              SAL       COMM PC_COM_SAL
---------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK            968
      7499 ALLEN      SALESMAN        1600        306
      7521 WARD       SALESMAN        1250        510
      7566 JONES      MANAGER      3599,75
      7654 MARTIN     SALESMAN        1250       1428
```

```
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
      7788 SCOTT      ANALYST         3630
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500              0
      7876 ADAMS      CLERK           1331
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3630
      7934 MILLER     CLERK           1300

14 linhas selecionadas.

SQL> update emp set sal = sal;

14 linhas atualizadas.

SQL>
```

Primeiramente, verificamos os dados da tabela `emp`. Logo após, executamos um `update` apenas para que nosso `trigger` dispare. Note que o `update` atualiza todos os salários dos empregados com o mesmo salário. Fizemos isso apenas para que todas as linhas sejam varridas e que o cálculo do percentual seja realizado para os registros já cadastrados. Vamos ver o resultado.

```
SQL> select empno, ename, job, sal, comm, pc_com_sal from emp;

     EMPNO ENAME      JOB              SAL       COMM PC_COM_SAL
---------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK            968
      7499 ALLEN      SALESMAN        1600        306     19,125
      7521 WARD       SALESMAN        1250        510       40,8
      7566 JONES      MANAGER      3599,75
      7654 MARTIN     SALESMAN        1250       1428     114,24
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
      7788 SCOTT      ANALYST         3630
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500          0          0
```

```
      7876 ADAMS      CLERK           1331
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3630
      7934 MILLER     CLERK           1300

14 linhas selecionadas.

SQL>
```

Veja que a coluna `pc_com_sal` recebeu o cálculo do percentual, conforme a ação disparada via `trigger`. Também podemos observar que somente os registros cujos empregados tenham como cargo `salesman` sofreram esta ação. Agora vamos inserir um novo empregado, com o cargo igual a `salesman` e outro com o cargo `clerk`, e vamos ver o que acontece.

```
SQL> insert into emp (empno, ename, job, mgr, hiredate, sal,
      comm, deptno)
  2   values (7940,'PEDRO','CLERK',7788,
      to_date('05/09/2011','dd/mm/rrrr'),750,100,30);

1 linha criada.

SQL> insert into emp (empno, ename, job, mgr, hiredate, sal,
      comm, deptno)
  2   values (7941,'JOAO','SALESMAN',7698,
      to_date('05/09/2011','dd/mm/rrrr'),1200,350,30)
  3  /

1 linha criada.

SQL>
```

Verificando a tabela `emp`.

```
SQL> select empno, ename, job, sal, comm, pc_com_sal from emp;

     EMPNO ENAME      JOB              SAL       COMM PC_COM_SAL
---------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK            968
```

```
      7499 ALLEN      SALESMAN        1600        306     19,125
      7521 WARD       SALESMAN        1250        510       40,8
      7566 JONES      MANAGER      3599,75
      7654 MARTIN     SALESMAN        1250       1428     114,24
      7698 BLAKE      MANAGER         2850
      7782 CLARK      MANAGER         2450
      7788 SCOTT      ANALYST         3630
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500          0          0
      7876 ADAMS      CLERK           1331
      7900 JAMES      CLERK            950
      7902 FORD       ANALYST         3630
      7934 MILLER     CLERK           1300
      7940 PEDRO      CLERK            750        100
      7941 JOAO       SALESMAN        1200        350 29,1666667

16 linhas selecionadas.

SQL>
```

Analisando as informações podemos constatar que o empregado `pedro`, embora tenha recebido valores de salário e comissão, não sofreu o cálculo do percentual. Isso ocorreu pelo fato de ele não ter como cargo, `salesman`. Neste caso, o `trigger` não disparou. Já para o empregado `joao`, o `trigger` disparou e calculou o percentual.

Aqui apresentamos alguns exemplos simples, apenas para mostrar o funcionamento dos `triggers`. Contudo, dependendo da necessidade podemos escrever códigos complexos para atender às mais variadas situações que podem surgir. Mesmo porque, como pode ser visto, um `trigger` nada mais é que um bloco PL/SQL, ou seja, um programa, que é programado para disparar de forma automática.

Predicados condicionais para triggers de banco de dados

Vimos que um mesmo `trigger` pode disparar por um ou mais tipos de ações, como no `insert`, `delete` e `update`. Contudo, dependendo da ação ocorrida na tabela, podem surgir necessidades diferentes para cada uma. Para isso, usamos os predicados condicionais que nos dizem qual ação resultou

no disparo do `trigger`. Estes predicados são: `inserting`, `updating` e `deleting`. Seus possíveis valores são `true` e `false`. Veja o exemplo a seguir, onde criaremos um histórico para gravar as ações de manipulação dos departamentos cadastrados na tabela `dept`. Criando a tabela:

```
SQL> create table tab_hist_dept
  2  ( deptno number
  3   ,dt_historico date
  4   ,ds_historico varchar2(2000)
  5  )
  6  /
Tabela criada.
SQL>
```

Nesta tabela será gravado o código do departamento, a data da ação e também um descritivo do histórico. Vamos criar o `trigger` que fará o controle de histórico.

```
SQL> create or replace trigger tig_hist_dept
  2    after insert or delete or update on dept
  3    for each row
  4  declare
  5    wacao varchar2(200) default null;
  6  begin
  7    --
  8    if inserting then
  9      wacao := 'inserido';
 10    elsif updating then
 11      wacao := 'atualizado';
 12    elsif deleting then
 13      wacao := 'excluído';
 14    end if;
 15    --
 16    insert into tab_hist_dept (deptno, dt_historico,
                                  ds_historico)
 17         values ( nvl(:new.deptno,:old.deptno)
 18                 ,sysdate
 19                 ,'O departamento '||
                    nvl(:new.dname,:old.dname)||'
```

```
                   foi '||wacao||'.');
 20    --
 21  end;
 22  /
Gatilho criado.
SQL>
```

O `trigger` em questão tem o objetivo de verificar qual a ação foi executada na tabela e a partir desta informação montar o histórico. Testando a geração do histórico:

```
SQL> update dept
  2  set dname = dname;

4 linhas atualizadas.

SQL> insert into dept values (50,'TI','BRASIL');

1 linha criada.

SQL> delete dept where deptno = 50;

1 linha deletada.

SQL>
```

Agora executamos uma série de ações em cima da tabela `dept`, para termos diferentes tipos de histórico. Vamos verificar:

```
SQL> select * from tab_hist_dept;

    DEPTNO DT_HISTO DS_HISTORICO
---------- -------- ----------------------------------------
        10 09/09/11 O departamento ACCOUNTING foi atualizado.
        20 09/09/11 O departamento RESEARCH foi atualizado.
        30 09/09/11 O departamento SALES foi atualizado.
        40 09/09/11 O departamento OPERATIONS foi atualizado.
        50 09/09/11 O departamento TI foi inserido.
        50 09/09/11 O departamento TI foi excluído.
```

```
6 linhas selecionadas.

SQL>
```

Através dos predicados é possível determinar qual ação disparou o `trigger` e, com isso, fazer com que o programa tome decisões e caminhos diferentes. Devemos observar algumas restrições e características com relação à criação e manipulação dos `triggers` que diferem de um bloco PL/SQL, sendo este anônimo ou não. Dentro de um `trigger` não podemos utilizar `commit` ou `rollback`, a menos que estejamos utilizando `pragma`. Seu uso será discutido mais adiante. Fora esta exceção, não é permitida a utilização destes comandos dentro do código dos `triggers`. Dessa forma, para validar as alterações realizadas pelo `trigger` é necessária uma efetivação externa, geralmente vindo do programa causador do disparo do seu disparo. Temos as seguintes premissas:

- Não são permitidos comandos de `ddl` dentro de `triggers`;
- Não são permitidos comandos de controle da transação, como `rollback`, `commit`, `savepoint` etc. (exceto com o uso de `pragma`).

Outro ponto que deve ser observado é com relação à criação de um `trigger`. Quando o criamos, o Oracle faz validações igualmente a quando criamos `procedures`, `functions` ou `packages`. Se este `trigger` apresentar algum erro de sintaxe, objetos não existentes, erros de referência ou qualquer outro problema que impossibilite seu uso, o Oracle permite a criação, mas o deixa inválido no banco de dados.

Desta forma, o `trigger` não fica apto para o uso e não será disparado mesmo que as ações satisfaçam suas definições. Neste caso, ele deve ser analisado, corrigido e recriado, novamente. Quando isso acontecer, não é necessário excluí-lo, basta recriá-lo. Basta usar o comando `replace` no momento em que estiver recriando. Na sequência, mostro um exemplo:

```
SQL> create or replace trigger tig_hist_cargo_emp
  2    after update of job on emp
  3    referencing old as v
```

```
 4                  new as n
 5     for each row
 6   begin
 7     insert into tab_hist_cargo_emp ( empno
 8                                     ,job_anterior
 9                                     ,job_atual
10                                     ,dt_alteracao_cargo
11                                     ,ds_historico)
12                         values ( :n.empno
13                                     ,:v.job
14                                     ,:n.job
15                                     ,sysdate
16                                     ,'O empregado '||:n.ename||
17                                      ' passou para o cargo '||:n.job||
18                                      ' em carater de promoção.');
19
20   end;
21   /

Gatilho criado.

SQL>
```

Caso seja necessário compilar um `trigger` pelo fato de ele estar inválido, use o comando `alter trigger`. Todavia, não é comum, pois ao criá-lo ou recriá-lo o Oracle já faz esta validação, a menos que ele tenha sido invalidado via dependência de algum outro objeto.

```
SQL> alter trigger tig_hist_cargo_emp compile;

Gatilho alterado.

SQL>
```

Também podemos ativar ou desativar um `trigger`. Quando o desativamos, ele permanece criado no banco de dados, mas fica inativo. Desta forma, ele não dispara caso a tabela de referência seja manipulada. Para ativar ou desativar um `trigger`, também utilizamos o comando `alter table`.

```
SQL> alter trigger tig_hist_cargo_emp disable -- desabilita o
                                                  trigger
  2  /

Gatilho alterado.

SQL> alter trigger tig_hist_cargo_emp enable -- habilita o
                                                 trigger
  2  /

Gatilho alterado.

SQL>
```

Também é possível habilitar ou desabilitar todos os `triggers` associados a uma tabela com um comando apenas.

```
SQL> alter table emp disable all triggers;

Tabela alterada.

SQL> alter table emp enable all triggers;

Tabela alterada.

SQL>
```

Para eliminar um `trigger`, utilizamos o comando `frop trigger`. Esse comando faz com que ele seja eliminado definitivamente.

```
SQL> drop trigger tig_hist_cargo_emp;

Gatilho eliminado.

SQL>
```

Para se criar um `trigger`, o usuário deve possuir os privilégios de criação: `create trigger` (ou `create any trigger`) e `alter table`. Este último está relacionado à tabela base para o `trigger`. Quando o usuário tem privilégios para criação de tabelas, implicitamente, já terá o poder

para alterá-las ou excluí-las. Veja um exemplo de concessão de privilégios para `triggers`.

> **Nota:** `create any trigger` permite que o usuário crie `triggers` em qualquer esquema, exceto `sys`. Isso não é recomendado para a criação deles em tabelas de dicionário de dados.

```
SQL> grant create trigger to tsql;

Concessão bem-sucedida.

SQL> grant create any trigger to tsql;

Concessão bem-sucedida.

SQL>
```

Para recuperar a definição e o código de um `trigger`, use as `views` `user_triggers`, `all_trigger` ou `dba_triggers`.

```
SQL> select trigger_body
  2  from user_triggers where trigger_name = 'TIG_PC_COM_SAL_EMP';

TRIGGER_BODY
--------------------------------------------------------------------
begin
  :n.pc_com_sal := nvl(:n.comm,0) * 100 / :n.sal;
end;


SQL>
```

No geral, os `triggers` de banco de dados podem servir, por exemplo, para controle, segurança e auditoria de informações de tabelas em um banco de dados, para a realização de backups ou replicação de dados, objetivando sempre a automação de algum processo.

Sequência de disparo de um trigger

Uma tabela não está limitada a apenas um `trigger`. Podemos ter vários associados a uma única tabela. Quando isso acontece, temos que nos preocupar com alguns aspectos, por exemplo, se a ordem de execução de um determinado `trigger` influencia em uma execução de um outro. Para isso, é preciso entender a ordem de execução dos `triggers`, que depende do tipo e do momento em que será disparado. Olhando o desenho a seguir, fica mais clara a ordem de execução, dependendo do tipo e do momento em que são disparados.

Imagine o seguinte cenário: temos dois `triggers`, um de tabela (chamado de `tgrtab`) e outro de linha (chamado de `tgrlin`). O `trigger` de tabela dispara tanto no momento `before` quanto no momento `after`. O `trigger` de linha segue a mesma lógica. Ambos são disparados na ação `update`. Vamos ver como fica a ordem de disparo quando executamos um `update` nos registros.

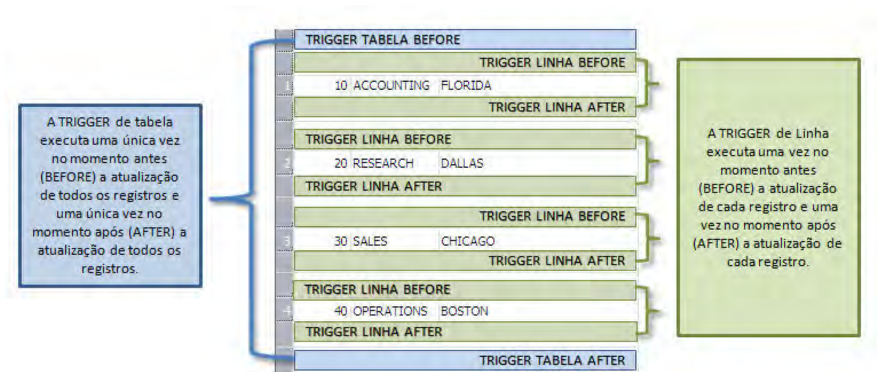

Fig. 19.1: Esquema mostrando a sequência de disparo das triggers de linha de tabela

Note que o `trigger tgrtab` executa uma única vez antes na atualização de todos os registros e uma única vez no final quando todos os registros já foram atualizados. Já o `trigger tgrlin` executa após a primeira execução do `trigger tgrtab` e antes e depois da atualização de cada registro.

Outro ponto a ser observado é que não há impedimento algum em termos dois `triggers` exatamente iguais, que sejam do mesmo tipo e que disparem no mesmo momento. Entretanto, para estes casos, não é possível determinar qual a ordem em que cada um será executado. A Oracle não garante a ordem da execução quando há `triggers` contendo características de disparo iguais. Caso sejam dependentes um do outro, o aconselhável é juntá-los, e que tais dependências sejam tratadas no código.

19.4 Mutante table

Existe uma limitação no uso dos *triggers de linha*, quando selecionamos dados de uma tabela, no evento `after`, que seja a base do `trigger`, ou seja, estamos tentando selecionar os dados de uma tabela que está sendo alterada e ao mesmo tempo é a causadora do disparo. Contudo, quando temps este cenário, mas o evento é o `before`, este problema não acontece.

Quando um `trigger` é disparado, isso indica que os dados de uma tabela estão sendo alterados. Neste momento podemos concluir que, enquanto as alterações não forem confirmadas, os dados que estão sendo manipulados na transação em questão não estão consistentes. Apenas quando forem confirmados, através de um `rollback` ou de um `commit`, os dados estarão íntegros.

Seguindo esta lógica, no momento antes da efetivação dos dados alterados, não é possível executar a manipulação deles,por exemplo, através de um comando `select`, mesmo se nossa intenção seja buscar apenas os registros que não estão sofrendo tais alterações. Isso confirma a limitação no nível de tabela e não no nível de dados. Quando tentamos realizar esta operação, um erro chamado `mutante table` é gerado. Veja o resumo das regras a seguir:

- O erro de `mutante table` não acontece quando a `trigger` é de linha e o momento de disparo for `before`, isso tanto para os comandos `delete`, `update`, `insert` ou `select`. Já no evento `after`, o erro de `mutante table` acontece para todos estes comandos (com exceção do uso de transações autônomas);

- Para as `triggers` de tabela, não precisamos nos preocupar com o erro de `mutante table`, pois independente de o evento ser `before`

ou `after`, e de qual seja o comando, `select`, `delete`, `update` ou `insert`, esse tipo de problema não ocorre.

Agora veremos o exemplo na sequência:

```
SQL> create or replace trigger tig_emp_pragma
  2    after update on emp
  3    for each row
  4  declare
  5    wcont_registro number default 0;
  6  begin
  7    --
  8    select count(*) into wcont_registro from emp;
  9    --
 10    dbms_output.put_line('Quantidade de registros na tabela
        EMP: '||wcont_registro);
 11    --
 12  end;
 13  /

Gatilho criado.

SQL>
```

Criamos um `trigger` do tipo linha, com o qual a cada atualização realizada nos registros da tabela `emp` é executado um `select count` para verificar e mostrar a quantidade de registros desta mesma tabela. Agora vamos executar o comando a seguir:

```
SQL> update emp
  2  set sal = sal
  3  /
update emp
       *
ERRO na linha 1:
ORA-04091: a tabela TSQL.EMP é mutante; talvez o gatilho/função
não possa localizá-la
ORA-06512: em "TSQL.TIG_EMP_PRAGMA", line 5
ORA-04088: erro durante a execução do gatilho
```

```
'TSQL.TIG_EMP_PRAGMA'


SQL>
```

Como era previsto, o erro ocorreu após tentarmos realizar a alteração. Agora vamos recriar o `trigger` como sendo um de tabela e executar a atualização novamente.

```
SQL> create or replace trigger tig_emp_pragma
  2    after update on emp
  3  declare
  4    wcont_registro number default 0;
  5  begin
  6    --
  7    select count(*) into wcont_registro from emp;
  8    --
  9    dbms_output.put_line('Quantidade de registros na tabela
      EMP: '||wcont_registro);
 10    --
 11  end;
 12  /

Gatilho criado.

SQL>

SQL> update emp
  2  set sal = sal;
Quantidade de registros na tabela EMP: 14
```

Note que quando utilizamos um `trigger` do tipo tabela não temos este problema. Mas aí vem a pergunta: e se tivermos esta necessidade e nos depararmos com este problema? O que fazer? Nestes casos, podemos utilizar o recurso de transações autônomas, informando-o na declaração do `trigger`. Sua função é fazer com que o Oracle abra uma nova sessão somente para atender o `trigger`, ou seja, neste momento existirão pelo menos duas transações em aberto. Uma é referente à transação que está realizando as alterações na

tabela em questão, e outra referente ao disparo do `trigger`. Veja o desenho a seguir onde tentamos ilustramos como o Oracle trata esta situação.

TRANSAÇÃO "A"

```
SQL> update emp
  2  set sal = sa
  3  /

Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14

14 linhas atualizadas.

SQL>
```

Uma nova Transação é aberta

TRANSAÇÃO "B"

```
SQL> create or replace trigger tig_emp_pragma
  2    before update on emp
  3    for each row
  4  declare
  5    --
  6    pragma autonomous_transaction;
  7    --
  8    wcont_registro number default 0;
  9  begin
 10    --
 11    select count(*) into wcont_registro from emp;
 12    --
 13    if nvl(:old.comm,0) < 300 then
 14      :new.comm := :new.sal * 10 / 100;
 15    end if;
 16    --
 17    dbms_output.put_line('Quantidade de registros na
tabela EMP: '||wcont_registro);
 18    --
 19    commit;
 20    --
 21  end;
 22  /

Gatilho criado.

SQL>
```

Fig. 19.2: Abrindo transações através de pragma autonomous_transaction

Contudo, conforme já explicado no capítulo anterior sobre transações autônomas, temos que nos atentar para a seguinte situação. Sabemos que, quando é aberta uma transação no Oracle, ela monta uma imagem dos objetos e dados disponíveis para o usuário em questão. Com relação aos dados, isso permite visualizar somente aquilo que está efetivado no banco de dados, ou seja, dados consistentes e íntegros. Isso quer dizer que dados pendentes de efetivação não poderão ser visualizados, isso é, dados excluídos, inseridos ou atualizados que ainda não tenham sido `commitados`.

Logo, quando isso acontece, no caso de um `trigger`, não conseguimos visualizar as alterações que estão sendo realizadas, pelo menos não as que fogem do escopo `old` e `new`. Portanto, quando temos um `trigger` com transação autônoma, conseguimos manipular os dados de `old` e `new`, mas

se realizarmos um comando `select` a fim de obter estes dados, não conseguiremos enxergá-los. Apenas para ressaltar, isto se dá ao fato de que dentro do `trigger` estamos em outro escopo, em uma nova transação.

Outro ponto muito importante é que quando utilizamos `trigger` com transação autônoma precisamos finalizá-la através de um `commit` ou de um `rollback`. Como a `autonomous_transaction` faz com que uma nova transação seja aberta, temos que finalizá-la. Caso contrário, ela ficaria pendente e isso poderia gerar um erro. Vamos verificar na prática:

```
SQL> create or replace trigger tig_emp_pragma
  2    before update on emp
  3    for each row
  4  declare
  5    --
  6    pragma autonomous_transaction;
  7    --
  8    wcont_registro number default 0;
  9  begin
 10    --
 11    select count(*) into wcont_registro from emp;
 12    --
 13    if nvl(:old.comm,0) < 300 then
 14      :new.comm := :new.sal * 10 / 100;
 15    end if;
 16    --
 17    dbms_output.put_line('Quantidade de registros na tabela
        EMP: '||wcont_registro);
 18    --
 19    commit;
 20    --
 21  end;
 22  /

Gatilho criado.

SQL>
```

Uma observação: para este exemplo alteramos o `trigger` de `after` para `before` para que pudéssemos alterar o valor de `new`.

Note que na área de declaração colocamos a seguinte linha: `pragma autonomous_transaction;`. Isso faz com que uma nova transação seja aberta para a execução do `trigger`. Agora vamos executar uma alteração na tabela `emp` e veremos como ele se comporta.

```
SQL> update emp
  2  set sal = sal
  3  /
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14
Quantidade de registros na tabela EMP: 14

14 linhas atualizadas.

SQL>
```

A execução não nos gerou erro. Vamos verificar a tabela `emp`. Antes da execução da atualização e disparo do `trigger` os dados estavam assim:

```
SQL> select empno, ename, sal, comm from emp;

     EMPNO ENAME             SAL       COMM
---------- ---------- ---------- ----------
      7369 SMITH             968
      7499 ALLEN            1600        306
      7521 WARD             1250        510
      7566 JONES         3599,75
      7654 MARTIN           1250       1428
```

```
      7698 BLAKE            2850
      7782 CLARK            2450
      7788 SCOTT            3630
      7839 KING             5000
      7844 TURNER           1500           0
      7876 ADAMS            1331
      7900 JAMES             950
      7902 FORD             3630
      7934 MILLER           1300

14 linhas selecionadas.

SQL>
```

Após a execução da atualização e disparo do `trigger` os dados ficaram assim:

```
SQL> select empno, ename, sal, comm from emp;

     EMPNO ENAME             SAL       COMM
---------- ---------- ---------- ----------
      7369 SMITH             968       96,8
      7499 ALLEN            1600        306
      7521 WARD             1250        510
      7566 JONES         3599,75     359,98
      7654 MARTIN           1250       1428
      7698 BLAKE            2850        285
      7782 CLARK            2450        245
      7788 SCOTT            3630        363
      7839 KING             5000        500
      7844 TURNER           1500        150
      7876 ADAMS            1331      133,1
      7900 JAMES             950         95
      7902 FORD             3630        363
      7934 MILLER           1300        130

14 linhas selecionadas.

SQL>
```

Agora vamos cadastrar um novo empregado através de um `insert` para vermos bem o controle de transação sendo consistido. Para isso, alteramos novamente nosso `trigger`, incluindo nas condições de disparo o comando `insert`.

```
SQL> create or replace trigger tig_emp_pragma
  2    before update or insert on emp
  3    for each row
  4  declare
  5    --
  6    pragma autonomous_transaction;
  7    --
  8    wcont_registro number default 0;
  9  begin
 10    --
 11    select count(*) into wcont_registro from emp;
 12    --
 13    if nvl(:old.comm,0) < 300 then
 14      :new.comm := :new.sal * 10 / 100;
 15    end if;
 16    --
 17    dbms_output.put_line('Quantidade de registros na tabela
       EMP: '||wcont_registro);
 18    --
 19    commit;
 20    --
 21  end;
 22  /

Gatilho criado.

SQL>
```

Executando o `insert`.

```
SQL> insert into emp
  2     (empno, ename, job, mgr, hiredate, sal, comm, deptno,
         pc_com_sal)
  3   values
```

```
  4     (7935, 'PAUL', 'SALESMAN', 7698, to_date('15-MAR-1980',
        'dd-mon-rrrr'),
   1000, 100, 30, null);

Quantidade de registros na tabela EMP: 14

1 linha criada.

SQL>
```

Note que a mensagem vinda do `trigger` nos informou que temos 14 registros, ou seja, ele não está contando com este que acabamos de inserir, que seria o registro 15. Mesmo que alteremos o `triiger` para que dispare no momento `after`, ele não vai enxergar o novo registro, pois o `insert` está acontecendo numa transação e a contagem do registro, feita pelo `trigger`, acontece em outra.

Se mesmo com o uso de transações autônomas não for possível resolver problemas de `mutante table`, talvez mesclar o uso de `triggers` de linha com os de tabela possa ser uma saída.

Considerações sobre triggers

- Um `trigger` tanto de linha quanto de tabela pode agir de forma recursiva. O Oracle garante até 50 níveis de recursividade para estes dois tipos de objeto;
- Um `trigger` não pode contemplar eventos diferentes de execução (`before` ou `after`) em um mesmo objeto. Devem ser criados dois `triggers` distintos;
- Ao excluir uma tabela, `triggers` associadas a ela também são excluídos automaticamente.

19.5 Trigger de sistema

Trigger de sistema são objetos criados em nível de sistema e não de tabelas. Quando falamos em sistemas nos referimos ao banco de dados, mais preci-

samente a ações que ocorrem nele. Seu uso é mais comum pelos DBAs (administradores de banco de dados) e é uma ferramenta poderosa quando utilizada com criatividade.

Os triggers de sistema disparam conforme um evento que acontece no banco de dados. Veja a seguir a lista de alguns eventos:

- `startup`: quando o banco de dados é aberto;
- `shutdown`: antes de o banco de dados iniciar o `shutdown`(fechamento). Se for `shutdown abort` este evento não é disparado;
- `servererror`: quando um erro ocorre. Com exceção dos erros: ORA-1034, ORA-1403, ORA-1422, ORA-1423 e ORA-4030;
- `after logon`: depois de uma conexão ser completada no banco de dados;
- `before logoff`: quando o usuário desconecta do banco de dados;
- `before create` / `after create`: quando um objeto é criado no banco de dados (comando `create`), exceto o comando `create database`;
- `before alter` / `after alter`: quando um objeto é alterado no banco de dados (comando `alter`), com exceção do comando `alter database`;
- `before drop` / `after drop`: quando um objeto é eliminado do banco de dados (comando `drop`);
- `before analyze` / `after analyze`: quando o comando `analyze` é executado. Este comando é utilizado para gerar estatísticas relacionadas ao desempenho de comandos SQL e processos do banco de dados;
- `before commit` / `after commit`: quando um `commit` é executado;

- `before ddl` / `after ddl`: quando um comando `ddl` é executado, com exceção dos comandos `alter database`, `create controlfile`, `create database` e `ddl` executados a partir de interface PL/SQL;
- `before grant` / `after grant`: quando o comando `grant` é executado;
- `before rename` / `after rename`: quando o `rename` é executado;
- `before revoke` / `after revoke`: quando o comando `revoke` é executado;
- `before truncate` / `after truncate`: quando o `truncate` é executado.

Além de contar com os eventos para determinar quando o *trigger de sistema* deve disparar, também temos a disposição os atributos de evento que nos dão informações sobre o banco de dados, transações e sobre as operações que disparam o `trigger`. Veja a lista a seguir.

- `ora_client_ip_address` (tipo: `varchar2`): retorna o IP da máquina cliente no evento de LOGON, se o protocolo for TCP/IP;
- `ora_database_name` (tipo: `varchar2(50)`): nome do banco de dados;
- `ora_dict_obj_name` (tipo: `varchar(30)`): nome do objeto que sofreu o evento `ddl`;
- `ora_dict_obj_name_list(name_list out ora_name_list_t)` (tipo: `binary_integer`): retorna lista de objetos afetados no evento;
- `ora_dict_obj_ownerR` (tipo: `varchar(30)`): proprietário (usuário/schema) do objeto que sofreu o evento `ddl`;
- `ora_dict_obj_owner_list(OWNER_LIST OUT ora_name_list_t)` (tipo: `binary_integer`): retorna lista de proprietários (usuário/schemas) dos objetos afetados no evento;

- ora_dict_obj_type (tipo: varchar(20)): tipo de objeto que sofreu o evento ddl;
- ora_grantee(user_list out ora_name_list_t) (tipo: binary_integer): retorna lista de usuários ou roles que receberam grant;
- ora_is_alter_column(column_name in varchar2) (tipo: boolean): retorna true se a coluna especificada foi alterada;
- ora_is_drop_column(column_name in varchar2) (tipo: boolean): retorna true se a coluna especificada foi eliminada;
- ora_login_user (tipo: varchar2(30)): nome do usuário que acabou de conectar em uma sessão do banco de dados;
- ora_privileges(privilege_list out ora_name_list_t) (tipo: binary_integer): retorna lista de privilégios dados ou revogados;
- ora_revokee(user_list out ora_name_list_t) (tipo: binary_integer): retorna lista dos que perderam privilégios;
- ora_sysevent (tipo: varchar2(20)): retorna o nome do evento que foi disparado. O nome é o mesmo usado na definição do trigger;
- ora_with_grant_option (tipo: boolean): retorna true se o grant tem with grant option.

Na sequência, alguns exemplos:

```
SQL> create table hist_usuario
  2  (
  3     nm_usuario   varchar2(50)
  4    ,dt_historico date
  5    ,ds_historico varchar2(4000)
  6  )
  7  /
```

```
Tabela criada.

SQL>
```

Criamos uma tabela para guardar os históricos gerados pelos `triggers`. No exemplo a seguir, temos um que registra quando o usuário conecta no banco de dados.

```
SQL> create or replace trigger tgr_hist_conexao_usuario
  2     after logon on database
  3   begin
  4     insert into hist_usuario values
  5         (ORA_LOGIN_USER,sysdate,'Conexão com o banco de dados
          '||ORA_DATABASE_NAME);
  6   end;
  7   /

Gatilho criado.

SQL>
```

Conectando ao banco de dados:

```
SQL> disconnect
Desconectado de Oracle Database 10g Express Edition Release
  10.2.0.1.0 - Production
SQL>
SQL> conn tsql/tsql@xe;
Conectado.
SQL>
SQL>
```

> **Nota:** Como já estávamos conectados ao banco de dados, primeiramente, desconectamos e refizemos a conexão logo após.

Verificando histórico:

```
SQL> select * from hist_usuario;

NM_USUARIO           DT_HISTO DS_HISTORICO
-------------------- -------- -----------------------------------
TSQL                 04/10/11 Conexão com o banco de dados XE

SQL>
```

Como pode ser visto, o `trigger` gravou o histórico com as informações pertinentes ao usuário que conectou no banco de dados.

O próximo registra alterações de `dml` realizados pelo usuário, mais precisamente a criação de objetos no sistema.

```
SQL> create or replace trigger tgr_hist_criatab_usuario
  2    after create on database
  3  begin
  4    insert into hist_usuario values
  5      (ORA_LOGIN_USER,sysdate,
          'O Objeto '||ORA_DICT_OBJ_NAME||' foi criado
  no banco de dados.');
  6  end;
  7  /

Gatilho criado.

SQL>
```

Criando o objeto tabela `teste_trigger`:

```
SQL> create table teste_trigger
  2  (
  3     id_campo    varchar2(50)
  4    ,nm_campo    date
  5    ,ds_campo    varchar2(4000)
  6  )
  7  /

Tabela criada.

SQL>
```

Verificando tabela de histórico:

```
SQL> select * from hist_usuario;

NM_USUARIO      DT_HISTO DS_HISTORICO
--------------- -------- ----------------------------------------
TSQL            04/10/11 O objeto TESTE_TRIGGER foi criado no
                         banco de dados.


SQL>
```

19.6 Trigger de view

Aprendemos anteriormente que um `trigger` pode estar associado a uma tabela ou ao avento de sistema. Contudo, quando estamos falando de views, que são bem semelhantes a tabelas, pelo menos na forma de apresentação dos dados, existe a possibilidade de criarmos um `trigger` e o associarmos também a este tipo de objeto.

Em regras gerais, só conseguimos criar um `trigger` para uma determinada view que seja composta de apenas uma tabela. No entanto, se criarmos um `trigger` usando a cláusula `instead of`, podemos trabalhar com `triggers` associadas a views compostas de uma ou mais tabelas. O `trigger` usado com `instead of` tem sua definição um pouco diferente, pois esta cláusula substitui os eventos `before` e `after`. Outro ponto que deve ser observado é que o uso desta cláusula só pode ser empregado com *triggers de linha*.

A questão de poder ou não criar um `trigger` para uma view está relacionada às operações que podemos efetuar em cima desta view. Esquecendo um pouco os `triggers`, sabemos que operações de DML (`insert`, `delete` ou `update`) só podem ocorrer quando a view for de apenas uma tabela. Portanto, a utilização de `triggers` em views segue os mesmos princípios. A única diferença é que quando utilizamos `triggers` para realizar comando DML em views podemos utilizá-los com a opção `instead of`, o que torna possível tratar de forma diferente as views que possuam mais de uma tabela na sua composição.

A seguir, um exemplo de uma view composta pelas tabelas `emp` e `dept` onde nós vamos inserir dados. Criando `view` de empregados por departamento:

```
SQL> create view emp_dept_v as
  2  select e.empno, e.ename, e.job, e.sal, d.dname
  3  from   emp  e
  4        ,dept d
  5  where  e.deptno = d.deptno
  6  ;

View criada.

SQL>
```

Selecionando dados:

```
SQL> select * from emp_dept_v;

     EMPNO ENAME      JOB              SAL DNAME
---------- ---------- --------- ---------- --------------
      7369 SMITH      CLERK            968 RESEARCH
      7499 ALLEN      SALESMAN        1600 SALES
      7521 WARD       SALESMAN        1250 SALES
      7566 JONES      MANAGER      3599,75 RESEARCH
      7654 MARTIN     SALESMAN        1250 SALES
      7698 BLAKE      MANAGER         2850 SALES
      7782 CLARK      MANAGER         2450 ACCOUNTING
      7788 SCOTT      ANALYST         3630 RESEARCH
      7839 KING       PRESIDENT       5000 ACCOUNTING
      7844 TURNER     SALESMAN        1500 SALES
      7876 ADAMS      CLERK           1331 RESEARCH
      7900 JAMES      CLERK            950 SALES
      7902 FORD       ANALYST         3630 RESEARCH
      7934 MILLER     CLERK           1300 ACCOUNTING
      7935 PAUL       SALESMAN        1000 SALES

15 linhas selecionadas.

SQL>
```

Criando `trigger` com `instead of` para manipulação dos dados:

```
SQL> create or replace trigger trg_emp_dept_v
  2   instead of
  3    insert or delete or update
  4   on emp_dept_v
  5  referencing new as new old as old
  6  declare
  7    cursor c1(pdeptno dept.deptno%type) is
  8      select deptno, dname
  9      from   dept
 10      where  deptno = pdeptno;
 11    --
 12    cursor c2(pempno emp.empno%type) is
 13      select empno, ename
 14      from   emp
 15      where  empno = pempno;
 16    --
 17    wdeptno dept.deptno%type;
 18    wdname  dept.dname%type;
 19    wempno  emp.empno%type;
 20    wename  emp.ename%type;
 21  begin
 22    --
 23    if inserting then
 24      --
 25      -- verifica se existe departamento.
 26      open c1 (:new.deptno);
 27      fetch c1 into wdeptno, wdname;
 28      --
 29      if c1%notfound then
 30        insert into dept (deptno, dname, loc)
          values (:new.deptno, :new.dname, null);
 31        --
 32        dbms_output.put_line(
          'Departamento cadastrado com sucesso.');
 33      else
 34        dbms_output.put_line(
          'Departamento existente: '||wdeptno||' -
```

```
               '||wdname||'.');
35          end if;
36          --
37          -- verifica se existe empregado.
38          open c2 (:new.empno);
39          fetch c2 into wempno, wename;
40          --
41          if c2%notfound then
42            insert into emp (empno, ename, job, sal, mgr, deptno)
43               values (:new.empno, :new.ename, :new.job, :new.sal,
                       :new.mgr, :new.deptno);
44            --
45            dbms_output.put_line(
             'Funcionário cadastrado com sucesso.');
46          else
47            dbms_output.put_line(
             'Funcionário existente: '||wempno||' -
             '||wename||'.');
48          end if;
49          --
50        end if;
51        --
52      end;
53      /

Gatilho criado.

SQL>
```

Na primeira parte da criação do `trigger` vamos tratar a ação de `insert` na view `emp_dept_v`. A seguir temos várias execuções onde testamos, primeiramente, o comando `insert`.

```
SQL>  insert into emp_dept_v (empno, ename, job, sal, mgr,
                              deptno, dname)
  2                    values (8000, 'BRUCE', 'MANAGER', 1000,
                              7839, 20, 'RESEARCH');
Departamento existente: 20 - RESEARCH.
Funcionário cadastrado com sucesso.
```

```
1 linha criada.

SQL>

SQL> insert into emp_dept_v (empno, ename, job, sal, mgr, deptno,
                             dname)
  2                   values (8000, 'BRUCE', 'MANAGER', 1000,
                             7839, 20, 'RESEARCH');
Departamento existente: 20 - RESEARCH.
Funcionário existente: 8000 - BRUCE.

1 linha criada.
SQL>

SQL> insert into emp_dept_v (empno, ename, job, sal, mgr, deptno,
                             dname)
  2                   values (8001, 'SILVESTER', 'MANAGER', 1000,
                             7839, 80, 'S&E');
Departamento cadastrado com sucesso.
Funcionário cadastrado com sucesso.

1 linha criada.
SQL>

SQL> insert into emp_dept_v (empno, ename, job, sal, mgr, deptno,
                             dname)
  2                   values (8001, 'SILVESTER', 'MANAGER', 1000,
                             7839, 90, 'FINANCIAL');
Departamento cadastrado com sucesso.
Funcionário existente: 8001 - SILVESTER.

1 linha criada.

SQL>

SQL> insert into emp_dept_v (empno, ename, job, sal, mgr, deptno,
                             dname)
  2                   values (8002, 'WILL', 'MANAGER', 1000, 7839,
                             90, 'FINANCIAL');
```

```
Departamento existente: 90 - FINANCIAL.
Funcionário cadastrado com sucesso.

1 linha criada.

SQL>
```

Agora vamos para a segunda parte do `trigger`, onde vamos tratar a ação `update`. Criação do `trigger`:

```
SQL> create or replace trigger trg_emp_dept_v
  2    instead of
  3      insert or delete or update
  4      on emp_dept_v
  5      referencing new as new old as old
  6  declare
  7    cursor c1(pdeptno dept.deptno%type) is
  8      select deptno, dname
  9      from   dept
 10      where  deptno = pdeptno;
 11    --
 12    cursor c2(pempno emp.empno%type) is
 13      select empno, ename
 14      from   emp
 15      where  empno = pempno;
 16    --
 17    wdeptno dept.deptno%type;
 18    wdname  dept.dname%type;
 19    wempno  emp.empno%type;
 20    wename  emp.ename%type;
 21  begin
 22    --
 23    ----------------------------------------------------------
 24    if inserting then
 25      --
 26      -- verifica se existe departamento.
 27      open c1 (:new.deptno);
 28      fetch c1 into wdeptno, wdname;
 29      --
```

```
30      if c1%notfound then
31        insert into dept (deptno, dname, loc)
          values (:new.deptno, :new.dname, null);
32        --
33        dbms_output.put_line('Departamento cadastrado com
                                sucesso.');
34      else
35        dbms_output.put_line('Departamento existente:
                               '||wdeptno||' - '||wdname||'.');
36      end if;
37      --
38      -- verifica se existe empregado.
39      open c2 (:new.empno);
40      fetch c2 into wempno, wename;
41      --
42      if c2%notfound then
43        insert into emp (empno, ename, job, sal, mgr, deptno)
44          values (:new.empno, :new.ename, :new.job,
                    :new.sal,:new.mgr, :new.deptno);
45        --
46        dbms_output.put_line('Funcionário cadastrado com
                               sucesso.');
47      else
48        dbms_output.put_line('Funcionário existente:
                               '||wempno||' - '||wename||'.');
49      end if;
50      --
51    ----------------------------------------------------------
52    elsif updating then
53      --
54      -- verifica se existe departamento.
55      open c1 (:new.deptno);
56      fetch c1 into wdeptno, wdname;
57      --
58      if c1%notfound then
59        --
60        dbms_output.put_line('Departamento não existe.');
61        --
62      else
```

```
63        --
64        update dept
65        set    dname  = :new.dname
66        where  deptno = :new.deptno;
67        --
68        dbms_output.put_line('Departamento atualizado com
                                    sucesso: '||:new.dname||'.');
69        --
70      end if;
71      --
72      -- verifica se existe empregado.
73      open c2 (:new.empno);
74      fetch c2 into wempno, wename;
75      --
76      if c2%notfound then
77        --
78        dbms_output.put_line('Funcionário não cadastrado.');
79        --
80      else
81        update emp
82        set    ename  = :new.ename
83              ,job    = :new.job
84              ,sal    = :new.sal
85              ,mgr    = :new.mgr
86              ,deptno = :new.deptno
87        where  empno  = :new.empno;
88        --
89        dbms_output.put_line('Funcionário atualizado com
                                    sucesso: '||:new.ename||'.');
90        --
91      end if;
92      --
93    end if;
94    --
95  end;
96  /

Gatilho criado.
```

```
SQL>
```

Testando o `trigger`:

```
SQL> select * from emp_dept_v where  empno = 8000;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      8000 BRUCE      MANAGER         1000       7839

    DEPTNO DNAME
---------- ----------
        20 RESEARCH
SQL>

SQL>  update emp_dept_v
  2   set     ename  = 'BRUCEW'
  3          ,job    = 'ANALYST'
  4          ,sal    = 1500
  5          ,deptno = 20
  6          ,dname  = 'RESEARCH'
  7   where   empno = 8000;
Departamento atualizado com sucesso: RESEARCH.
Funcionário atualizado com sucesso: BRUCEW.

1 linha atualizada.

SQL> select * from emp_dept_v where  empno = 8000;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      8000 BRUCEW     ANALYST         1500       7839

    DEPTNO DNAME
---------- ----------
        20 RESEARCH
SQL>

SQL> select * from emp_dept_v where deptno = 20;
```

```
     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            968       7902
      7566 JONES      MANAGER      3599,75       7839
      7788 SCOTT      ANALYST         3630       7566
      7876 ADAMS      CLERK           1331       7788
      7902 FORD       ANALYST         3630       7566
      8000 BRUCEW     ANALYST         1500       7839

SQL> select * from emp_dept_v where deptno = 20;

    DEPTNO DNAME
---------- -----------
        20 RESEARCH
        20 RESEARCH
        20 RESEARCH
        20 RESEARCH
        20 RESEARCH
        20 RESEARCH

6 linhas selecionadas.
SQL>

SQL> update emp_dept_v
  2  set    dname  = 'PESQUISA'
  3  where  deptno = 20;
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: SMITH.
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: JONES.
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: SCOTT.
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: ADAMS.
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: FORD.
Departamento atualizado com sucesso: PESQUISA.
Funcionário atualizado com sucesso: BRUCEW.
```

```
6 linhas atualizadas.

SQL> select * from emp_dept_v where deptno = 20;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            968       7902
      7566 JONES      MANAGER      3599,75       7839
      7788 SCOTT      ANALYST         3630       7566
      7876 ADAMS      CLERK           1331       7788
      7902 FORD       ANALYST         3630       7566
      8000 BRUCEW     ANALYST         1500       7839

    DEPTNO DNAME
---------- -----------
        20 PESQUISA
        20 PESQUISA
        20 PESQUISA
        20 PESQUISA
        20 PESQUISA
        20 PESQUISA

6 linhas selecionadas.

SQL>
```

A exclusão finaliza a última parte do nosso `trigger` de view.

```
SQL> ed
Gravou arquivo afiedt.buf

  1  create or replace trigger trg_emp_dept_v
  2    instead of
  3      insert or delete or update
  4      on emp_dept_v
  5      referencing new as new old as old
  6  declare
  7    cursor c1(pdeptno dept.deptno%type) is
  8      select deptno, dname
  9      from   dept
```

```
10        where  deptno = pdeptno;
11      --
12      cursor c2(pempno emp.empno%type) is
13        select empno, ename
14        from   emp
15        where  empno = pempno;
16      --
17      cursor c3(pdeptno dept.deptno%type) is
18        select count(*)
19        from   emp
20        where  deptno = pdeptno;
21      --
22      wdeptno  dept.deptno%type;
23      wdname   dept.dname%type;
24      wempno   emp.empno%type;
25      wename   emp.ename%type;
26      qt_empno number default 0;
27    begin
28      --
29      ----------------------------------------------------------
30      if inserting then
31        --
32        -- verifica se existe departamento.
33        open c1 (:new.deptno);
34        fetch c1 into wdeptno, wdname;
35        --
36        if c1%notfound then
37          insert into dept (deptno, dname, loc)
             values (:new.deptno, :new.dname, null);
38          --
39          dbms_output.put_line(
             'Departamento cadastrado com sucesso.');
40        else
41          dbms_output.put_line(
             'Departamento existente: '||wdeptno||' -
                                      '||wdname||'.');
42        end if;
43        --
44        -- verifica se existe empregado.
```

```
45      open c2 (:new.empno);
46      fetch c2 into wempno, wename;
47      --
48      if c2%notfound then
49        insert into emp (empno, ename, job, sal, mgr, deptno)
50          values (:new.empno, :new.ename, :new.job, :new.sal,
                  :new.mgr, :new.deptno);
51        --
52        dbms_output.put_line(
         'Funcionário cadastrado com sucesso.');
53      else
54        dbms_output.put_line(
        'Funcionário existente: '||wempno||' - '||wename||'.');
55      end if;
56      --
57    ----------------------------------------------------------
58    elsif updating then
59      --
60      -- verifica se existe departamento.
61      open c1 (:new.deptno);
62      fetch c1 into wdeptno, wdname;
63      --
64      if c1%notfound then
65        --
66        dbms_output.put_line('Departamento não existe.');
67        --
68      else
69        --
70        update dept
71        set     dname   = :new.dname
72        where   deptno = :new.deptno;
73        --
74        dbms_output.put_line('Departamento atualizado com
                            sucesso: '||:new.dname||'.');
75        --
76      end if;
77      --
78      -- verifica se existe empregado.
79      open c2 (:new.empno);
```

```
80      fetch c2 into wempno, wename;
81      --
82      if c2%notfound then
83        --
84        dbms_output.put_line('Funcionário não cadastrado.');
85        --
86      else
87        update emp
88        set    ename  = :new.ename
89              ,job    = :new.job
90              ,sal    = :new.sal
91              ,mgr    = :new.mgr
92              ,deptno = :new.deptno
93        where  empno  = :new.empno;
94        --
95        dbms_output.put_line('Funcionário atualizado com
                               sucesso: '||:new.ename||'.');
96        --
97      end if;
98      --
99    -----------------------------------------------------------
100   elsif deleting then
101     --
102     -- verifica se existe empregado.
103     open c2 (:old.empno);
104     fetch c2 into wempno, wename;
105     --
106     if c2%notfound then
107       --
108       dbms_output.put_line('Funcionário não cadastrado.');
109       --
110     else
111       delete from emp where empno = :old.empno;
112       --
113       dbms_output.put_line('Funcionário excluído com
                               sucesso: '||:old.ename||'.');
114       --
115     end if;
116     --
```

```
117       -- verifica se existe departamento.
118       open c1 (:old.deptno);
119       fetch c1 into wdeptno, wdname;
120       --
121       if c1%notfound then
122         --
123         dbms_output.put_line('Departamento não existe.');
124         --
125       else
126         --
127         -- verifica se existe departamento.
128         open c3 (:old.deptno);
129         fetch c3 into qt_empno;
130         --
131         if qt_empno = 0 then
132           --
133           delete from dept where deptno = :old.deptno;
134           --
135           dbms_output.put_line('Departamento excluído com
                                     sucesso: '||:old.dname||'.');
136           --
137         else
138           --
139           dbms_output.put_line('O departamento
                                     '||:old.dname||'
                                     não pode ser excluído.
Ainda existem
140           --
141         end if;
142         --
143       end if;
144       --
145     end if;
146     --
147* end;
SQL> /

Gatilho criado.
```

```
SQL>
```

Vamos testar a ação de exclusão:

```
SQL> select * from emp_dept_v;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            968       7902
      7499 ALLEN      SALESMAN        1600       7698
      7521 WARD       SALESMAN        1250       7698
      7566 JONES      MANAGER      3599,75       7839
      7654 MARTIN     SALESMAN        1250       7698
      7698 BLAKE      MANAGER         2850       7839
      7782 CLARK      MANAGER         2450       7839
      7788 SCOTT      ANALYST         3630       7566
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500       7698
      7876 ADAMS      CLERK           1331       7788
      7900 JAMES      CLERK            950       7698
      7902 FORD       ANALYST         3630       7566
      7934 MILLER     CLERK           1300       7782
      7935 PAUL       SALESMAN        1000       7698
      8001 SILVESTER  MANAGER         1000       7839
      8000 BRUCEW     ANALYST         1500       7839

    DEPTNO DNAME
---------- --------------
        20 PESQUISA
        30 SALES
        30 SALES
        20 PESQUISA
        30 SALES
        30 SALES
        10 ACCOUNTING
        20 PESQUISA
        10 ACCOUNTING
        30 SALES
        20 PESQUISA
        30 SALES
```

```
        20 PESQUISA
        10 ACCOUNTING
        30 SALES
        80 S&E
        20 PESQUISA

17 linhas selecionadas.

SQL>

SQL> delete from emp_dept_v where empno = 8000;
Funcionário excluído com sucesso: BRUCEW.
O departamento PESQUISA não pode ser excluído. Ainda existem
funcionários agregados a ele.

1 linha deletada.

SQL> select * from emp_dept_v;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- ---------- ---------- ----------
      7369 SMITH      CLERK            968       7902
      7499 ALLEN      SALESMAN        1600       7698
      7521 WARD       SALESMAN        1250       7698
      7566 JONES      MANAGER      3599,75       7839
      7654 MARTIN     SALESMAN        1250       7698
      7698 BLAKE      MANAGER         2850       7839
      7782 CLARK      MANAGER         2450       7839
      7788 SCOTT      ANALYST         3630       7566
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500       7698
      7876 ADAMS      CLERK           1331       7788
      7900 JAMES      CLERK            950       7698
      7902 FORD       ANALYST         3630       7566
      7934 MILLER     CLERK           1300       7782
      7935 PAUL       SALESMAN        1000       7698
      8001 SILVESTER  MANAGER         1000       7839

    DEPTNO DNAME
```

```
---------- --------------
        20 PESQUISA
        30 SALES
        30 SALES
        20 PESQUISA
        30 SALES
        30 SALES
        10 ACCOUNTING
        20 PESQUISA
        10 ACCOUNTING
        30 SALES
        20 PESQUISA
        30 SALES
        20 PESQUISA
        10 ACCOUNTING
        30 SALES
        80 S&E

16 linhas selecionadas.

SQL>

SQL> delete from emp_dept_v where deptno = 20;
Funcionário excluído com sucesso: SMITH.
O departamento PESQUISA não pode ser excluído. Ainda existem
funcionários agregados a ele.
Funcionário excluído com sucesso: JONES.
O departamento PESQUISA não pode ser excluído. Ainda existem
funcionários agregados a ele.
Funcionário excluído com sucesso: SCOTT.
O departamento PESQUISA não pode ser excluído. Ainda existem
funcionários agregados a ele.
Funcionário excluído com sucesso: ADAMS.
O departamento PESQUISA não pode ser excluído. Ainda existem
uncionários agregados a ele.
Funcionário excluído com sucesso: FORD.
Departamento excluído com sucesso: PESQUISA.

5 linhas deletadas.
```

```
SQL> select * from emp_dept_v;

     EMPNO ENAME      JOB              SAL        MGR
---------- ---------- --------- ---------- ----------
      7499 ALLEN      SALESMAN        1600       7698
      7521 WARD       SALESMAN        1250       7698
      7654 MARTIN     SALESMAN        1250       7698
      7698 BLAKE      MANAGER         2850       7839
      7782 CLARK      MANAGER         2450       7839
      7839 KING       PRESIDENT       5000
      7844 TURNER     SALESMAN        1500       7698
      7900 JAMES      CLERK            950       7698
      7934 MILLER     CLERK           1300       7782
      7935 PAUL       SALESMAN        1000       7698
      8001 SILVESTER  MANAGER         1000       7839

    DEPTNO DNAME
---------- --------------
        30 SALES
        30 SALES
        30 SALES
        30 SALES
        10 ACCOUNTING
        10 ACCOUNTING
        30 SALES
        30 SALES
        10 ACCOUNTING
        30 SALES
        80 S&E

11 linhas selecionadas.

SQL>
```

Com isso, encerramos o assunto sobre `triggers`. Como pôde ser visto, este tipo de objeto é muito interessante e útil para implementar diversos tipos de rotinas que possam ser executadas sem a intervenção direta do usuário e com segurança, o que faz dele um artifício muito poderoso.

Capítulo 20

PL/SQL Tables (estruturas homogêneas)

Conforme comentamos anteriormente, a PL/SQL contempla em sua estrutura, todos os recursos encontrados em outras linguagens de programação. Um recurso muito bacana é o uso de dados intrínsecos, através das estruturas homogenias, comumente, chamadas de `vetor`. Em PL/SQL chamamos este recurso de *PL/SQL Table*.

Para quem não conhece vetores, um vetor é uma estrutura array de um tipo definido de dado. Por exemplo, podemos ter um vetor de numéricos, caracteres ou datas. Podemos imaginar um array, ou melhor, um vetor, como se fosse a coluna de uma determinada tabela que possui um tipo definido, que pudesse ser preenchida por vários valores.

A diferença entre um e outro é que os vetores são definidos apenas em

memória, bem como os dados que são atribuídos a eles. Contudo, como estão em memória, proporcionam uma velocidade maior de acesso aos dados, ao contrário do uso de tabelas, no qual é necessário, na maioria das vezes, acessá-los fisicamente no disco.

Para utilizarmos uma PL/SQL Table, temos que primeiro defini-la como se fosse um tipo de dado sendo criado, chamado de `type`. Isso deve ser feito na área de declaração (`declare`) do bloco PL/SQL, sendo um bloco anônimo ou através de uma `procedure`, `function` ou `package`. Após a definição do `type` PL/SQL Table, é necessário declarar uma variável que utilizará esta definição. Vale salientar que inicialmente a definição do `type` não ocupa espaço na memória, apenas quando declaramos uma variável com base neste `type` é que isto acontece.

Veja o exemplo a seguir.

```
SQL> declare
  2     --
  3     type deptnotab is table of number index by binary_integer;
  4     type dnametab  is table of dept.dname%type index by
         binary_integer;
  5     type loctab    is table of varchar2(200) index by
         binary_integer;
  6     --
  7     wdeptnotab   deptnotab;
  8     wdnametab    dnametab;
  9     wloctab      loctab;
 10     --
 11     idx          binary_integer default 0;
 12   begin
 13     --
 14     for r1 in (select deptno, dname, loc from dept) loop
 15       idx := idx + 1;
 16       wdeptnotab(idx) := r1.deptno;
 17       wdnametab(idx)  := r1.dname;
 18       wloctab(idx)    := r1.loc;
 19     end loop;
 20     --
 21     for i in 1..wdeptnotab.last loop
 22       dbms_output.put_line('Departamento: '||wdeptnotab(i)||
```

```
23                               ' - '||wdnametab(i)||
24                               ' - Local: '||wloctab(i));
25     end loop;
26   end;
27   /

Departamento: 10 - ACCOUNTING - Local: FLORIDA
Departamento: 30 - SALES - Local: CHICAGO
Departamento: 40 - OPERATIONS - Local: BOSTON
Departamento: 80 - S&E - Local:
Departamento: 90 - FINANCIAL - Local:
Departamento: 99 - RH - Local: ARGENTINA
Departamento: 88 - RH - Local: ARGENTINA
Departamento: 41 - GENERAL LEDGER - Local:
Departamento: 42 - PURCHASING - Local:

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Pois bem, o exemplo anterior é muito simples. Ele seleciona dados da tabela `dept`. A intenção aqui não é mostrar exemplos complexos que dificultem o entendimento do assunto. Pelo contrário, utilizamos exemplos simples, focando nos pontos que devemos aprender. Desta forma, analisando o programa, temos nas linhas 3 a 4, três definições de tabela PL/SQL. A definição de uma tabela PL/SQL deve iniciar pelo comando `type` seguido do nome do tipo que queremos criar. Por exemplo, na linha 3, criamos com o nome `deptnotab`.

Logo após o nome, colocamos a expressão `is table of`, que vai nos indicar de que tipo será nossa PL/SQL Table, que neste caso definimos que é do tipo `number`. Depois de informarmos o tipo, devemos utilizar a cláusula `index by binary_integer`, que é uma cláusula padrão para a criação de PL/SQL Tables. Ela tem a a ver com o tipo de indexação (no caso, binária) usada para a criação da tabela e acesso aos dados na memória.

As linhas 4 e 5 seguem o mesmo formato de definição, sendo que uma foi definida com o mesmo tipo da coluna `deptno` da tabela `dept` e a outra com o tipo `varchar2(200)`. Veja que, nesse exemplo, a definição do `type` da

tabela PL/SQL é parecida com as definições usadas em variáveis.

Nas linhas 7 a 8, estão declaradas variáveis dos tipos criados nas linhas 3 a 5, ou seja, variáveis definidas com o tipo PL/SQL Table. São estas, as variáveis que serão manipuladas em nosso programa. Nas linhas 14 a 19, temos um cursor `for loop` que lê todas as informações da tabela `dept` e as armazena em nossas tabelas PL/SQL.

Uma observação importante é que, embora neste exemplo tenha sido criado um `type` de tabela PL/SQL para cada informação (código do departamento, nome do departamento e localização) é possível criar uma tabela PL/SQL usando `%rowtype` referenciando cursores ou tabelas. Nesses casos, é possível definir um `type` de tabela PL/SQL constituído por vários arrays, ou melhor, campos, para uma única PL/SQL Table.

Note que, dessa forma, a tabela PL/SQL ficará limitada à definição da tabela ou cursor utilizados na declaração. Contudo, neste capítulo, estamos falando de vetores, ou seja, estruturas homogêneas de dados. Assim sendo, como nesse exemplo foram recuperados dados referentes a três colunas da tabela `dept`, foi preciso definir três tabelas PL/SQL para guardá-los, e assim mantermos os conceitos referentes a vetores.

Continuando, conforme pode ser visto nas linhas 16 a 18, a forma de armazenamento precisa ser do tipo indexada para que possamos navegar entre as posições da tabela PL/SQL. Para isso, criamos uma variável chamada `idx` que será alimentada dentro do `loop` e servirá como índice de navegação dentro das nossas PL/SQL Tables. Para armazenar determinado valor, informamos o nome da tabela PL/SQL seguido do indexador entre parênteses, conforme pôde ser visto no exemplo (linhas 16 a 18).

Seguindo o exemplo, utilizamos outro `for loop` que lê nossas tabelas PL/SQL e imprime os valores na tela (linhas 21 a 25). Algumas funções estão disponíveis para o `type` PL/SQL Table. Uma delas é a função `last` que retorna o último registro preenchido no array. Também temos a função `count` que retorna a quantidade de registros contidos na tabela. No exemplo, foi utilizada a função `last` para indicar a posição final do `for loop`, ou seja, ele deve varrer a tabela PL/SQL da posição 1 até a última posição dela.

Repare que usamos o próprio indexador do `for loop (i)` para navegar pelos valores das tabelas, gravados anteriormente. Como nossas tabelas

foram preenchidas utilizando o mesmo `for loop`, ou seja, todas possuindo a mesma quantidade de registros, nós utilizamos apenas uma delas para servir como base para sabermos a quantidade exata que deve ser lida pelo cursor `for loop`. No entanto, através do indexador `(i)` do cursor, todos os registros de todas as tabelas foram lidos e utilizados na geração da saída feita pelo `dbms_output`.

Veja a figura a seguir para melhor entendimento de como é realizada a gravação e leitura dos dados, usando como base o programa do exemplo.

Cursor: gravação da tabela PL/SQL

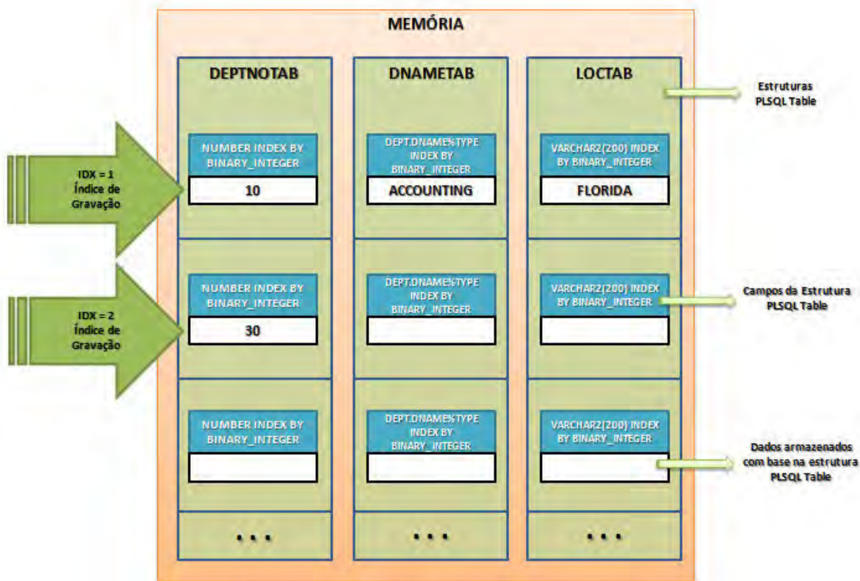

Fig. 20.1: Esquema de gravação de uma PL/SQL Table

Cursor: leitura da tabela PL/SQL

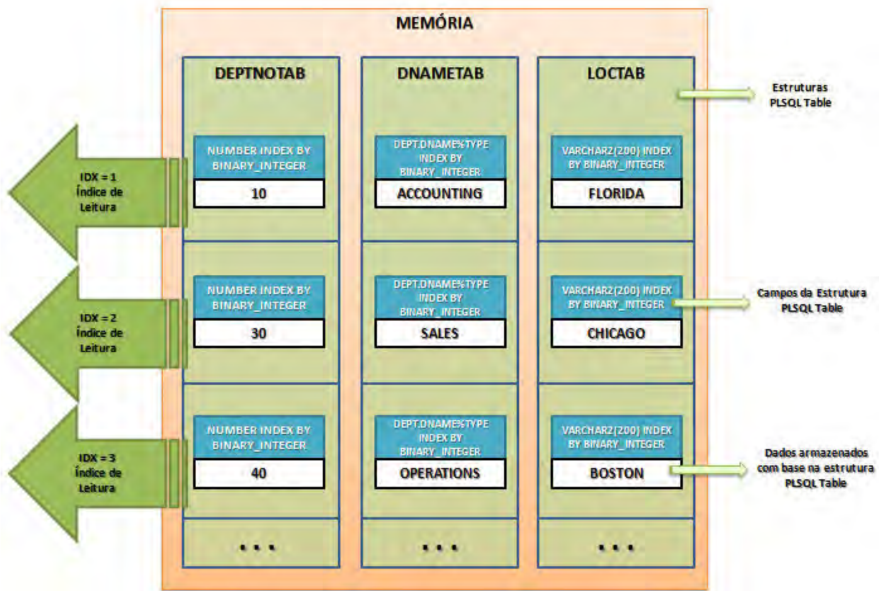

Fig. 20.2: Esquema de leitura de uma PL/SQL Table

Para concluir, seguem dois exemplos onde foram definidas PL/SQL Tables com base em tabelas e cursores.

Exemplo utilizando cursor:

```
SQL> declare
  2    --
  3    cursor c1 is
  4      select  d.department_id
  5             ,department_name
  6             ,first_name
  7             ,hire_date
  8             ,salary
  9      from   departments d
 10            ,employees e
 11      where  d.manager_id = e.employee_id
 12      order  by department_name;
```

```
13     --
14     type tab is table of c1%rowtype index by binary_integer;
15     --
16     tbgerente tab;
17     n  number;
18     --
19   begin
20     for r1 in c1 loop
21       tbgerente(r1.department_id) := r1;
22     end loop;
23     --
24     n := tbgerente.first;
25     --
26     while n <= tbgerente.last loop
27       dbms_output.put_line(
            'Depto: '||tbgerente(n).department_name||' '||
28           'Gerente: '||tbgerente(n).first_name||' '||
29           'Dt. Admi.: '||tbgerente(n).hire_date||' '||
30          'Sal.: '||to_char(tbgerente(n).salary,
            'fm$999g999g990d00'));
31       n := tbgerente.next(n);
32     end loop;
33   end;
34   /
Depto: Administration Gerente: Jennifer Dt. Admi.: 17/09/97
Sal.: $4.400,00
Depto: Marketing Gerente: Michael Dt. Admi.: 17/02/06
Sal.: $13.000,00
Depto: Purchasing Gerente: Den Dt. Admi.: 07/12/04
Sal.: $11.000,00
Depto: Human Resources Gerente: Susan Dt. Admi.: 07/06/04
Sal.: $6.500,00
Depto: Shipping Gerente: Adam Dt. Admi.: 10/04/07
Sal.: $8.200,00
Depto: IT Gerente: Alexander Dt. Admi.: 03/01/00 Sal.: $9.000,00
Depto: Public Relations Gerente: Hermann Dt. Admi.: 07/06/04
Sal.: $10.000,00
Depto: Sales Gerente: John Dt. Admi.: 01/10/06 Sal.: $14.000,00
Depto: Executive Gerente: Steven Dt. Admi.: 17/06/97
```

```
Sal.: $24.000,00
Depto: Finance Gerente: Nancy Dt. Admi.: 17/08/04
Sal.: $12.000,00
Depto: Accounting Gerente: Shelley Dt. Admi.: 07/06/04
Sal.: $12.000,00

Procedimento PL/SQL concluído com sucesso.

SQL>
```

No exemplo anterior, temos um programa que lê as informações de departamento e seus gerentes. Para tal, criamos um cursor (linhas 3 a 12) que busca cada departamento cadastrado na tabela `departments` que possua gerente associado na tabela `employees`. São trazidas pelo cursor as informações: código do departamento, nome do departamento, primeiro nome do gerente do departamento, data de admissão e salário do gerente.

Na linha 14, definimos um `type` PL/SQL Table chamado `tab` e atribuímos a ele o tipo `c1%rowtype`. Este tipo consiste em montar a mesma estrutura do cursor `c1` para o tipo `tab`, ou seja, serão criados para a PL/SQL Table os seguintes arrays: `department_id`, `department_name`, `first_name`, `hire_date` e `salary`.

Nas linhas 20 a 22, encontra-se a abertura do cursor e atribuição dos resultados do mesmo para a tabela PL/SQL `tbgerente`, que foi declarada como sendo do tipo `tab` na linha 16 do programa. Note que usamos como indexador o próprio valor do campo `department_id`, sendo que ele é único para cada linha. Veja também, que atribuímos à PL/SQL Table o array `r1`. `r1` possui os dados de todas as colunas do cursor. Logo, nossa tabela PL/SQL recebe todos estes valores, respeitando a estrutura definida.

Na linha 24, temos o uso da função `first` indicando que queremos que o ponteiro, que indica a posição em que o foco se encontra na PL/SQL Table, volte para a primeira posição. Só para entendimento, cada vez que um dado é carregado em uma linha da PL/SQL Table, o foco permanece nesta linha. Este retorno para a primeira linha da tabela PL/SQL tem a ver com o `loop while` que temos entre as linhas 26 a 32. No exemplo, com o `for loop`, nós utilizamos a própria estrutura de repetição para controlar a faixa de início e fim das linhas.

Como agora estamos utilizando o `while`, nós precisamos controlar isso. Para tanto, testamos a variável `N`, que recebe a primeira posição do array na linha 24. Logo, estamos solicitando que a estrutura `while` fique em loop enquanto não chegar a última linha do array. O comando utiliza a função `last` da tabela PL/SQL. Veja também que, na linha 31, utilizamos a função `next` para fazer com que o ponteiro vá para a próxima linha da tabela, fazendo sucessivamente com que todas as linhas sejam varridas.

Exemplo utilizando tabela (`%rowtype`):

```
SQL> declare
  2    --
  3    type deptab  is table of dept%rowtype index by
        binary_integer;
  4    --
  5    wdeptab   deptab;
  6    --
  7    idx          binary_integer default 0;
  8  begin
  9    --
 10    for r1 in (select * from dept) loop
 11      idx := idx + 1;
 12      wdeptab(idx) := r1;
 13    end loop;
 14    --
 15    for i in 1..wdeptab.last loop
 16      dbms_output.put_line(
         'Departamento: '||wdeptab(i).deptno||
 17       ' - '||wdeptab(i).dname||
 18       ' - Local: '||wdeptab(i).loc);
 19    end loop;
 20  end;
 21  /
Departamento: 10 - ACCOUNTING - Local: FLORIDA
Departamento: 30 - SALES - Local: CHICAGO
Departamento: 40 - OPERATIONS - Local: BOSTON
Departamento: 80 - S&E - Local:
Departamento: 90 - FINANCIAL - Local:
Departamento: 99 - RH - Local: ARGENTINA
```

```
Departamento: 88 - RH - Local: ARGENTINA
Departamento: 41 - GENERAL LEDGER - Local:
Departamento: 42 - PURCHASING - Local:

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Esse exemplo é muito semelhante aos demais. A única diferença é a forma como foi criado o `type` PL/SQL Table. Note que a tabela foi definida como tendo a mesma estrutura da tabela `dept`. Logo, estamos criando em memória a mesma estrutura da `dept` que existe fisicamente no banco de dados.

Capítulo 21

PL/SQL Records (estruturas heterogêneas)

Vimos que em PL/SQL Table trabalhamos com dados intrínsecos através das estruturas homogêneas. Em PL/SQL Records, vamos trabalhar com estruturas heterogêneas. Com PL/SQL Table, estávamos limitados a criar `types` Table constituídas apenas de um tipo de dado (numérico, caractere ou data), ou utilizando tabelas e cursores, onde, embora, tivéssemos vários campos com tipos de dados diferentes, cada qual possuía seus tipos já definidos, sem a possibilidade de alterarmos conforme a necessidade.

Com o recurso de PL/SQL Records, podemos definir tipos e estruturas de dados distintas dentro de um mesmo `type`, sem precisar criar vários `types` para cada um, ou sem necessitar associá-los a tabelas ou cursores.

Veja o exemplo a seguir:

```
SQL> declare
  2     --
  3     type deprec is record ( deptno number(2,0)
  4                           ,dname  varchar2(14)
  5                           ,loc    varchar2(13));
  6     --
  7     wdeprec   deprec;
  8     --
  9  begin
 10     --
 11     select *
 12     into   wdeprec
 13     from   dept
 14     where  deptno = 10;
 15     --
 16     dbms_output.put_line('Departamento: '||wdeprec.deptno||
 17                           ' - '||wdeprec.dname||
 18                           ' - Local: '||wdeprec.loc);
 19  end;
 20  /
Departamento: 10 - ACCOUNTING - Local: FLORIDA

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Nesse exemplo, começamos pela criação, linha 3, onde informamos a expressão `type` seguida pelo nome, `deprec`. Logo após, informamos o tipo do `type`, `is record`, que o define como PL/SQL Record. Após isto, entre parênteses, informamos qual a estrutura que este `type` terá. Aqui, definimos que ele será constituído por três campos, `deptno`, `dname` e `loc`. Desta forma, nos é permitido, através deste `type`, definir quantos campos quisermos e utilizar tipos diferentes para eles, possibilitando criar uma estrutura heterogênea.

Vale salientar que, embora os nomes dos campos sejam semelhantes aos da tabela `dept` existente no banco de dados, eles não estão referenciando-os. Aqui poderíamos colocar qualquer nome, respeitando apenas as mesmas regras para definição de variáveis.

Depois da criação do `type`, declaramos uma variável chamada `wdeprec` que será definida com este tipo (linha 7). Nas linhas 11 a 14, estamos selecionando os dados da tabela `dept` cujo departamento seja igual a 10. Estamos utilizando a cláusula `into`, onde os dados vindos do `select` estão sendo atribuídos ao nosso PL/SQL Record. Nas linhas 16 a 18, temos a impressão do resultado contido em nosso `type` Record. Veja a imagem a seguir, onde é mostrada de forma simbólica a definição e armazenamento dos dados em memória no PL/SQL Record.

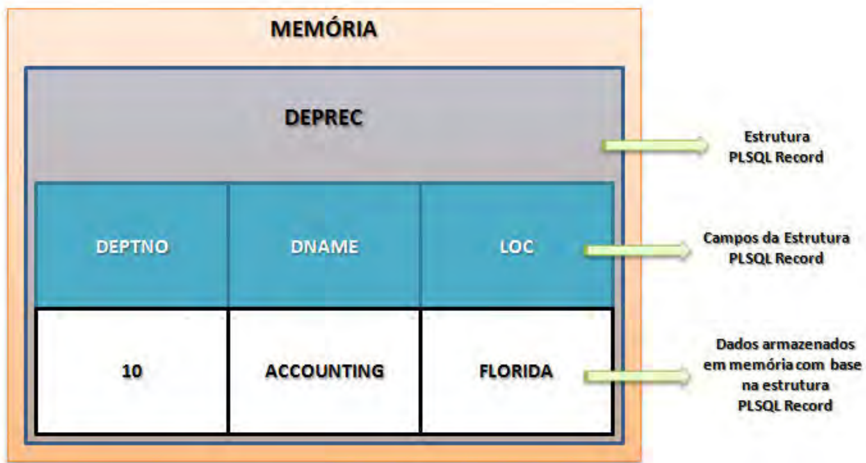

Fig. 21.1: Esquema de gravação de um PL/SQL Record

Você deve ter notado que não utilizamos estruturas de repetição e que, ao contrário disto, nosso exemplo trouxe apenas uma linha de registro. Pois bem, uma característica do `type` Record é que ele por si só pode guardar apenas um registro por vez, ao contrário do `type` Table, no qual podemos inserir várias linhas.

Desta forma, na maioria das vezes vemos os `types` Record sendo utilizados em conjunto com os `types` Table, sendo que este último nos permite gravar várias linhas. Vamos utilizar o exemplo anterior, agora, trazendo todos os dados da tabela `dept` e usando o recurso de `type` Table juntamente com o `type` Record.

```
SQL> declare
  2    --
  3    type deprec is record ( deptno number(2,0)
  4                           ,dname  varchar2(14)
  5                           ,loc    varchar2(13));
  6    --
  7    type deptab is table of deprec index by binary_integer;
  8    --
  9    wdeptab   deptab;
 10    idx       binary_integer default 0;
 11    --
 12  begin
 13    --
 14    for r1 in (select * from dept) loop
 15      --
 16      idx := idx + 1;
 17      wdeptab(idx) := r1;
 18      --
 19    end loop;
 20    --
 21    for i in 1..wdeptab.last loop
 22      dbms_output.put_line(
          'Departamento: '||wdeptab(i).deptno||
 23       ' - '||wdeptab(i).dname||
 24       ' - Local: '||wdeptab(i).loc);
 25    end loop;
 26  end;
 27  /
Departamento: 10 - ACCOUNTING - Local: FLORIDA
Departamento: 30 - SALES - Local: CHICAGO
Departamento: 40 - OPERATIONS - Local: BOSTON
Departamento: 80 - S&E - Local:
Departamento: 90 - FINANCIAL - Local:
Departamento: 99 - RH - Local: ARGENTINA
Departamento: 88 - RH - Local: ARGENTINA
Departamento: 41 - GENERAL LEDGER - Local:
Departamento: 42 - PURCHASING - Local:

Procedimento PL/SQL concluído com sucesso.
```

```
SQL>
```

Conforme mencionado, neste exemplo empregamos o uso de dois recursos em um mesmo programa, para que pudéssemos definir uma estrutura heterogênea e poder armazenar mais de um registro em memória.

Vale salientar que os `type` Tables e Records podem guardar qualquer tipo de dados, não necessariamente, dados vindos de tabelas do banco de dados. Podemos gerar e processar dados dentro de um programa e ir alimentando dentro destes objetos sem ter a necessidade de varrer dados que estão armazenados em tabelas, por exemplo.

Mas, voltando ao exemplo, vemos que nas linhas 3 a 5, está criado o `type` Record `deprec`, igualmente ao exemplo anterior. Já na linha 7 temos um `type` Table criado com base no `type` Record `deprec`. Dessa forma, é possível termos registros em forma de tabela, podendo armazenar mais de uma linha na PL/SQL Table, conforme a estrutura do PL/SQL Record criado. Veja a seguir a imagem representativa do uso destes recursos juntos.

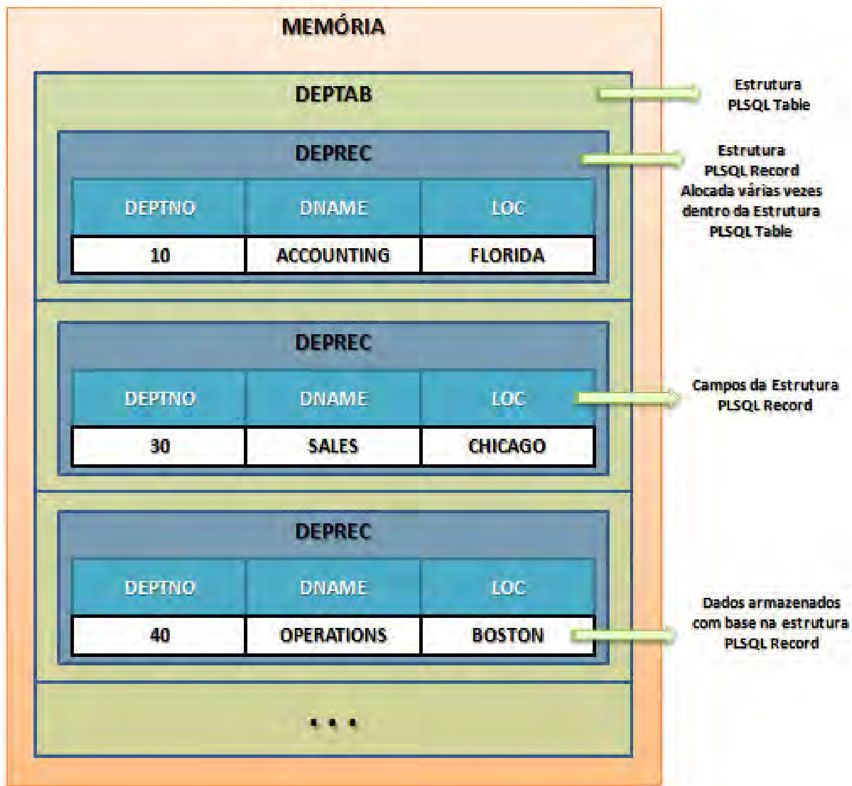

Fig. 21.2: Esquema do uso combinado de PL/SQL Table e PL/SQL Record

Ficou visto que a forma de definição e manipulação dos `type` Tables e Records é bastante semelhante. O entendimento dos conceitos apresentado no capítulo referente a PL/SQL Tables se faz necessário para compreender todo o contexto apresentado nestes dois capítulos. No mais, salientamos que o uso destes recursos pode ajudar e muito na manipulação dos dados, principalmente, em grandes volumes, quando o desempenho pode ser prejudicado, devido a muitos acessos a rede ou ao disco na busca pelas informações.

Capítulo 22

Pacote utl_file

Um recurso bem interessante e muito utilizado em PL/SQL é a leitura e escrita de arquivos. Em muitas empresas a exportação e importação de arquivos são muito utilizadas para alimentar sistemas paralelos (também chamados de sistemas satélites) ou para gerar informações gerenciais. A possibilidade de extrair informações do sistema e trabalhá-las usando ferramentas específicas para análise é uma prática muito comum.

Para este trabalho existe um pacote dentro do Oracle chamado `utl_file`. Este pacote trabalha com recursos que possibilita a comunicação entre o banco de dados Oracle e o sistema operacional, fazendo com que consigamos ler ou gerar arquivos externamente ao banco de dados.

Este pacote possui uma série de funções e procedimentos que são utilizados dentro dos blocos PL/SQL para a geração ou leitura de dados. De forma bem estruturada e clara, é possível não só ler as informações como também

trabalhar com elas, alterando-as e formatando-as antes de inseri-las em tabelas ou gravá-las em arquivo.

Algumas premissas devem ser observadas antes de utilizar este recurso. A primeira delas é ter privilégio para executar o pacote `utl_file`. A segunda premissa faz referência a como o `utl_file` faz a comunicação entre o banco de dados e o sistema operacional, para obter acesso aos diretórios do servidor onde serão gravados ou lidos os arquivos com os dados.

O banco de dados Oracle possui uma série de parametrizações que garante seu funcionamento. Estes parâmetros podem ser visualizados através da view `v$parameter`. Esta view lista todos os parâmetros do banco de dados Oracle. Um destes parâmetros é `utl_file_dir`, responsável por informar ao `utl_file` a quais diretórios ele tem acesso no servidor. A partir daí, o pacote `utl_file` só terá acesso em gravar e ler arquivos nestes diretórios. Esses diretórios não necessitam estar no mesmo servidor do banco de dados, entretanto, através de algum compartilhamento, os dois precisam enxergar um ao outro.

As opções de configuração deste parâmetro são:

- `utl_file_dir=C:\sistema\ERP`: indica um diretório específico para a gravação e leitura de arquivos (compatível com sistema Windows).
- `utl_file_dir=/ora/dat`: indica um diretório específico para a gravação e leitura de arquivos (compatível com sistema Unix).
- `utl_file_dir=C:\sistema\ERP, /ora/dat`: indica vários diretórios, separados por vírgula, para a gravação e leitura de arquivos.
- `utl_file_dir=*` : indica que a gravação ou leitura de arquivos pode ser realizada em todos os diretórios disponíveis no servidor.

> **Nota:** mesmo que os diretórios já estejam devidamente configurados para o parâmetro `utl_file_dir`, as permissões de acesso aos diretórios no nível de servidor também precisam ser realizadas.

Contudo, esta configuração em nível de parâmetro de banco de dados é meio inconveniente, pois o DBA para alterar esta definição precisa parar o banco de dados para realizá-la. Nem todos os parâmetros do banco de dados Oracle necessitam uma parada do sistema, mas para o parâmetro `utl_file_dir` é obrigatório que isto seja feito. Cada vez que o `utl_file` necessita acessar um diretório diferente é necessário realizar todo este trabalho.

Para evitar isto, a partir de versões mais novas do banco de dados a Oracle disponibilizou um recurso chamado `directory` com o qual é possível criar um objeto no banco de dados e apontá-lo para um diretório do servidor, sem a necessidade de alterarmos isso via parâmetro de banco. Com isto, o trabalho ficou muito mais simples, inclusive, não exige conhecimentos específicos de DBA.

Desta forma, antes de tudo, vamos aprender como se cria um objeto `directory`, para utilizarmos em nossa aplicação `utl_file`.

```
SQL> create directory dir_principal as 'C:\tmp\arquivos';

Diretório criado.

SQL>
```

Usamos o comando `create directory` para criar um objeto que aponte para o diretório onde queremos gravar ou ler determinados arquivos. Em seguida, informamos um nome para o objeto. Este é o nome que será utilizado dentro dos nossos programas. Seguindo a expressão `as` vem o caminho referente ao diretório. Para nossos exemplos usaremos este recurso.

Além de criamos o objeto, precisamos dar privilégios de acesso aos usuários do banco de dados. Caso queria disponibilizar acesso para todos os usuários do banco, dê privilégios através do usuário `public`.

```
SQL> grant read, write on directory dir_principal to TSQL;

Concessão bem-sucedida.

SQL>
```

Atenção: para dar acesso ao objeto `directory`, é necessário estar conectado com o usuário `system`, que é o usuário administrador do banco de dados Oracle.

Para ver os objetos `directory` cadastrados, use o comando `select`:

```
SQL> select *
  2  from   all_directories
  3  where  directory_name = 'DIR_PRINCIPAL';

OWNER   DIRECTORY_NAME    DIRECTORY_PATH
------- ----------------- ------------------------------
SYS     DIR_PRINCIPAL     C:\tmp\arquivos


SQL>
```

Para alterar um objeto `directory`, utilizamos o comando `replace`. Através dele podemos alterar o caminho do diretório.

```
SQL> create or replace directory dir_principal as 'C:\tmp\';

Diretório criado.

SQL>
```

Para dar privilégios de execução do `utl_file` é necessário estar conectado com o usuário `sys`.

```
SQL> grant execute on UTL_FILE to TSQL;

Concessão bem-sucedida.

SQL>
```

Nota: para conectar com o usuário `sys` utilize o comando `concc sys/<senha>@<string_banco> as sysdba;`.

Muito bem. Agora já criamos nosso `directory` e já concedemos privilégio para o usuário. Vamos colocar a mão na massa. Para trabalhar com arquivos, precisamos conhecer um pouco sobre as funções e procedimentos que fazem parte deste pacote. Além disso, precisamos conhecer as exceções que podem ser geradas em caso de erros.

Na sequência, veja funções e procedimentos no uso do `utl_file`.

fclose

Fecha o arquivo. Exemplo:

```
procedure fclose(file in out file_type);
```

Onde: `file` é o nome do arquivo lido ou gerado.

fclose_all

Fecha todos os arquivos. Exemplo:

```
procedure fclose_all;
```

Neste caso todos os arquivos aberto serão fechados.

fflush

Descarrega todos os dados em buffer para serem gravados em disco imediatamente. Exemplo:

```
procedure fflush(file in file_type);
```

Onde: `file` é o nome do arquivo lido ou gerado.

fopen

Abre o arquivo. Exemplo:

```
function fopen(location in varchar2,
               filename in varchar2,
               openmode in varchar2)
return file_type;%%
```

Onde: `location` é o nome do diretório onde se encontra o arquivo ou onde o arquivo deve ser gerado. `filename` é o nome do arquivo a ser lido ou gerado. `openmode` indica qual modo deve ser aplicado no arquivo. `R` = Read (ler), `W` = Write (escrever) ou `A` = Append (adicionar).

```
function fopen(location in varchar2,
               filename in varchar2,
               openmode in varchar2,
               max_linesize in binary_integer) return file_type;
```

Onde: `location` é o nome do diretório onde se encontra o arquivo ou onde o arquivo deve ser gerado. `filename` nome do arquivo a ser lido ou gerado. `openmode` indica qual modo deve ser aplicado no arquivo. `R` = Read (ler), `W` = Write (escrever) ou `A` = Append (adicionar). `max_linesize` permite especificar o tamanho máximo de linha. O intervalo permitido é de 1 até 32.767. Se você omitir esse parâmetro, o padrão 1.023 é usado.

get_line

Lê uma linha de um arquivo. Exemplo:

```
procedure get_line(file in file_type,
                   buffer out varchar2);
```

Onde: `file` é o nome do arquivo onde a linha deve ser lida. `buffer` é o lugar onde o conteúdo lido da linha deve ser inserido.

is_open

Verifica se um arquivo está aberto. Exemplo:

```
procedure is_open(file in file_type) return boolean;
```

Onde: `file` é o nome do arquivo que deve ser verificado.

new_line

Grava um caractere *newline* em um arquivo. Exemplo:

```
procedure new_line(file in file_type,
                   lines in natural := 1);
```

Onde: `file` é o nome do arquivo que deve receber a nova linha. `lines` é o número total de caracteres *newline* que você quer gravar no arquivo. O padrão é gravar uma *newline* nova. Este argumento é opcional.

put

Grava uma string de caracteres em um arquivo, mas não coloca uma *newline* depois dela. Exemplo:

```
procedure put(file in file_type,
              buffer in varchar2);
```

Onde: `file` é o nome do arquivo que deve receber o conteúdo. `buffer` é o conteúdo a ser gravado no arquivo.

put_line

Grava uma linha em um arquivo. Exemplo:

```
procedure put_line(file in file_type,
                   buffer in varchar2);
```

Onde: `file` é o nome do arquivo que deve receber o conteúdo. `buffer` contém o texto que você deseja gravar no arquivo. O número máximo de caracteres que você pode gravar usando uma chamada se limita ao tamanho de linha que você especificou na chamada de `fopen` e tem como padrão 1.023.

putf

Formata e grava saída. Essa é uma imitação bruta do procedimento `PRINTF()` do C. Exemplo:

```
procedure putf(file in file_type,
               format in varchar2,
               arg1 in varchar2 default null,
               arg2 in varchar2 default null,
```

```
            arg3 in varchar2 default null,
            arg4 in varchar2 default null,
            arg5 in varchar2 default null);
```

Onde: `file` é o nome do arquivo que receberá o conteúdo. `format` representa a string que você deseja gravar no arquivo. `arg1` ao `arg5` são argumentos opcionais contendo os valores que são substituídos na string de formato antes dela ser gravada no arquivo.

Exceções de fclose e fclose_all

- `utl_file.invalid_filehandle`: você passou um *handle* de arquivo que não representava um arquivo aberto.
- `utl_file.write_error`: o sistema operacional não pode gravar no arquivo.
- `utl_file.internal_error`: um erro interno ocorreu.

Exceções de fopen

- `utl_file.invalid_path`: o diretório não é válido. Você deve verificar em `utl_file_dir`.
- `utl_file.invalid_mode`: um modo inválido foi especificado. O modo open deve ser `R`, `W` ou `A`.
- `utl_file.invalid_operation`: o arquivo não pode ser aberto por algum outro motivo. Verifique se o proprietário do software do Oracle tem acesso ao diretório (isso pode ser uma questão de permissão) e entre em contato com o administrador do seu banco de dados para obter ajuda.
- `utl_file.internal_error`: um erro interno ocorreu.

Exceções de get_line

- `utl_file.invalid_filehandle`: você passou *handle* de arquivo inválido. Possivelmente você esqueceu de abrir o arquivo primeiro.
- `utl_file.invalid_operation`: o arquivo não está aberto para leitura (modo `R`), ou existem problemas com as permissões de arquivo.
- `utl_file.value_error`: o buffer não é suficientemente longo para conter a linha que está sendo lida do arquivo. O tamanho do buffer é aumentado.
- `utl_file.no_data_found`: o final do arquivo foi atingido.
- `utl_file.internal_error`: ocorreu um erro interno do sistema `utl_file`.
- `utl_file.read_error`: ocorreu um erro do sistema operacional durante a leitura do arquivo.

Exceções de put, put_line, putf e fflush

- `utl_file.invalid_filehandle`: você usou um *handle* de arquivo inválido. Essa exceção pode ser levantada quando você esquecer de abrir o arquivo.
- `utl_file.invalid_operation`: você tentou gravar em um arquivo que não estava aberto para gravação (modo `W` ou `A`).
- `utl_file.write_error`: ocorreu um erro de sistema operacional, tal como um erro de disco cheio, ao tentar gravar em um arquivo.
- `utl_file.internal_error`: ocorreu um erro interno.

Agora vamos ver um exemplo onde vamos ler os registros da tabela `emp` e `dept` e gerar um arquivo para ser gravado no diretório definido em nosso `directory`. Vamos gravar as informações no arquivo de forma linear e separando as informações por ponto e vírgula.

```
SQL> declare
  2     cursor c1 is
  3        select a.deptno, dname, empno, ename
  4        from   dept a
  5               ,emp  b
  6        where  a.deptno = b.deptno
  7        order by a.deptno;
  8     r1 c1%rowtype;
  9     --
 10     meu_arquivo  utl_file.file_type;
 11     --
 12  BEGIN
 13    --
 14    meu_arquivo := utl_file.fopen('DIR_PRINCIPAL',
                                      'empregados.txt', 'w');
 15    --
 16    open c1;
 17    --
 18    loop
 19      fetch c1 into r1;
 20      exit when c1%notfound;
 21      utl_file.put_line(meu_arquivo, r1.deptno||';'||
 22                                     r1.dname||';'||
 23                                     r1.empno||';'||
 24                                     r1.ename);
 25    end loop;
 26    --
 27    close c1;
 28    --
 29    utl_file.fclose(meu_arquivo);
 30    --
 31  exception
 32    when utl_file.invalid_path then
 33      utl_file.fclose(meu_arquivo);
 34      dbms_output.put_line ('Caminho ou nome do arquivo
                              inválido');
 35    when utl_file.invalid_mode then
 36      utl_file.fclose(meu_arquivo);
 37      dbms_output.put_line ('Modo de abertura inválido');
```

```
38  end;
39  /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Analisando o programa, vemos que na linha 10 declaramos uma variável do tipo `utl_file.file_type` que vai ser o ponteiro (*handle*) para o nosso arquivo físico. Quando isso é feito, o Oracle monta uma estrutura de arquivo em memória referente ao arquivo físico com que iremos trabalhar. Já na linha 14, atribuímos à variável declarada à abertura do arquivo passando como parâmetro o `directory` criado anteriormente, que faz referência ao local do diretório onde o arquivo deve ser gravado, o nome do arquivo e, por último, o modo, que neste caso é modo de escrita, representado por `W`. Quando executamos um `fopen` utilizando o modo escrita, caso não exista o arquivo ele o cria.

Na linha 21, utilizamos a chamada `put_line` para inserir linhas ao arquivo. Neste caso, estamos concatenando uma série de informações vindas das tabelas `emp` e `dept` e enviando para o arquivo. Após o termino da leitura das tabelas, ou seja, no final do cursor, fechamos o arquivo na linha 29. Note que entre as linhas 31 a 37 estão tratadas as possíveis exceções relacionadas ao pacote `utl_file`. Segue o resultado:

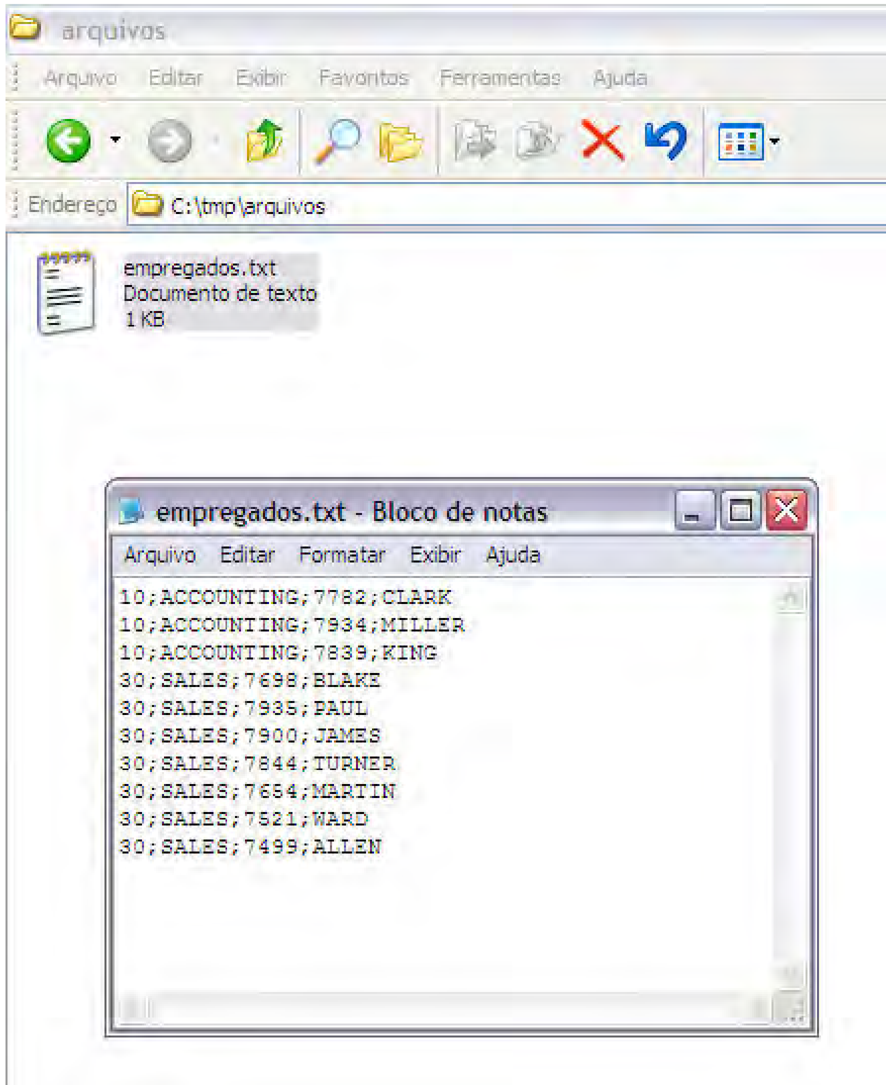

Fig. 22.1: Arquivo gerado através do pacote utl_file

Após termos visto o programa que gera as informações e as joga em um arquivo, vamos ver a contrapartida onde lemos o arquivo gerado e mostramos as informações na tela.

```
SQL> declare
  2     --
  3     meu_arquivo  utl_file.file_type;
  4     linha        varchar2(32000);
  5     --
  6     wdeptno   emp.deptno%type;
  7     wdname    dept.dname%type;
  8     wempno    emp.empno%type;
  9     wename    emp.ename%type;
 10     --
 11  begin
 12    --
 13    meu_arquivo := utl_file.fopen('DIR_PRINCIPAL',
                     'empregados.txt', 'r');
 14    --
 15    loop
 16      --
 17      utl_file.get_line(meu_arquivo,linha);
 18      --
 19      exit when linha is null;
 20      --
 21      wdeptno   :=
            rtrim(substr(linha,1,(instr(linha,';',1,1) -1)));
 22
 23      wdname    :=
            rtrim(substr(linha,(instr(linha,';',1,1) + 1)
 24                            ,(instr(linha,';',1,2) -1) -
 25                             (instr(linha,';',1,1) + (1 -1))));
 26
 27      wempno    :=
            rtrim(substr(linha,(instr(linha,';',1,2) + 1)
 28                            ,(instr(linha,';',1,3) -1) -
 29                             (instr(linha,';',1,2) + (1 -1))));
 30
 31      wename    := rtrim(rtrim(substr(linha,(
                        instr(linha,';',1,3) + 1)),chr(13)));
 32      --
 33      dbms_output.put_line('Cód. Departamento: '||wdeptno);
 34      dbms_output.put_line('Nome Departamento: '||wdname);
```

```
35       dbms_output.put_line('Cód. Empregado: '||wempno);
36       dbms_output.put_line('Nome Empregado: '||wename);
37       dbms_output.put_line('_');
38       --
39     end loop;
40     --
41     utl_file.fclose(meu_arquivo);
42     --
43   exception
44     when no_data_found then
45       utl_file.fclose(meu_arquivo);
46       dbms_output.put_line ('Final do Arquivo.');
47     when utl_file.invalid_path then
48       utl_file.fclose(meu_arquivo);
49       dbms_output.put_line ('Caminho ou nome do arquivo
                               inválido');
50     when utl_file.invalid_mode then
51       utl_file.fclose(meu_arquivo);
52       dbms_output.put_line ('Modo de abertura inválido');
53   end;
54   /
```

Vamos analisar o código do programa. Na linha 3, declaramos uma variável do tipo `utl_file.file_type` que vai ser o ponteiro para o nosso arquivo físico. Na linha criamos uma variável chamada `linha` que receberá os dados de cada linha vinda do arquivo. Abrimos o arquivo através da função `fopen` utilizando o modo `R` (leitura). Já na linha 17, utilizamos a chamada de procedimento `get_lines` que lê uma linha (a primeira) do arquivo e guarda seu valor dentro da variável `linha`. Neste caso, ele busca todo o conteúdo da linha e joga dentro da variável, ou seja, todos os valores da linha concatenados por ponto e vírgula.

Na linha 19, temos o seguinte código `exit when linha is null`. Conforme visto nas exceções, vimos que ao chegar ao final de um arquivo a exceção `no_data_found` é acionada, ou seja, quando não houver mais linhas para ler, é gerada uma exceção informando que não há mais linhas para serem lidas no arquivo. Contudo, dependendo o sistema operacional e/ou da codificação do arquivo gerado, pode acontecer, no caso de termos

um caractere `enter` no final do arquivo, de o `utl_file` ler este caractere e considerá-lo como uma linha. Caso o programa não esteja tratando esta situação, erros podem ocorrer.

Desta forma, a linha 19 serve para tratar este tipo de situação. Caso a exceção `no_data_found` não for acionada, esta cláusula garante a saída do `loop`. Mas atenção: isso serve para arquivos que não estejam considerando linhas em branco em sua estrutura. Caso linhas em branco façam parte da formatação dos dados, esta linha não deve ser colocada. Depois, nas linhas 21 a 31 utilizamos as funções `substr` e `instr` para separar os dados, tendo como base o caractere ponto e vírgula. Finalizamos a leitura do arquivo na linha 41. Note que neste exemplo tratamos a exceção `no_data_found`, na linha 44.

Veja o resultado.

```
Cód. Departamento: 10
Nome Departamento: ACCOUNTING
Cód. Empregado: 7782
Nome Empregado: CLARK
-
Cód. Departamento: 10
Nome Departamento: ACCOUNTING
Cód. Empregado: 7934
Nome Empregado: MILLER
-
Cód. Departamento: 10
Nome Departamento: ACCOUNTING
Cód. Empregado: 7839
Nome Empregado: KING
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7698
Nome Empregado: BLAKE
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7935
Nome Empregado: PAUL
```

```
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7900
Nome Empregado: JAMES
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7844
Nome Empregado: TURNER
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7654
Nome Empregado: MARTIN
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7521
Nome Empregado: WARD
-
Cód. Departamento: 30
Nome Departamento: SALES
Cód. Empregado: 7499
Nome Empregado: ALLEN
-
Final do Arquivo.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Por curiosidade, vamos apresentar o mesmo exemplo de geração e leitura de arquivo utilizando a separação de informação não por ponto e vírgula, mas sim, por número de caracteres fixos. Este método visa termos número fixos para o tamanho das colunas. Vale lembrar que não muda em nada a forma de utilizar o pacote `utl_file`. É apenas para mostrar uma forma diferente, porém muito utilizada, de gerar layouts diferentes para arquivos textos

Geração do arquivo:

```
SQL> declare
  2     cursor c1 is
  3        select a.deptno, dname, empno, ename
  4        from   dept a
  5              ,emp  b
  6        where  a.deptno = b.deptno
  7        order by a.deptno;
  8     r1 c1%rowtype;
  9     --
 10     meu_arquivo  utl_file.file_type;
 11     --
 12  BEGIN
 13    --
 14    meu_arquivo := utl_file.fopen('DIR_PRINCIPAL',
                                   'empregados.txt', 'w');
 15    --
 16    open c1;
 17    --
 18    loop
 19      fetch c1 into r1;
 20      exit when c1%notfound;
 21      utl_file.put_line(meu_arquivo, rpad(r1.deptno,2,' ') ||
 22                                     rpad(r1.dname,14,' ') ||
 23                                     rpad(r1.empno,4,' ')  ||
 24                                     rpad(r1.ename,10,' ')
 25                                     );
 26    end loop;
 27    --
 28    close c1;
 29    --
 30    utl_file.fclose(meu_arquivo);
 31    --
 32  exception
 33    when utl_file.invalid_path then
 34      utl_file.fclose(meu_arquivo);
 35      dbms_output.put_line ('Caminho ou nome do arquivo
                              inválido');
```

```
36     when utl_file.invalid_mode then
37       utl_file.fclose(meu_arquivo);
38       dbms_output.put_line ('Modo de abertura inválido');
39   end;
40   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Leitura do arquivo:

```
SQL> declare
  2      --
  3      meu_arquivo  utl_file.file_type;
  4      linha        varchar2(32000);
  5      --
  6      wdeptno   emp.deptno%type;
  7      wdname    dept.dname%type;
  8      wempno    emp.empno%type;
  9      wename    emp.ename%type;
 10      --
 11   begin
 12     --
 13     meu_arquivo := utl_file.fopen('DIR_PRINCIPAL',
                                      'empregados.txt', 'r');
 14     --
 15     loop
 16       --
 17       utl_file.get_line(meu_arquivo,linha);
 18       --
 19       exit when linha is null;
 20       --
 21       wdeptno   := rtrim(ltrim(substr(linha,1,2)));
 22       wdname    := rtrim(ltrim(substr(linha,3,14)));
 23       wempno    := rtrim(ltrim(substr(linha,17,4)));
 24       wename    := rtrim(ltrim(substr(linha,21,10)));
 25       --
 26       dbms_output.put_line('Cód. Departamento: '||wdeptno);
 27       dbms_output.put_line('Nome Departamento: '||wdname);
```

```
28       dbms_output.put_line('Cód. Empregado: '||wempno);
29       dbms_output.put_line('Nome Empregado: '||wename);
30       dbms_output.put_line('_');
31       --
32     end loop;
33     --
34     utl_file.fclose(meu_arquivo);
35     --
36   exception
37     when no_data_found then
38       utl_file.fclose(meu_arquivo);
39       dbms_output.put_line ('Final do Arquivo.');
40     when utl_file.invalid_path then
41       utl_file.fclose(meu_arquivo);
42       dbms_output.put_line (
          'Caminho ou nome do arquivo inválido');
43     when utl_file.invalid_mode then
44       utl_file.fclose(meu_arquivo);
45       dbms_output.put_line ('Modo de abertura inválido');
46   end;
47   /
Cód. Departamento: 10
Nome Departamento: ACCOUNTING
Cód. Empregado: 7782
Nome Empregado: CLARK
....
_
Final do Arquivo.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Conforme pôde ser visto nesse exemplo, a leitura toma como base, posições fixas que indicam a localização exata de cada informação na linha, extraindo-as.

Capítulo 23

SQL dinâmico

Na maioria das vezes, trabalharemos com comandos SQL previamente definidos dentro de nossos programas. Geralmente, o corpo dos comandos não se altera, apenas os parâmetros que passamos a eles. Podemos ter o mesmo comando SQL diversas vezes dentro do nosso programa, mas mediante a passagem de parâmetros com valores distintos podemos ter resultados diferentes.

Contudo, pode acontecer de, dependendo da situação, não termos estes comandos SQL definidos, ou seja, mediante determinadas ações talvez precisemos modificar a estrutura do comando SQL para atender as necessidades, mas com um detalhe: tudo isso em tempo de execução.

Para isso, a linguagem PL/SQL disponibiliza um recurso chamado de *SQL dinâmico* que nos dá a possibilidade de executar um comando SQL a partir de uma string, ou seja, o comando passa a ser uma string de caracteres armazenados em uma variável. Através de um comando, a variável é lida e o SQL que está dentro dela é executado. Isso é feito através do procedimento `execute`

immediate. Por meio deste procedimento, podemos executar comandos de insert, delete, update, select, create, alter, drop, bem como executar procedimentos e funções. É possível também realizar passagem de parâmetros para esses comandos. Veja os exemplos a seguir.

Uso com insert:

```
SQL> declare
  2     --
  3     winsert_emp  varchar2(4000) default null;
  4     winsert_dept varchar2(4000) default null;
  5     --
  6     wempno      emp.empno%type;
  7     wename      emp.ename%type;
  8     wjob        emp.job%type;
  9     wmgr        emp.mgr%type;
 10     whiredate   emp.hiredate%type;
 11     wsal        emp.sal%type;
 12     wcomm       emp.comm%type;
 13     wdeptno     emp.deptno%type;
 14     --
 15   begin
 16     --
 17     wempno      := 9999;
 18     wename      := 'BONO';
 19     wjob        := 'SALESMAN';
 20     wmgr        := 7698;
 21     whiredate   := SYSDATE;
 22     wsal        := 1000;
 23     wcomm       := 0;
 24     wdeptno     := 30;
 25     --
 26     winsert_emp := 'insert into emp ( empno
 27                                     ,ename
 28                                     ,job
 29                                     ,mgr
 30                                     ,hiredate
 31                                     ,sal
 32                                     ,comm
 33                                     ,deptno)
```

```
34                              values ( :empno
35                                      ,:ename
36                                      ,:job
37                                      ,:mgr
38                                      ,:hiredate
39                                      ,:sal
40                                      ,:comm
41                                      ,:deptno)';
42     --
43     execute immediate winsert_emp using wempno
44                                         ,wename
45                                         ,wjob
46                                         ,wmgr
47                                         ,whiredate
48                                         ,wsal
49                                         ,wcomm
50                                         ,wdeptno;
51     --
52     winsert_dept :=
        'insert into dept values (:deptno, :dname, :loc)';
53     --
54     execute immediate winsert_dept using 99, 'RH', 'BRASIL';
55     --
56   end;
57   /

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo, na linha 26, temos a variável `winsert_emp` que recebe como string o comando `insert`. Note que os valores que serão inseridos (linha 34 a 41) estão representados por variáveis chamadas `bind`. Este tipo de variável é caracterizado por iniciar com dois pontos antes do seu nome. Quando declaramos uma variável do tipo `bind` a PL/SQL interpreta que algum valor deve ser informado para ela, em um dado momento. Caso este valor não seja passado por parâmetro é aberto um *prompt* para que ele seja digitado.

Nas linhas 43 a 50, temos o comando `execute immediate`. Ele aparece precedido da string ou variável referente ao comando que deve ser executado. Caso parâmetros devam ser passados, utilizamos a cláusula `using` para isto. Veja no exemplo, que estamos utilizando `using` para passar os parâmetros para o comando `insert`, substituindo as variáveis `bind`. Esta substituição acontece de forma sequencial.

Além da inserção de empregados, também temos, no mesmo bloco PL/SQL, a inserção de departamentos. Embora bem parecido com a inserção de empregado, este exemplo serve para mostrar que os valores referentes aos parâmetros podem ser passados diretamente no comando `execute immediate`. Veja o resultado na sequência:

```
SQL> select * from emp;

     EMPNO ENAME    JOB              MGR HIREDATE        SAL      COMM
---------- -------- --------- -------- -------- ---------- --------
      7499 ALLEN    SALESMAN     7698  20/02/81       1600       306
      7521 WARD     SALESMAN     7698  22/02/81       1250       510
      9999 BONO     SALESMAN     7698  25/10/11       1000         0
      7654 MARTIN   SALESMAN     7698  28/09/81       1250      1428
      7698 BLAKE    MANAGER      7839  01/05/81       2850       285
      7782 CLARK    MANAGER      7839  09/06/81       2450       245
      7839 KING     PRESIDENT          17/11/81       5000       500
      7844 TURNER   SALESMAN     7698  08/09/81       1500       150
      7900 JAMES    CLERK        7698  03/12/81        950        95
      7934 MILLER   CLERK        7782  23/01/82       1300       130
      7935 PAUL     SALESMAN     7698  15/03/80       1000       100

    DEPTNO PC_COM_SAL
---------- ----------
        30     19,125
        30       40,8
        30
        30     114,24
        30
        10
        10
        30         10
```

```
        30
        10
        30          10

11 linhas selecionadas.

SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        80 S&E
        90 FINANCIAL
        99 RH             BRASIL

6 linhas selecionadas.

SQL> COMMIT;

Validação completa.

SQL>
```

Uso com delete e update:

```
SQL> declare
  2    --
  3  begin
  4    --
  5    execute immediate 'delete from emp
  6                       where empno = :empno ' using 9999;
  7    --
  8    execute immediate 'update dept set loc = :loc
  9                where deptno = :depnto' using 'AUSTRALIA', 99;
 10    --
 11  end;
 12  /
```

```
Procedimento PL/SQL concluído com sucesso.

SQL>
```

Nesse exemplo, estamos mostrando a exclusão e atualização de registros via `execute immediate`. A seguir temos o resultado:

```
SQL> select * from emp;

     EMPNO ENAME    JOB             MGR HIREDATE        SAL      COMM
---------- ------- --------- ------- -------- --------- ---------
      7499 ALLEN   SALESMAN     7698  20/02/81       1600       306
      7521 WARD    SALESMAN     7698  22/02/81       1250       510
      7654 MARTIN  SALESMAN     7698  28/09/81       1250      1428
      7698 BLAKE   MANAGER      7839  01/05/81       2850       285
      7782 CLARK   MANAGER      7839  09/06/81       2450       245
      7839 KING    PRESIDENT          17/11/81       5000       500
      7844 TURNER  SALESMAN     7698  08/09/81       1500       150
      7900 JAMES   CLERK        7698  03/12/81        950        95
      7934 MILLER  CLERK        7782  23/01/82       1300       130
      7935 PAUL    SALESMAN     7698  15/03/80       1000       100

    DEPTNO PC_COM_SAL
---------- ----------
        30     19,125
        30       40,8
        30     114,24
        30
        10
        10
        30         10
        30
        10
        30         10

10 linhas selecionadas.

SQL> select * from dept;
```

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        80 S&E
        90 FINANCIAL
        99 RH             AUSTRALIA

6 linhas selecionadas.

SQL> COMMIT;
Validação completa.
SQL>
```

Uso com os comandos `create` e `alter`. `execute immediate` pode ser usado também para a criação e alteração dinâmica de tabelas:

```
SQL> declare
  2    --
  3  begin
  4    --
  5    execute immediate 'create table func
  6                      (cd_func number, nm_func varchar2(50))';
  7    --
  8    execute immediate 'alter table func
  9                       modify cd_func number not null';
 10    --
 11  end;
 12  /

Procedimento PL/SQL concluído com sucesso.

SQL> desc func
 Nome                                      Nulo?    Tipo
 ----------------------------------------- -------- -------------
 CD_FUNC                                   NOT NULL NUMBER
 NM_FUNC                                            VARCHAR2(50)

SQL>
```

Por meio do uso do `execute immediate` é possível retornar informações provenientes de comandos SQL. Veja este exemplo da execução de um `update`:

```
SQL> declare
  2    --
  3    watualiza_dept varchar2(2000);
  4    --
  5    wdname   dept.dname%type;
  6    wloc_re dept.loc%type;
  7    --
  8    wloc    dept.loc%type    default 'CHILE';
  9    wdeptno dept.deptno%type default 99;
 10  begin
 11    --
 12    watualiza_dept := 'update dept set loc = :1
                         where deptno = :2
 13                      returning dname, loc into :3, :4';
 14    --
 15    execute immediate watualiza_dept using wloc, wdeptno
 16                                          ,out wdname, out wloc_re;
 17    --
 18    dbms_output.put_line('Localização atualizada para o
                            departamento '||
 19                         wdname||' (Loc: '||wloc||').');
 20    --
 21  end;
 22  /
Localização atualizada para o departamento RH (Loc: CHILE).

Procedimento PL/SQL concluído com sucesso.

SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

```
        80 S&E
        90 FINANCIAL
        99 RH             CHILE

6 linhas selecionadas.

SQL>
```

Analisando esse exemplo, temos na linha 12 a variável `watualiza_dept` recebendo a string referente ao comando `update`. Também é possível observar que, na linha 13, está incluída a cláusula `returning` seguida de dois campos, os quais serão o nosso retorno, seguida da cláusula `into` informando duas variáveis `bind`, as quais receberão os valores retornados.

Uma observação importante, é que na linha 15 e 16, referentes ao comando `execute immediate`, temos duas passagens de parâmetros do tipo `in` (entrada) e duas do tipo `out` (saída), sendo que estas duas últimas guardam nosso retorno. Vale lembrar que quando não informado o tipo da variável na cláusula `using`, por padrão, seu tipo será de entrada (`in`).

Veja outro exemplo utilizando retorno de informação, mas agora com o uso da cláusula `returning` também no `execute immediate`. Note que através desta notação não necessitamos informar o tipo das variáveis de retorno.

```
SQL> declare
  2    --
  3    watualiza_dept varchar2(2000);
  4    --
  5    wdname  dept.dname%type;
  6    wloc_re dept.loc%type;
  7    --
  8    wloc    dept.loc%type    default 'ARGENTINA';
  9    wdeptno dept.deptno%type default 99;
 10  begin
 11    --
 12    watualiza_dept := 'update dept set loc = :1 where
                         deptno = :2
 13                      returning dname, loc into :3, :4';
 14    --
```

```
15      execute immediate watualiza_dept using wloc, wdeptno
16                      returning into wdname, wloc_re;
17    --
18    dbms_output.put_line('Localização atualizada para o
                          departamento '||
19                        wdname||' (Loc: '||wloc||').');
20    --
21  end;
22  /
Localização atualizada para o departamento RH (Loc: ARGENTINA).

Procedimento PL/SQL concluído com sucesso.

SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        80 S&E
        90 FINANCIAL
        99 RH             ARGENTINA

6 linhas selecionadas.

SQL>
```

O exemplo a seguir mostra o uso do `execute immediate` na execução dinâmica de um comando `delete`:

```
SQL> create function rows_deleted ( table_name in varchar2
  2                      ,condition  in varchar2) return integer as
  3  begin
  4     execute immediate 'delete from ' || table_name || '
        where ' || condition;
  5     return sql%rowcount;
  6  end;
  7  /
```

```
Função criada.

SQL> declare
  2     wnr_linhas number;
  3   begin
  4     wnr_linhas := rows_deleted('EMP',' empno = 7935');
  5     --
  6     dbms_output.put_line('Linhas excluídas: '||wnr_linhas);
  7     --
  8   end;
  9   /
Linhas excluídas: 1

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Note que utilizamos o cursor implícito `sql%rowcount` para retornar a quantidade de linhas excluídas. Através deste recurso também é possível executar procedimentos PL/SQL. Para isso, criamos uma `procedure` para realizar inserções na tabela `dept`.

```
SQL> create or replace procedure cria_dept (
        pdeptno in     number
  2     ,pdname  in     varchar2
  3     ,ploc    in     varchar2
  4     ,pstatus in out varchar2) is
  5   begin
  6     --
  7     insert into dept values (pdeptno, pdname, ploc);
  8     --
  9     pstatus := 'OK';
 10     --
 11     commit;
 12     --
 13   exception
 14     when others then
 15       pstatus := sqlerrm;
```

```
16  end;
17  /

Procedimento criado.

SQL>
```

Agora chamamos a `procedure` criada através de outro bloco PL/SQL, utilizando uma chamada dinâmica.

```
SQL> declare
  2    --
  3    winsere_dept varchar2(2000);
  4    --
  5    wdname  dept.dname%type  default 'RH';
  6    wloc    dept.loc%type    default 'ARGENTINA';
  7    wdeptno dept.deptno%type default 88;
  8    --
  9    wstatus varchar2(4000);
 10    --
 11  begin
 12    --
 13     winsere_dept := 'begin cria_dept(:a, :b, :c, :d); end;';
 14     --
 15     execute immediate winsere_dept
 16        using in wdeptno, in wdname, in wloc, in out wstatus;
 17     --
 18     if wstatus = 'OK' then
 19       --
 20       dbms_output.put_line('Departamento inserido com
                                sucesso.');
 21       --
 22     else
 23       --
 24       dbms_output.put_line('Erro ao inserir departamento.
                                Erro: '||wstatus);
 25       --
 26     end if;
 27  end;
```

```
 28  /
Departamento inserido com sucesso.

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Veja o resultado:

```
SQL> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        80 S&E
        90 FINANCIAL
        99 RH             ARGENTINA
        88 RH             ARGENTINA

7 linhas selecionadas.

SQL>
```

A seguir, diferentes usos de SQL dinâmico para comandos `select`, com retorno de informações. Nestes exemplos, não utilizaremos `execute immediate`; em vez disso, usaremos cursores do tipo `ref cursor`.

23.1 Ref cursor

Uma variável do tipo `ref cursor`, como é comumente chamada, é utilizada para referenciar a estrutura de uma `query`, de forma dinâmica, ou seja, ela não está ligada exatamente a um único comando SQL, diferente do uso de cursores, no qual ao declararmos, necessitamos informar um comando `select` específico.

Variáveis do tipo `ref cursor`, por não terem uma estrutura predefinida, podem ser usadas várias vezes dentro de um programa para diferentes comandos `select` e para um número ilimitado de linhas. Variáveis

do tipo `ref cursor` podem ainda serem definidas como parâmetros em `procedures`. Sua declaração é semelhante à declaração de variáveis do tipo tabela ou registro (`Index-By Table`, `Record`).

A principal vantagem no uso de variáveis do tipo `ref cursor` se dá pelo fato de ela apenas apontar para a estrutura ou linha retornada pelo comando `select`. Sendo assim, o resultado não é armazenado, apenas referenciado. Isso é muito útil na passagem de informações entre vários sistemas, pois não há uma passagem de informações propriamente dita, apenas o compartilhamento através do que chamamos de um ponteiro.

```
SQL> declare
  2     type empcurtyp is ref cursor;
  3     emp_cv    empcurtyp;
  4     --
  5     my_ename  varchar2(15);
  6     my_sal    number        default 1000;
  7  begin
  8     open emp_cv for
        'select ename, sal from emp where sal > :s' using my_sal;
  9     loop
 10       fetch emp_cv into my_ename, my_sal;
 11
 12       exit when emp_cv%notfound;
 13       --
 14       dbms_output.put_line(
            'Empregado: '||my_ename||' Salário: '||my_sal);
 15       --
 16     end loop;
 17  end;
 18  /
Empregado: ALLEN Salário: 1600
Empregado: WARD Salário: 1250
Empregado: MARTIN Salário: 1250
Empregado: BLAKE Salário: 2850
Empregado: CLARK Salário: 2450
Empregado: KING Salário: 5000
Empregado: TURNER Salário: 1500
Empregado: MILLER Salário: 1300
```

```
Procedimento PL/SQL concluído com sucesso.

SQL>
```

Neste exemplo, estamos utilizando um `ref cursor` que recebe uma string de um comando SQL (linha 8). Utilizamos as cláusulas `open` e `for` para atribuir o comando à variável `ref cursor emp_cv`. Depois, utilizamos um `loop` simples (linha 9) para manipular as informações, que são atribuídas a duas variáveis através da cláusula `into` (linha 10). Note que também utilizamos a cláusula `using` para passagem de parâmetro.

```
SQL> declare
  2      type empcurtyp is ref cursor;
  3      emp_cv    empcurtyp;
  4      emp_rec   emp%rowtype;
  5      --
  6      sql_stmt varchar2(200);
  7      my_job   varchar2(15) := 'CLERK';
  8  begin
  9      sql_stmt := 'select * from emp where job = :j';
 10      --
 11      open emp_cv for sql_stmt using my_job;
 12      loop
 13         fetch emp_cv into emp_rec;
 14         exit when emp_cv%notfound;
 15         --
 16         dbms_output.put_line('Empregado: '||emp_rec.ename||'
                                  Salário: '||emp_rec.sal);
 17         --
 18      end loop;
 19      close emp_cv;
 20  end;
 21  /
Empregado: JAMES Salário: 950
Empregado: MILLER Salário: 1300

Procedimento PL/SQL concluído com sucesso.
```

```
SQL>
```

Esse exemplo é bem semelhante ao anterior. A diferença está nas linhas 4 e 13. Declaramos uma variável do tipo tabela, utilizando a estrutura da tabela `emp` (linha 4), que receberá o resultado do `select` através da cláusula `into` na linha 13.

O próximo exemplo mostra que podemos montar um comando SQL através de strings passadas por parâmetro.

```
SQL> create procedure print_table (tab_name varchar2) is
  2      type refcurtyp is ref cursor;
  3      cv refcurtyp;
  4      wdname  dept.dname%type;
  5      wloc    dept.loc%type;
  6  begin
  7    open cv for 'select dname, loc from ' || tab_name;
  8    --
  9    loop
 10      fetch cv into wdname, wloc;
 11      exit when cv%notfound;
 12      --
 13      dbms_output.put_line('Departamento: '||wdname||'
                             Localização: '||wloc);
 14      --
 15    end loop;
 16    --
 17    close cv;
 18    --
 19  end;
 20  /

Procedimento criado.

SQL> begin print_table (tab_name => 'DEPT'); end;
  2  /
Departamento: ACCOUNTING Localização: FLORIDA
Departamento: SALES Localização: CHICAGO
Departamento: OPERATIONS Localização: BOSTON
Departamento: S&E Localização:
```

```
Departamento: FINANCIAL Localização:
Departamento: RH Localização: ARGENTINA
Departamento: RH Localização: ARGENTINA

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Já no exemplo a seguir temos um mix de variável `ref cursor` e variáveis `table` e `record`. Aqui utilizamos a cláusula `bulk collect`, que é responsável por preparar toda a coleção de saída de dados. Esta cláusula pode ser usada junto com a cláusula `into` nos comandos `select into`, `fetch into`, `returning into` e variáveis `table`. Esta cláusula não pode ser usada para variáveis do tipo `record`. Dessa forma, caso tenhamos informações referentes a várias colunas, será necessária a criação de uma variável do tipo `table` para cada uma.

```
SQL> declare
  2      type empcurtyp is ref cursor;
  3      type numlist   is table of number;
  4      type namelist  is table of varchar2(15);
  5      --
  6      emp_cv empcurtyp;
  7      empnos numlist;
  8      enames namelist;
  9      --
 10      sals   numlist;
 11  begin
 12    open emp_cv for 'select empno, ename from emp';
 13    fetch emp_cv bulk collect into empnos, enames;
 14    close emp_cv;
 15    --
 16    for r in 1..empnos.count loop
 17      dbms_output.put_line('Cód.: '||empnos(r)||' -
            Empregado: '||enames(r));
 18    end loop;
 19    --
 20    execute immediate 'select sal from emp'
 21       bulk collect into sals;
```

```
22    --
23    for r in 1..sals.count loop
24      dbms_output.put_line('Salário: '||sals(r));
25    end loop;
26  end;
27  /
Cód.: 7499 - Empregado: ALLEN
Cód.: 7521 - Empregado: WARD
Cód.: 7654 - Empregado: MARTIN
Cód.: 7698 - Empregado: BLAKE
Cód.: 7782 - Empregado: CLARK
Cód.: 7839 - Empregado: KING
Cód.: 7844 - Empregado: TURNER
Cód.: 7900 - Empregado: JAMES
Cód.: 7934 - Empregado: MILLER
Cód.: 7935 - Empregado: PAUL
Salário: 1600
Salário: 1250
Salário: 1250
Salário: 2850
Salário: 2450
Salário: 5000
Salário: 1500
Salário: 950
Salário: 1300
Salário: 1000

Procedimento PL/SQL concluído com sucesso.

SQL>
```

Na linha 2, temos a definição de uma variável do tipo `ref cursor` que receberá um comando SQL. Já nas linhas 3 e 4, temos a declaração de duas variáveis do tipo `table`. Esta declaração define dois tipos array, um array numérico e outro de caracteres. Nas linhas 7 e 8, são declaradas duas variáveis com base nestes tipos. Nas linhas 12 e 13, foi utilizado um cursor que lê as informações de um `select` dinâmico que, através do `bulk collect`, carrega as informações para as variáveis do tipo array.

Note que não foi necessário utilizar `loop` para atribuir as linhas vindas do `select`. O `bulk collect` já faz esta atribuição. Nas linhas 16 e 18, utilizamos um `for loop` para ler os dados das variáveis array. Observe que o `for loop` toma como base a quantidade de linhas que o array possui. Neste caso, utilizamos o `count` de uma das variáveis para determinar valor final do `for loop`, mesmo porque carregamos as duas variáveis em um mesmo momento e utilizando o mesmo `select`. Logo, a quantidade de dados carregados nas duas variáveis é a mesma.

Dentro no mesmo programa temos outra execução dinâmica (linhas 20 a 21), onde é possível usar o `bulk collect` também para receber o resultado vindo através da execução via `execute immediate`.

Por último, um exemplo de `function` para retornar a quantidade de registros em uma determinada tabela, passada por parâmetro, usando `execute immediate` com retorno utilizando `into`.

```
SQL>
create function row_count(tab_name varchar2) return integer as
  2    rows integer;
  3   begin
  4     execute immediate 'select count(*) from ' ||
           tab_name into rows;
  5     return rows;
  6   end;
  7   /

Função criada.

SQL> select row_count('EMP') from dual;

ROW_COUNT('EMP')
----------------
              10

SQL>
```

Capítulo 24

Apêndice: SQL – Primeiros passos

24.1 Como iniciar no SQL

Uma das dificuldades que encontramos quando estamos aprendendo uma linguagem de programação é saber por onde começar. No caso do aprendizado da SQL, não seria diferente. É uma situação normal. Até sentirmos segurança e termos conhecimento suficiente sobre a linguagem, é interessante termos um roteiro contendo os primeiros passos para iniciar um programa ou comando SQL.

Desta forma, demonstro aqui uma técnica que utilizo bastante, mesmo tendo um bom conhecimento na linguagem. Na verdade, já está implícito na minha forma de pensar, tanto que acabo executando-a mentalmente, en-

quanto escrevo nesta linguagem. Este método não foi retirado de nenhum livro, foi algo que, entendendo a lógica, fui seguindo e deu certo. Espero que ajude vocês.

Vamos tomar como exemplo a questão a seguir:

Escreva um `select` que apresente as colunas, nome da região (`REGION_NAME`: tabela `REGIONS`), nome da cidade (`CITY`: tabela `LOCATIONS`), nome do departamento (`DEPARTMENT_NAME`: tabela `DEPARTMENTS`) e o nome dos empregados (`FIRST_NAME`), mas somente daqueles que possuam o salário maior que 11.000. Ordene os dados pelos mesmos campos do `select`, seguindo a mesma ordem do enunciado.

Primeiro passo

Identifico no enunciado as fontes de dados, ou seja, as tabelas que farão parte do comando SQL.

Neste nosso exemplo, as tabelas são: REGIONS, COUNTRIES, LOCATIONS, DEPARTMENTS e EMPLOYEES, pois o enunciado nos diz isto. Contudo, se ele não diz os nomes das tabelas, pelo menos deve mencionar do que se tratam tais informações, por exemplo, empregados, departamentos, países etc.

Se você tiver o *MER* (Modelo Entidade Relacionamento, veja anexos 26) fica mais fácil, pois nele você consegue verificar a localização de tais informações referentes aos nomes das tabelas ou os nomes dos campos os quais o enunciado pede. Exemplo: REGION_NAME, CITY etc.

Caso não tenha o MER, você pode localizar tais tabelas no banco de dados, executando um `select` na `user_tables` ou `all_tables` para tentar localizá-las através dos seus nomes. Tendo as localizado, você pode executar o comando `desc` em cima de cada uma delas para visualizar suas colunas e identificar as informações solicitadas. Veja o exemplo a seguir:

```
SQL> select table_name
  2  from   user_tables;

TABLE_NAME
------------------------------
EMP
```

```
DEPT
BONUS
SALGRADE
DUMMY
REGIONS
LOCATIONS
DEPARTMENTS
JOBS
EMPLOYEES
JOB_HISTORY
COUNTRIES

23 linhas selecionadas.

SQL>
```

Caso você queira ver se há comentários sobre tais tabelas, por exemplo, indicando sua finalidade, você pode executar um `select` na `user_tab_comments` ou `all_tab_comments`, pesquisando pelo nome da tabela. Lembrando que utilizando a tabela com prefixo `all` você deve informar a que usuário, ou melhor, a qual dono, a tabela pertence. Veja o exemplo:

```
SQL> select comments
  2  from   user_tab_comments
  3  where  table_name = 'EMPLOYEES';

COMMENTS
----------------------------------------------------------------
employees table. Contains 107 rows. References with departments,
jobs, job_history tables. Contains a self reference.


SQL>
```

Segundo passo

Conectado ao banco de dados, através de uma ferramenta, exemplo SQL*Plus, faço um breve reconhecimento das tabelas envolvidas no enun-

ciado.

Seguindo o exemplo, vamos analisar através do comando `desc` as tabelas e colunas que identificamos, como as tabelas EMPLOYEES e DEPARTMENTS.

```
SQL> desc employees
 Nome                                      Nulo?    Tipo
 ----------------------------------------- -------- -------------
 EMPLOYEE_ID                               NOT NULL NUMBER(6)
 FIRST_NAME                                         VARCHAR2(20)
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 EMAIL                                     NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                       VARCHAR2(20)
 HIRE_DATE                                 NOT NULL DATE
 JOB_ID                                    NOT NULL VARCHAR2(10)
 SALARY                                             NUMBER(8,2)
 COMMISSION_PCT                                     NUMBER(2,2)
 MANAGER_ID                                         NUMBER(6)
 DEPARTMENT_ID                                      NUMBER(4)
 BIRTH_DATE                                         DATE
 HIRE_DATE_BKP                                      DATE

SQL> desc departments
 Nome                                      Nulo?    Tipo
 ----------------------------------------- -------- -------------
 DEPARTMENT_ID                             NOT NULL NUMBER(4)
 DEPARTMENT_NAME                           NOT NULL VARCHAR2(30)
 MANAGER_ID                                         NUMBER(6)
 LOCATION_ID                                        NUMBER(4)

SQL>
```

Terceiro passo

Inicio a escrita do comando SQL, preenchendo a cláusula `select` com as colunas que devo mostrar, e a cláusula `from` informando às tabelas que vou precisar para recuperar todas as informações.

```
select r.region_name
```

```
       ,l.city
       ,d.department_name
       ,e.first_name
from    regions      r
       ,countries    c
       ,locations    l
       ,departments  d
       ,employees    e
```

Note que nem sempre são mencionadas no enunciado todas as tabelas de que precisaremos para retornar todas as informações. Desta forma, precisamos consultar novamente o MER, para entender quais são todas as ligações e de quais objetos vamos necessitar. Vale salientar que entre as tabelas, na maioria dos casos, existe uma ou mais colunas que são as responsáveis pelas ligações entre elas. Necessitamos fazer estas ligações para que o resultado retornado seja o resultado correto, e a integridade seja mantida.

Outro fator importante: sempre utilizo apelidos (*aliasl*) para identificar as tabelas e colunas em um comando SQL. Isso serve para evitar erros de colunas ambíguas (colunas com nomes iguais entre as tabelas, pois, neste caso, o Oracle não sabe dizer quem é quem), e para tornar a leitura do comando SQL mais inteligível.

Quarto passo

Fazer as ligações entre as tabelas que estamos utilizando na cláusula from.

```
10  where r.region_id      = c.region_id
11  and   c.country_id     = l.country_id
12  and   l.location_id    = d.location_id
13  and   d.department_id = e.department_id
```

Caso não se tenha o MER, podemos fazer um `select` nas views `user_constraints` e `user_cons_columns`, para tentar localizar as ligações entre as tabelas. Exemplo:

```
SQL> select cons.table_name||'.'||cons_col.column_name||
  2          ' faz ligação com '||
```

```
 3          cons_depend.table_name||'.
            '||cons_col_depend.column_name||' '||
 4          'através da chave '||cons.constraint_name "
            Dependências"
 5   from   -- tabela pesquisa
 6          user_constraints   cons
 7         ,user_cons_columns  cons_col
 8          -- tabela dependencia
 9         ,user_constraints   cons_depend
10         ,user_cons_columns  cons_col_depend
11   where  cons.constraint_name = cons_col.constraint_name
12   and    cons.table_name      = cons_col.table_name
13   and    cons.table_name      =
              'EMPLOYEES' -- tabela para pesquisa
14   and    cons.constraint_type =
              'R' -- Foreign key (ligação entre as tabelas)
15   --
16   and    cons_depend.constraint_name =
              cons_col_depend.constraint_name
17   and    cons_depend.table_name      =
              cons_col_depend.table_name
18   and    cons_depend.constraint_name = cons.r_constraint_name
19   order  by 1
20   /

Dependências
-----------------------------------------------------------------
EMPLOYEES.DEPARTMENT_ID faz ligação com
DEPARTMENTS.DEPARTMENT_ID através da chave
EMP_DEPT_FK

EMPLOYEES.JOB_ID faz ligação com JOBS.JOB_ID através da chave
EMP_JOB_FK
EMPLOYEES.MANAGER_ID faz ligação com EMPLOYEES.EMPLOYEE_ID
através da chave EMP_MANAGER_FK

SQL>
```

Observação: Você pode utilizar este `select` sempre que possível.

Guarde-o!

Quinto passo

Colocar as demais críticas (restrições) solicitadas pelo enunciado.

```
14  and     e.salary> 11000
15  order   by r.region_name
16             ,l.city
17             ,d.department_name
18             ,e.first_name;
```

Segue o comando SQL completo:

```
SQL> select r.region_name
  2         ,l.city
  3         ,d.department_name
  4         ,e.first_name
  5  from    regions      r
  6         ,countries    c
  7         ,locations    l
  8         ,departments  d
  9         ,employees    e
 10  where r.region_id      = c.region_id
 11  and c.country_id      = l.country_id
 12  and l.location_id     = d.location_id
 13  and d.department_id = e.department_id
 14  and e.salary> 11000
 15  order  by r.region_name
 16             ,l.city
 17             ,d.department_name
 18             ,e.first_name;
```

Observações: para os demais comandos, como o `insert`, o `delete` e o `update`, podemos seguir o mesmo raciocínio. Primeiramente, identificamos as tabelas, as colunas para as ligações e restrições, se for o caso, e depois escrevemos o comando.

Capítulo 25

Referências bibliográficas

- DATE, C. J. *Introdução a Sistemas de Bancos de Dados*. Rio de Janeiro: Campus, 1991.
- FERNANDES, Lúcia. *Oracle 9i Para Desenvolvedores - Curso Completo*. Rio de Janeiro: Axcel Books, 2002.
- GENNICK, Jonathan, LUERS, Tom. *Aprenda em 21 dias PL/SQL*. 2. ed. Rio de Janeiro: Campus, 2000.
- HEUSER, Carlos Alberto. *Projeto de Banco de Dados*. 4. ed. Porto Alegre: Sagra Luzzatto, 2001.
- LIMA, Adilson da Silva. *Erwin 4.0 Modelagem de Dados*. 2. ed. São Paulo: Érica, 2002.

- LONEY, Kevin et al. *Oracle 9i: O Manual do DBA*. Rio de Janeiro: Campus, 2002.
- MOLINARI, Leonardo. *BTO Otimização da Tecnologia de Negócio*. São Paulo: Érica, 2003.
- ORACLE Database Online Documentation 10g Release 1 (10.1). Disponível em: <http://www.oracle.com/pls/db10g/portal.portal_demo3?selected=1> . Acesso em: 16 mar. 2011.
- Oracle Database New Features Guide 11g, Release 2 (11.2)
- Oracle Database Advanced Application Developer's Guide 11g Release 2 (11.2)
- Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)
- Oracle Database Reference 11g Release 2 (11.2)
- Oracle Database SQL Language Reference 11g Release 2 (11.2)
- Oracle Database Concepts 11g, Release 2 (11.2)
- Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)
- Oracle Database SQL Developer User's Guide Release 3.1

Capítulo 26

Anexos

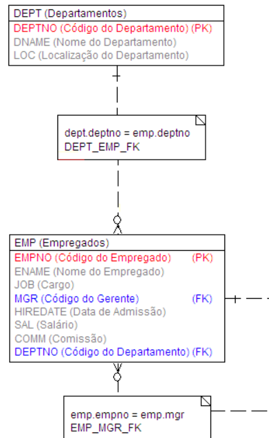

Fig. 26.1: Esquema da base de dados SCOTT utilizada nos exemplos

Schema HR

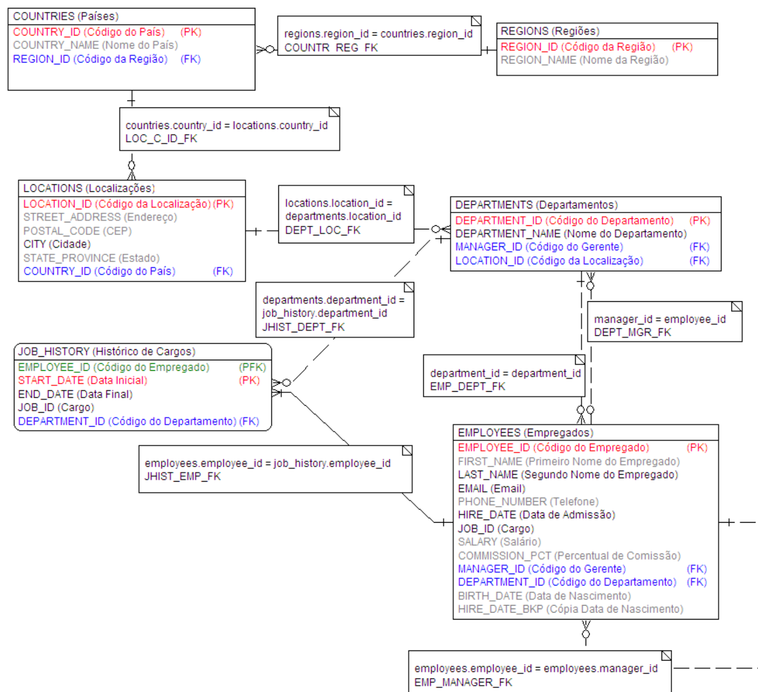

Fig. 26.2: Esquema da base de dados HR utilizada nos exemplos