

PROGRAMANDO EM ORACLE 9i

PL/SQL

Desenvolvida Por:  
Rondinele Santos de Moraes



<b>INTRUÇÃO À LINGUAGEM PL/SQL.....</b>	<b>11</b>
Interagindo com o usuário .....	11
PL/SQL e tráfego de rede .....	11
Estrutura do bloco PL/SQL.....	12
Blocos aninhados .....	15
Identificadores .....	16
Literais .....	16
Comentários .....	16
Declaração de variável.....	17
Tipos PL/SQL .....	17
Character Datatype .....	17
Numeric Datatype .....	17
Date Datatype .....	18
LOB Datatypes .....	18
Outros Datatypes.....	19
Tipo booleano .....	19
Tipos compostos .....	19
Tipos de referencia.....	19
Tipos de objeto.....	19
Utilizando %TYPE .....	19
Subtipos definidos pelo usuário .....	20
Convertendo entre tipos de dados .....	20
Escopo de variável e visibilidade.....	21
Operadores .....	21
Expressões booleanas .....	22
Estruturas de controle PL/SQL .....	22
IF-THEN- ELSE .....	22
CASE .....	23
Loops while.....	24
Loops FOR numéricos .....	25
GOTOS e rótulos .....	26
Rotulando LOOPS.....	26
NULO como uma instrução .....	27
Registros PL/SQL .....	27
Utilizando %ROWTYPE .....	29
<b>SQL DENTRO DA LINGUAGEM PL/SQL .....</b>	<b>30</b>
Select.....	30
Insert .....	30
Update .....	31
Delete .....	31
A cláusula RETURNING .....	32
Referências de tabelas.....	33
Database Links.....	33
Sinônimos .....	33
Controle de transações .....	33
Transações versus blocos .....	34

Transações autônomas .....	34
Privilégios: GRANT e REVOKE .....	34
Roles .....	36
<b>TRATAMENTO DE ERROS .....</b>	<b>37</b>
O que é uma exceção .....	37
Tipos de erros PL/SQL .....	37
Declarando Exceções .....	39
Exceções definidas pelo usuário .....	39
Exceções predefinidas.....	40
Exceções predefinidas pelo Oracle .....	40
Levantando exceções .....	43
Tratando exceções.....	44
O handler de exceção OTHERS .....	46
SQLCODE e SQLERRM .....	47
O pragma EXCEPTION_INIT .....	49
Utilizando RAISE_APPLICATION_ERROR .....	50
Exceções levantadas na seção de exceção .....	52
<b>FUNÇÕES SQL PREDEFINIDAS .....</b>	<b>53</b>
Funções de caractere que retornam valores de caracteres .....	53
CHR (x[using nchar_cs]) .....	53
CONCAT (string1, string2) .....	53
INITCAP (string).....	53
LOWER (string) .....	53
LPAD (String1, x[string2]).....	53
LTRIM (String1,String2).....	54
REPLACE (string, string_a_pesquisar [string_substituta]) .....	54
RPAD (string1, x, [string2]) .....	54
TRANSLATE (string, str_de, str_para).....	54
TRIM([{ {LEADING TRAILING BOTH}[aparar_char] aparar_char}FROM]string)	54
UPPER (string) .....	54
SUBSTR .....	54
SOUNDEX .....	55
Funções de caractere que retornam valores numéricos.....	56
ASCII (string) .....	56
INSTR (string1,string2 [,a] [,b]) .....	56
LENGTH (string).....	56
INSTR .....	56
LENGTH .....	56
Funções de NLS .....	57
CONVERT (string, conjunto_de_caracteres_dest[,conjunto_de_caracteres_orig]).....	57
NCHR(x).....	57
NLS_CHARSET_DECL_LEN .....	57
NLS_CHARSET_ID .....	57
NLS_CHARSET_NAME.....	57
NLS_INITCAP .....	57
NLS_LOWER.....	57
NLS_UPPER .....	57

NLSSORT .....	58
TRANSLATE .....	58
UNISTR(s).....	58
Funções Numéricas.....	58
ABS(x) .....	58
ACOS(x) .....	58
ASIN(x) .....	58
ATAN(x).....	58
ATAN2 (x,y).....	58
BITAND(x,y).....	58
CEIL(x).....	58
COS(x) .....	58
COSH(x) .....	58
EXP(x) .....	59
FLOOR(x).....	59
LN(x) .....	59
LOG (x,y).....	59
MOD(x,y) .....	59
POWER(x,y).....	59
ROUND(x,[y]) .....	59
SIGN(x) .....	59
SIN(x) .....	59
SINH(x) .....	59
SQRT(x).....	59
TAN(x).....	59
TANH(x).....	59
TRUNC (x,[y]).....	59
WIDTH_BUCKET .....	60
Funções de data e hora.....	60
ADD_MONTHS(d,x) .....	60
CURRENT_DATE .....	60
CURRENT_TIMESTAMP.....	61
DBTIMEZONE .....	61
LAST_DAY .....	61
LOCALTIMESTAMP .....	61
MONTHS_BETWEEN .....	61
NEW_TIME (d, zona1,zona2).....	61
NEXTDAY .....	61
ROUND (d[,formato]) .....	61
SESSIONTIMEZONE.....	61
SYS_EXTRACT_UTC.....	61
SYSDATE .....	61
SYSTIMETAMP .....	61
TRUNC (d,[,formato]) .....	61
TZ_OFFSET(fuso horário) .....	62
Funções de conversão .....	62
ASCIIISTR (string) .....	62

BIN_TO_NUM (num[,num]...)	62
CHARTOROWID .....	62
COMPOSE.....	62
DECOMPOSE .....	62
FROM_TZ (timestamp,fuso horário) .....	62
HEXTORAW.....	62
NUMTODSINTERVAL.....	62
NUMTOYMINTEGER .....	62
REFTOHEX.....	62
RAWTOHEX.....	62
RAWTONHEX.....	62
ROWIDTOCHAR .....	63
ROWIDTONCHAR.....	63
TO_CHAR (datas e horas).....	63
TO_CHAR (Números).....	63
TO_DATE .....	64
TO_NUMBER .....	64
TO_TIMESTAMP e TO_TIMESTAMP_TZ.....	64
<b>Funções de grupo .....</b>	<b>65</b>
AVG ([DISTINCT ALL] col).....	65
COUNT (* [DISTINCT ALL]col) .....	65
GROUP_ID() .....	65
GROUPING .....	65
GROUPING_ID.....	65
MAX([DISTINCT ALL]col) .....	65
MIN([DISTINCT ALL]col).....	65
STDDEV ([DISTINCT ALL]col).....	65
SUM([DISTINCT ALL]col).....	66
<b>Outras funções .....</b>	<b>66</b>
BFILENAME (diretório nome_de_arquivo) .....	66
COALESCE .....	66
DECODE (expressão_de_base, comparação1,valor1,comparação2,valor2...padrão) .....	66
EMPTY_BLOB/EMPTY_CLOB .....	66
GREATEST (expr1 [,expr2]).....	66
LEAST (expr1 [,expr2]).....	66
NULLIF (a,b).....	66
NVL (expr1,expr2) .....	66
NVL2 (expr1,expr2,expr3) .....	66
SYS_CONNECT_BY_PATH .....	66
SYS_CONTEXT .....	66
SYS_GUID .....	67
SYS_TYPEID (tipo_de_objecto) .....	67
TREAT (expr AS [REF] [esquema.]tipo ) .....	67
UID .....	67
USER .....	67
DUMP.....	67
<b>CURSORES .....</b>	<b>68</b>

Cursos explícitos.....	69
Parâmetros de cursor.....	70
Atributos de cursor.....	71
Cursos implícitos .....	71
Loops de Busca de Cursos .....	72
Loop Simples .....	73
Loops WHILE.....	73
Loops FOR de cursor.....	74
Loops FOR implícitos.....	75
NO_DATA_FOUND versus %NOTFOUND .....	75
Cursos SELECT FOR UPDATE .....	76
FOR UPDATE .....	76
WHERE CURRENT OF .....	76
COMMIT dentro de um Loop de cursor FOR UPDATE .....	77
Variáveis de cursor .....	78
COLEÇÕES.....	81
Tabelas Index-by.....	81
Elementos inexistentes.....	81
Tabelas index-by de tipos compostos .....	81
Tabelas index-by de registros .....	82
Tabelas index-by de tipos de objeto.....	82
Tabelas aninhadas (Nested tables).....	82
Inicialização de uma tabela aninhada .....	83
Tabelas vazias .....	84
Adicionando elementos a uma tabela existente .....	85
VARRAYS .....	85
Inicialização de varray .....	86
Manipulando os elementos de um varray .....	86
Coleções de múltiplos níveis .....	87
Coleções no banco de dados .....	88
A estrutura de varrays armazenados .....	88
Estrutura das tabelas aninhadas armazenadas.....	89
Manipulando coleções inteiras.....	89
INSERT .....	89
UPDATE.....	90
DELETE .....	91
SELECT .....	91
Operadores de tabela SQL .....	92
Métodos de coleção .....	92
EXISTS .....	93
COUNT.....	93
LIMIT .....	94
FIRST e LAST.....	95
NEXT e PRIOR .....	95
EXTEND .....	96
TRIM .....	97
DELETE .....	98

CRIANDO PROCEDURES, FUNÇÕES E PACOTES .....	101
Procedures e funções .....	101
Criação de subprograma .....	102
Criando uma procedure.....	102
Corpo da procedure.....	102
Criando uma função.....	103
A instrução RETURN .....	104
Eliminando procedures e funções .....	105
Parâmetros de subprograma.....	106
Modo de parâmetro .....	106
Passando valores entre parâmetros formais e reais.....	108
Literais ou constantes como parametros reais .....	109
Lendo de parâmetros OUT .....	110
Restrições quanto aos parâmetros formais.....	110
Parâmetros de %TYPE e de procedure.....	111
Exceções levantadas dentro de subprogramas .....	111
Passando parâmetro por referência e por valor.....	111
Utilizando o NOCOPY .....	111
Semântica de exceção com NOCOPY.....	112
Restrições de NOCOPY .....	112
Os benefícios de NOCOPY .....	112
Subprogramas sem parâmetros .....	115
Notação posicional e identificada .....	115
Valores padrão do parâmetro .....	116
A instrução CALL .....	117
Procedures versus funções .....	118
Pacotes .....	119
Especificação de pacote .....	119
Corpo de pacote .....	120
Pacotes e escopo .....	122
Escopo de objetos no corpo do pacote.....	122
Sobrecarregando subprogramas empacotados .....	125
Inicialização do pacote.....	127
UTILIZANDO PROCEDURES, FUNÇÕES E PACOTES.....	130
Localizações do subprograma.....	130
Subprogramas armazenados e o dicionário de dados .....	130
Compilação nativa .....	130
Subprogramas locais .....	130
Subprogramas locais como parte de subprogramas armazenados .....	131
Localização de subprogramas locais.....	131
Declarações prévias .....	132
Sobrecarregando subprogramas locais.....	132
Subprogramas locais <i>versus</i> armazenados .....	133
Considerações sobre subprogramas e pacotes armazenados .....	133
Recompilação automática .....	133
Pacotes e dependências .....	133
Como as invalidações são determinadas.....	133

Estado em tempo de execução de pacote.....	135
Privilégios e subprogramas armazenados .....	135
Privilégio EXECUTE .....	135
Direito do chamador <i>versus</i> direito do definidor .....	135
Triggers, visualizações e direitos do chamador .....	136
Utilizando funções armazenadas em instruções SQL.....	136
Níveis de pureza para as funções .....	137
Chamando funções armazenadas a partir da SQL no Oracle8i.....	138
Chamando funções a partir de instruções de DML.....	139
Fixando no pool compartilhado .....	140
KEEP .....	140
UNKEEP.....	141
SIZES .....	141
ABORTED_REQUEST_THRESHOLD.....	141
<b>TRIGGERS DE BANCO DE DADOS .....</b>	<b>142</b>
Sintaxe para criação de triggers .....	142
Criando triggers de DML.....	142
Identificadores de correlação em triggers de nível de linha .....	146
Cláusula REFERENCING .....	147
A cláusula WHEN .....	147
Predicados de trigger: INSERTING, UPDATING e DELETING .....	147
Criando triggers Instead-of.....	148
Criando triggers de sistema.....	150
Triggers de banco de dados versus esquema .....	151
Funções do atributo de evento .....	151
Corpos de triggers .....	153
Privilégios de trigger.....	153
Views do dicionário de dados .....	154
Descartando e desativando triggers .....	154
Tabelas que sofrem mutação.....	154
<b>RECURSOS AVANÇADOS .....</b>	<b>158</b>
SQL Dinâmica nativa.....	158
Consultas com EXECUTE IMMEDIATE.....	160
Executando consultas com cursores utilizando OPEN FOR .....	160
Vinculação em volume .....	163
Questões transacionais .....	164
A cláusula BULK COLLECT.....	166
Tipos de objeto.....	168
Armazenando objetos no banco de dados .....	170
Referências de objeto.....	172
Funções de tabela em pipeline .....	173
Pacotes avançados.....	174
DBMS_SQL.....	174
DBMS_PIPE.....	175
DBMS_ALERT .....	176
UTL_FILE .....	177
UTL_TCP .....	178

UTL_SMTP .....	179
UTL_HTTP.....	179
UTL_INADDR .....	180
DBMS_JOB .....	180
DBMS_LOB .....	181

## INTRUDUÇÃO À LINGUAGEM PL/SQL

**A** PL/SQL é uma linguagem de programação sofisticada utilizada para acessar um banco de dados Oracle a partir de vários ambientes. Ela é integrada com o servidor do banco de dados de modo que o código PL/SQL possa ser processado de maneira rápida e eficiente. Essa linguagem também está disponível em algumas ferramentas Oracle do lado do cliente.

Em linhas gerais, a PL/SQL (Procedural Language/SQL) combina o poder e flexibilidade da SQL com as construções procedurais de uma linguagem de 3<sup>a</sup> geração.

### Interagindo com o usuário

A PL/SQL não tem nenhuma funcionalidade de entrada ou de saída construída diretamente na linguagem. Para retificar isso, o SQL\*Plus, em combinação com o pacote DBMS\_OUTPUT, fornece a capacidade de dar saída para mensagens em tela. Isso é feito em dois passos:

1. Permitir a saída no SQL\*Plus com o comando *set serveroutput on* :  
**SET SERVEROUTPUT {ON | OFF} [SIZE n]**

Onde o *n* é o tamanho do buffer de saída. Seu valor padrão é 2.000 bytes

2. Dentro do seu programa PL/SQL, utilize a procedure **DBMS\_OUTPUT.PUT\_LINE(msg)**.

Exemplo:

```
SQL> SET SERVEROUTPUT ON
SQL> begin
 2      DBMS_OUTPUT.PUT_LINE('Teste de pl/sql!!!');
 3  end;
 4 /
Teste de pl/sql!!!
PL/SQL procedure successfully completed.
```

### PL/SQL e tráfego de rede

No modelo cliente/servidor, o próprio programa reside na máquina cliente e envia solicitações de informações para um servidor de banco de dados. As solicitações são feitas utilizando SQL. Em geral, isso resulta em várias viagens pela rede, uma para cada instrução SQL, diferente do uso da PL/SQL que pode estar armazenada no banco de dados ou mesmo permitir que vários comandos SQL sejam empacotados em bloco PL/SQL e enviados ao servidor como uma única unidade.

## Estrutura do bloco PL/SQL

A unidade básica em um programa PL/SQL é um bloco. Todos os programas da PL/SQL são compostos por blocos, que podem ser aninhados dentro do outro. Em geral , cada bloco realiza uma unidade lógica de trabalho no programa, assim separando um do outro diferente tarefas. Um bloco tem a seguinte estrutura:

**DECLARE**

*/\* Seção declarativa – variáveis, tipos, cursores e subprogramas locais \*/*

**BEGIN**

*/\* Seção executável - instruções SQL e procedurais entram aqui. Essa é a principal sessão do bloco PL/SQL, e é a única obrigatória. \*/*

**EXCEPTION**

*/\* Seção de tratamento de exceções – instruções de tratamento de erros entram aqui. \*/*

**END;**

Tipos de blocos PL/SQL:

- Blocos anônimos: construídos dinamicamente e ejecutados apenas uma vez. Podem ser rotulados ou não. Blocos rotulados são geralmente utilizados da mesma maneira que os blocos anônimos, mas os rotulados permitem referenciar variáveis que de outro modo não seriam visíveis.
- Subprogramas: consistem em procedures e funções. Podem ser armazenados no banco de dados como objetos independentes, como parte de um pacote ou como métodos de um tipo de objeto.
- Triggers: consistem em um bloco PL/SQL que está associado a um evento que ocorre no banco de dados.

Exemplos:

```
REM BLOCO ANONIMO NÃO ROTULADO
DECLARE
    /* Declare variables to be used in this block. */
    v_Num1      NUMBER := 1;
    v_Num2      NUMBER := 2;
    v_String1   VARCHAR2(50) := 'Hello World!';
    v_String2   VARCHAR2(50) :=
        '-- This message brought to you by PL/SQL!';
    v_OutputStr VARCHAR2(50);
BEGIN
    /* First, insert two rows into temp_table, using the values
       of the variables. */
    INSERT INTO temp_table (num_col, char_col)
```

```

        VALUES (v_Num1, v_String1);
INSERT INTO temp_table (num_col, char_col)
        VALUES (v_Num2, v_String2);

/* Now query temp_table for the two rows we just inserted,
and
   output them to the screen using the DBMS_OUTPUT package.
*/
SELECT char_col
    INTO v_OutputStr
    FROM temp_table
   WHERE num_col = v_Num1;
DBMS_OUTPUT.PUT_LINE(v_OutputStr);

SELECT char_col
    INTO v_OutputStr
    FROM temp_table
   WHERE num_col = v_Num2;
DBMS_OUTPUT.PUT_LINE(v_OutputStr);

/* Rollback our changes */
ROLLBACK;
END;
/

REM BLOCO ANONIMO ROTULADO
<<1_InsertIntoTemp>>
DECLARE
    /* Declare variables to be used in this block. */
    v_Num1      NUMBER := 1;
    v_Num2      NUMBER := 2;
    v_String1   VARCHAR2(50) := 'Hello World!';
    v_String2   VARCHAR2(50) :=
        '-- This message brought to you by PL/SQL!';
    v_OutputStr VARCHAR2(50);
BEGIN
    /* First, insert two rows into temp_table, using the values
       of the variables. */
    INSERT INTO temp_table (num_col, char_col)
        VALUES (v_Num1, v_String1);
    INSERT INTO temp_table (num_col, char_col)
        VALUES (v_Num2, v_String2);

    /* Now query temp_table for the two rows we just inserted,
and
       output them to the screen using the DBMS_OUTPUT package.
*/

```

```

SELECT char_col
  INTO v_OutputStr
  FROM temp_table
 WHERE num_col = v_Num1;
DBMS_OUTPUT.PUT_LINE(v_OutputStr);

SELECT char_col
  INTO v_OutputStr
  FROM temp_table
 WHERE num_col = v_Num2;
DBMS_OUTPUT.PUT_LINE(v_OutputStr);

/* Rollback our changes */
ROLLBACK;

END l_InsertIntoTemp;
/
REM PROCEDIMENTO
CREATE OR REPLACE PROCEDURE InsertIntoTemp AS
  /* Declare variables to be used in this block. */
  v_Num1      NUMBER := 1;
  v_Num2      NUMBER := 2;
  v_String1   VARCHAR2(50) := 'Hello World!';
  v_String2   VARCHAR2(50) :=
    '-- This message brought to you by PL/SQL!';
  v_OutputStr VARCHAR2(50);
BEGIN
  /* First, insert two rows into temp_table, using the values
     of the variables. */
  INSERT INTO temp_table (num_col, char_col)
    VALUES (v_Num1, v_String1);
  INSERT INTO temp_table (num_col, char_col)
    VALUES (v_Num2, v_String2);

  /* Now query temp_table for the two rows we just inserted,
and
     output them to the screen using the DBMS_OUTPUT package.
*/
  SELECT char_col
    INTO v_OutputStr
    FROM temp_table
   WHERE num_col = v_Num1;
DBMS_OUTPUT.PUT_LINE(v_OutputStr);

SELECT char_col
  INTO v_OutputStr

```

```

        FROM temp_table
        WHERE num_col = v_Num2;
        DBMS_OUTPUT.PUT_LINE(v_OutputStr);

/* Rollback our changes */
ROLLBACK;

END InsertIntoTemp;
/
REM BLOCO ANONIMO PARA CHAMAR A PROCEDURE
BEGIN
    InsertIntoTemp;
END;
/

```

## Blocos aninhados

Um bloco pode ser aninhado dentro da seção executável ou de exceção de um bloco externo, como no exemplo abaixo:

```

DECLARE
    /* Start of declarative section */
    v_StudentID NUMBER(5) := 10000;    -- Numeric variable
    initialized                                         -- to 10,000
    v_FirstName VARCHAR2(20);           -- Variable length
    character string                      -- with maximum length of 20
BEGIN
    /* Start of executable section */
    -- Retrieve first name of student with ID 10,000
    SELECT first_name
        INTO v_FirstName
        FROM students
        WHERE id = v_StudentID;

    -- Start of a nested block, which contains only an
    -- executable
    -- section
    BEGIN
        INSERT INTO log_table (info)
            VALUES ('Hello from a nested block!');
    END;
EXCEPTION
    /* Start of exception section */
    WHEN NO_DATA_FOUND THEN

```

```

-- Start of a nested block, which itself contains an
executable
-- and exception section
BEGIN
    -- Handle the error condition
    INSERT INTO log_table (info)
        VALUES ('Student 10,000 does not exist!');
EXCEPTION
    WHEN OTHERS THEN
        -- Something went wrong with the INSERT
        DBMS_OUTPUT.PUT_LINE('Error inserting into
log_table!');
    END;
END;
/

```

## **Identificadores**

Os identificadores são usados para nomear objetos da PL/SQL como variáveis, cursores, subprogramas, etc. Como regra geral, os identificadores:

- Consistem em uma letra, opcionalmente seguida por qualquer seqüência de caracteres incluindo números e (\$), (#) e (\_).
- Devem possuir comprimento máximo de 30 caracteres
- Não há distinção entre letras maiúscula e minúscula
- Não pode possuir nome igual uma palavra reservada, ex: BEGIN, END
- Podem ser identificados com aspas para possuírem espaços e distinção entre letras maiúsculas e minúsculas. EX: “X / Y”, “variavel A”

## **Literais**

São valores constantes, podendo ser Caracter : (‘Teste literal’), Numérico: (132, -44), ou Booleanos: (TRUE, FALSE, NULL)

## **Comentários**

Melhoram a legibilidade e tornam os programas mais compreensíveis.

- Comentários de uma única linha, através de dois traços “--”:
 

```
v_data := SYSDATE; --variável recebe data atual
```
- Comentários de múltiplas linhas, usando “/\*” para iniciar e “\*/” para fechar:
 

```
BEGIN
        /* Estamos agora dentro de um comentário.
           Aqui é a continuação do comentário.
        */
        NULL;
    END;
```

## Declaração de variável

*Nome\_da\_variável [CONSTANT] datatype [NOT NULL] [{:= / DEFAULT} valor];*

Onde *nome\_da\_variável* é o identificador, *datatype* é tipo e *valor* é o conteúdo inicial da variável. EX:

```
DECLARE
    v_dataInicial DATE;
    v_contador BINARY_INTEGER NOT NULL := 0;
    v_nome VARCHAR2(20);
    c_PI CONSTANT NUMBER DEFAULT 3.14;
```

## Tipos PL/SQL

O Oracle PL/SQL possui diferentes tipos de dados (datatypes) para atender suas necessidades, que são divididos nas seguintes categorias: CHARACTER, NUMBER, DATE, LOB, BOOLEANOS, TIPOS COMPOSTOS, TIPOS DE OBJETO e TIPOS DE REFERÊNCIA.

### Character Datatype

Usados para armazenar dados alfanuméricos.

- **CHAR(<n>)** armazena string de tamanho fixo. Tamanho default 1, máximo 32.767. Subtipo: **CHARACTER**
- **VARCHAR2(<n>)** armazena string de tamanho variável. É possível armazenar string de até 32.767 bytes. Subtipo: **STRING**
- **VARCHAR(<n>)** sinônimo para o tipo VARCHAR2.
- **NCHAR(<n>)** e **NVARCHAR2(<n>)** possuem as mesmas características dos tipos CHAR e VARCHAR2 e são usados para armazenar dados NLS (National Language Support). A arquitetura Oracle NLS permite armazenar, processar e recuperar informações em linguagens nativas.
- **LONG** é um tipo de dados que se tornou “obsoleto” com a chegada dos tipos LOB (Large Object). O tipo LONG armazena strings de tamanho variável de no máximo 32.760 bytes.

### Numeric Datatype

Usado para armazenar dados numéricos com precisão de até 38 dígitos.

- **NUMBER(<x>, <y>)** onde <X> corresponde ao número de dígitos e <Y> o número de casas decimais. Valores inseridos em colunas numéricas com número de casas decimais menor que o dado inserido serão arredondados. Subtipos: **DEC**, **DECIMAL**, **DOUBLE PRECISION**, **FLOAT**, **INTEGER**, **INT**, **NUMERIC**, **REAL**, **SMLLINT**.

- **BINARY\_INTEGER** utilizado para armazenar inteiros com sinal, que variam de -2147483647 a 2147483647. Requerem menos memória que tipos NUMBER. Subtipos: **NATURAL (n>=0)**, **NATURALN (n>=0 not null)**, **POSITIVE (n>0)**, **POSITIVEN (n>0 not null)**, **SIGNTYPE (-1, 0, 1)**.
- **PLS\_INTEGER** Possui as mesmas características do tipo BINARY\_INTEGER, entretanto possui melhor performance para cálculos.

## Date Datatype

O tipo **DATE** permite valores de data e hora. O formato padrão é definido pelo parâmetro **NLS\_DATE\_FORMAT**. O Oracle armazena internamente a data em formato de número juliano com a parte fracionária usada para controlar a hora. Uma data Juliana corresponde ao número de dias desde 1 de Janeiro de 4712 A.C.

Para operações aritméticas com datas no Oracle, basta adicionar ou subtrair números inteiros ou fracionários. Por exemplo, SYSDATE + 1 para somar uma dia, 1/24 para acrescentar uma hora, 1/(24x60) ou 1/1440 para acrescentar 1 minuto e 1/(24x60x60) ou 1/86400 para um segundo.

No Oracle 9i há também 5 novos tipos de dados para armazenar tipos de data:

- **TIMESTAMP** semelhante ao tipo DATE, com a diferença de armazenar fração de segundos com precisão de até 9 dígitos.
- **TIMESTAMP WITH TIME ZONE** armazena data/hora com informações de fuso horário.
- **TIMESTAMP WITH LOCAL TIME ZONE** armazena data/hora no fuso horário do servidor. Quando o usuário seleciona os dados, o valor é ajustado para as configurações da sua sessão.
- **INTERVAL YEAR TO MONTH** usado para armazenar espaço de tempo em anos e meses.
- **INTERVAL DAY TO SECOND** permite especificar intervalos em dias, horas, minutos e segundos.

## LOB Datatypes

Large Object (LOB) datatypes são usados para armazenar dados não estruturados como imagens, arquivos binários. Os tipos LOBs podem armazenar até 4GB de informação. A manipulação dos tipos LOB é feita através do package DBMS\_LOB.

Datatype		Descrição
BLOB	Binary Large Object	Armazena até 4GB de dados binários no banco
CLOB	Character Large Object	Armazena até 4GB de dados caráter
BFILE	Binary File	Armazena até 4GB de dados em arquivos binários externos. Uma coluna BFILE armazena um ponteiro para o arquivo armazenado no sistema operacional.

## Outros Datatypes

- **RAW** é um tipo para dados binários, não interpretados pelo banco. Podem armazenar até 32.767 bytes de informação e seus dados não passam por conversão de conjunto de caracteres entre cliente e servidor.
- **LONGRAW** semelhante ao tipo LONG, é um tipo de dados “obsoleto” que pode armazenar até 32.760 bytes de dados. Seu uso foi depreciado pelos tipos BLOB e BFILE.
- **ROWID** Oracle utiliza o datatype ROWID para armazenar o endereço de cada linha no banco de dados. Toda tabela contém uma coluna oculta chamada ROWID que retorna um identificador único do endereço da linha no banco de dados no formato OOOOOOFFBBBBBRRR onde “O” representa o número do objeto, “F” o número do datafile, “B” a identificação do bloco Oracle e “R” a identificação da linha no bloco.
- **UROWID** Universal ROWID, suporta todos os tipos de ROWID (físicas ou lógicas) bem como de tabelas não Oracle acessadas através de gateway.

## Tipo booleano

O único tipo de dados na família booleana é o BOOLEAN. Variáveis booleanas são utilizadas em estruturas de controle da PL/SQL como as instruções IF-THEN-ELSE e de LOOP. Podem conter apenas os valores TRUE, FALSE ou NULL.

## Tipos compostos

Os tipos compostos disponíveis em PL/SQL são registros, tabelas e varrays. Um tipo composto é um que tem componentes dentro dele. Uma variável do tipo composta contém uma ou mais variáveis.

## Tipos de referencia

Um tipo de referência na PL/SQL é semelhante a um ponteiro em C. Uma variável que seja declarada como um tipo de referência pode apontar para posições de memória diferentes na vida do programa. Em PL/SQL usamos os tipos REF e REF CURSOR.

## Tipos de objeto

Consiste em um tipo composto que possui atributos (variáveis de outros tipos) e métodos (subprogramas) dentro dele.

## Utilizando %TYPE

Utilizado para declarar uma variável com o mesmo tipo de uma coluna de alguma tabela, ex:

```

DECLARE
    v_Nome STUDENTS.FIRST_NAME%TYPE;
    v_Idade PLS_INTEGER NOT NULL :=0;
    v_IdadeTemp v_Idade%TYPE; --não herda restrição nem valor default

```

### Subtipos definidos pelo usuário

Utilizado para fornecer um tipo alternativo para um tipo, descrevendo sua utilização pretendida. Sintaxe:

```
SUBTYPE novo_tipo IS tipo_original;
```

Exemplo:

```

DECLARE
    SUBTYPE T_Contador IS NUMBER(3);
    v_Contador T_Contador;
    SUBTYPE T_Nome IS STUDENTS.FIRST_NAME%TYPE;

```

### Convertendo entre tipos de dados

Quando possível, a PL/SQL converterá automaticamente (conversão implícita) tipos dentro da mesma família e entre as famílias de tipos de dados:

- Caracteres e números
- Caracteres e datas

Funções para conversão explícita de tipos de dados:

Função	Descrição
TO_CHAR	Converte seu argumento em um tipo VARCHAR2
TO_DATE	Converte seu argumento em um tipo DATE
TO_TIMESTAMP	Converte seu argumento em um tipo TIMESTAMP
TO_TIMESTAMP_TZ	Converte seu argumento em um tipo TIMESTAMP WITH TIMEZONE
TO_DSINTERVAL	Converte seu argumento em um tipo INTERVAL DAY TO SECOND
TO_YMINTERVAL	Converte seu argumento em um tipo INTERVAL YEAR TO MONTH
TO_NUMBER	Converte seu argumento em um tipo NUMBER
HEXTORAW	Converte uma representação hexadecimal na quantidade binária equivalente

Função	Descrição
RAWTOHEX	Converte um valor RAW em uma representação hexadecimal da quantidade binária
CHARTOROWID	Converte uma representação de caractere de um ROWID em formato binário interno
ROWIDTOCHAR	Converte uma variável binária interna de ROWID em formato externo de 18 caracteres

## Escopo de variável e visibilidade

O escopo de uma variável é a parte do programa onde a variável pode ser acessada antes de ser liberada da memória.

A visibilidade de uma variável é a parte do programa onde a variável pode ser acessada sem ter de qualificar a referência.

```
<<bloco1>>
DECLARE
    v_flag BOOLEAN;
    v_SSN NUMBER(9);
BEGIN
    /* Aquí v_flag e v_SSN estão visíveis */

    DECLARE
        v_Data DATE;
        v_SSN CHAR(11);
    BEGIN
        /* Aquí v_flag, v_Data e v_SSN CHAR(11) estão visíveis. Para acessar v_SSN NUMBER(9) é necessário informar o bloco a que pertence: bloco1.v_SSN */
    END;

    /* Aquí v_flag e v_SSN estão visíveis */

END;
```

## Operadores

O operador mais básico na PL/SQL é o de atribuição. A sintaxe é:

*Variável := valor;*

Veja a tabela abaixo com os operadores PL/SQL:

Operador	Tipo	Descrição
<code>**, NOT</code>	Binário	Exponenciação, negação lógica
<code>+, -</code>	Unário	Identidade, negação
<code>*, /</code>	Binário	Multiplicação, divisão
<code>+, -,   </code>	Binário	Adição, subtração, concatenação
<code>=, !=, &lt;, &gt;, &lt;=, &gt;=, IS NULL, LIKE, BETWEEN, IN</code>	Binário (exceto IS NULL que é unário)	Comparação lógica
<code>AND</code>	Binário	Conjunção lógica
<code>OR</code>	Binário	Inclusão lógica

## Expressões booleanas

Todas as estruturas de controle PL/SQL envolvem expressões booleanas, também conhecidas como condições. Uma expressão booleana é qualquer expressão que é avaliada como um valor booleano (TRUE, FALSE ou NULL).

Três operadores (AND, OR, NOT) recebem argumentos booleanos e retornam valores booleanos. Seus comportamentos são descritos na tabela abaixo:

<b>NOT</b>	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
	FALSE	TRUE	NULL

<b>AND</b>	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	FALSE	FALSE	NULL
<i>FALSE</i>	FALSE	FALSE	FALSE
<i>NULL</i>	NULL	FALSE	NULL

<b>OR</b>	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	TRUE	TRUE	TRUE
<i>FALSE</i>	TRUE	FALSE	NULL
<i>NULL</i>	TRUE	NULL	NULL

## Estruturas de controle PL/SQL

Permitem controlar o comportamento do bloco à medida que ele está sendo executado.

### IF-THEN- ELSE

```

DECLARE
    v_NumberSeats rooms.number_seats%TYPE;
    v_Comment VARCHAR2(35);
BEGIN
    SELECT number_seats
    INTO v_NumberSeats
    FROM rooms
    WHERE room_id = 20008;

```

```

IF v_NumberSeats < 50 THEN
    v_Comment := 'Fairly small';
ELSIF v_NumberSeats < 100 THEN
    v_Comment := 'A little bigger';
ELSE
    v_Comment := 'Lots of room';
END IF;
END;
/

```

## CASE

```

DECLARE
    v_Major students.major%TYPE;
    v_CourseName VARCHAR2(10);
BEGIN
    SELECT major
        INTO v_Major
        FROM students
        WHERE ID = 10011;

    CASE v_Major
        WHEN 'Computer Science' THEN
            v_CourseName := 'CS 101';
        WHEN 'Economics' THEN
            v_CourseName := 'ECN 203';
        WHEN 'History' THEN
            v_CourseName := 'HIS 101';
        WHEN 'Music' THEN
            v_CourseName := 'MUS 100';
        WHEN 'Nutrition' THEN
            v_CourseName := 'NUT 307';
        ELSE
            v_CourseName := 'Unknown';
    END CASE;

    DBMS_OUTPUT.PUT_LINE(v_CourseName);
END;
/

```

Quando uma condição não for validada por uma condição CASE, a PL/SQL interromperá a execução com o erro CASE\_NOT\_FOUND, ou ORA-6592:

```

SQL> DECLARE
  2      v_TestVar NUMBER := 1;
  3  BEGIN
  4      -- Não haverá teste para condição = 1

```

```

5      -- this will raise ORA-6592.
6      CASE v_TestVar
7          WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Two!');
8          WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Three!');
9          WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Four!');
10     END CASE;
11    END;
12  /
DECLARE
*
ERROR at line 1:
ORA-06592: CASE not found while executing CASE statement
ORA-06512: at line 6

```

Devemos então utilizar a cláusula ELSE:

```

SQL> DECLARE
2      v_TestVar NUMBER := 1;
3      BEGIN
4          -- This CASE statement is labeled.
5          <<MyCase>>
6          CASE v_TestVar
7              WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Two!');
8              WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Three!');
9              WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Four!');
10             ELSE DBMS_OUTPUT.PUT_LINE('One!');
11         END CASE MyCase;
12     END;
13  /
One!

PL/SQL procedure successfully completed.

```

## Loops while

A condição é avaliada antes de cada iteração do loop. Sintaxe:

```

WHILE condição LOOP
    Seqüência_de_instruções;
END LOOP;

```

Exemplo:

```

DECLARE
    v_Counter BINARY_INTEGER := 1;
BEGIN
    -- Test the loop counter before each loop iteration to

```

```

-- insure that it is still less than 50.
WHILE v_Counter <= 50 LOOP
    INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
    v_Counter := v_Counter + 1;
END LOOP;
END;
/

```

Se desejado, as instruções **EXIT** ou **EXIT WHEN** podem ainda ser utilizadas dentro de um loop WHILE para sair prematuramente do loop.

Tenha em mente que se a condição de loop não for avaliada como TRUE na primeira vez que ela for verificada, o loop não será executado.

Exemplo:

```

DECLARE
    v_Counter BINARY_INTEGER;
BEGIN
    -- This condition will evaluate to NULL, since v_Counter
    -- is initialized to NULL by default.
    WHILE v_Counter <= 50 LOOP
        INSERT INTO temp_table
            VALUES (v_Counter, 'Loop index');
        v_Counter := v_Counter + 1;
    END LOOP;
END;
/

```

## Loops FOR numéricos

Loop que possui um número definido de iterações. Sintaxe:

```

FOR contador IN [REVERSE] limite_inferior .. limite_superior LOOP
    Seqüência_de_instruções;
END LOOP;

```

Exemplo:

```

BEGIN
    --não é necessário declarer a variável v_counter
    --ela será automaticamente declarada como BINARY_INTEGER
    FOR v_Counter IN 1..50 LOOP
        INSERT INTO temp_table
            VALUES (v_Counter, 'Loop Index');
    END LOOP;
END;

```

Se a palavra REVERSE estiver presente no loop FOR, o índice de loop irá iterar a partir do valor alto para o valor baixo. Exemplo:

```
DECLARE
    v_Baixo NUMBER := 10;
    v_Alto   NUMBER := 40;
BEGIN
    FOR v_Counter in REVERSE v_Baixo .. v_Alto LOOP
        INSERT INTO temp_table VALUES(v_Counter, 'Teste');
    END LOOP;
END;
/
```

## GOTOS e rótulos

A linguagem PL/SQL também possui uma instrução GOTO para passar o controle para uma área específica do bloco. A PL/SQL não permite utilizar GOTO para desviar o controle para um bloco interno, para dentro de uma condição IF ou sair da seção de exceção do bloco para uma seção executável.

Exemplo:

```
DECLARE
    v_Counter BINARY_INTEGER := 1;
BEGIN
    LOOP
        INSERT INTO temp_table
            VALUES (v_Counter, 'Loop count');
        v_Counter := v_Counter + 1;
        IF v_Counter >= 50 THEN
            GOTO 1_EndOfLoop;
        END IF;
    END LOOP;

    <<1_EndOfLoop>>
    INSERT INTO temp_table (char_col)
        VALUES ('Done!');
END;
/
```

## Rotulando LOOPS

Os próprios Loops podem ser rotulados. Podemos rotular um loop e utilizá-lo em uma instrução EXIT para indicar qual loop deve ser terminado. Exemplo:

```

BEGIN
    <<loop_pai>>
    FOR v_Pai IN 1..50 LOOP
        ...
        <<loop_filho>>
        FOR v_filho IN 2..10 LOOP
            ...
            if v_Pai > 40 then
                EXIT loop_pai; --termina ambos os loops
            end if;
            END LOOP loop_filho;
        END LOOP loop_pai;
    END;

```

## NULO como uma instrução

Em alguns casos, você talvez queira indicar explicitamente que nenhuma ação deve acontecer. Isso pode ser feito via a instrução NULL. Exemplo:

```

DECLARE
    v_TempVar  NUMBER := 7;
BEGIN
    IF v_TempVar < 5 THEN
        INSERT INTO temp_table (char_col)
        VALUES ('Too small');
    ELSIF v_TempVar < 20 THEN
        NULL; -- Do nothing
    ELSE
        INSERT INTO temp_table (char_col)
        VALUES ('Too big');
    END IF;
END;
/

```

## Registros PL/SQL

Os registros PL/SQL são semelhantes a estruturas C. Um registro fornece uma maneira de lidar com variáveis separadas, mas relacionadas como uma unidade. Sintaxe:

```

TYPE nome_tipo IS RECORD(campo1 tipo1 [NOT NULL] [:= valor],
                           campo2 tipo2 [NOT NULL] [:= valor], ...);

```

Exemplo:

```

DECLARE
    TYPE t_Rec1Type IS RECORD (
        Field1 NUMBER,

```

```

        Field2 VARCHAR2(5));
TYPE t_Rec2Type IS RECORD (
    Field1 NUMBER,
    Field2 VARCHAR2(5));
v_Rec1 t_Rec1Type;
v_Rec2 t_Rec2Type;
v_Rec3 t_Rec1Type;
v_Rec4 t_Rec1Type;
BEGIN
    /*Apesar de possuírem a mesma estrutura, a atribuição
abaixo não é permitida */
    v_Rec1 := v_Rec2;

    --A forma correta seria:
    v_Rec1.Field1 := v_Rec2.Field1;
    v_Rec2.Field2 := v_Rec2.Field2;

    /*Essa atribuição é permitida pois ambas variáveis são do
mesmo tipo */

    v_Rec3 := v_Rec4;

END;
/

```

Um registro também pode ser atribuído para uma instrução SELECT que retorna vários campos desde que seus atributos estejam na mesma ordem. Exemplo:

```

DECLARE
    TYPE t_StudentRecord IS RECORD (
        FirstName    students.first_name%TYPE,
        LastName     students.last_name%TYPE,
        Major        students.major%TYPE);

        v_Student    t_StudentRecord;
BEGIN
    SELECT first_name, last_name, major
        INTO v_Student
        FROM students
        WHERE ID = 10000;
END;
/

```

O Oracle 9i também permite que os registros sejam utilizados nas instruções INSERT e UPDATE.
--

## Utilizando %ROWTYPE

É comum declarar um registro com os mesmos tipos e nomes das colunas de uma tabela do banco de dados. Semelhante ao %TYPE, %ROWTYPE retornará um tipo com base na definição de tabela. Exemplo:

```
DECLARE
    v_Student  students%ROWTYPE;
BEGIN
    SELECT first_name, last_name, major
        INTO v_Student
        FROM students
        WHERE ID = 10000;
END;
/
```

Como com %TYPE, qualquer restrição NOT NULL definida na coluna não é copiada para o registro. Entretanto, o comprimento das colunas de VARCHAR2 e CHAR, a precisão e a escala para as colunas de NUMBER são copiadas.

Se a definição da tabela mudar no banco de dados, %ROWTYPE também mudará. %TYPE e %ROWTYPE são avaliados toda vez que um bloco anônimo for submetido para o mecanismo PL/SQL.

## SQL DENTRO DA LINGUAGEM PL/SQL

A linguagem SQL (Structure Query Language) define como os dados do Oracle são manipulados. As construções procedurais com PL/SQL tornam-se mais úteis quando combinadas com o poder de processamento da SQL, permitindo que os programas manipulem os dados no Oracle.

As únicas instruções SQL permitidas diretamente em um programa PL/SQL são DMLs (SELECT, INSERT, UPDATE, DELETE) instruções de controle de transação (COMMIT, ROLLBACK, SAVEPOINT...).

### Select

Recupera os dados do banco de dados para as variáveis PL/SQL.

Exemplo:

```
DECLARE
    v_StudentRecord    students%ROWTYPE;
    v_Department       classes.department%TYPE;
    v_Course           classes.course%TYPE;
BEGIN
    SELECT *
        INTO v_StudentRecord
        FROM students
        WHERE id = 10000;

    SELECT department, course
        INTO v_Department, v_Course
        FROM classes
        WHERE room_id = 20003;
END;
/
```

### Insert

Insere novas linhas na tabela a partir de variáveis, registros, subquerys, etc.

Exemplo:

```
DECLARE
    v_StudentID    students.id%TYPE;
BEGIN
    SELECT student_sequence.NEXTVAL
        INTO v_StudentID
        FROM dual;

    INSERT INTO students (id, first_name, last_name)
```

```

        VALUES (v_StudentID, 'Timothy', 'Taller');

    INSERT INTO students (id, first_name, last_name)
        VALUES (student_sequence.NEXTVAL, 'Patrick', 'Poll');
END;
/

```

## Update

Atualiza colunas das tabelas a partir de variáveis, subquerys, registros, etc.

Exemplo:

```

DECLARE
    v_Major          students.major%TYPE;
    v_CreditIncrease NUMBER := 3;
BEGIN
    v_Major := 'History';
    UPDATE students
        SET current_credits = current_credits + v_CreditIncrease
        WHERE major = v_Major;

    UPDATE temp_table
        SET num_col = 1, char_col = 'abcd';
END;
/

```

## Delete

Remove linhas de uma tabela do banco de dados.

Exemplo:

```

DECLARE
    v_StudentCutoff  NUMBER;
BEGIN
    v_StudentCutoff := 10;
    DELETE FROM classes
        WHERE current_students < v_StudentCutoff;

    DELETE FROM students
        WHERE current_credits = 0
        AND   major = 'Economics';
END;
/

```

## A cláusula RETURNING

Utilizada para obter as informações sobre a linha ou linhas que acabaram de ser processadas por um comando DML, como por exemplo conhecer a ROWID da linha que acabou de ser incluída sem a necessidade de submeter um comando SELECT para o banco de dados.

Exemplo:

```
set serveroutput on

DECLARE
    v_NewRowid ROWID;
    v_FirstName students.first_name%TYPE;
    v_LastName students.last_name%TYPE;
    v_ID students.ID%TYPE;
BEGIN
    INSERT INTO students
        (ID, first_name, last_name, major, current_credits)
    VALUES
        (student_sequence.NEXTVAL,           'Xavier',      'Xemes',
        'Nutrition', 0)
    RETURNING rowid INTO v_NewRowid;

    DBMS_OUTPUT.PUT_LINE('Newly inserted rowid is ' || v_NewRowid);

    UPDATE students
        SET current_credits = current_credits + 3
        WHERE rowid = v_NewRowid
    RETURNING first_name, last_name INTO v_FirstName,
v_LastName;

    DBMS_OUTPUT.PUT_LINE('Name: ' || v_FirstName || ' ' || v_LastName);

    DELETE FROM students
        WHERE rowid = v_NewRowid
    RETURNING ID INTO v_ID;

    DBMS_OUTPUT.PUT_LINE('ID of new row was ' || v_ID);
END;
/
```

## Referências de tabelas

Todas as operações DML referenciam uma tabela. Em geral, essa referência se parece com:

```
[esquema].tabela[@dblink]
```

onde *esquema* identifica o proprietário da tabela e *dblink* é um link para um banco de dados remoto.

## Database Links

Um link de banco de dados é uma referência para um banco de dados remoto, que pode estar localizado em um sistema completamente diferente do banco de dados local. A sintaxe para criação de um Database Link é a seguinte:

```
CREATE DATABASE LINK nome_do_link
  CONNECT TO nome_do_usuário IDENTIFIED BY senha_do_usuário
  USING string_do_sqlnet;
```

Exemplo:

```
CREATE DATABASE LINK filial2
  CONNECT TO scott IDENTIFIED BY tiger
  USING 'ORACURSO2';
```

```
UPDATE emp@FILIAL2
  SET sal = 2050.10
 WHERE empno = 1005;
```

## Sinônimos

Permite criar um *alias* para a referência de uma tabela. Sintaxe:

```
CREATE SYNONYM nome FOR referência;
```

Exemplo:

```
CREATE SYNONYM emp2 FOR emp@FILIAL2;
```

## Controle de transações

Uma transação é uma série de instruções SQL que podem ser aplicadas com sucesso, falhas ou canceladas. Os comandos para controlar transações são:

- COMMIT [WORK] para confirmar uma transação
- ROLLBACK [WORK] [TO SAVEPOINT x] para cancelar parte ou toda uma transação

- SAVEPOINT para criar um ponto de salvamento na transação
- SET TRANSACTION para alterar o modo de consistência de leitura de uma transação

## Transações versus blocos

Devemos observar a distinção entre transações e blocos PL/SQL. Quando um bloco inicia, não significa que uma transação inicia. Da mesma forma, o início de uma transação não precisar coincidir com o início de um bloco.

## Transações autônomas

Permite que determinadas operações SQL sejam confirmadas independente do restante das operações de um bloco. Uma transação autônoma é iniciada dentro do contexto de uma outra transação podendo ser confirmada ou cancelada independentemente do estado da transação que a invocou.

Um bloco PL/SQL é marcado como autônomo utilizando um pragma (diretivas de um compilador) na seção declarativa como no exemplo abaixo:

```
CREATE OR REPLACE PROCEDURE autonomous AS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO temp_table VALUES(-10, 'Hello');
  COMMIT;
END autonomous;
/

--bloco para chamar procedure autônoma
BEGIN
  INSERT INTO temp_table VALUES(-20, 'Hello 2');

  Autonomous; --chama procedure autônoma
  ROLLBACK; --desfaz apenas a transação atual (-20)
END;
/
```

## Privilégios: GRANT e REVOKE

Para realizar operações como INSERT ou DELETE em uma tabela Oracle, você precisa de permissão. Essas permissões são manipuladas via comandos GRANT e REVOKE da linguagem SQL.

Há dois tipos diferentes de privilégios: objeto e sistema. Um privilégio de objeto permite uma operação em um objeto em particular (tabela, view, sequence...). Um privilégio de sistema permite operações em uma classe inteira de objetos.

## Principais privilégios de objeto SQL

Privilégio	Tipos de objetos	Descrição
ALTER	Tabelas, sequences	Permite ALTER TABLE no objeto
DELETE	Tabelas, views	Permite DELETE contra o objeto
EXECUTE	Procedures, funções e packages	Permite executar um objeto PL/SQL armazenado
INDEX	Tabelas	Permite criar índice na tabela especificada
INSERT	Tabelas, views	Permite inserir linhas no objeto
READ	Diretórios	Permite ler a partir do diretório especificado
REFERENCES	Tabelas	Permite criar FKs para referenciar a tabela
SELECT	Tabelas, views, sequences	Permite selecionar dados do objeto
UPDATE	Tabelas, views	Permite atualizar linhas do objeto

Você poderá conhecer todos os privilégios de objeto através da tabela de sistema **TABLE\_PRIVILEGE\_MAP** e os privilégios de sistema através da tabela **SYSTEM\_PRIVILEGE\_MAP**.

### Grant

A instrução GRANT é utilizada para permitir algum privilégio para determinado usuário. A sintaxe é a seguinte para permissões de objeto:

*GRANT privilégio ON objeto TO usuário/role [WITH GRANT OPTION];*

Já para permissões de sistema:

*GRANT privilégio TO usuário/role [WITH ADMIN OPTION];*

Exemplo:

```
CONNECT ALUNO3/SENHA3@ORACURSO;
GRANT UPDATE, DELETE ON classes TO aluno2;
GRANT ALL ON empregado TO scott;
GRANT SELECT ON classes TO aluno8 WITH GRANT OPTION;
```

```
CONNECT ALUNO8/SENHA8@ORACURSO;
GRANT SELECT ON aluno3.classes TO aluno9;
```

Como WITH GRANT OPTION foi especificada, então aluno8 poderá repassar o privilégio para outro usuário. Caso aluno8 perca o privilégio, aluno9 também o perderá. Isso não vale para WITH ADMIN OPTION, pois o privilégio continuará concedido.

## Revoke

A instrução REVOKE revoga um privilégio concedido a um usuário. Sintaxe:

REVOKE *privilegio* ON *objeto* FROM *usuário/role* [CASCADE CONSTRAINTS];

Exemplo:

```
REVOKE SELECT ON classes FROM aluno8;
```

Se a cláusula CASCADE CONSTRAINTS foi incluída e o privilégio REFERENCES estiver sendo revogado, todas as restrições de integridade referencial criadas pelo usuário que detinha o privilégio serão eliminadas.

## Roles

Permite facilitar a administração dos privilégios. Consiste em uma coleção de privilégios, tanto de objetos quanto de sistema. Exemplo:

```
CREATE ROLE pesquisa_tabelas;

GRANT SELECT ON students TO pesquisa_tabelas;
GRANT SELECT ON classes TO pesquisa_tabelas;
GRANT SELECT ON rooms TO pesquisa_tabelas;

GRANT pesquisa_tabelas TO aluno7;
GRANT pesquisa_tabelas TO PUBLIC;
```

A role PUBLIC é internamente definida pelo Oracle e é concedida automaticamente para todos os usuários. Portanto, ao conceder algum privilégio para PUBLIC você estará concedendo para todos os usuários do seu banco de dados de uma vez.

As roles mais comuns predefinidas pelo Oracle são: CONNECT, RESOURCE, DBA, EXP\_FULL\_DATABASE e IMP\_FULL\_DATABASE.

## TRATAMENTO DE ERROS

A PL/SQL implementa tratamento de erro por meio de exceções e handlers de exceção. As exceções podem estar associadas com erros do Oracle ou com os próprios erros definidos pelo usuário. Neste capítulo, discutiremos a sintaxe para exceções e handlers de exceção, como as exceções são levantadas e tratadas, e as regras de propagação de exceção. O capítulo termina com as diretrizes sobre a utilização das exceções.

### O que é uma exceção

A PL/SQL está baseada na linguagem Ada. Um dos recursos Ada, que também foi incorporado na PL/SQL, é o mecanismo de exceção. Utilizando as exceções e handlers de exceção, você pode tornar seus programas PL/SQL mais poderosos e capazes de lidar durante a execução tanto com erros esperados como inesperados. As exceções PL/SQL também são semelhantes a exceções em Java. Por exemplo, exceções em Java são lançadas e capturadas da mesma maneira como na PL/SQL. Entretanto, diferente do que ocorre em Java, as exceções PL/SQL não são objetos e não têm nenhum método para defini-las. As exceções são projetadas para tratamento de erros em tempo de execução, em vez de tratamento de erros na compilação. Os erros que ocorrem durante a fase de compilação são detectados pelo mecanismo PL/SQL e informados ao usuário. O programa não pode tratar esses erros, pelo fato de o programa ainda não ter sido executado. Por exemplo, o seguinte bloco levantará o erro de compilação:

- PLS-201: Identifier ‘SSTUDENTS’ must be declared

```
DECLARE
    v_NumStudents NUMBER;
BEGIN
    SELECT count(*)
        INTO v_NumStudents
        FROM students;
END;
```

### Tipos de erros PL/SQL

Tipo de Erro	Informado pelo	Como é tratado
Na compilação	Compilador PL/SQL	Interativamente – o compilador informa os erros e você tem de corrigi-los.
Em tempo de execução	Mecanismo de Execução da PL/SQL	Programaticamente - as exceções são levantadas e capturadas pelos handlers de exceção.

As exceções e handlers de exceção são o método pelo qual o programa em tempo de execução reage e lida com os erros. Os Erros em tempo de execução incluem erros SQL como :

- ORA-1: unique constraint violated

e erros procedurais como:

- ORA-06502: PL/SQL: numeric or value error

- **NOTA**

A PL/SQL tem um recurso conhecido como SQL dinâmico que permite que você crie e execute arbitrariamente as instruções SQL, bem como blocos PL/SQL em tempo de execução. Se você executar um bloco PL/SQL dinamicamente, que contenha um erro de compilação, esse erro será levantado em tempo de execução e poderá ser capturado por um handler de exceção.

Se ocorrer um erro, uma exceção é levantada. Quando isso acontecer, o controle passa para o handler de exceção, que é uma seção separada do programa. Isso separa o tratamento de erro do restante do programa, o que torna a lógica do programa mais fácil de entender. Isso também assegura que todos os erros serão interceptados.

Em uma linguagem que não utilize o modelo de exceção para o tratamento de erro (como C), a fim de assegurar que seu programa possa tratar todos erros, você deve inserir explicitamente o código de tratamento do erro. Examine o seguinte código para um exemplo:

- Int x = 1 , y = 2 , z = 3;  
f(x); /\* chamada de função que passa x como um argumento.\*/  
if <an error occurred>  
 handle\_error(...);  
y = 1 / z;  
if <an error occurred>  
 handle\_error(...);  
z = x + y;  
if <an error occurred>  
 handle\_error(...);

Observe que uma verificação de erros deve ocorrer depois de cada instrução no programa. Se você se esquecer de inserir a verificação, o programa não tratará adequadamente uma situação de erro. Além disso, o tratamento de erros pode confundir o programa, tornando difícil de entender a lógica do programa. Compare o exemplo anterior como este exemplo semelhante na PL/SQL:

```

DECLARE
  x NUMBER := 1;
  y NUMBER := 2;
  z NUMBER := 3;
BEGIN
  f(x);
  y := 1 / z;
  z := x + y;
EXCEPTION
  WHEN OTHERS THEN
    /* Handler para executar todos os erros */
    Handle_error(...);
END;

```

Observe que o tratamento de erros está separado da lógica do programa. Isso resolver ambos os problemas com o exemplo feito em C, a saber:

- A lógica do programa é mais fácil de entender porque está claramente visível.
- Independente de qual instrução falhar, o programa irá detectar e tratar o erro. Entretanto, observe que a execução do programa não continuará a partir da instrução que levantou o erro. Em vez disso, a execução continuará no handler de exceção e em seguida para qualquer bloco externo.

## **Declarando Exceções**

As exceções são declaradas na seção declarativa do bloco, levantadas na seção executável e tratadas na seção de exceção. Há dois tipos de exceções : definidas pelo usuário e predefinidas.

### **Exceções definidas pelo usuário**

Uma exceção definida pelo usuário é um erro que é definido pelo programador. O que em erro denota não e necessariamente um erro Oracle – por exemplo, ele pode ser um erro com os dados. Por outro lado, as exceções predefinidas correspondem aos erros comuns SQL e da PL/SQL.

Exceções definidas pelo usuário são declaradas na seção declarativa de um bloco PL/SQL. Da mesma forma como as variáveis, as exceções tem um tipo (EXCEPTION) e um escopo. Por exemplo:

```

DECLARE
  e_TooManyStudents EXCEPTION;

```

e\_TooManyStudents é um identificador que estará visível até o final desse bloco.

Observe que o escopo de uma exceção é o mesmo que o escopo de qualquer outra variável ou cursor na mesma seção declarativa.

### Exceções predefinidas

O Oracle tem várias exceções predefinidas que correspondem aos erros mais comuns do Oracle. Como os tipos predefinidos (NUMBER, VARCHAR2 e outros), os identificadores dessas exceções são definidas no pacote STANDART. Por causa disso, eles já estão disponíveis no programa, não sendo necessário declará-los na seção declarativa.

- **NOTA**

Também é possível associar as exceções definidas pelo usuário com os erros do Oracle. Consulte a seção “O pragma EXCEPTION\_INIT”, mais adiante neste capítulo para obter informações adicionais..

### Exceções predefinidas pelo Oracle

Erro do Oracle	Exceção Equivalente	Descrição
ORA-0001	DUP_VAL_ON_INDEX	Uma única restrição violada.
ORA-0051	TIMEOUT_ON_RESOURCE	O tempo limite ocorreu ao esperar pelo recurso.
ORA-0061	TRANSACTION_BACKED_OUT	A transação foi revertida devido a um impasse.
ORA-1001	INVALID_CURSOR	Operação ilegal de cursor.
ORA-1012	NOT_LOGGED_ON	Não conectado ao Oracle.
ORA-1017	LOGIN_DENIED	Nome usuário/senha invalida.
ORA-1403	NO_DATA_FOUND	Nenhum dado localizado.
ORA-1410	SYS_INVALID_ROWID	Conversão para um ROWID universal falhou.
ORA-1422	TOO_MANY_ROWS	Uma instrução SELECT....INTO corresponde a mais de uma linha.
ORA-1476	ZERO_DIVIDE	Divisão por zero.
ORA-1722	INAVLID_NUMBER	Conversão para um número falhou – por exemplo. “IA” não é valido.
ORA-6500	STORAGE_ERROR	Erro interno PL/SQL é levantado se a PL/SQL ficar na memória.
ORA-6501	PROGRAM_ERROR	Erro interno PL/SQL.
ORA-6502	VALUE_ERROR	Erro de truncamento, aritmética ou de conversão.
ORA-6504	ROQTYPE_MISMATCH	Variável do cursor do host e variável de cursor PL/SQL que tem tipos de linhas incompatíveis;
ORA-6511	CURSOR_ALREADY_OPEN	Tentativa de abrir um cursor já aberto.

ORA-6530	ACESS_INTO_NULL	Tentativa para atribuir valores para atributos de um objeto null.
ORA-6531	COLLECTION_IS_NULL	Tentativa para aplicar métodos de coleções diferentes de EXISTS em uma tabela ou varray NULL PL/SQL.
ORA-6532	SUBSCRIPT_OUTSIDE_LIMIT	Relatório de uma tabela animada ou um índice varray fora do intervalo ( como - I ).
ORA-6533	SUBSCRIPT_BEYOND_COUNT	Referência a uma tabela aninhada ou um índice varray maior que o número de elementos da coleção.
ORA-6592	CASE_NOT_FOUND	Nenhuma correspondencia com uma clausula WHEN quando uma instrução CASE é localizada.
ORA-30625	SELF_IS_NULL2	Tentativa para chamar um método em uma instancia NULL de Objeto.

Abaixo algumas descrições de exceções predefinidas.

**INVALID\_CURSOR** Esse erro é levantado quando é realizada uma operação ilegal de cursor como ao tentar fechar um cursor que já está fechado, uma situação igual ao tentar abrir um cursor que já está aberto CURSOR\_ALREADY\_OPEN.

**NO\_DATA\_FOUND** Essa exceção pode ser levantada em duas diferentes situações. A primeira é quando uma instrução SELECT... INTO não retorna nenhuma linha. Se a instrução retornar mais de uma linha, TOO\_MANY\_ROWS é levantada. A segunda situação é uma tentativa de referenciar um elemento “index-by table” da PL/SQL ao qual não foi atribuído um valor. Por exemplo, o seguinte bloco anônimo levantará NO\_DATA\_FOUND:

```

DECLARE
    TYPE t_NumberTableType IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
    v_NumberTable t_NumberTableType;
    v_TempVar NUMBER;
BEGIN
    v_TempVar := v_NumberTable(1);
END;

DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 7

```

**VALUE\_ERROR** Essa exceção é levantada quando um erro de aritmética, conversão, truncamento, ou de restrição ocorrer em uma instrução procedural. Se o erro ocorrer em uma instrução SQL, um erro como INVALID\_NUMBER é levantado. O erro pode ocorrer como resultado de uma instrução de atribuição, uma instrução SELECT... INTO, parâmetros RETURNING INTO de uma instrução SQL ou parâmetros de um subprograma. Todas essas situações são resultados de um valor atribuído para uma variável PL/SQL. Se houver um problema com essa atribuição, VALUE\_ERROR é levantado.

```
DECLARE
    v_TempVar VARCHAR2(3);
BEGIN
    v_TempVar := 'ABCD';
END;

DECLARE
*
ERROR at line 1:
ORA-06502:PL/SQL:numeric or value error: character string
buffer too small
ORA-06512: at line 4

=====

DECLARE
    v_TempVar NUMBER(2);
BEGIN
    SELECT id
        INTO v_TempVar
        FROM students
        WHERE last_name = 'Smith';
END;

DECLARE
*
ERROR at line 1:
ORA-06502:PL/SQL:
numeric or value error: number precision too large
ORA-06512: at line 4
```

**ROWTYPE\_MISMATCH** Essa exceção é levantada quando os tipos de uma variável de cursor de host e de uma variável de cursor PL/SQL não correspondem. Por exemplo, se os tipos de retornos reais e formais não correspondem a um procedimento que recebe uma variável de cursor como um argumento, ROWTYPE\_MISMATCH é levantado.

## Levantando exceções

Quando o erro associado com uma exceção ocorrer, a exceção é levantada. Exceções definidas pelo usuário são levantadas explicitamente via instrução RAISE, enquanto as exceções predefinidas são levantadas implicitamente quando seus erros associados com Oracle ocorrem. Se ocorrer um erro Oracle que não esteja associado com uma exceção, uma exceção também é levantada. Essa exceção pode ser capturada com um handler OTHERS , exceções predefinidas também podem ser levantadas explicitamente via instrução RAISE, se desejado.

```
DECLARE
    -- Exception to indicate an error condition
    e_TooManyStudents EXCEPTION;

    -- Current number of students registered for HIS-101
    v_CurrentStudents NUMBER(3);

    -- Maximum number of students allowed in HIS-101
    v_MaxStudents NUMBER(3);
BEGIN
    /* Find the current number of registered students, and the
maximum
    number of students allowed. */
    SELECT current_students, max_students
        INTO v_CurrentStudents, v_MaxStudents
        FROM classes
        WHERE department = 'HIS' AND course = 101;

    /* Check the number of students in this class. */
    IF v_CurrentStudents > v_MaxStudents THEN
        /* Too many students registered -- raise exception. */
        RAISE e_TooManyStudents;
    END IF;
END;
```

Quando a instrução é levantada o controle passa para o Bloco de instrução. Se não houver nenhuma seção de exceção, a exceção é propagada para o bloco de inclusão, uma vez que o controle foi passado para o handler de exceção, não há nenhuma maneira de retornar à seção executável do bloco.

Exceções predefinidas são automaticamente levantadas quando o erro do associado com Oracle ocorrer. Por exemplo, o seguinte bloco PL/SQL levantará a execução DUP\_VAL\_ON\_INDEX:

```

BEGIN
  INSERT INTO students (id, first_name, last_name)
    VALUES (20000, 'John', 'Smith');
  INSERT INTO students (id, first_name, last_name)
    VALUES (20000, 'Susan', 'Ryan');
END;

```

A exceção é levantada por o Id da coluna student e chave primária e, portanto, tem uma restrição única definida nele. Quando a segunda instrução INSERT tentar inserir 20000 na nessa coluna, o erro

- ORA-0001: unique constraint (constraint name) violated

## Tratando exceções

Quando uma exceção é levantada, o controle para a seção de exceção do bloco, como vimos acima. A seção de exceção consiste em handlers para algumas ou todas as exceções. A sintaxe para a seção de exceção é a seguinte:

```

EXCEÇÃO
  WHEN nome_da_seção_ THEN
    Seqüência_de_instruções1;
  WHEN nome_da_seção_ THEN
    Seqüência_de_instruções2;
  WHEN OUTROS THEN
    Seqüência_de_instruções3;
END;

```

Veja o exemplo abaixo:

```

DECLARE
  -- Exception to indicate an error condition
  e_TooManyStudents EXCEPTION;

  -- Current number of students registered for HIS-101
  v_CurrentStudents NUMBER(3);

  -- Maximum number of students allowed in HIS-101
  v_MaxStudents NUMBER(3);
BEGIN
  /* Find the current number of registered students, and the
maximum
   * number of students allowed. */
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
   FROM classes

```

```

        WHERE department = 'HIS' AND course = 101;

/* Check the number of students in this class. */
IF v_CurrentStudents > v_MaxStudents THEN
    /* Too many students registered -- raise exception. */
    RAISE e_TooManyStudents;
END IF;
EXCEPTION
WHEN e_TooManyStudents THEN
    /* Handler which executes when there are too many
students
       registered for HIS-101. We will insert a log message
       explaining what has happened. */
    INSERT INTO log_table (info)
        VALUES ('History 101 has ' || v_CurrentStudents ||
               'students: max allowed is ' || v_MaxStudents);
END;

```

Um único handler também pode ser executado em mais de uma exceção. Simplesmente liste os nomes de exceção na cláusula WHEN separada pela palavra-chave OR:

```

EXCEPTION
WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
    INSERT INTO log_table (info)
        VALUES ('A select error occurred');
END;

```

Uma dada exceção pode ser no máximo tratada por um handler em uma seção de exceção. Se houver mais de um handler para uma exceção, o compilador PL/SQL levantará o PLS-486, como abaixo:

```

DECLARE
    -- Declare 2 user defined exceptions
    e_Exception1 EXCEPTION;
    e_Exception2 EXCEPTION;
BEGIN
    -- Raise just exception 1.
    RAISE e_Exception1;
EXCEPTION
    WHEN e_Exception2 THEN
        INSERT INTO log_table (info)
            VALUES ('Handler 1 executed!');
    WHEN e_Exception1 THEN

```

```

    INSERT INTO log_table (info)
        VALUES ('Handler 3 executed!');
WHEN e_Exception1 OR e_Exception2 THEN
    INSERT INTO log_table (info)
        VALUES ('Handler 4 executed!');
END;

WHEN e_Exception1 OR e_Exception2 THEN
*
ERROR at line 15:
ORA-06550 line 15, column 3:
PLS-00483: exception 'E_EXCEPTION2' may appear in at most one exception handler in
this block
ORA-06550: line 0, column 0:
PL/SQL: compilation unit analysis terminated

```

## O handler de exceção OTHERS

A PL/SQL define um handler especial de exceção, conhecido como WHEN OTHERS. Esse handler será executado em todas as exceções levantadas que forem tratadas por quaisquer outras cláusulas WHEN definidas na seção atual de exceção. Ele sempre deve ser o último handler no bloco, de modo que todos os handlers anteriores, serão varridos primeiros. WHEN OTHERS interromperá todas as exceções definidas pelo usuário ou predefinidas.

Abaixo um exemplo com o handler OTHERS:

```

DECLARE
    -- Exception to indicate an error condition
    e_TooManyStudents EXCEPTION;

    -- Current number of students registered for HIS-101
    v_CurrentStudents NUMBER(3);

    -- Maximum number of students allowed in HIS-101
    v_MaxStudents NUMBER(3);
BEGIN
    /* Find the current number of registered students, and the
maximum
    number of students allowed. */
    SELECT current_students, max_students
        INTO v_CurrentStudents, v_MaxStudents
        FROM classes
        WHERE department = 'HIS' AND course = 101;

    /* Check the number of students in this class. */

```

```

IF v_CurrentStudents > v_MaxStudents THEN
    /* Too many students registered -- raise exception. */
    RAISE e_TooManyStudents;
END IF;
EXCEPTION
    WHEN e_TooManyStudents THEN
        /* Handler which executes when there are too many
        students
            registered for HIS-101. We will insert a log message
            explaining what has happened. */
        INSERT INTO log_table (info)
            VALUES ('History 101 has ' || v_CurrentStudents ||
                'students: max allowed is ' || v_MaxStudents);
WHEN OTHERS THEN
    /* Handler which executes for all other errors. */
    INSERT INTO log_table (info) VALUES ('Another error
    occurred');
END;

```

## SQLCODE e SQLERRM

Dentro de um handler OTHERS, freqüentemente é útil saber qual erro Oracle levantou a exceção, quer o erro tenha ou não uma exceção predefinida para ele. Uma das razões seria registrar em LOG o erro que ocorreu, em vez do fato de que um erro aconteceu.. Ou você talvez queira fazer coisas diferentes dependendo de qual erro foi levantado. A PL/SQL fornece essas informações via duas funções predefinidas:SQLCODE e SQLERRM. SQLCODE retorna o código do erro atual e SQLERRM retorna o texto da mensagem do erro atual. Em uma exceção definida pelo usuário, a SQLCODE retorna 1 e a SQLERRM retorna ‘Exceção definida pelo Usuário’.

- **NOTA**

A função DBMS\_UTILITY.FORMAT\_ERRO\_STACK também retorna uma mensagem do erro atual e pode ser utilizado além da SQLERRM.

Abaixo o bloco completo PL/SQL que desenvolvemos até agora, com um handler de exceção OTHERS completo:

```

-- This block contains a WHEN OTHERS handler that records
which
-- runtime error occurred.
DECLARE
    -- Exception to indicate an error condition
    e_TooManyStudents EXCEPTION;

    -- Current number of students registered for HIS-101
    v_CurrentStudents NUMBER(3);

```

```

-- Maximum number of students allowed in HIS-101
v_MaxStudents NUMBER(3);

-- Code and text of other runtime errors
v_ErrorCode log_table.code%TYPE;
v_ErrorText log_table.message%TYPE;

BEGIN
    /* Find the current number of registered students, and the
maximum
    number of students allowed. */
    SELECT current_students, max_students
        INTO v_CurrentStudents, v_MaxStudents
        FROM classes
        WHERE department = 'HIS' AND course = 101;

    /* Check the number of students in this class. */
    IF v_CurrentStudents > v_MaxStudents THEN
        /* Too many students registered -- raise exception. */
        RAISE e_TooManyStudents;
    END IF;
EXCEPTION
    WHEN e_TooManyStudents THEN
        /* Handler which executes when there are too many
students
        registered for HIS-101. We will insert a log message
        explaining what has happened. */
        INSERT INTO log_table (info)
            VALUES ('History 101 has ' || v_CurrentStudents ||
                    'students: max allowed is ' || v_MaxStudents);
    WHEN OTHERS THEN
        /* Handler which executes for all other errors. */
        v_ErrorCode := SQLCODE;
        -- Note the use of SUBSTR here.
        v_ErrorText := SUBSTR(SQLERRM, 1, 200);
        INSERT INTO log_table (code, message, info) VALUES
            (v_ErrorCode, v_ErrorText, 'Oracle error occurred');
END;

```

O comprimento máximo de uma mensagem de erro do Oracle é de 512 caracteres. Observe que os valores SQLCODE e SQLERRM são atribuídos primeiramente para as variáveis locais; em seguida essas variáveis são utilizadas em uma instrução SQL. Pelo fato de essas funções serem procedurais, elas não podem ser utilizadas diretamente dentro de uma instrução SQL.

SQLERRM também pode ser chamada com um único argumento de número. Nesse caso, ela retorna o texto associado com o número. Esse argumento sempre deve ser negativo. Se SQLERRM for chamada com zero, a mensagem:

- ORA-0000 normal, successful completion

Se SQLERRM for chamada com qualquer valor positivo que não seja +100, a mensagem:

- Non-Oracle Exception

Se SQLERRM (100) retorna:

- ORA-1403: no data found

O valor 100 positivo e o caracter ANSI do erro NO DATA FOUND.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('SQLERRM(0): ' || SQLERRM(0));
    DBMS_OUTPUT.PUT_LINE('SQLERRM(100): ' || SQLERRM(100));
    DBMS_OUTPUT.PUT_LINE('SQLERRM(10): ' || SQLERRM(10));
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('SQLERRM(-1): ' || SQLERRM(-1));
    DBMS_OUTPUT.PUT_LINE('SQLERRM(-54): ' || SQLERRM(-54));
END;
```

SQLERRM(0): ORA-0000: normal, successful completion

SQLERRM(100): ORA-1403 no data found

SQLERRM(10): -10Ç non-ORACLE exception

SQLERRM: ORA-0000: normal, successful completion

SQLERRM(-1): ORA-0001: unique constraint(.) violated

SQLERRM(-54): ORA-0054: resource busy and acquire with NOWAIT specified

PL/SQL procedure successfully completed

## O pragma EXCEPTION\_INIT

Você pode associar uma exceção nomeada com um erro do Oracle em particular. Isso dá a capacidade de interromper especificamente esse erro, sem ser via um handler OTHERS. Isso é feito via pragma EXCEPTION\_INIT. O pragma EXCEPTION\_INIT é utilizado como a seguir,

```
PRAGMA EXCEPTION_INIT(nome_da_exceção, numero_erro_do_Oracle);
```

onde nome\_da\_exceção é o nome de uma exceção declarada antes do pragma e numero\_erro\_do\_Oracle é o código de erro do desejado a ser associado com essa exceção nomeada. Esse pragma deve estar na seção declarativa. O seguinte exemplo levantara a exceção definida pelo usuário:

```
DECLARE
    e_MissingNull EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_MissingNull, -1400);
BEGIN
    INSERT INTO students (id) VALUES (NULL);
EXCEPTION
    WHEN e_MissingNull then
        INSERT INTO log_table (info) VALUES ('ORA-1400
occurred');
END;
```

## Utilizando RAISE\_APPLICATION\_ERROR

Você pode utilizar a função predefinida RAISE\_APPLICATION\_ERROR para criar suas próprias mensagens de erro, que podem ser mais descritivas que as exceções identificadas. Os erros definidos pelo usuário são passados para o bloco da mesma maneira como os erros do Oracle são passados para o ambiente de chamada. A sintaxe é:

```
RAISE_APPLICATION_ERROR (numero_do_erro, mensagem_do_erro, "manter_erros");
```

onde numero\_do\_erro é o valor entre -20.000 e -20.999, mensagem\_do\_erro é o texto associado, e manter\_erros é um valor booleano.

Exemplo:

```
/* Registers the student identified by the p_StudentID
parameter in
the class identified by the p_Department and p_Course
parameters. */
CREATE OR REPLACE PROCEDURE Register (
    p_StudentID IN students.id%TYPE,
    p_Department IN classes.department%TYPE,
    p_Course IN classes.course%TYPE) AS

    v_CurrentStudents classes.current_students%TYPE;
    v_MaxStudents classes.max_students%TYPE;
    v_NumCredits classes.num_credits%TYPE;
    v_Count NUMBER;

BEGIN
```

```

/* Determine the current number of students registered, and
the
maximum number of students allowed to register. */
BEGIN
    SELECT current_students, max_students, num_credits
        INTO v_CurrentStudents, v_MaxStudents, v_NumCredits
        FROM classes
        WHERE course = p_Course
        AND department = p_Department;

    /* Make sure there is enough room for this additional
student. */
    IF v_CurrentStudents + 1 > v_MaxStudents THEN
        RAISE_APPLICATION_ERROR(-20000,
            'Can''t add more students to ' || p_Department || ' '
            || p_Course);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        /* Class information passed to this procedure doesn't
exist. */
        RAISE_APPLICATION_ERROR(-20001,
            p_Department || ' ' || p_Course || ' doesn''t
exist');
    END;

/* Ensure that the student is not currently registered */
SELECT COUNT(*)
    INTO v_Count
    FROM registered_students
    WHERE student_id = p_StudentID
    AND department = p_Department
    AND course = p_Course;
IF v_Count = 1 THEN
    RAISE_APPLICATION_ERROR(-20002,
        'Student ' || p_StudentID || ' is already registered
for ' ||
        p_Department || ' ' || p_Course);
END IF;

/* There is enough room, and the student is not already in
the
class. Update the necessary tables. */
INSERT INTO registered_students (student_id, department,
course)
    VALUES (p_StudentID, p_Department, p_Course);
UPDATE students

```

```

SET current_credits = current_credits + v_NumCredits
WHERE ID = p_StudentID;
UPDATE classes
SET current_students = current_students + 1
WHERE course = p_Course
AND department = p_Department;
END Register;

```

Execute abaixo os comandos para verificar os erros ocorridos pelo RAISE\_APPLICATION\_ERROR:

```

-- Illustrate the ORA-2001 and ORA-2002 errors
exec Register(10000, 'CS', 999);
exec Register(10000, 'CS', 102);

-- Register 2 students for MUS 410, which will raise ORA-2003
exec Register(10002, 'MUS', 410);
exec Register(10005, 'MUS', 410);

```

Compare a saída anterior com o próximo bloco anônimo que simplesmente levanta a exceção NO\_DATA\_FOUND:

```

BEGIN
  RAISE NO_DATA_FOUND;
END;

BEGIN
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 2

```

### **Exceções levantadas na seção de exceção**

As exceções também podem ser levantadas quando ainda estiverem em um handler de exceção, tanto explicitamente via instrução RAISE como implicitamente via um erro em tempo de execução. Em qualquer um dos casos, a exceção é imediatamente propagada para o bloco “pai”.

Um problema com as exceções é que quando uma exceção ocorre, não há nenhuma maneira fácil para informar qual parte do código estava sendo executado nesse momento. A PL/SQL fornece uma solução para isso, com a função **DBMS\_UTLILITY.FORMAT\_CALL\_STACK**. Essa função retornará um “trace” para se chegar ao ponto onde a exceção foi gerada.

## FUNÇÕES SQL PREDEFINIDAS

A linguagem SQL fornece diversas funções predefinidas que podem ser chamadas a partir de uma instrução SQL. Por exemplo, a seguinte instrução SELECT utiliza a função UPPER para retornar os primeiros nomes dos alunos, escritos em letras maiúsculas, em vez da maneira como que eles foram armazenados.

```
Select upper (first_name)  
From students;
```

Várias funções SQL também podem ser denominadas de instruções procedurais PL/SQL . Por exemplo, o seguinte bloco também utiliza a função UPPER, mas em uma instrução de atribuição:

```
Declare  
  V(firstName students.first_name%type);  
  
Begin  
  V.firstName := upper('charlie');  
End;
```

### Funções de caractere que retornam valores de caracteres

Todas essas funções recebem argumentos da família de caractere (exceto CHR e NCHR) e retornam valores de caractere. A maior parte das funções retornam um valor VARCHAR2.

#### **CHR (x[using nchar\_cs])**

Retorna o caractere que tem o valor equivalente ao x no conjunto de caracteres do bando de dados. CHR e ASCII são funções opostas.

#### **CONCAT (string1, string2)**

Retorna string1 concatenada com string2. Essa função é idêntica ao operador || .

#### **INITCAP (string)**

Retorna string com o primeiro caractere de cada palavra em letra maiúscula e os caracteres restantes de cada palavra em letras minúsculas.

#### **LOWER (string)**

Retorna string com todos os caracteres em letras minúsculas. Quaisquer caracteres que não forem letras permanecem intactos. Se string tiver o tipo de dados CHAR, o resultado também será CHAR. Se string for VARCHAR2, o resultado será VARCHAR2.

#### **LPAD (String1, x[string2])**

Retorna string1 preenchida à esquerda até o comprimento x com os caracteres em string2.

### **LTRIM (String1,String2)**

Retorna string1 com os caracteres mais à esquerda aparecendo em string2 removidos.  
string2 assume o padrão de um espaço em branco.

### **REPLACE (string, string\_a\_pesquisar [string\_substituta])**

Retorna string com cada ocorrência de string\_a\_pesquisar susbstituída por string\_substituta.

### **RPAD (string1, x, [string2])**

Retorna string1, com os caracteres mais à direita que aparecem em string2, removidos. A string2 assume o padrão de um espaço em branco.

### **TRANSLATE (string, str\_de, str\_para)**

Retorna string com todas as ocorrências de cada caractere em str\_de\_substituído pelo caractere correspondente em str\_para.

### **TRIM([{LEADING|TRAILING|BOTH}{aparar\_char}]|aparar\_char}FROM]string)**

Retorna string com as ocorrências inicial, final ou ambas de aparar\_char removidas.

### **UPPER (string)**

Retorna string com todas as letras em maiúsculas.

## **SUBSTR**

Sintaxe:

### **SUBSTR (string, a[,b])**

Retorna uma parte da string que inicia no caractere a, com o comprimento dos caracteres b. Se a for 0, é tratado como 1 (o início da string). Se a for positivo, os caracteres que retornam são contados da esquerda. Se a for negativo, os caracteres retornam iniciando do final da string e são contados da direita. Se b não estiver presente, a string inteira é assumida como padrão. Se b for menor que 1, NULL é retornado. Se um valor de ponto flutuante for passado para a e b, o valor primeiro é truncado para um inteiro. O tipo de retorno é sempre do mesmo tipo da string.

Exemplo:

```
Select substr('abc123def', 4, 4) "First"  
From dual;
```

```
First  
----  
123d
```

```
Select substr('abc123def ', -4,4) "Second"  
From dual;
```

```
Second  
-----  
3def
```

```
Select substr('abc123def ', 5) "Third"  
From dual;
```

```
Third  
-----  
23def
```

## SOUNDEX

Sintaxe:

**SOUNDEX(string)**

Retorna a representação fonética de string. Isso é útil para comparar palavras que apresentam grafia diferente, mas são pronunciadas de modo semelhante. A representação fonética é definida no livro. The Art of Computer Programming, Volume 3: Sorting and Searching de Donald E. Knuth.

Exemplo:

```
Select first_name, soundex(first_name)  
From students;
```

First_name	SOUN
-----	-----
Scott	S300
Margaret	M626

```
Select first_name  
From students  
Where soundex(first_name) = soundex('skit');
```

First_Name	
-----	
Scott	

## **Funções de caractere que retornam valores numéricos**

Essas funções recebem argumentos de caractere e retornam resultados numéricos. Os argumentos por ser CHAR ou VARCHAR2. Embora vários dos resultados sejam de fato valores inteiros, o valor de retorno é simplesmente NUMBER, sem definição de precisão ou escala.

### **ASCII (string)**

Retorna a representação decimal do primeiro byte de string no conjunto de caracteres do banco de dados.

### **INSTR (string1,string2 [,a] [,b])**

Retorna a posição dentro de string1 onde string2 está contida, com a e b medidos em caracteres.

### **LENGTH (string)**

Retorna o comprimento de string medido em caracteres.

## **INSTR**

Sintaxe:

```
INSTR (string1, string2 [,a] [,b])
```

Retorna a posição dentro de string1 em que string2 está contida.

Exemplo:

```
SELECT INSTR('Scott"s spot' , 'ot', 1, 2) "First"  
From dual;
```

```
First  
-----  
11
```

## **LENGTH**

Sintaxe:

```
LENGTH(string)
```

Retorna o comprimento de string. Uma vez que valores de CHAR são preenchidos por espaços em branco, se string tiver tipo de dados de CHAR, os espaços em branco finais são incluídos no comprimento.

Exemplo:

```
Select length ('Mary had a little lamb') "Length"  
From dual;  
Length  
-----  
22
```

## Funções de NLS

Exceto por NCHR, essas funções recebem argumentos de caractere e retornam os valores em caracteres.

### **CONVERT (string, conjunto\_de\_caracteres\_dest[,conjunto\_de\_caracteres\_orig])**

Converte a entrada de string no conjunto de caracteres conjunto\_de\_caracteres\_dest especificado.

#### **NCHR(x)**

Retorna o caractere que tem o valor equivalente ao x no banco de dados do conjunto de caracteres nacional. NCHR(x) é equivalente a CHR (x USING NCHAR\_CS).

#### **NLS\_CHARSET\_DECL\_LEN**

Retorna (em caracteres) a largura da declaração de um valor de NCHAR, largura\_de\_byte é o comprimento do valor em bytes e conjunto\_de\_caracteres é o ID do conjunto de caracteres do valor.

#### **NLS\_CHARSET\_ID**

Retorna o ID numérico do conjunto especificado de caracteres nomeado nome\_do\_conjunto\_de\_caracteres.

#### **NLS\_CHARSET\_NAME**

Retorna o ID do conjunto de caracteres especificado chamado ID\_do\_conjunto\_de\_caracteres.

#### **NLS\_INITCAP**

Retorna string com o primeiro caractere de cada palavra em letras maiúsculas e os caracteres restantes de cada palavra em letras minúsculas.

#### **NLS\_LOWER**

Retorna string com todas as letras em minúsculas. Os caracteres que não são letras permanecem intactos.

#### **NLS\_UPPER**

Retorna string com todas as letras em Maiúsculas.

## **NLSSORT**

Retorna a string de bytes utilizada para classificar string.

## **TRANSLATE**

Translate...USING converte o argumento de string de entrada tanto em um conjunto de caracteres do banco de dados (se CHAR\_CS estiver especificado) como no conjunto de caracteres nacional do banco de dados (se NCHAR\_CS estiver especificado).

## **UNISTR(s)**

Retorna a string s traduzida pelo conjunto de caracteres Unicode do banco de dados.

## **Funções Numéricas**

Essas funções recebem argumentos de NUMBER e retornam valores de NUMBER.

### **ABS(x)**

Retorna o valor absoluto de x.

### **ACOS(x)**

Retorna o co-seno do arco de x.x de -1 a 1 e a saída deve variar de 0 a II, expressa em radianos.

### **ASIN(x)**

Retorna o seno do arco de x.x deve variar de -1 a 1 e a saída de -II/2 a -II/2, expressa em radianos.

### **ATAN(x)**

Retorna a tangente do arco de x. A saída varia de -II/2 a -II/2, expressa em radianos.

### **ATAN2 (x,y)**

Retorna a tangente do arco de x e y.

### **BITAND(x,y)**

Retorna o bitwise AND de x e de y, cada um dos quais devem ser valores não-negativos de inteiro.

### **CEIL(x)**

Retorna o menor inteiro maior ou igual que x.

### **COS(x)**

Retorna o co-seno de x. o x é um ângulo expresso em radianos.

### **COSH(x)**

Retorna o co-seno da hipérbole de x.

**EXP(x)**

Retorna e elevado a. x. e= 2.71828183...

**FLOOR(x)**

Retorna o maior inteiro igual ao menor que x.

**LN(x)**

Retorna o logaritmo natural x. x deve ser maior que 0.

**LOG (x,y)**

Retorna a base de logaritmo de x e de y. A base deve ser um número positivo, menos 0 ou 1 e y pode ser qualquer número positivo.

**MOD(x,y)**

Retorna o resto de x dividido por y. Se y for 0, x é retomado.

**POWER(x,y)**

Retorna x elevado a y. A base x e o expoente y não precisam ser inteiros positivos, mas se x for negativo, y deve ser um inteiro.

**ROUND(x,[y])**

Retorna x arredondado para y casas à direita do ponto decimal. Y assume o padrão de 0, o qual arredonda x para o inteiro mais próximo. Se y for negativo, os dígitos à esquerda do ponto decimal são arredondados. Y deve ser um inteiro.

**SIGN(x)**

Se  $x < 0$ , retorna -1. Se  $x = 0$ , retorna 0. Se  $x > 0$ , retorna 1.

**SIN(x)**

Retorna o seno de x, que é um ângulo expresso em radianos.

**SINH(x)**

Retorna o seno hiperbólico de x.

**SQRT(x)**

Retorna a raiz quadrada de x. x não pode ser negativo.

**TAN(x)**

Retorna a tangente de x, que é um ângulo expresso em radianos.

**TANH(x)**

Retorna a tangente hiperbólica de x.

**TRUNC (x,[y])**

Retorna x truncado (em oposição a arredondado ) para y casas decimais. Y assume o padrão de 0, o que trunca x para um valor de inteiro. Se y for negativo, os dígitos à esquerda do ponto decimal são truncados.

## **WIDTH\_BUCKET**

Sintaxe:

```
WIDTH_BUCKET(x,mín, máx, num_buckets)
```

**WIDTH\_BUCKET** permite que você crie histogramas com comprimento igual com base nos parâmetros de entrada. O intervalo *mín...máx* é dividido em *num\_buckets* seções, tendo cada seção o mesmo tamanho. A seção em que x cai então é retornada. Se x for menor que *mín*, 0 é retornado. Se x for maior que ou igual a *máx*, *num\_buckets*+1 é retornado. Nem *mín* nem *máx* podem ser NULL; e *num\_buckets* deve ser avaliado como um inteiro positivo. Se x for NULL, então NULL é retornado.

Exemplo:

O seguinte exemplo configura 20 buckets, cada um com um tamanho de 50 (1.000/20):

```
Select number_seats, WIDTH_BUCKET(number_seats, 1,1000,20)
Bucket
FROM rooms;
```

NUMBER_SEATS	BUCKET
1000	21
500	10
50	1
1000	21

## **Funções de data e hora**

As funções de data recebem argumentos do tipo DATE. Exceto pela função MONTHS\_BETWEEN, que retorna um NUMBER, todas as funções retornam valores DATE ou datetime.

### **ADD\_MONTHS(d,x)**

Retorna a data d mais x meses. X pode ser qualquer inteiro. Se o mês resultante tiver menos dias que o mês d, o último dia do mês resultante é retornado. Se não, o resultado tem o mesmo componente de dia que d. O componente de hora de d e o resultado são os mesmos.

### **CURRENT\_DATE**

Retorna a data atual na sessão do fuso horário com um valor de DATE. Essa função é semelhante a SYSDATE, exceto que SYSDATE não é sensível ao fuso horário da sessão atual.

**CURRENT\_TIMESTAMP**

Retorna a data atual na sessão do fuso horário como um valor de TIMESTAMP WITH TIMEZONE. Se precisão for especificada, ela representa a precisão decimal do número de segundo que é retornado. O valor de 6 é assumido por padrão.

**DBTIMEZONE**

Retorna o fuso horário do banco de dados. O formato é o mesmo utilizado pela instrução CREATE DATABASE ou mais recente ALTER DATABASE.

**LAST\_DAY**

Retorna a data do último dia do mês que contém d. Essa função pode ser utilizada para determinar quantos dias restam no mês atual.

**LOCALTIMESTAMP**

Retorna a data atual no fuso horário da sessão como um valor TIMESTAMP.

**MONTHS\_BETWEEN**

Retorna o número de meses entre a data1 e a data2.

**NEW\_TIME (d, zona1,zona2)**

Retorna a data e a hora do fuso horário da zona2 quando a data e a hora no fuso horário da zona1 forem d.

**NEXTDAY**

Retorna a data do primeiro dia nomeado por string que é mais antiga que a data d.

**ROUND (d[,formato])**

Arredonda a data d para a unidade especificada por formato.

**SESSIONTIMEZONE**

Retorna o fuso horário da sessão atual.

**SYS\_EXTRACT\_UTC**

Retorna a hora em UTC (Coordinated Universal Time, antigamente Greenwich Mean Time) a partir do datetime fornecido, o qual deve incluir um fuso horário.

**SYSDATE**

Retorna a data e a hora atual no tipo DATE.

**SYSTIMESTAMP**

Retorna a data e hora atual do tipo TIMESTAMP WITH TIMEZONE.

**TRUNC (d,[,formato])**

Retorna a data d truncada para a unidade especificada por formato.

**TZ\_OFFSET(fuso horário)**

Retorna o deslocamento como uma string de caractere entre o fuso horário fornecido e UTC.

**Funções de conversão****ASCIISTR (string)**

Retorna uma string contendo apenas caracteres SQL válidos mais uma barra.

**BIN\_TO\_NUM (num[,num]...)**

Converte um vetor de bit em seu número equivalente.

**CHARTOROWID**

Converte um valor de CHAR ou de VARCHAR2 que contém o formato externo de um ROWID em formato binário interno.

**COMPOSE**

Retorna string (que pode ser qualquer conjunto de caracteres) em sua forma Unicode completamente normalizada no mesmo conjunto de caracteres.

**DECOMPOSE**

Retorna uma string Unicade a qual é a decomposição canônica de string (que pode ser qualquer conjunto de caracteres).

**FROM\_TZ (timestamp,fuso horário)**

Retorna um valor de TIMESTAMP WITH TIMEZONE, que é a combinação de timestamp (registro de data/hora).

**HEXTORAW**

Converte o valor binário representado por string em um valor RAW.

**NUMTODSINTERVAL**

Converte x, que deve ser um número, em um valor de INTERVAL DAY TO SECOND.

**NUMTOYMINTEGER**

Converte x, que dever ser um número, em um valor de INTERVAL YEAR TO MONTH.

**REFTOHEX**

Retorna uma representação hexadecimal do valorref de REF.

**RAWTOHEX**

Converte valorbruto de RAW em uma string de caractere contendo a representação hexadecimal.

**RAWTONHEX**

Converte o valorbruto RAW em uma string de caractere contendo a representação hexadecimal.

## **ROWIDTOCHAR**

Converte o valor ROWID de idLinha em sua representação externa de string de caractere (que pode ter formas diferentes, dependendo do valor original de idLinha).

## **ROWIDTONCHAR**

Converte o valor ROWID de idLinha em sua representação externa se string de caractere ( que pode ter formas diferentes, dependendo do valor original de idLinha).

## **TO\_CHAR (datas e horas)**

Sintaxe:

```
TO_CHAR (d[,formato[,params_de_nls]])
```

Converte a data ou registro de data/hora (timestamp) d em uma string de caractere VARCHAR2. Se o formato for especificado, ele é utilizado para controlar como o resultado é estruturado.

Exemplo:

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YY HH24:MI:SS') "Right Now"  
      FROM dual;  
  
Right Now  
-----  
10-AUG-01    15:44:54
```

## **TO\_CHAR (Números)**

Sintaxe:

```
TO_CHAR(num[,formato[,params_de_nls]])
```

Converte o argumento NUMBER de num em um VARCHAR2. Se especificado, formato governa a conversão. Se o formato não for especificado, a string resultante terá exatamente tantos caracteres quantos necessários para conter os dígitos significativos de num.

Exemplo:

```
SELECT TO_CHAR(123456, '99G99G99') "Resul"  
      FROM dual;  
  
Result  
-----  
12,34,56
```

## **TO\_DATE**

Sintaxe:

```
TO_DATE(string[,formato[,params_de_nls]])
```

Converte a string CHAR ou VARCHAR2 em uma DATE.

Exemplo:

```
DECLARE  
  
V_CurrentDate DATE;  
  
BEGIN  
V_CurrentDate := TO_DATE ('January 7, 1973', 'Month DD,  
YYYY');
```

## **TO\_NUMBER**

Sintaxe:

```
TO_NUMBER(string[,formato[,params_de_nls]])
```

Converte string de CHAR ou de VARCHAR2 para um valor de NUMBER.

Exemplo:

```
DECLARE  
  
V_Num NUMBER;  
  
BEGIN  
V_Num := TO_NUMBER('$12345.67', '$99999.99');  
END;
```

## **TO\_TIMESTAMP e TO\_TIMESTAMP\_TZ**

Sintaxe

```
TO_TIMESTAMP (string[,formato[,params_de_nls]])  
TO_TIMESTAMP_TZ(string[,formato[,params_de_nls]])
```

Converte string CHAR ou VARCHAR2 em uma string TIMESTAMP ou  
TIMESTAMPWITHTIMEZONE.

Exemplo:

```
DECLARE
  V_CurrentDate TIMESTAMP;
BEGIN
  V_CurrentDate := TO_TIMESTAMP ('January 7,1973', 'Month
DD, YYYY');
END;
```

## Funções de grupo

As funções agregadas retornam um único resultado com base em várias linhas em oposição às funções de uma única linha, que retornam um resultado para cada linha.

### **AVG ([DISTINCT|ALL] col)**

Retorna a media dos valores da coluna.

### **COUNT (\*|[DISTINCT|ALL]col)**

Retorna o número de linhas na consulta. Se \* for passado, o número total de linhas é retornado.

### **GROUP\_ID()**

Retorna um valor único de número utilizado para distinguir grupos em uma cláusula GROUP BY.

### **GROUPING**

Distingue entre linhas de superagregados e linhas regulares agrupadas. Consulte o SQL Reference para obter mais detalhes.

### **GROUPING\_ID**

Retorna um número correspondente com o vetor de bit de GROUPING para uma linha. Consulte o SQL Reference para obter mais detalhes.

### **MAX([DISTINCT|ALL]col)**

Retorna o valor máximo do item da lista de seleção. Observe que DISTINCT e ALL não têm nenhum efeito, uma vez que o valor máximo seria o mesmo em qualquer um dos casos.

### **MIN([DISTINCT|ALL]col)**

Retorna o valor mínimo do item da lista de seleção. Observe que DISTINCT e ALL não têm nenhum efeito, uma vez que o valor mínimo seria o mesmo em qualquer um dos casos.

### **STDDEV ([DISTINCT|ALL]col)**

Retorna a desvio padrão do item da lista de seleção. Isso é definido como a raiz quadrada da variância.

**SUM([DISTINCT|ALL]col)**

Retorna a soma dos valores para o item da lista da seleção.

**Outras funções****BFILENAME (diretório nome\_de\_arquivo)**

Retorna o localizador BFILE associado com o arquivo físico nome\_de\_arquivo no sistema operacional. O diretório deve ser um objeto de DIRECTORY no dicionário de dados.

**COALESCE**

Retorna o primeiro expr não-NULO na lista de argumentos. Se todas as expressões forem NULL COALESCE retorna NULL.

**DECODE (expressão\_de\_base, comparação1,valor1,comparação2,valor2..padrão)**

A função DECODE é semelhante a uma série de instruções IF-THEN-ELSE aninhadas.

**EMPTY\_BLOB/EMPTY\_CLOB**

Retorna um localizador de LOB vazio. EMPTY\_CLOB retorna um localizador de caractere e EMPTY\_BLOB retorna um localizador binário.

**GREATEST (expr1 [,expr2])...**

Retorna a maior expressão dos seus argumentos. Cada expressão é implicitamente convertida para o tipo de expr1 antes que as comparações sejam feitas. Se expr1 for um tipo de caractere, são utilizadas as comparações de caracteres sem preenchimento com espaços em branco e o resultado tem o tipo de dados VARCHAR2.

**LEAST (expr1 [,expr2])...**

Retorna o menor valor na lista de expressões. Least comporta-se de maneira semelhante a GREATEST, pelo fato de que todas as expressões são implicitamente convertidas no tipo de dados do primeiro. Todas as comparações de caracteres são feitas com uma semântica de comparação de caractere sem preenchimento de espaços em branco.

**NLLIF (a,b)**

Retorna NULL se a for igual a b; e a, caso contrário.

**NVL (expr1,expr2)**

Se expr1 for NULL, retorna expr2; caso contrário, retorna expr1.

**NVL2 (expr1,expr2,expr3)**

Se expr1 for NULL, então retorna expr2, caso contrario, retorna expr3.

**SYS\_CONNECT\_BY\_PATH**

Retorna o caminho do valor de uma coluna da raiz para o nó. É valido apenas em consultas hierárquicas. Consulte o SQL Reference para obter mais detalhes.

**SYS\_CONTEXT**

Retorna o valor do parâmetro associado com o contexto do espaço de nome.

**SYS\_GUID**

Retorna um identificador globalmente único como um valor de RAW de 16 bytes.

**SYS\_TYPEID (tipo\_de\_objecto)**

Retorna o ID do tipo do mais específico de tipo\_de\_objeto.

**TREAT (expr AS [REF] [esquema.]tipo )**

TREAT é utilizado para alterar o tipo declarado de uma expressão.

**UID**

Retorna um inteiro que identifica de maneira exclusiva o usuário atual do banco de dados.

UID não recebe nenhum argumento.

**USER**

Retorna um valor de VARCHAR2 contendo o nome do usuário atual do Oracle. User não recebe nenhum argumento.

**DUMP**

Sintaxe:

DUMP (expr[,formato\_de\_número[,posição\_inicial] [,comprimento]])

Retorna um valor de VARCHAR2 que contém as informações sobre a representação interna de expr.

## CURSORES

A fim de processar uma instrução SQL, o Oracle alocará uma área de memória conhecida como área de contexto. A área de contexto contém as informações necessárias para completar o processamento, incluindo o número de linhas processadas pela instrução, e no caso de uma consulta, o conjunto ativo, que é o conjunto de linhas retornado pela consulta.

```
DECLARE
    /* Output variables to hold the results of the query */
    v_StudentID      students.id%TYPE;
    v_FirstName       students.first_name%TYPE;
    v_LastName        students.last_name%TYPE;

    /* Bind variable used in the query */
    v_Major           students.major%TYPE := 'Computer Science';

    /* Cursor declaration */
    CURSOR c_Students IS
        SELECT id, first_name, last_name
        FROM students
        WHERE major = v_Major;
BEGIN
    /* Identify the rows in the active set, and prepare for
    further
    processing of the data */
    OPEN c_Students;
    LOOP
        /* Retrieve each row of the active set into PL/SQL
        variables */
        FETCH c_Students INTO v_StudentID, v_FirstName,
        v_LastName;

        /* If there are no more rows to fetch, exit the loop */
        EXIT WHEN c_Students%NOTFOUND;
    END LOOP;

    /* Free resources used by the query */
    CLOSE c_Students;
END;
```

Cursos são trechos alocados de memória destinados a processar as declarações SELECT. Podem ser definidos pelo próprio PL/SQL, chamados de **Cursos Implícitos**, ou podem ser definidos manualmente, são os chamados de **Cursos Explícitos**.

## Cursos explícitos

Os cursos explícitos são chamados dessa forma porque são declarados formalmente na Área de declarações do módulo, ao contrário do que ocorre com os cursos implícitos. São tipicamente mais flexíveis e poderosos que os cursos implícitos, podendo substituí-los em qualquer situação.

Para sua utilização são necessários alguns passos básicos:

- declarar o cursor ;
- declarar as variáveis que receberão os dados ;
- abrir (uma espécie de preparação) o cursor na área de instruções ;
- ler os dados produzidos pelo cursor ;
- fechar (desalocar a memória) do cursor.

O exemplo a seguir ilustra a utilização de um cursor explícito equivalente àquele utilizado para demonstrar o cursor implícito. Analise as principais diferenças entre essa solução e a anterior.

Inicialmente, existe a declaração do cursor nas linhas 004 a 007, onde não aparece a cláusula INTO. O nome dado ao cursor segue as mesmas regras para os nomes de variáveis. A sintaxe básica dessa declaração é a seguinte:

**CURSOR** <nome do cursor> IS <declaração SELECT> ;

Ao fazermos a declaração, apenas foi definida uma estrutura que será utilizada posteriormente, quando o SELECT for executado e a recuperação das linhas for realizada. A primeira instrução realmente executável relativa ao cursor é a sua abertura, feita na área de instruções através do comando OPEN. É no momento da abertura que o SELECT é executado e as linhas recuperadas tornam-se disponíveis para uso. A sintaxe do comando OPEN é:

OPEN <nome-do-cursor>;

```
DECLARE
    v_StudentID students.ID%TYPE;

CURSOR c_AllStudentIDs IS
    SELECT ID FROM students;
BEGIN
    OPEN c_AllStudentIDs;

    -- Open it again. This will raise ORA-6511.
    OPEN c_AllStudentIDs;
END;
```

Para obtermos as linhas que foram recuperadas pela consulta, devemos buscá-las, uma a uma, na estrutura do cursor e copiar seus dados para as variáveis correspondentes. Isso é obtido através do comando FETCH:

```
FETCH <nome-do-cursor> INTO <lista de variáveis> ;  
  
DECLARE  
    v_Department    classes.department%TYPE;  
    v_Course        classes.course%TYPE;  
    CURSOR c_AllClasses IS  
        SELECT *  
        FROM classes;  
    v_ClassesRecord  c_AllClasses%ROWTYPE;  
BEGIN  
    OPEN c_AllClasses;  
  
    -- This is a legal FETCH statement, returning the first  
    -- row into a PL/SQL record which matches the select list  
    -- of the query.  
    FETCH c_AllClasses INTO v_ClassesRecord;  
  
    -- This FETCH statement is illegal, since the select list  
    -- of the query returns all 7 columns in the classes table  
    -- but we are only fetching into 2 variables.  
    -- This will raise the error "PLS-394: wrong number of  
    values  
    -- in the INTO list of a FETCH statement".  
    FETCH c_AllClasses INTO v_Department, v_Course;  
END;
```

Ao final do processamento, o cursor é fechado através de um comando CLOSE, cuja sintaxe é:CLOSE <nome-do-cursor> ;

Além dos recursos básicos ligados aos cursores, existem três outros que merecem atenção: a possibilidade da passagem de parâmetros para os cursores; os chamados atributos de cursor e cursores explícitos via looping FOR.

### Parâmetros de cursor

Geralmente o comando SELECT de um cursor possui uma cláusula WHERE que especifica uma seleção de linhas a serem retornadas. Muitas vezes, temos necessidade de variar um dado a ser comparado nessa cláusula, e isso pode ser feito através de uma espécie de parâmetro passado para o cursor no momento de sua abertura. Observe o exemplo a seguir:

```

DECLARE
    CURSOR c_Classes(p_Department classes.department%TYPE ,
                      p_Course classes.course%TYPE) IS
        SELECT *
        FROM classes
        WHERE department = p_Department
        AND course = p_Course;
BEGIN
    OPEN c_Classes('HIS', 101);
END;

```

## Atributos de cursor

Durante a utilização de um cursor em uma rotina, uma série de valores pode ser testada, de maneira a permitir a monitoração do estado corrente do processamento. Esses valores são obtidos através de variáveis especiais mantidas pelo sistema, chamadas de Atributos do cursor. Todos eles têm seu nome começando com o símbolo “%” (sinal de porcentagem) e são referenciados colocando-se o nome do cursor imediatamente antes do “%”. A seguir um pequeno resumo com esses atributos e suas características principais.

Atributos:

**%ISOPEN** (BOOLEAN): Indica se o cursor referenciado está aberto (TRUE) ou fechado (FALSE).

**%ROWCOUNT** (NUMBER): É um contador que indica quantas linhas já foram recuperadas através de um comando FETCH.

**%NOTFOUND** (BOOLEAN): Indica o resultado do último FETCH: se foi bem sucedido, seu valor é FALSE, senão TRUE

**%FOUND** (BOOLEAN): Indica o resultado do último FETCH: se foi bem sucedido, seu valor é TRUE, senão FALSE.

## Cursos implícitos

Um cursor deste tipo é implementado através da colocação da cláusula INTO no SELECT, conforme pode ser visto no exemplo a seguir:

Dois cuidados básicos devem ser tomados ao utilizar-se cursos implícitos:

- as variáveis que receberão os dados obtidos pelo SELECT deverão ser declaradas com tipo igual ao do campo correspondente na tabela, o que torna bastante indicado

nesses casos utilizar o atributo %TYPE ao declarar a variável, para evitar problemas de incompatibilidade;

- o comando deve retornar no máximo uma única linha, senão uma exceção TOO\_MANY\_ROWS será gerada.

Não há uma declaração formal do cursor, apenas das variáveis a serem atualizadas pelo comando.

Exemplo:

```
BEGIN
    UPDATE rooms
        SET number_seats = 100
        WHERE room_id = 99980;
    -- If the previous UPDATE statement didn't match any rows,
    -- insert a new row into the rooms table.
    IF SQL%NOTFOUND THEN
        INSERT INTO rooms (room_id, number_seats)
            VALUES (99980, 100);
    END IF;
END;

-----
BEGIN
    UPDATE rooms
        SET number_seats = 100
        WHERE room_id = 99980;
    -- If the previous UPDATE statement didn't match any rows,
    -- insert a new row into the rooms table.
    IF SQL%ROWCOUNT = 0 THEN
        INSERT INTO rooms (room_id, number_seats)
            VALUES (99980, 100);
    END IF;
END;
```

## Loops de Busca de Cursos

A operação mais comum com os cursores é busca todas as linhas no conjunto ativo. Isso é feito via um loop de busca, o que é simplesmente um loop que processa cada uma das linhas no conjunto ativo, uma por uma.

## **Loop Simples**

```
DECLARE
    v_StudentID      students.id%TYPE;
    v_FirstName      students.first_name%TYPE;
    v_LastName       students.last_name%TYPE;

    students
    CURSOR c_HistoryStudents IS
        SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';
BEGIN
    OPEN c_HistoryStudents;
    LOOP
        FETCH c_HistoryStudents INTO v_StudentID, v_FirstName,
v_LastName;

        EXIT WHEN c_HistoryStudents%NOTFOUND;

        INSERT INTO registered_students (student_id, department,
course)
        VALUES (v_StudentID, 'HIS', 301);

        INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentID, v_FirstName || ' ' || v_LastName);

    END LOOP;

    CLOSE c_HistoryStudents;
END;
/
```

## **Loops WHILE**

```
DECLARE
    CURSOR c_HistoryStudents IS
        SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';

    v_StudentData   c_HistoryStudents%ROWTYPE;
BEGIN
    OPEN c_HistoryStudents;
```

```

FETCH c_HistoryStudents INTO v_StudentData;

WHILE c_HistoryStudents%FOUND LOOP
    INSERT INTO registered_students (student_id, department,
course)
        VALUES (v_StudentData.ID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentData.ID,
                v_StudentData.first_name || ' ' ||
v_StudentData.last_name);

    FETCH c_HistoryStudents INTO v_StudentData;
END LOOP;

CLOSE c_HistoryStudents;
END;
/

```

## Loops FOR de cursor

Esta é uma variação do cursor explícito em que este fica embutido em uma estrutura FOR responsável pelo seu processamento. É uma espécie de simplificação, pois nessa estrutura o cursor é aberto uma vez, as linhas são processadas (uma a cada passagem do looping) e o cursor é fechado automaticamente no final das iterações.

O <nome do registro> é um nome de variável do tipo RECORD criada automaticamente na entrada do looping. O cursor propriamente dito é declarado normalmente na área de declarações da rotina, e pode prever a utilização de parâmetros. Os atributos de cursor vistos anteriormente estão disponíveis normalmente neste tipo de estrutura.

A sintaxe básica é a seguinte:

```

FOR <nome do registro> IN <nome do cursor> LOOP
...
END LOOP ;

```

Exemplo:

```

DECLARE
    CURSOR c_HistoryStudents IS
        SELECT id, first_name, last_name
        FROM students
        WHERE major = 'History';
BEGIN
    FOR v_StudentData IN c_HistoryStudents LOOP

```

```

    INSERT INTO registered_students (student_id, department,
course)
        VALUES (v_StudentData.ID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentData.ID,
                v_StudentData.first_name || ' ' ||
v_StudentData.last_name);
END LOOP;

END;
/

```

## Loops FOR implícitos

A sintaxe para um loop FOR pode ser abreviada ainda mais. Além do registro, o próprio cursor pode ser implicitamente declarado, como o seguinte exemplo ilustra:

```

BEGIN
  FOR v_StudentData IN (SELECT id, first_name, last_name
                        FROM students
                        WHERE major = 'History') LOOP
    INSERT INTO registered_students (student_id, department,
course)
        VALUES (v_StudentData.ID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
        VALUES (v_StudentData.ID,
                v_StudentData.first_name || ' ' ||
v_StudentData.last_name);
  END LOOP;
END;

```

A consulta é contida dentro de parênteses, dentro da própria instrução FOR. Nesse caso, tanto o registro v\_StudentData como o cursor é implicitamente declarado. Entretanto, o cursor não tem nenhum nome.

## NO\_DATA\_FOUND versus %NOTFOUND

A exceção NO\_DATA\_FOUND é levantada apenas em instruções SELECT...INTO, quando a cláusula WHERE da consulta não corresponde a nenhuma linha. Quando nenhuma linha é retornada por um cursor explícito ou uma instrução UPDATE ou DELETE, o atributo %NOTFOUND é configurado como TRUE, não levantando a exceção NO\_DATA\_FOUND.

## Cursos SELECT FOR UPDATE

Frequentemente, o processamento feito em um loop de busca modifica as linhas que foram recuperadas pelo cursor. A PL/SQL fornece uma sintaxe conveniente para fazer isso. Esse método consiste em duas partes: a cláusula FOR UPDATE na declaração do cursor e a cláusula WHERE CURRENT em uma instrução UPDATE ou DELETE.

### FOR UPDATE

A cláusula FOR UPDATE é a parte de uma instrução SELECT. Ela é válida como a última cláusula da instrução, depois da cláusula ORDER BY (caso exista).

Sintaxe:

```
SELECT ... FROM ... FOR UPDATE [OF colunas] [{NOWAIT | WAIT n}]
```

Exemplo:

```
DECLARE
    CURSOR c_AllStudents IS
        SELECT * FROM students
        FOR UPDATE OF first_name, last_name;
```

### WHERE CURRENT OF

Se o cursor for declarado FOR UPDATE, a cláusula WHERE CURRENT OF pode ser utilizada em uma instrução UPDATE ou DELETE para referenciar a linha corrente do cursor.

Sintaxe:

```
{UPDATE | DELETE} tabela... WHERE CURRENT OF cursor
```

Exemplo:

```
DECLARE
    v_NumCredits    classes.num_credits%TYPE;

    CURSOR c_RegisteredStudents IS
        SELECT *
        FROM students
        WHERE id IN (SELECT student_id
                      FROM registered_students
                      WHERE department= 'HIS'
                        AND course = 101)
```

```

        FOR UPDATE OF current_credits;

BEGIN
    FOR v_StudentInfo IN c_RegisteredStudents LOOP

        SELECT num_credits
            INTO v_NumCredits
            FROM classes
            WHERE department = 'HIS'
            AND course = 101;

        UPDATE students
            SET current_credits = current_credits + v_NumCredits
            WHERE CURRENT OF c_RegisteredStudents;
    END LOOP;

    COMMIT;
END;
/

```

### **COMMIT dentro de um Loop de cursor FOR UPDATE**

Quando houver um COMMIT dentro de um loop de cursor SELECT ... FOR UPDATE, quaisquer tentativa de busca após o COMMIT retornará erro ORA-1002 pois o cursor é invalidado após essa operação para liberar os bloqueios nas linhas das tabelas. Contudo, você pode utilizar a chave primária da tabela na cláusula WHERE de UPDATE, como ilustrado pelo seguinte exemplo:

```

DECLARE
    v_NumCredits    classes.num_credits%TYPE;

CURSOR c_RegisteredStudents IS
    SELECT *
        FROM students
        WHERE id IN (SELECT student_id
                      FROM registered_students
                      WHERE department= 'HIS'
                      AND course = 101);

BEGIN
    FOR v_StudentInfo IN c_RegisteredStudents LOOP

        SELECT num_credits
            INTO v_NumCredits
            FROM classes
            WHERE department = 'HIS'
            AND course = 101;

```

```

UPDATE students
    SET current_credits = current_credits + v_NumCredits
    WHERE id = v_Studentinfo.id;

    COMMIT;
END LOOP;
END;
/

```

## Variáveis de cursor

Uma variável de cursor é um tipo referência. Um tipo referência pode referir-se a posições diferentes de memória à medida que o programa é executado. A sintaxe para definir um tipo de variável de cursor é:

```
TYPE nome_do_tipo IS REF CURSOR [RETURN tipo_de_retorno]
```

Onde *nome\_do\_tipo* é o nome do novo tipo de referência e o *tipo\_de\_retorno* é um tipo de registro que indica os tipos da lista SELECT que por fim retornação pela variável cursor.

A fim de associar uma variável de cursor com uma instrução SELECT em particular, a sintaxe OPEN é estendida para permitir que a consulta seja especificada. Veja a sintaxe:

```
OPEN variável_de_cursor FOR instrução_select;
```

Exemplo:

```

SQL> set serveroutput on
SQL> DECLARE
2      TYPE TMeuCursor IS REF CURSOR;
3      v_Cursor TMeuCursor;
4      v_EmpRec emp%ROWTYPE;
5      v_DeptRec dept%ROWTYPE;
6  BEGIN
7      --Utilizar tabela EMP
8      OPEN v_Cursor FOR select * from emp;
9      FETCH v_Cursor INTO v_EmpRec;
10     DBMS_OUTPUT.PUT_LINE(v_EmpRec.empno || '-'
' || v_EmpRec.ename);
11     CLOSE v_Cursor;
12     --Utilizar tabela DEPT
13     OPEN v_Cursor FOR select * from dept;
14     FETCH v_Cursor INTO v_DeptRec;

```

```

15      DBMS_OUTPUT.PUT_LINE(v_DeptRec.deptno || '-
' || v_DeptRec.dname);
16      CLOSE v_Cursor;
17  end;
18 /

```

7369-SMITH  
10-ACCOUNTING

PL/SQL procedure successfully completed.

O exemplo abaixo é uma procedure armazenada que seleciona linhas da tabela classes ou rooms, dependendo da sua entrada.

```

CREATE OR REPLACE PROCEDURE ShowCursorVariable
(p_Table IN VARCHAR2) AS

TYPE t_ClassesRooms IS REF CURSOR;

v_CursorVar t_ClassesRooms;

v_Department    classes.department%TYPE;
v_Course        classes.course%TYPE;
v_RoomID        rooms.room_id%TYPE;
v_Description   rooms.description%TYPE;

BEGIN
  IF p_Table = 'classes' THEN
    OPEN v_CursorVar FOR
      SELECT department, course
        FROM classes;
  ELSIF p_table = 'rooms' THEN
    OPEN v_CursorVar FOR
      SELECT room_id, description
        FROM rooms;
  ELSE
    RAISE_APPLICATION_ERROR(-20000,
      'Input must be ''classes'' or ''rooms''' );
  END IF;

LOOP
  IF p_Table = 'classes' THEN
    FETCH v_CursorVar INTO
      v_Department, v_Course;
    EXIT WHEN v_CursorVar%NOTFOUND;

    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_Course, v_Department);
  END IF;
END;

```

```
ELSE
  FETCH v_CursorVar INTO
    v_RoomID, v_Description;
  EXIT WHEN v_CursorVAR%NOTFOUND;

  INSERT INTO temp_table (num_col, char_col)
    VALUES (v_RoomID, SUBSTR(v_Description, 1, 60));
END IF;
END LOOP;

CLOSE v_CursorVar;

COMMIT;
END ShowCursorVariable;
```

## COLEÇÕES

Com freqüência, é conveniente manipular várias variáveis de uma vez como uma unidade. Esses tipos de dados são conhecidos como coleções. O Oracle7 fornecia um tipo de coleção: tabela index-by. O Oracle8i acrescentou outros dois tipos de coleção: tabelas aninhadas e varrays. O Oracle9i adiciona a capacidade de criar coleções de múltiplos níveis, isto é, coleção de coleções.

### Tabelas Index-by

Uma tabela index-by é semelhante a uma tabela do banco de dados, com duas colunas: key e value. O tipo de key é BINARY\_INTEGER e o tipo de value é qualquer tipo de dados especificado na definição. Sintaxe:

```
TYPE nome_do_tipo IS TABLE OF tipo INDEX BY BINARY_INTEGER;
```

Exemplo:

```
DECLARE
    TYPE NameTab IS TABLE OF students.first_name%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE DateTab IS TABLE OF DATE
        INDEX BY BINARY_INTEGER;
    v_Names NameTab;
    v_Dates DateTab;
BEGIN
    v_Names(1) := 'Scott';
    v_Dates(-4) := SYSDATE - 1;
END;
/
```

Não há limites sobre o número de elementos de uma tabela index-by. O único limite está sobre o número de linhas que uma chave BINARY\_INTEGER pode assumir (-2147483674 ... +2147483647). É importante lembrar também que os elementos de uma tabela index-by não estão necessariamente em uma ordem específica, qualquer valor do tipo BINARY\_INTEGER pode ser utilizado como índice de elemento da tabela.

### Elementos inexistentes

Qualquer tentativa de referenciar um elemento não existente em uma tabela index-by retornará o erro ORA-1403: NO DATA FOUND.

### Tabelas index-by de tipos compostos

O Oracle9i permite tabelas index-by de qualquer tipo de coleção.

## Tabelas index-by de registros

```
DECLARE
    TYPE StudentTab IS TABLE OF students%ROWTYPE
        INDEX BY BINARY_INTEGER;

    v_Students StudentTab;
BEGIN

    SELECT *
        INTO v_Students(10001)
        FROM students
        WHERE id = 10001;

    v_Students(1).first_name := 'Larry';
    v_Students(1).last_name := 'Lemon';
END;
/
```

## Tabelas index-by de tipos de objeto

```
CREATE OR REPLACE TYPE MyObject AS OBJECT (
    field1 NUMBER,
    field2 VARCHAR2(20),
    field3 DATE);
/

DECLARE
    TYPE ObjectTab IS TABLE OF MyObject
        INDEX BY BINARY_INTEGER;
    v_Objects ObjectTab;
BEGIN

    v_Objects(1) := MyObject(1, NULL, NULL);
    v_Objects(1).field2 := 'Hello World!';
    v_Objects(1).field3 := SYSDATE;
END;
/
```

## Tabelas aninhadas (Nested tables)

A funcionalidade básica de uma tabela aninhada é a mesma de uma tabela index-by, com a diferença de possuírem chaves seqüenciais e não negativas. Além disso, as tabelas aninhadas podem ser armazenadas como colunas no banco de dados, enquanto as tabelas index-by não podem. O número máximo de linhas em uma tabela aninhada é de 2 gigabytes. Sintaxe:

```
TYPE nome_do_tipo IS TABLE OF tipo [NOT NULL];
```

O *tipo* da tabela não pode ser BOOLEAN, NCHAR, NCLOB, NVARCHAR2 ou REF CURSOR. Se NOT NULL estiver presente, os elementos da tabela aninhada não podem ser NULL.

Exemplo:

```
DECLARE
    TYPE ObjectTab IS TABLE OF MyObject;
    TYPE StudentsTab IS TABLE OF students%ROWTYPE;
    v_StudentList StudentsTab;
    v_ObjectList ObjectTab;
```

### Inicialização de uma tabela aninhada

Quando uma tabela index-by é criada, mas ainda não tem nenhum elemento, ela simplesmente está vazia. Contudo, quando uma tabela aninhada é declarada, mas ainda não tem nenhum elemento, ela é inicializada como NULL. Se você tentar adicionar um elemento em uma tabela aninhada NULL, receberá o erro ORA-6531 que corresponde à exceção COLLECTION\_IS\_NULL.

Exemplo:

```
DECLARE
    TYPE ObjectTab IS TABLE OF MyObject;
    TYPE StudentsTab IS TABLE OF students%ROWTYPE;
    v_StudentList StudentsTab;
    v_ObjectList ObjectTab;
BEGIN
    v_ObjectList(1) :=
        MyObject(-17,    'Goodbye' ,    TO_DATE('01-01-2001',    'DD-MM-
YYYY'));
END;

ORA-06531: Reference to uninitialized collection
ORA-06512: at line 7
```

Portanto você deve inicializar uma tabela aninhada através de seu construtor, definindo o número de elementos disponíveis para referenciar.

Exemplo:

```
DECLARE
    TYPE NumbersTab IS TABLE OF NUMBER;
```

```

-- Create a table with one element.
v_Tab1 NumbersTab := NumbersTab(-1);
-- Create a table with five elements.
v_Primes NumbersTab := NumbersTab(1, 2, 3, 5, 7);
-- Create a table with no elements.
v_Tab2 NumbersTab := NumbersTab();
BEGIN
-- Assign to v_Tab1(1). This will replace the value already
-- in v_Tab1(1), which was initialized to -1.
v_Tab1(1) := 12345;

-- Print out the contents of v_Primes.
FOR v_Count IN 1..5 LOOP
    DBMS_OUTPUT.PUT(v_Primes(v_Count) || ' ');
END LOOP;
DBMS_OUTPUT.NEW_LINE;
END;
/

```

## Tabelas vazias

```

DECLARE
    TYPE WordsTab IS TABLE OF VARCHAR2(50);

    -- Create a NULL table.
    v_Tab1 WordsTab;

    -- Create a table with one element, which itself is NULL.
    v_Tab2 WordsTab := WordsTab();
BEGIN
    IF v_Tab1 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('v_Tab1 is NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('v_Tab1 is not NULL');
    END IF;

    IF v_Tab2 IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('v_Tab2 is NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('v_Tab2 is not NULL');
    END IF;
END;
/
v_Tab1 is NULL
v_Tab2 is not NULL
PL/SQL procedure successfully completed.

```

## Adicionando elementos a uma tabela existente

Não é possível atribuir valor a um elemento que ainda não existe em uma tabela aninhada. Caso você tente fazer isso, receberá o erro ORA-6533 correspondente à exceção SUBSCRIPT\_BEYOND\_COUNT.

Exemplo:

```
DECLARE
    TYPE NumbersTab IS TABLE OF NUMBER;
    v_Numbers NumbersTab := NumbersTab(1, 2, 3);
BEGIN
    -- v_Numbers was initialized to have 3 elements. So the
    -- following assignments are all legal.
    v_Numbers(1) := 7;
    v_Numbers(2) := -1;

    -- However, this assignment will raise ORA-6533.
    v_Numbers(4) := 4;
END;
```

ORA-06533: Subscript beyond count  
ORA-06512: at line 11

Você pode aumentar o tamanho de uma tabela aninhada utilizando o método EXTEND

## VARRAYS

Um varray (array de comprimento variável) é um tipo de dados bastante semelhante a um array em C ou Java. Sintaticamente é acessado semelhantemente a uma tabela aninhada ou index-by. Contudo, possui um limite superior fixo em relação ao seu tamanho. Seus elementos iniciam no índice 1 e vão até o comprimento definido em sua declaração, com tamanho máximo de 2 gigabytes. Os varrays também podem ser armazenados como colunas no banco de dados. Sintaxe:

```
TYPE nome_do_tipo IS VARRAY(tamanho_Maximo) OF tipo [NOT NULL];
```

Exemplo:

```
DECLARE
    TYPE NumberList IS VARRAY(10) OF NUMBER(3) NOT NULL;

    -- A list of PL/SQL records.
    TYPE StudentList IS VARRAY(100) OF students%ROWTYPE;

    -- A list of objects.
    TYPE ObjectList is VARRAY(25) OF MyObject;
```

```
BEGIN
    NULL;
END;
/
```

## Inicialização de varray

Semelhantes às tabelas aninhadas, os varrays são inicializados utilizando um construtor. O número de argumentos passado ao construtor torna-se o comprimento inicial do varray e deve ser menor ou igual ao comprimento máximo especificado no tipo de varray.

Exemplo:

```
DECLARE
    -- Define a VARRAY type.
    TYPE Numbers IS VARRAY(20) OF NUMBER(3);

    -- Declare a NULL varray.
    v_NullList Numbers;

    -- This varray has 2 elements.
    v_List1 Numbers := Numbers(1, 2);

    -- This varray has one element, which itself is NULL.
    v_List2 Numbers := Numbers(NULL);
BEGIN
    IF v_NullList IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('v_NullList is NULL');
    END IF;

    IF v_List2(1) IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('v_List2(1) is NULL');
    END IF;
END;
/
v_NullList is NULL
v_List2(1) is NULL
PL/SQL procedure successfully completed.
```

## Manipulando os elementos de um varray

A manipulação dos elementos de um varray devem respeitar o número de elementos utilizados no construtor. As atribuições para elementos fora desse intervalo levantarão o erro ORA-6533.

Exemplo:

```
DECLARE
    TYPE Strings IS VARRAY(5) OF VARCHAR2(10);

    -- Declare a varray with three elements
    v_List Strings :=
        Strings('One', 'Two', 'Three');
BEGIN
    -- Subscript between 1 and 3, so this is legal.
    v_List(2) := 'TWO';

    -- Subscript beyond count, raises ORA-6533.
    v_List(4) := '!!!!';
END;
```

ORA-06533: Subscript beyond count

ORA-06512: at line 12

O tamanho de varray pode ser aumentado utilizando o método EXTEND

Tentativas de atribuição para os elementos fora do tamanho máximo do varray ou tentativas para estender além do tamanho máximo de um varray levantarão o erro ORA-6532 equivalente à exceção SUBSCRIPT\_OUTSIDE\_LIMIT.

### Coleções de múltiplos níveis

O Oracle9i permite coleções de mais de uma dimensão, isto é, coleção de coleções.

Exemplo:

```
DECLARE
    TYPE t_Numbers IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;

    TYPE t_MultiNumbers IS TABLE OF t_Numbers
        INDEX BY BINARY_INTEGER;

    TYPE t_MultiVarray IS VARRAY(10) OF t_Numbers;

    TYPE t_MultiNested IS TABLE OF t_Numbers;

    v_MultiNumbers t_MultiNumbers;
BEGIN
    v_MultiNumbers(1)(1) := 12345;
END;
/
```

## Coleções no banco de dados

Armazenar coleções no banco de dados tem implicações relacionadas à maneira como os tipos de tabela precisam ser declarados, bem como com relação à sintaxe para criar tabelas com colunas de coleção.

A fim de armazenar e recuperar uma coleção a partir de uma tabela no banco de dados, o tipo de coleção deve ser conhecido tanto para a PL/SQL como para a SQL. Isso significa que ele não pode ser local para um bloco PL/SQL e, em vez disso deve ser declarado utilizando uma instrução SQL CREATE TYPE, semelhante a um tipo de objeto.

Exemplo:

```
SQL> CREATE OR REPLACE TYPE NameList AS
  2  VARRAY(20) OF VARCHAR2(30);
  3 /
Type created

SQL> DECLARE
  2      -- This type is local to this block.
  3      TYPE DateList IS VARRAY(10) OF DATE;
  4
  5
  6      v_Dates DateList;
  7      v_Names NameList;
  8  BEGIN
  9      NULL;
 10 END;
 11 /
PL/SQL procedure successfully completed

SQL> DECLARE
  2
  3
  4      v_Names2 NameList;
  5  BEGIN
  6      NULL;
  7  END;
  8 /
PL/SQL procedure successfully completed
```

## A estrutura de varrays armazenados

Um varray pode ser utilizado como o tipo em uma coluna do banco de dados. Nesse

caso, o varray inteiro é armazenado dentro de uma linha do banco de dados, ao lado das outras colunas.

Exemplo:

```
CREATE OR REPLACE TYPE BookList AS VARRAY(10) OF NUMBER(4);
/
CREATE TABLE class_material (
    department      CHAR(3),
    course         NUMBER(3),
    required_reading BookList
);
```

### Estrutura das tabelas aninhadas armazenadas

Da mesma maneira como um varray, uma tabela aninhada pode ser armazenada como uma coluna de banco de dados.

Exemplo:

```
CREATE OR REPLACE TYPE StudentList AS TABLE OF NUMBER(5);
/
CREATE TABLE library_catalog (
    catalog_number NUMBER(4),
    FOREIGN KEY (catalog_number) REFERENCES
books(catalog_number),
    num_copies      NUMBER,
    num_out         NUMBER,
    checked_out     StudentList)
NESTED TABLE checked_out STORE AS co_tab;
```

A cláusula NESTED TABLE é requerida para cada tabela aninhada em uma dada tabela do banco de dados. Essa cláusula indica o nome da tabela de armazenamento. Uma tabela de armazenamento é uma tabela gerada pelo sistema que é utilizado para armazenar os dados reais na tabela aninhada.

### Manipulando coleções inteiras

Você pode manipular uma coleção armazenada em sua totalidade utilizando as instruções SQL DML.

#### INSERT

```
DECLARE
    v_CSBooks BookList := BookList(1000, 1001, 1002);
```

```

v_HistoryBooks BookList := BookList(2001);
BEGIN
  INSERT INTO class_material
    VALUES ('MUS', 100, BookList(3001, 3002));

  INSERT INTO class_material VALUES ('CS', 102, v_CSBooks);

  INSERT INTO class_material VALUES ('HIS', 101,
v_HistoryBooks);
END;
/

```

## UPDATE

```

DECLARE
  v_StudentList1 StudentList := StudentList(10000, 10002, 10003);
  v_StudentList2 StudentList := StudentList(10000, 10002, 10003);
  v_StudentList3 StudentList := StudentList(10000, 10002, 10003);
BEGIN
  -- First insert rows with NULL nested tables.
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (1000, 20, 3);
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (1001, 20, 3);
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (1002, 10, 3);
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (2001, 50, 0);
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (3001, 5, 0);
  INSERT INTO library_catalog (catalog_number, num_copies, num_out)
    VALUES (3002, 5, 1);

  -- Now update using the PL/SQL variables.
  UPDATE library_catalog
    SET checked_out = v_StudentList1
    WHERE catalog_number = 1000;
  UPDATE library_catalog
    SET checked_out = v_StudentList2
    WHERE catalog_number = 1001;
  UPDATE library_catalog
    SET checked_out = v_StudentList3
    WHERE catalog_number = 1002;

  -- And update the last row using a new variable.
  UPDATE library_catalog
    SET checked_out = StudentList(10009)
    WHERE catalog_number = 3002;
END;
/

```

## **DELETE**

```
DELETE FROM library_catalog
WHERE catalog_number = 3001;
```

## **SELECT**

### **Consultando varrays**

```
DECLARE
  v_Books class_material.required_reading%TYPE;
  v_Title books.title%TYPE;
BEGIN

  SELECT required_reading
    INTO v_Books
   FROM class_material
  WHERE department = :p_Department
    AND course = :p_Course;

  FOR v_Index IN 1..v_Books.COUNT LOOP
    SELECT title
      INTO v_Title
     FROM books
    WHERE catalog_number = v_Books(v_Index);

    DBMS_OUTPUT.PUT_LINE(v_Books(v_Index) || ':' || 
v_Title);
  END LOOP;
END;
/
```

### **Consultando tabelas aninhadas**

```
DECLARE
  v_StudentList StudentList;
  v_Student students%ROWTYPE;
  v_Book     books%ROWTYPE;
  v_FoundOne BOOLEAN := FALSE;
BEGIN
  -- Select the entire nested table into a PL/SQL variable.
  SELECT checked_out
    INTO v_StudentList
   FROM library_catalog
```

```

        WHERE catalog_number = :p_CatalogNumber;

-- Loop over the nested table, and print out the student
names.
IF v_StudentList IS NOT NULL THEN
    FOR v_Index IN 1..v_StudentList.COUNT LOOP
        SELECT *
        INTO v_Student
        FROM students
        WHERE ID = v_StudentList(v_Index);

        DBMS_OUTPUT.PUT_LINE(v_Student.first_name || ' ' ||
v_Student.last_name);
    END LOOP;
END IF;
END;

```

Os elementos de uma tabela aninhada é iniciado pelo índice 1 e o último pode ser determinado pelo método COUNT

## Operadores de tabela SQL

Você também pode manipular os elementos de uma tabela aninhada armazenada utilizando diretamente a linguagem SQL com o operador TABLE. Os elementos de varrays armazenados não podem ser manipulados diretamente com a DML, e sim na PL/SQL.

Exemplos:

```

select column_value ID from
    TABLE(select checked_out from library_catalog
          where catalog_number=1003);

select department, course, column_value
    FROM class_material, TABLE(required_reading);
INSERT INTO TABLE(select checked_out from library_catalog
where catalog_number=1000)
VALUES(60);

UPDATE TABLE(select checked_out from library_catalog where
catalog_number=1000)
set column_value = 30  where column_value=6

```

## Métodos de coleção

Tabelas aninhadas e varrays são tipos de objeto e como tal tem métodos definidos para eles. Da mesma forma, tabelas index-by têm atributos.

## EXISTS

É utilizado para determinar se o elemento referenciado está presente na coleção.

Exemplo:

```
DECLARE
    v_NestedTable NumTab := NumTab(-7, 14.3, 3.14159, NULL, 0);
    v_Count BINARY_INTEGER := 1;
    v_IndexByTable IndexBy.NumTab;
BEGIN
    LOOP
        IF v_NestedTable.EXISTS(v_Count) THEN
            DBMS_OUTPUT.PUT_LINE(
                'v_NestedTable(' || v_Count || '): ' ||
                v_NestedTable(v_Count));
            v_Count := v_Count + 1;
        ELSE
            EXIT;
        END IF;
    END LOOP;

    v_IndexByTable(1) := -7;
    v_IndexByTable(2) := 14.3;
    v_IndexByTable(3) := 3.14159;
    v_IndexByTable(4) := NULL;
    v_IndexByTable(5) := 0;

    v_Count := 1;
    LOOP
        IF v_IndexByTable.EXISTS(v_Count) THEN
            DBMS_OUTPUT.PUT_LINE(
                'v_IndexByTable(' || v_Count || '): ' ||
                v_IndexByTable(v_Count));
            v_Count := v_Count + 1;
        ELSE
            EXIT;
        END IF;
    END LOOP;
END;
/
```

## COUNT

Retorna o número de elementos atualmente em uma coleção.

Exemplo:

```
DECLARE
    v_NestedTable NumTab := NumTab(1, 2, 3);
    v_Varray NumVar := NumVar(-1, -2, -3, -4);
    v_IndexByTable IndexBy.NumTab;
BEGIN
    v_IndexByTable(1) := 1;
    v_IndexByTable(8) := 8;
    v_IndexByTable(-1) := -1;
    v_IndexByTable(100) := 100;

    DBMS_OUTPUT.PUT_LINE(
        'Nested Table Count: ' || v_NestedTable.COUNT);
    DBMS_OUTPUT.PUT_LINE(
        'Varray Count: ' || v_Varray.COUNT);
    DBMS_OUTPUT.PUT_LINE(
        'Index-By Table Count: ' || v_IndexByTable.COUNT);
END;
/
```

## LIMIT

Retorna o número máximo atual de elementos. Para tabelas aninhadas que são ilimitados, retornará sempre NULL. Tabelas index-by não possuem tal método.

Exemplo:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      type NumTab IS TABLE OF NUMBER;
  3      type NumVar IS VARRAY(25) OF NUMBER;
  4      v_Table NumTab := NumTab(1, 2, 3);
  5      v_Varray NumVar := NumVar(1234, 4321);
  6  BEGIN
  7      DBMS_OUTPUT.PUT_LINE('Varray limit: ' ||
v_Varray.LIMIT);
  8      DBMS_OUTPUT.PUT_LINE('Varray count: ' ||
v_Varray.COUNT);
  9      IF v_Table.LIMIT IS NULL THEN
10          DBMS_OUTPUT.PUT_LINE('Table limit is NULL');
11      ELSE
12          DBMS_OUTPUT.PUT_LINE('Table limit: ' ||
v_Table.LIMIT);
13      END IF;
```

```

14      DBMS_OUTPUT.PUT_LINE('Table count: ' || 
v_Table.COUNT);
15  END;
16 /
Varray limit: 25
Varray count: 2
Table limit is NULL
Table count: 3

PL/SQL procedure successfully completed.

```

## FIRST e LAST

FIRST retorna o índice do primeiro elemento de uma coleção e LAST retorna o índice do último elemento. Esses métodos podem ser usados juntamente com NEXT e PRIOR para fazer um loop em uma coleção.

## NEXT e PRIOR

**São usados para incrementar e decrementar a chave de uma coleção. Caso não houver nenhum elemento antes ou depois do índice atual, esses métodos retornarão NULL.**

Exemplo:

```

DECLARE
  TYPE CharTab IS TABLE OF CHAR(1);
  v_Characters CharTab :=
    CharTab('M', 'a', 'd', 'a', 'm', ' ', ' ', ' ',
            'I', ' ', 'm', ' ', 'A', 'd', 'a', 'm');
  v_Index INTEGER;
BEGIN
  v_Index := v_Characters.FIRST;
  WHILE v_Index <= v_Characters.LAST LOOP
    DBMS_OUTPUT.PUT(v_Characters(v_Index));
    v_Index := v_Characters.NEXT(v_Index);
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;

  v_Index := v_Characters.LAST;
  WHILE v_Index >= v_Characters.FIRST LOOP
    DBMS_OUTPUT.PUT(v_Characters(v_Index));
    v_Index := v_Characters.PRIOR(v_Index);
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;
END;

```

/

## EXTEND

É utilizado para adicionar elementos NULL ao final de uma tabela aninhada ou de um varray. Não é válido para tabelas index-by. EXTEND tem três formas:

EXTEND  
EXTEND(n)  
EXTEND(n,i)

Uma tabela aninhada não tem tamanho máximo, já um varray pode ser estendido apenas até o tamanho máximo definido em sua declaração.

### Exemplo:

```
SQL> set serveroutput on
SQL>
SQL> DECLARE
 2   type NumTab IS TABLE OF NUMBER;
 3   type NumVar IS VARRAY(25) OF NUMBER;
 4
 5   v_NumbersTab NumTab := NumTab(1, 2, 3, 4, 5);
 6   v_NumbersList NumVar := NumVar(1, 2, 3, 4, 5);
 7 BEGIN
 8   BEGIN
 9     v_NumbersTab(26) := -7;
10   EXCEPTION
11     WHEN SUBSCRIPT_BEYOND_COUNT THEN
12       DBMS_OUTPUT.PUT_LINE(
13         'ORA-6533 raised for assignment to
v_NumbersTab(26)');
14   END;
15
16   v_NumbersTab.EXTEND(30);
17
18   v_NumbersTab(26) := -7;
19
20   BEGIN
21     v_NumbersList.EXTEND(30);
22   EXCEPTION
23     WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN
24       DBMS_OUTPUT.PUT_LINE(
25         'ORA-6532 raised for v_NumbersList.EXTEND(30)');
26   END;
27
28   v_NumbersList.EXTEND(20);
```

```

29
30      v_NumbersList(25) := 25;
31  END;
32 /
ORA-6533 raised for assignment to v_NumbersTab(26)
ORA-6532 raised for v_NumbersList.EXTEND(30)

```

PL/SQL procedure successfully completed.

## TRIM

É utilizado para remover elementos do final de uma tabela aninhada ou de um varray.

Exemplo:

```

SQL> set serveroutput on
SQL>
SQL> DECLARE
2      type NumTab IS TABLE OF  NUMBER;
3
4      v_Numbers NumTab := NumTab(-3, -2, -1, 0, 1, 2, 3);
5
6      -- Local procedure to print out a table.
7      PROCEDURE Print(p_Table IN NumTab) IS
8          v_Index INTEGER;
9      BEGIN
10         v_Index := p_Table.FIRST;
11         WHILE v_Index <= p_Table.LAST LOOP
12             DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
13             DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
14             v_Index := p_Table.NEXT(v_Index);
15         END LOOP;
16         DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
17         DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
18     END Print;
19
20 BEGIN
21     DBMS_OUTPUT.PUT_LINE('At initialization, v_Numbers
contains');
22     Print(v_Numbers);
23
24     v_Numbers.DELETE(6);
25     DBMS_OUTPUT.PUT_LINE('After delete , v_Numbers
contains');
26     Print(v_Numbers);
27     v_Numbers.TRIM(3);

```

```

28      DBMS_OUTPUT.PUT_LINE('After trim, v_Numbers
contains');
29      Print(v_Numbers);
30  END;
31 /
At initialization, v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 6: 2
Element 7: 3
COUNT = 7
LAST = 7
After delete , v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 7: 3
COUNT = 6
LAST = 7
After trim, v_Numbers contains
Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
COUNT = 4
LAST = 4

PL/SQL procedure successfully completed.

```

## **DELETE**

Remove um ou mais elementos de uma tabela index-by ou de uma tabela aninhada.  
 Não tem nenhum efeito em um varray por causa do seu tamanho fixo. Possui três formas:

- DELETE
- DELETE(n)
- DELETE(m,n)

**Depois de DELETE, COUNT será menor, refletindo o novo tamanho da tabela.**

Exemplo:

```

SQL> set serveroutput on
SQL>
SQL> DECLARE
 2      type NumTab IS TABLE OF  NUMBER;
 3
 4      v_Numbers NumTab := NumTab(10, 20, 30, 40, 50, 60, 70,
 5      80, 90, 100);
 6
 7      PROCEDURE Print(p_Table IN NumTab) IS
 8          v_Index INTEGER;
 9      BEGIN
10          v_Index := p_Table.FIRST;
11          WHILE v_Index <= p_Table.LAST LOOP
12              DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
13              DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
14              v_Index := p_Table.NEXT(v_Index);
15          END LOOP;
16          DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
17          DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
18      END Print;
19
20      BEGIN
21          DBMS_OUTPUT.PUT_LINE('At initialization, v_Numbers
contains');
22          Print(v_Numbers);
23          DBMS_OUTPUT.PUT_LINE('After delete(6), v_Numbers
contains');
24          v_Numbers.DELETE(6);
25          Print(v_Numbers);
26
27          DBMS_OUTPUT.PUT_LINE('After delete(7,9), v_Numbers
contains');
28          v_Numbers.DELETE(7,9);
29          Print(v_Numbers);
30      END;
31  /
At initialization, v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 6: 60
Element 7: 70
Element 8: 80
Element 9: 90

```

```
Element 10: 100
COUNT = 10
LAST = 10
After delete(6), v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 7: 70
Element 8: 80
Element 9: 90
Element 10: 100
COUNT = 9
LAST = 10
After delete(7,9), v_Numbers contains
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
Element 10: 100
COUNT = 6
LAST = 10
```

PL/SQL procedure successfully completed.

## CRIANDO PROCEDURES, FUNÇÕES E PACOTES

Há dois tipos principais de blocos PL/SQL:

- Anônimo  
Bloco PL/SQL iniciado com DECLARE ou BEGIN, não é armazenado diretamente no banco de dados, não pode ser chamado diretamente de outros blocos PL/SQL.
- Identificado  
Bloco PL/SQL armazenado no banco de dados e executado quando apropriado.  
Referem-se as seguintes construções – procedures, functions, pacotes e triggers.

### Procedures e funções

As procedures e funções PL/SQL assemelham-se muito com as procedures e funções em outras 3GLs (linguagens de terceira geração). Coletivamente, também são conhecidas como *subprogramas*. O exemplo a seguir cria um procedimento no banco de dados:

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
    p_FirstName    students.first_name%TYPE ,
    p_LastName     students.last_name%TYPE ,
    p_Major        students.major%TYPE ) AS
BEGIN
    -- Insert a new row in the students table. Use
    -- student_sequence to generate the new student ID, and
    -- 0 for current_credits.
    INSERT INTO students (ID, first_name, last_name,
                          major, current_credits)
        VALUES (student_sequence.nextval, p_FirstName,
p_LastName,
                p_Major, 0);
END AddNewStudent;
```

Uma vez criada, a procedure pode ser chamada de outro bloco PL/SQL:

```
BEGIN
    AddNewStudent( 'Zelda' , 'Zudnik' , 'Computer Science' );
END;
```

Pontos importantes a serem verificados:

- Quando a procedure é criada, ela é primeiramente compilada e então armazenada no banco de dados. Este código compilado pode ser executado de outro bloco PL/SQL.
- Parâmetros podem ser passados quando uma procedure é chamada. No exemplo anterior, o primeiro nome, sobrenome e a especialização dos alunos são passados à procedure em tempo de execução, e os parâmetros *p\_FirstName*, *p\_LastName*,

*p\_Major*, terão os seguintes valores, 'Zelda', 'Zudnik', 'Computer Science', respectivamente.

## Criação de subprograma

As procedures são criadas com CREATE PROCEDURE e as funções são criadas com CREATE FUNCTION.

### Criando uma procedure

A sintaxe básica da instrução CREATE [OR REPLACE] PROCEDURE é:

```
CREATE [OR REPLACE] PROCEDURE nome_de_procedure [(argumento [{IN / OUT} IN OUT]) tipo] {IS | AS}  
corpo_de_procedure
```

onde *nome\_de\_procedure* é o nome da procedure a ser criada, *argumento* é nome do parâmetro da procedure, *tipo* é o tipo do parâmetro associado e *corpo\_de\_procedure* é um bloco PL/SQL que compõe o código da procedure.

A fim de alterar o código de uma procedure, a procedure deve ser descartada e então recriada. As palavras-chave OR REPLACE permitem que a operação de recriar uma procedure seja feito.

Se a procedure existir e se as palavras-chave OR REPLACE não estiverem presentes, a instrução CREATE retornará o erro Oracle “ORA-955: Name is already used by na existing object”.

Assim como outras instruções CREATE, criar uma procedure é uma operação de DDL, portanto um COMMIT隐式 é feito tanto antes como depois do procedimento ser criado. Qualquer uma das palavras-chave IS ou AS pode ser utilizada – elas são equivalentes.

### Corpo da procedure

O corpo de uma procedure é um bloco PL/SQL com seções executáveis declarativas e de exceção. A seção declarativa esta localizada entre as palavras-chave IS ou AS e a palavra-chave BEGIN. A seção executável (a única requerida) está localizada entre as palavras-chave BEGIN e EXCEPTION ou entre as palavras-chave BEGIN e END se não houver nenhuma seção de tratamento de exceções. Se a seção de exceção estiver presente, ela é colocada entre as palavras-chave EXCEPTION e END.

A estrutura da instrução para criação de uma procedure é:

```
CREATE OR REPLACE PROCEDURE nome_da_procedure [lista_de_parametros] AS  
/* A seção declarativa entra aqui*/  
BEGIN  
/* A seção executável entra aqui*/
```

```

EXCEPTION
/* A seção de exceção entra aqui*/
END [nome_da_procedure];

```

## Criando uma função

Uma função é bem semelhante a uma procedure (parametros, seções, etc.), entretanto uma chamada de procedure é uma instrução PL/SQL por si própria, enquanto uma chamada de função é chamada como parte de uma expressão. Por exemplo, as seguintes funções retornam TRUE se a classe especificada estiver cheia em 80% ou mais e FALSE caso contrário:

```

CREATE OR REPLACE FUNCTION AlmostFull (
    p_Department classes.department%TYPE,
    p_Course      classes.course%TYPE)
RETURN BOOLEAN IS

    v_CurrentStudents NUMBER;
    v_MaxStudents      NUMBER;
    v_ReturnValue      BOOLEAN;
    v_FullPercent      CONSTANT NUMBER := 80;

BEGIN
    -- Get the current and maximum students for the requested
    -- course.
    SELECT current_students, max_students
        INTO v_CurrentStudents, v_MaxStudents
        FROM classes
        WHERE department = p_Department
        AND course = p_Course;

    -- If the class is more full than the percentage given by
    -- v_FullPercent, return TRUE. Otherwise, return FALSE.
    IF (v_CurrentStudents / v_MaxStudents * 100) >=
    v_FullPercent THEN
        v_ReturnValue := TRUE;
    ELSE
        v_ReturnValue := FALSE;
    END IF;

    RETURN v_ReturnValue;
END AlmostFull;

```

A função retorna um valor booleano. Ela pode ser chamada a partir do bloco PL/SQL, por exemplo:

```

DECLARE
    CURSOR c_Classes IS

```

```

SELECT department, course
  FROM classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Output all the classes which don't have very much room
    IF AlmostFull(v_ClassRecord.department,
v_ClassRecord.course) THEN
      DBMS_OUTPUT.PUT_LINE(v_ClassRecord.department || ' ' ||
v_ClassRecord.course || ' is almost full!');
    END IF;
  END LOOP;
END;

```

A sintaxe da função é bem semelhante à sintaxe para uma procedure:

```

CREATE [OR REPLACE] FUNCTION nome_de_função
[argumento [{IN | OUT| IN OUT}] tipo] ,
...
argumento [{IN | OUT| IN OUT}] tipo]
RETURN tipo_de_retorno {IS |AS}
corpo_da_função

```

onde *nome\_de\_função* é o nome da função, *argumento* e o *tipo* são os mesmos das procedures, *tipo\_de\_retorno* é o tipo do valor que a função retorna (requerido) e *corpo\_da\_função* é um bloco PL/SQL contendo o código da função. Assim como ocorre com as procedures, a lista de argumentos é opcional. O tipo da função é utilizado para determinar o tipo de expressão que contém a chamada de função.

### A instrução RETURN

É utilizada para retornar o controle para o ambiente de chamada com um valor. A sintaxe geral da instrução é:

```
RETURN expressão;
```

onde *expressão* é o valor a ser retornado. Pode haver mais de uma instrução RETURN em uma função, embora apenas uma delas é executada. Por exemplo:

```

CREATE OR REPLACE FUNCTION ClassInfo(
  /* Returns 'Full' if the class is completely full,
   'Some Room' if the class is over 80% full,
   'More Room' if the class is over 60% full,
   'Lots of Room' if the class is less than 60% full, and
   'Empty' if there are no students registered. */
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
RETURN VARCHAR2 IS

```

```

v_CurrentStudents NUMBER;
v_MaxStudents      NUMBER;
v_PercentFull     NUMBER;

BEGIN
    -- Get the current and maximum students for the requested
    -- course.
    SELECT current_students, max_students
        INTO v_CurrentStudents, v_MaxStudents
        FROM classes
        WHERE department = p_Department
        AND course = p_Course;

    -- Calculate the current percentage.
    v_PercentFull := v_CurrentStudents / v_MaxStudents * 100;

    IF v_PercentFull = 100 THEN
        RETURN 'Full';
    ELSIF v_PercentFull > 80 THEN
        RETURN 'Some Room';
    ELSIF v_PercentFull > 60 THEN
        RETURN 'More Room';
    ELSIF v_PercentFull > 0 THEN
        RETURN 'Lots of Room';
    ELSE
        RETURN 'Empty';
    END IF;
END ClassInfo;

```

A instrução RETURN também pode ser utilizada em uma procedure. Nesse caso, ela não tem nenhum argumento, o que faz com que o controle retorne imediatamente para o ambiente de chamada. Os valores atuais dos parametros formais declarados como OUT ou IN OUT são passados de volta aos parametros reais e a execucao prossegue a partir da instrução seguinte à chamada da procedure (mais adiante será melhor detalhado).

### **Eliminando procedures e funções**

A sintaxe para descartar uma procedure é:

DROP PROCEDURE *nome\_de\_procedure*;

e a sintaxe para descartar uma função é:

DROP FUNCTION *nome\_de\_função*;

Da mesma maneira como CREATE, DROP é um comando de DDL, assim um COMMIT隐式的 é feito tanto antes como depois da instrução. Se o subprograma não existir, a instrução DROP levantará o erro "ORA-4043: Object does not exist".

## Parâmetros de subprograma

As procedures e funções podem receber parâmetros por diferentes modos e podem ser passados por valor ou por referência.

### Modo de parâmetro

Dada a procedure:

```
DECLARE
    -- Variables describing the new student
    v_NewFirstName    students.first_name%TYPE := 'Cynthia';
    v_NewLastName     students.last_name%TYPE := 'Camino';
    v_NewMajor        students.major%TYPE := 'History';
BEGIN
    -- Add Cynthia Camino to the database.
    AddNewStudent(v_NewFirstName, v_NewLastName, v_NewMajor);
END;
```

As variáveis declaradas (v\_NewFirstName, v\_NewLastName, v\_NewMajor) são passadas como argumentos em AddNewStudent, sendo conhecidas como *parametros reais*, ao passo que os parametros na declaração da procedure (p\_FirstName, p\_LastName, p\_Major) são conhecidos como *parametros formais*.

Os parametros formais pode ter três modos - IN, OUT, ou IN OUT. Se o modo não for especificado, o parametro IN é adotado como padrão.

```
CREATE OR REPLACE PROCEDURE ModeTest (
    p_InParameter      IN NUMBER,
    p_OutParameter     OUT NUMBER,
    p_InOutParameter   IN OUT NUMBER) IS

    v_LocalVariable    NUMBER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside ModeTest:');
    IF (p_InParameter IS NULL) THEN
        DBMS_OUTPUT.PUT('p_InParameter is NULL');
    ELSE
        DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
    END IF;

    IF (p_OutParameter IS NULL) THEN
        DBMS_OUTPUT.PUT('p_OutParameter is NULL');
    ELSE
        DBMS_OUTPUT.PUT('p_OutParameter = ' || p_OutParameter);
    END IF;

    IF (p_InOutParameter IS NULL) THEN
        DBMS_OUTPUT.PUT('p_InOutParameter is NULL');
    ELSE
        DBMS_OUTPUT.PUT('p_InOutParameter = ' || p_InOutParameter);
    END IF;
```

```

        DBMS_OUTPUT.PUT_LINE(' p_InOutParameter is NULL');
ELSE
    DBMS_OUTPUT.PUT_LINE(' p_InOutParameter = ' || p_InOutParameter);
END IF;

/* Assign p_InParameter to v_LocalVariable. This is legal,
   since we are reading from an IN parameter and not
writing
   to it. */
v_LocalVariable := p_InParameter; -- Legal

/* Assign 7 to p_InParameter. This is ILLEGAL, since we
   are writing to an IN parameter. */
-- p_InParameter := 7; -- Illegal

/* Assign 7 to p_OutParameter. This is legal, since we
   are writing to an OUT parameter. */
p_OutParameter := 7; -- Legal

/* Assign p_OutParameter to v_LocalVariable. In Oracle7
version
   7.3.4, and Oracle8 version 8.0.4 or higher (including
8i),
   this is legal. Prior to 7.3.4, it is illegal to read
from an
   OUT parameter. */
v_LocalVariable := p_OutParameter; -- Possibly illegal

/* Assign p_InOutParameter to v_LocalVariable. This is
legal,
   since we are reading from an IN OUT parameter. */
v_LocalVariable := p_InOutParameter; -- Legal

/* Assign 8 to p_InOutParameter. This is legal, since we
   are writing to an IN OUT parameter. */
p_InOutParameter := 8; -- Legal

DBMS_OUTPUT.PUT_LINE('At end of ModeTest:');
IF (p_InParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('p_InParameter is NULL');
ELSE
    DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
END IF;

IF (p_OutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT(' p_OutParameter is NULL');

```

```

    ELSE
        DBMS_OUTPUT.PUT(' p_OutParameter = ' || p_OutParameter);
    END IF;

    IF (p_InOutParameter IS NULL) THEN
        DBMS_OUTPUT.PUT_LINE(' p_InOutParameter is NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' p_InOutParameter = ' ||
                                p_InOutParameter);
    END IF;

END ModeTest;

```

### **Passando valores entre parâmetros formais e reais**

Podemos chamar ModeTest com o seguinte bloco:

```

DECLARE
    v_In NUMBER := 1;
    v_Out NUMBER := 2;
    v_InOut NUMBER := 3;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before calling ModeTest:');
    DBMS_OUTPUT.PUT_LINE('v_In = ' || v_In ||
                           ' v_Out = ' || v_Out ||
                           ' v_InOut = ' || v_InOut);
    ModeTest(v_In, v_Out, v_InOut);
    DBMS_OUTPUT.PUT_LINE('After calling ModeTest:');
    DBMS_OUTPUT.PUT_LINE(' v_In = ' || v_In ||
                           ' v_Out = ' || v_Out ||
                           ' v_InOut = ' || v_InOut);
END;

```

Modo	Descrição
IN	O valor do parâmetro real é passado para a procedure quando a procedure é invocada. Dentro da procedure, o parâmetro formal atua como uma constante PL/SQL - ele é considerado de <i>leitura</i> e não pode ser alterado. Quando a procedure conclui e o controle retorna ao ambiente de chamada, o parâmetro real não é alterado.
OUT	Qualquer valor que o parametro real tenha é ignorado quando a procedure é chamada. Dentro da procedure, o parametro formal atua como uma variavel nao inicializada PL/SQL e, portanto tem um valor NULL. Ele pode ser lido e escrito. Quando a procedure conclui e o controle retorna ao ambiente de chamada, o conteúdo do parametro formal é atribuido ao paramentro real.
IN OUT	Combina IN e OUT. O valor do parametro real é passado para a procedure

	quando a procedure é invocada. Dentro da procedure, o parametro formal atua como uma variável inicializada e poder ser lido e gravado. Quando a procedure conclui e o controle retorna ao ambiente de chamada, o conteúdo do parametro formal é atribuído ao parametro real.
--	--

Isso introduz a saída mostrada a seguir:

```
Befor calling ModeTest:  
v_In=1 v_Out=2 v_InOut=3 Inside ModeTest:  
p_InParameter=1 p_OutParameter is NULL p_InOutParameter=3  
At end of ModeTest:  
p_InParameter=1 p_OutParameter=7 p_InOutParameter=8  
After calling ModeTest:  
v_In=1 v_Out=7 v_InOut=8
```

Essa saída mostra que o parametro OUT foi inicializado como NULL dentro da procedure. Além disso, os valores dos parametros formais IN e IN OUT no final da procedure foram copiados de volta para os parametros reais quando a procedure conclui.

### Literais ou constantes como parametros reais

Por causa dessa copia, o parametro real que corresponde a um parametro IN OUT ou OUT deve ser uma variável e não pode ser uma expressão ou um constante. Deve haver um lugar onde o valor retornado possa ser armazenado. Por exemplo, podemos substituir v\_In por um literal quando chamamos ModeTest:

```
DECLARE  
    v_Out NUMBER := 2;  
    v_InOut NUMBER := 3;  
BEGIN  
    ModeTest(1, v_Out, v_InOut);  
END;
```

Mas se substituirmos v\_Out por um literal, teríamos o seguinte exemplo ilegal:

```
DECLARE  
    v_InOut NUMBER := 3;  
BEGIN  
    ModeTest(1, 2, v_InOut);  
END;  
*Error at line 1:  
ORA-06550: line 4, column 15:  
PLS-00363: expression '2' cannot be used as na assignment  
target  
ORA-06550: line 4, column 3:  
PL/SQL: Statement ignored
```

## Lendo de parâmetros OUT

Em uma procedure, anterior as versões 7.3.4 e especificamente na versão 8.0.3, é ilegal ler de um parâmetro OUT. Uma maneira de contornar esse problema é declara os parâmetros OUT como IN OUT.

## Restrições quanto aos parâmetros formais

Quando uma procedure é chamada, são passados os valores dos parâmetros reais para os parâmetros formais dentro da procedure, também são passadas as restrições quanto às variáveis como parte do mecanismo de passagem de parâmetro. Na declaração de uma procedure, é ilegal restringir os parametros de CHAR e de VARCHAR2 com um comprimento e os parâmetros de NUMBER com uma precisão e/ou escala. O exemplo a seguir irá gerar um erro de compilação.

```
CREATE OR REPLACE PROCEDURE ParameterLength (
    p_Parameter1 IN OUT VARCHAR2(10),
    p_Parameter2 IN OUT NUMBER(3,1)) AS
BEGIN
    p_Parameter1 := 'abcdefghijklm';
    p_Parameter2 := 12.3;
END ParameterLength;
```

A declaração correta para esse procedimento seria:

```
CREATE OR REPLACE PROCEDURE ParameterLength (
    p_Parameter1 IN OUT VARCHAR2,
    p_Parameter2 IN OUT NUMBER) AS
BEGIN
    p_Parameter1 := 'abcdefghijklmno';
    p_Parameter2 := 12.3;
END ParameterLength;
```

Como já mencionado, as restrições vêm dos parâmetros reais. Por exemplo, se chamarmos ParameterLength com

```
DECLARE
    v_Variable1 VARCHAR2(40);
    v_Variable2 NUMBER(7,3);
BEGIN
    ParameterLength(v_Variable1, v_Variable2);
END;
```

então p\_Parameter1 terá um comprimento máximo de 40 (vindo do parâmetro real v\_Variable1), da mesma forma p\_Parameter2 terá uma precisão 7 e escala 3 (vinda do parâmetro real v\_Variable2).

## Parâmetros de %TYPE e de procedure

Embora os parâmetros formais não possam ser declarados com restrições, eles podem ser restringidos utilizando %TYPE, dessa forma restrição estará no parâmetro formal em vez de no parâmetro real.

```
CREATE OR REPLACE PROCEDURE ParameterLength (
    p_Parameter1 IN OUT VARCHAR2,
    p_Parameter2 IN OUT students.current_credits%TYPE) AS
BEGIN
    p_Parameter2 := 12345;
END ParameterLength;
```

então, p\_Parameter2 será restrito com precisão de 3, porque essa é a precisão da coluna current\_credits.

## Exceções levantadas dentro de subprogramas

Uma exceção é levantada se um erro ocorrer dentro de um subprograma. Essa exceção talvez seja definida pelo usuário ou predefinida. Se a procedure não tiver um handler de exceção, o controle passa imediatamente da procedure para o ambiente de chamada, de acordo com as regras de propagação de exceção. Nesse caso, os valores formais OUT e IN OUT *não* retornam para os parâmetros reais. Os parâmetros reais terão os mesmos valores que teriam se a procedure não tivesse sido chamada.

## Passando parâmetro por referência e por valor

Um parâmetro de subprograma pode ser passado de uma entre duas maneiras - por referência ou por valor. Quando uma parâmetro é passado *por referência*, um ponteiro para o parâmetro real é passado para o parâmetro formal correspondente. Por outro lado, quando um parâmetro é passado *por valor* ele é copiado do parâmetro real para o parâmetro formal. Passar por referência é geralmente mais rápido, pois evita cópia. Por padrão, a PL/SQL passará parâmetros IN por referência e parâmetros OUT e IN OUT por valor. Isso é feito para preservar a semântica de exceção e de tal modo que as restrições sobre os parâmetros reais possam ser verificados. Antes do Oracle8i, não havia nenhuma maneira de modificar esse comportamento.

## Utilizando o NOCOPY

O Oracle8i inclui um compilador de dica conhecido como NOCOPY. A sintaxe para declarar um parâmetro com essa dica é:

*nome\_de\_parâmetro* [*modo*] NOCOPY *tipo\_de\_dados*

onde *nome\_de\_parâmetro* é o nome do parâmetro, *modo* é o modo do parâmetro (IN, OUT ou IN OUT) e *tipo\_de\_dados* é o tipo de dados do parâmetro. Se NOCOPY estiver

presente, o compilador PL/SQL tentará passar o parâmetro por referência, em vez de por valor. Por exemplo:

```
CREATE OR REPLACE PROCEDURE NoCopyTest (
    p_InParameter      IN NUMBER,
    p_OutParameter     OUT NOCOPY VARCHAR2,
    p_InOutParameter   IN OUT NOCOPY CHAR) IS
BEGIN
    NULL;
END NoCopyTest;
```

NOCOPY não é aplicado para o parâmetro IN pois os parâmetros IN sempre serão passados por referência.

### Semântica de exceção com NOCOPY

Quando um parâmetro é passado por referência, quaisquer modificações no parâmetro formal também modificam o parâmetro real, pois ambos apontam para um mesmo lugar. Isso significa que se um procedimento for concluído com uma exceção não-tratada depois que o parâmetro formal foi alterado, o valor original do parâmetro real será perdido.

### Restrições de NOCOPY

Como NOCOPY é uma dica que o compilador não é obrigado a seguir, em alguns casos, ele será ignorado e o parametro sera passado como valor. NOCOPY sera ignorado nas seguintes situações:

- O parametro real é membro de uma tabela index-by. Exceto se o parametro real for uma tabela inteira.
- O parametro real esta escrito por uma precisao, escala ou NOT NULL.
- Os parametros formais e reais sao registros; e eles foram declarados implicitamente com uma variavel de controle de loop ou utilizando %ROWTYPE, e as restrições diferem nos campos correspondentes.
- Passar o parâmetro real requer uma conversão implícita para os tipos de dados.
- O subprograma é envolvido em uma chamada de procedimento remoto (RPC). Uma RPC é uma chamada de procedure realizada de um link do banco de dados para um servidor remoto. Pelo fato de os parametros deverem ser transferidos na rede, nao é possivel passa-los por referencia.

### Os benefícios de NOCOPY

A principal vantagem é o aumento do desempenho. Especialmente ao passar grandes tabelas PL/SQL, como por exemplo:

```
CREATE OR REPLACE PACKAGE CopyFast AS
-- PL/SQL table of students.
```

```

TYPE StudentArray IS
  TABLE OF students%ROWTYPE;

  -- Three procedures which take a parameter of StudentArray,
in
  -- different ways. They each do nothing.
  PROCEDURE PassStudents1(p_Parameter IN StudentArray);
  PROCEDURE PassStudents2(p_Parameter IN OUT StudentArray);
  PROCEDURE PassStudents3(p_Parameter IN OUT NOCOPY
StudentArray);

  -- Test procedure.
  PROCEDURE Go;
END CopyFast;

CREATE OR REPLACE PACKAGE BODY CopyFast AS
  PROCEDURE PassStudents1(p_Parameter IN StudentArray) IS
BEGIN
  NULL;
END PassStudents1;

  PROCEDURE PassStudents2(p_Parameter IN OUT StudentArray) IS
BEGIN
  NULL;
END PassStudents2;

  PROCEDURE PassStudents3(p_Parameter IN OUT NOCOPY
StudentArray) IS
BEGIN
  NULL;
END PassStudents3;

  PROCEDURE Go IS
    v_StudentArray StudentArray := StudentArray(NULL);
    v_StudentRec students%ROWTYPE;
    v_Time1 NUMBER;
    v_Time2 NUMBER;
    v_Time3 NUMBER;
    v_Time4 NUMBER;
BEGIN
  -- Fill up the array with 50,001 copies of David
Dinsmore's
  -- record.
  SELECT *
    INTO v_StudentArray(1)
    FROM students
    WHERE ID = 10007;

```

```

v_StudentArray.EXTEND(50000, 1);

-- Call each version of PassStudents, and time them.
-- DBMS_UTLILITY.GET_TIME will return the current time, in
-- hundredths of a second.
v_Time1 := DBMS_UTLILITY.GET_TIME;
PassStudents1(v_StudentArray);
v_Time2 := DBMS_UTLILITY.GET_TIME;
PassStudents2(v_StudentArray);
v_Time3 := DBMS_UTLILITY.GET_TIME;
PassStudents3(v_StudentArray);
v_Time4 := DBMS_UTLILITY.GET_TIME;

-- Output the results.
DBMS_OUTPUT.PUT_LINE('Time to pass IN: ' ||
                      TO_CHAR((v_Time2 - v_Time1) / 100));
DBMS_OUTPUT.PUT_LINE('Time to pass IN OUT: ' ||
                      TO_CHAR((v_Time3 - v_Time2) /
100));
DBMS_OUTPUT.PUT_LINE('Time to pass IN OUT NOCOPY: ' ||
                      TO_CHAR((v_Time4 - v_Time3) / 100));
END Go;
END CopyFast;

```

Todas as procedures PassStudents não fazem nada - as procedures simplesmente recebem um parametro que é uma tabela PL/SQL dos alunos. O parametro tem 50.001 registros, razoavelmente grande. A diferenca entre as procedures é que PassStudents1 recebe o parametro como um IN, PassStudents2 como um IN OUT, e PassStudents3 como IN OUT NOCOPY. Portanto, PassStudents2 deve passar o parametro por valor e os outros dois por referencia. Conforme os resultados da chamada CopyFast.Go:

```

SQL> BEGIN
2      CopyFast.Go
3      END;
4      /
Time to pass IN = 0
Time to pass IN OUT = 4.28
Time to pass IN OUT NOCOPY = 0
PL/SQL procedure successfully completed.

```

Embora no seu sistema os resultados possam ser diferentes, o tempo para passar o parametro IN OUT por valor devera ser significativamente maior do que passar os parametros IN e IN OUT NOCOPY por referencia.

## Subprogramas sem parâmetros

Se uma procedure não houver nenhum parametrom, não há nenhum parentese na declaracao da procedure, nem na chamada de procedure. Isso também é verdadeiro para as funções.

## Notação posicional e identificada

A notação posicional é aquela onde os parametros reais são associados com os parâmetros formais por posição. Em todos os exemplos utilizou-se dessa notação, onde v\_Variable1 é associado com p\_ParameterA, v\_Variable2 com p\_ParameterB e assim por diante. Este tipo de notação é comumente utilizada e é também a notação utilizada em outras 3GLs como C.

```
CREATE OR REPLACE PROCEDURE CallMe(
    p_ParameterA VARCHAR2,
    p_ParameterB NUMBER,
    p_ParameterC BOOLEAN,
    p_ParameterD DATE) AS
BEGIN
    NULL;
END CallMe;
/

DECLARE
    v_Variable1 VARCHAR2(10);
    v_Variable2 NUMBER(7,6);
    v_Variable3 BOOLEAN;
    v_Variable4 DATE;
BEGIN
    CallMe(v_Variable1, v_Variable2, v_Variable3, v_Variable4);
END;
```

Alternativamente, podemos chamar a procedure utilizando a notação identificada. Neste caso o parâmetro formal e o parâmetro real sao incluídos em cada argumento. Pode-se também reorganizar a ordem dos argumentos.

```
DECLARE
    v_Variable1 VARCHAR2(10);
    v_Variable2 NUMBER(7,6);
    v_Variable3 BOOLEAN;
    v_Variable4 DATE;
BEGIN
    CallMe(p_ParameterA => v_Variable1,
```

```

    p_ParameterC => v_Variable3,
    p_ParameterD => v_Variable4,
    p_ParameterB => v_Variable2);
END;

```

Se desejado, ambas notações podem ser combinadas em uma mesma chamada. Qualquer combinação de notações segue a mesma eficácia, a preferência é de estilo.

## Valores padrão do parâmetro

De maneira semelhante às declarações de variável, os parâmetros formais para um procedimento ou função podem ter valores padrão. Se um parâmetro tiver um valor padrão, ele não tem de ser passado a partir do ambiente de chamada. Se o for, o valor do parâmetro real será utilizado em vez do padrão. Um valor padrão para um parâmetro é incluído utilizando a sintaxe:

```

nome_do_parâmetro [modo] tipo_de_parâmetro
{ := | DEFAULT } valor_inicial

```

onde *nome\_do\_parâmetro* é o nome do parâmetro formal, *modo* é o modo de parâmetro (IN, OUT ou IN OUT), *tipo\_de\_parâmetro* é o tipo de parâmetro (tanto predefinido pelo usuário) e *valor\_inicial* é o valor a ser atribuído ao parâmetro formal por padrão. Tanto := como a palavra-chave DEFAULT podem ser utilizadas. Por exemplo:

```

CREATE OR REPLACE PROCEDURE AddNewStudent (
    p_FirstName    students.first_name%TYPE,
    p_LastName     students.last_name%TYPE,
    p_Major        students.major%TYPE DEFAULT 'Economics' ) AS
BEGIN
    -- Insert a new row in the students table. Use
    -- student_sequence to generate the new student ID, and
    -- 0 for current_credits.
    INSERT INTO students VALUES (student_sequence.nextval,
        p_FirstName, p_LastName, p_Major, 0);
END AddNewStudent;

```

O valor padrão será utilizado se o parâmetro formal p\_Major não tiver um parâmetro real associado a ele na chamada de procedure. Podemos fazer isso por meio da notação posicional:

```

BEGIN
    AddNewStudent('Simon', 'Salovitz');
END;

```

ou com notação identificada:

```

BEGIN

```

```

    AddNewStudent(p_FirstName => 'Veronica',
                  p_LastName  => 'Vassily');
END;

```

Ao utilizar valores padrão, se possível faça com que eles sejam os últimos parâmetros na lista de argumentos. Dessa maneira, tanto a notação posicional como a identificada podem ser utilizados.

## A instrução CALL

A partir do Oracle8i a instrução CALL surge como nova instrução SQL para chamar subprogramas armazenados (o comando EXECUTE em versões anteriores pode também ser utilizado). Pode ser utilizada para chamar subprogramas tanto PL/SQL como de Java por meio de um empacotador PL/SQL. Sintaxe:

```

CALL nome_do_subprograma ([lista_de_argumentos])
[INTO variavel_do_host];

```

onde *nome\_do\_subprograma* é um subprograma de terceiros ou empacotado, *lista\_de\_argumentos* é uma lista de argumentos delimitada por vírgulas; e *variavel\_do\_host* é uma variável de host utilizada para recuperar o valor de retorno das funções. A seguir algumas utilizações válidas e ilegais da instrução CALL:

```

CREATE OR REPLACE PROCEDURE CallProc1(p1 IN VARCHAR2 := NULL)
AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('CallProc1 called with ' || p1);
END CallProc1;

CREATE OR REPLACE PROCEDURE CallProc2(p1 IN OUT VARCHAR2) AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('CallProc2 called with ' || p1);
    p1 := p1 || ' returned!';
END CallProc2;

CREATE OR REPLACE FUNCTION CallFunc(p1 IN VARCHAR2)
    RETURN VARCHAR2 AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('CallFunc called with ' || p1);
    RETURN p1;
END CallFunc;

-- Algumas chamadas validas via SQL*Plus.
SQL> CALL CallProc1('Hello!');
SQL> CALL CallProc1();
SQL> VARIABLE v_Output VARCHAR2(50);
SQL> CALL CallFunc('Hello!') INTO :v_Output;

```

```

SQL> PRINT v_Output
Hello!

SQL> CALL CallProc2(:v_Output);
SQL> PRINT v_Output
Hello! returned!

--Chamada ilegal
BEGIN
    CALL CallProc1();
END;
Erro ORA-06550

-- Mas esses são válidos
DECLARE
    v_Result VARCHAR2(50);
BEGIN
    EXECUTE IMMEDIATE 'CALL CallProc1('''Hello from PL/SQL''')';
    EXECUTE IMMEDIATE
        'CALL CallFunc('''Hello from PL/SQL''') INTO :v_Result'
        USING OUT v_Result;
END;

```

O exemplo ilustra os seguintes pontos:

- CALL é uma instrução SQL. Ela não é válida dentro de um bloco PL/SQL, mas é válida quando executada utilizando a SQL dinâmica.
- Os parênteses são sempre requeridos, mesmo se o subprograma não receber nenhum argumento (ou tiver valores padrão para todos os argumentos).
- A cláusula INTO é utilizada apenas para as variáveis de saída de funções. Os parâmetros IN OUT ou OUT são especificados como parte de *lista\_de\_argumentos*.

### **Procedures versus funções**

As procedures e funções compartilham vários dos mesmos recursos:

- Ambas podem retornar mais de um valor via parâmetros OUT
- Ambas podem ter seções executáveis, declarativas e de tratamento de exceções.
- Ambas podem aceitar valores padrão.
- Ambas podem ser chamadas utilizando a notação posicional ou identificada.
- Ambas podem aceitar parâmetros de NOCOPY (Oracle8i e superior).

Geralmente se houver mais de um valor de retorno utiliza-se uma procedure e se houver apenas um valor de retorno, uma função pode ser utilizada. Embora seja válido que uma função tenha os parâmetros OUT (e assim retorne mais de um valor), geralmente isso é considerado como estilo de programação pobre.

## Pacotes

Um pacote é uma construção PL/SQL que permite que objetos relacionados sejam armazenados juntos. Um pacote tem duas partes separadas: a especificação e o corpo. Cada uma das quais é armazenada no dicionário de dados separadamente. Os pacotes permitem que objetos relacionados sejam agrupados, juntos, os pacotes também são úteis no que diz respeito às dependências e, por fim, vantagens de desempenho.

Um pacote é essencialmente uma seção declarativa identificada. Qualquer coisa que possa entrar na parte declarativa de um bloco pode entrar em um pacote. Isso inclui procedures, funções, cursores, tipos e variáveis. Uma vantagem em colocar esses objetos em um pacote é a capacidade de referenciá-los a partir de outros blocos PL/SQL, portanto na PL/SQL os pacotes também fornecem variáveis globais (dentro de uma única sessão do banco de dados) para PL/SQL.

### Especificação de pacote

Também conhecida como *cabeçalho de pacote*, contém as informações sobre o conteúdo do pacote. Entretanto, não contém o código de nenhum subprograma. Veja no exemplo:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
    -- Add a new student into the specified class.
    PROCEDURE AddStudent(p_StudentID    IN students.id%TYPE,
                         p_Department   IN
classes.department%TYPE,
                         p_Course       IN classes.course%TYPE);

    -- Removes the specified student from the specified class.
    PROCEDURE RemoveStudent(p_StudentID   IN students.id%TYPE,
                           p_Department  IN
classes.department%TYPE,
                           p_Course      IN
classes.course%TYPE);

    -- Exception raised by RemoveStudent.
    e_StudentNotRegistered EXCEPTION;

    -- Table type used to hold student info.
    TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
        INDEX BY BINARY_INTEGER;

    -- Returns a PL/SQL table containing the students currently
    -- in the specified class.
    PROCEDURE ClassList(p_Department  IN
classes.department%TYPE,
                         p_Course      IN  classes.course%TYPE,
                         p_IDs         OUT t_StudentIDTable,
```

```

    p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;

```

ClassPackage contém três procedures, um tipo e uma exceção. A sintaxe para criar um cabeçalho de pacote é:

```

CREATE [OR REPLACE] PACKAGE nome_de_pacote {IS | AS}
definição_de_tipo /
especificação_de_procedimento /
especificação_de_função /
declaração_de_variável /
declaração_de_exceção /
declaração_de_cursor /
declaração_de_prgama /
END [nome_do_pacote]

```

Os *elementos* dentro do pacote (especificações de procedures e de função, variáveis e assim por diante) são os mesmos que estariam na seção declarativa de um bloco anônimo. As mesmas regras de sintaxe aplicadas para um cabeçalho são aplicadas a uma seção declarativa, exceto quanto às declarações de procedures e de função. Essas regras são as seguintes:

- Os elementos do pacote podem aparecer em qualquer ordem. Entretanto, como em uma seção declarativa, um objeto deve ser declarado antes que seja referenciado.
- Todos os tipos de elementos não têm de estar presentes. Por exemplo, um pacote pode conter apenas especificações de procedure e de função sem declarar nenhuma exceção ou tipo.

## Corpo de pacote

O *corpo de pacote* é um objeto do dicionário de dados separado do cabeçalho do pacote. Ele não pode ser compilado com sucesso a menos que o cabeçalho do pacote já tenha sido compilado com sucesso. O corpo contém o código para as declarações introdutórias do subprograma no cabeçalho de pacote. Pode conter declarações adicionais que são globais para o corpo de pacote, mas não são visíveis na especificação. Exemplo de corpo de pacote:

```

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Add a new student for the specified class.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN
  classes.department%TYPE,
                       p_Course      IN classes.course%TYPE)
  IS
  BEGIN
    INSERT INTO registered_students (student_id, department,
    course)
    VALUES (p_StudentID, p_Department, p_Course);

```

```

END AddStudent;

-- Removes the specified student from the specified class.
PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                        p_Department IN
classes.department%TYPE,
                        p_Course      IN
classes.course%TYPE) IS
BEGIN
    DELETE FROM registered_students
    WHERE student_id = p_StudentID
    AND department = p_Department
    AND course = p_Course;

    -- Check to see if the DELETE operation was successful.
If
    -- it didn't match any rows, raise an error.
    IF SQL%NOTFOUND THEN
        RAISE e_StudentNotRegistered;
    END IF;
END RemoveStudent;

-- Returns a PL/SQL table containing the students currently
-- in the specified class.
PROCEDURE ClassList(p_Department  IN
classes.department%TYPE,
                     p_Course      IN classes.course%TYPE,
                     p_IDs         OUT t_StudentIDTable,
                     p_NumStudents IN OUT BINARY_INTEGER) IS

v_StudentID  registered_students.student_id%TYPE;

-- Local cursor to fetch the registered students.
CURSOR c_RegisteredStudents IS
    SELECT student_id
        FROM registered_students
        WHERE department = p_Department
        AND course = p_Course;
BEGIN
    /* p_NumStudents will be the table index. It will start
at
        * 0, and be incremented each time through the fetch
loop.
        * At the end of the loop, it will have the number of
rows
        * fetched, and therefore the number of rows returned in
        * p_IDs.

```

```

*/  

p_NumStudents := 0;  
  

OPEN c_RegisteredStudents;  

LOOP  

  FETCH c_RegisteredStudents INTO v_StudentID;  

  EXIT WHEN c_RegisteredStudents%NOTFOUND;  
  

  p_NumStudents := p_NumStudents + 1;  

  p_IDs(p_NumStudents) := v_StudentID;  

END LOOP;  

END ClassList;  

END ClassPackage;

```

O corpo de pacote é opcional. Se o cabeçalho de pacote não contiver nenhuma procedure ou função (apenas declarações de variável, cursores, tipos e outros), o corpo não tem de estar presente. Essa técnica é valiosa para a declaração de variáveis e tipos globais, porque todos os objetos em um pacote são visíveis fora do pacote (escopo e visibilidade serão vistos adiante).

## Pacotes e escopo

Qualquer objeto declarado em um cabeçalho de pacote estará em escopo e visível fora do pacote, qualificando o objeto como o nome de pacote. Exemplo:

```

BEGIN
  ClassPackage.RemoveStudent(10006, 'HIS', 101);
END;

```

A chamada de procedure é a mesma que seria para uma procedure independente, a diferença é que ela é prefixada pelo nome do pacote. Porém, dentro do corpo de pacote, os objetos no cabeçalho podem ser referenciados sem o nome do pacote.

As procedures empacotadas podem ter parâmetros padrão e podem ser chamadas utilizando tanto notação posicional como identificada, da mesma forma que as procedures armazenadas independentes.

## Escopo de objetos no corpo do pacote

No exemplo ClassPackage, se quisermos atualizar outras tabelas além da tabela *registered\_students*, pode-se fazer isso adicionando uma procedure ao corpo do pacote, como no exemplo a seguir:

```

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
  -- Utility procedure that updates students and classes to
  -- reflect
  -- the change. If p_Add is TRUE, then the tables are
  -- updated for

```

```

-- the addition of the student to the class. If it is
FALSE,
-- then they are updated for the removal of the student.
PROCEDURE UpdateStudentsAndClasses(
    p_Add      IN BOOLEAN,
    p_StudentID IN students.id%TYPE,
    p_Department IN classes.department%TYPE,
    p_Course    IN classes.course%TYPE) IS

    -- Number of credits for the requested class
    v_NumCredits  classes.num_credits%TYPE;
BEGIN
    -- First determine NumCredits.
    SELECT num_credits
        INTO v_NumCredits
        FROM classes
        WHERE department = p_Department
        AND course = p_Course;

    IF (p_Add) THEN
        -- Add NumCredits to the student's course load
        UPDATE STUDENTS
            SET current_credits = current_credits + v_NumCredits
            WHERE ID = p_StudentID;

        -- And increase current_students
        UPDATE classes
            SET current_students = current_students + 1
            WHERE department = p_Department
            AND course = p_Course;
    ELSE
        -- Remove NumCredits from the students course load
        UPDATE STUDENTS
            SET current_credits = current_credits - v_NumCredits
            WHERE ID = p_StudentID;

        -- And decrease current_students
        UPDATE classes
            SET current_students = current_students - 1
            WHERE department = p_Department
            AND course = p_Course;
    END IF;
END UpdateStudentsAndClasses;

-- Add a new student for the specified class.
PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,

```

```

          p_Department IN
classes.department%TYPE,
                  p_Course      IN classes.course%TYPE)
IS
BEGIN
    INSERT INTO registered_students (student_id, department,
course)
    VALUES (p_StudentID, p_Department, p_Course);

UpdateStudentsAndClasses(TRUE, p_StudentID, p_Department,
p_Course);
END AddStudent;

-- Removes the specified student from the specified class.
PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                        p_Department IN
classes.department%TYPE,
                        p_Course      IN
classes.course%TYPE) IS
BEGIN
    DELETE FROM registered_students
    WHERE student_id = p_StudentID
    AND department = p_Department
    AND course = p_Course;

-- Check to see if the DELETE operation was successful.
If
    -- it didn't match any rows, raise an error.
    IF SQL%NOTFOUND THEN
        RAISE e_StudentNotRegistered;
    END IF;

UpdateStudentsAndClasses(FALSE, p_StudentID,p_Department,
p_Course);

END RemoveStudent;
...
END ClassPackage;

```

No corpo de pacote UpdateStudentAndClasses é declarado local. O seu escopo é, portanto o próprio corpo de pacote. Consequentemente, ele pode ser chamado a partir de outras procedures no corpo (a saber, AddStudent e RemoveStudent), mas não estará visível fora do corpo.

## Sobrecarregando subprogramas empacotados

Dentro de um pacote, as procedures e funções podem ser *sobre carregadas*, ou seja, pode haver mais de uma procedure ou função com o mesmo nome, porém com diferentes parâmetros. Isto permite que uma mesma operação seja aplicada para objetos de diferentes tipos. Por exemplo, ao querer adicionar um aluno para uma classe tanto especificando o seu ID ou seu nome e sobrenome; pode-se fazer isso modificando ClassPackage, como a seguir:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
    -- Add a new student into the specified class.
    PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                         p_Department IN
classes.department%TYPE,
                         p_Course      IN classes.course%TYPE);

    -- Also adds a new student, by specifying the first and
last
    -- names, rather than ID number.
    PROCEDURE AddStudent(p_FirstName IN
students.first_name%TYPE,
                         p_LastName   IN
students.last_name%TYPE,
                         p_Department IN
classes.department%TYPE,
                         p_Course      IN classes.course%TYPE);

    -- Removes the specified student from the specified class.
    PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                           p_Department IN
classes.department%TYPE,
                           p_Course      IN
classes.course%TYPE);

    -- Exception raised by RemoveStudent.
    e_StudentNotRegistered EXCEPTION;

    -- Table type used to hold student info.
    TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
        INDEX BY BINARY_INTEGER;

    -- Returns a PL/SQL table containing the students currently
    -- in the specified class.
    PROCEDURE ClassList(p_Department  IN
classes.department%TYPE,
                         p_Course      IN classes.course%TYPE,
```

```

        p_IDs          OUT t_StudentIDTable,
        p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
    -- Add a new student for the specified class.
    PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN
classes.department%TYPE,
                          p_Course      IN classes.course%TYPE)
IS
    BEGIN
        INSERT INTO registered_students (student_id, department,
course)
        VALUES (p_StudentID, p_Department, p_Course);
    END AddStudent;

    -- Add a new student by name, rather than ID.
    PROCEDURE AddStudent(p_FirstName IN
students.first_name%TYPE,
                          p_LastName   IN
students.last_name%TYPE,
                          p_Department IN
classes.department%TYPE,
                          p_Course      IN classes.course%TYPE)
IS
    v_StudentID students.ID%TYPE;
    BEGIN
        /* First we need to get the ID from the students table.
 */
        SELECT ID
        INTO v_StudentID
        FROM students
        WHERE first_name = p_FirstName
        AND last_name = p_LastName;

        -- Now we can add the student by ID.
        INSERT INTO registered_students (student_id, department,
course)
        VALUES (v_StudentID, p_Department, p_Course);
    END AddStudent;
...
END ClassPackage;

```

Agora pode-se adicionar um aluno para Música 410 tanto com um:

```

BEGIN
    ClassPackage.AddStudent(10000, 'MUS', 410);
END;

```

como com:

```

BEGIN
    ClassPackage.AddStudent('Rita', 'Ramirez', 'MUS', 410);
END;

```

Sobrecarregar subprogramas é bastante útil, porém está sujeito a várias restrições:

- Não se pode sobrecarregar dois subprogramas se os seus parâmetros diferirem apenas em nome ou modo (IN, OUT ou IN OUT).
- Não se pode sobrecarregar duas funções apenas em seu tipo de retorno diferentes (por exemplo, RETURN DATE em uma e RETURN NUMBER na outra).
- Os parametros de funções sobrecarregadas devem diferir pela família do tipo - não se pode sobrecarregar na mesma família (CHAR como VARCHAR2).

Obs. O compilador PL/SQL permite criar um pacote com subprogramas que violam restrições, porém na execução ele não será capaz de resolver as referências e irá gerar um erro (PLS-307: *too many declarations of 'subprogram' match this call*).

### **Inicialização do pacote**

Na primeira vez em que um subprograma empacotado é chamado, ou qualquer referência para uma variável ou tipo empacotado ocorre, o pacote é *instanciado*. Isso significa que o pacote é lido do disco para a memória e o código compilado é executado a partir do subprograma chamado. Nesse ponto, é alocada memória para todas as variáveis definidas no pacote.

Em várias situações, o código de inicialização precisa ser executado na primeira vez que o pacote é instanciado dentro de uma sessão. Isso pode ser feito adicionando uma seção de inicialização ao corpo do pacote, depois de todos os outros objetos, com a sintaxe:

```

CREATE OR REPLACE PACKAGE BODY nome_de_pacote { IS | AS}
...
BEGIN
    código_de_inicialização
END [nome_de_pacote];

```

onde *nome\_de\_pacote* é o nome do pacote e *código\_de\_inicialização* é o código a ser executado. O exemplo seguinte implementa uma função de número aleatório.

```

CREATE OR REPLACE PACKAGE Random AS
    -- Random number generator.  Uses the same algorithm as the
    -- rand() function in C.

    -- Used to change the seed.  From a given seed, the same

```

```

-- sequence of random numbers will be generated.
PROCEDURE ChangeSeed(p_NewSeed IN NUMBER);

-- Returns a random integer between 1 and 32767.
FUNCTION Rand RETURN NUMBER;

-- Same as Rand, but with a procedural interface.
PROCEDURE GetRand(p_RandomNumber OUT NUMBER);

-- Returns a random integer between 1 and p_MaxVal.
FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER;

-- Same as RandMax, but with a procedural interface.
PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                     p_MaxVal IN NUMBER);
END Random;

CREATE OR REPLACE PACKAGE BODY Random AS

/* Used for calculating the next number. */
v_Multiplier CONSTANT NUMBER := 22695477;
v_Increment   CONSTANT NUMBER := 1;

/* Seed used to generate random sequence. */
v_Seed         number := 1;

PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
BEGIN
    v_Seed := p_NewSeed;
END ChangeSeed;

FUNCTION Rand RETURN NUMBER IS
BEGIN
    v_Seed := MOD(v_Multiplier * v_Seed + v_Increment,
                  (2 ** 32));
    RETURN BITAND(v_Seed/(2 ** 16), 32767);
END Rand;

PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
BEGIN
    -- Simply call Rand and return the value.
    p_RandomNumber := Rand;
END GetRand;

FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER IS
BEGIN
    RETURN MOD(Rand, p_MaxVal) + 1;

```

```

END RandMax;

PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                      p_MaxVal IN NUMBER) IS
BEGIN
  -- Simply call RandMax and return the value.
  p_RandomNumber := RandMax(p_MaxVal);
END GetRandMax;

BEGIN
  /* Inicialização do pacote. Inicializa a semente com a hora
  atual em segundos */
  ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE, 'SSSSS')));
END Random;

```

Ao chamar Random.Rand, a sequência dos números aleatórios é controlada pela semente inicial - a mesma sequencia é gerada para uma dada semente. Assim, para fornecer mais valores aleatórios é preciso inicializar a semente com um valor diferente todas as vezes que o pacote for instanciado. Para isso, o procedimento ChangeSeed é chamado a partir da seção de inicialização do pacote.

## **UTILIZANDO PROCEDURES, FUNÇÕES E PACOTES**

### **Localizações do subprograma**

Os subprogramas e pacotes podem ser armazenados no dicionário de dados. Contudo, um subprograma pode ser definido dentro da seção declarativa de um bloco, neste caso, ele é conhecido *subprograma local*. Os pacotes podem ser armazenados no dicionário de dados e não podem ser locais.

### **Subprogramas armazenados e o dicionário de dados**

Quando um subprograma é criado com CREATE OR REPLACE, ele é armazenado no dicionário de dados. Além do texto de origem, o subprograma é armazenado de uma forma compilada, conhecida como *p-code* (lido a partir do disco e armazenado na SGA podendo ser acessado por vários usuários).

A visão *user\_objects* contém informações sobre todos os objetos possuídos pelo usuário atual, incluindo subprogramas armazenados (data de criação, última modificação, tipo de objeto, validade do objeto, etc.)

A visão *user\_objects* contém o código fonte-original para o objeto.

A visão *user\_errors* contém informações sobre erros de compilação.

Observação: na SQL\*Plus, o comando *show errors* consultará *user\_errors* sobre informações do último objeto criado.

### **Compilação nativa**

No Oracle9i pode-ser ter a PL/SQL compilada no código nativo criando uma biblioteca compartilhada então executada pelo processo interno do Oracle. Para utilizar este recurso é necessário ter um compilador C instalado no sistema. Para obter detalhes sobre como fazer isso deve-se consultar a documentação do Oracle.

### **Subprogramas locais**

Um subprogramma local é declarado na seção na seção declarativa de um bloco PL/SQL, como por exemplo:

```
DECLARE
    CURSOR c_AllStudents IS
        SELECT first_name, last_name
        FROM students;

    v_FormattedName VARCHAR2(50);

    /* Function which will return the first and last name
       concatenated together, separated by a space. */
    FUNCTION FormatName(p_FirstName IN VARCHAR2,
                        p_LastName IN VARCHAR2)
        RETURN VARCHAR2 IS
    BEGIN
        RETURN p_FirstName || ' ' || p_LastName;
```

```

END FormatName;

-- Begin main block.
BEGIN
  FOR v_StudentRecord IN c_AllStudents LOOP
    v_FormattedName :=
      FormatName(v_StudentRecord.first_name,
                  v_StudentRecord.last_name);
    DBMS_OUTPUT.PUT_LINE(v_FormattedName);
  END LOOP;
END;

```

Especificamente, o subprograma local é visível apenas no bloco em que é declarado.

### **Subprogramas locais como parte de subprogramas armazenados**

Da mesma forma, subprogramas locais podem ser declarados como parte da seção declarativa de um subprograma armazenado, por exemplo:

```

CREATE OR REPLACE PROCEDURE StoredProc AS
  /* Local declarations, which include a cursor, variable, and a
     function. */
  CURSOR c_AllStudents IS
    SELECT first_name, last_name
      FROM students;

  v_FormattedName VARCHAR2(50);

  /* Function which will return the first and last name
     concatenated together, separated by a space. */
  FUNCTION FormatName(p_FirstName IN VARCHAR2,
                      p_LastName IN VARCHAR2)
    RETURN VARCHAR2 IS
  BEGIN
    RETURN p_FirstName || ' ' || p_LastName;
  END FormatName;

-- Begin main block.
BEGIN
  FOR v_StudentRecord IN c_AllStudents LOOP
    v_FormattedName :=
      FormatName(v_StudentRecord.first_name,
                  v_StudentRecord.last_name);
    DBMS_OUTPUT.PUT_LINE(v_FormattedName);
  END LOOP;
END StoredProc;

```

### **Localização de subprogramas locais**

A localização de um subprograma local deve ser a última na seção declarativa caso exista outras declarações (cursos, variáveis, etc.)

## Declarações prévias

Uma vez que os nomes de subprogramas locais PL/SQL são identificadores, eles devem ser declarados antes de serem referenciados. No caso de subprogramas mutuamente referencias, considere o exemplo:

```
DECLARE
    v_TempVal BINARY_INTEGER := 5;

    -- Declaração introdutória da procedure B
    PROCEDURE B(p_Counter IN OUT BINARY_INTEGER);

    PROCEDURE A(p_Counter IN OUT BINARY_INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('A(' || p_Counter || ')');
        IF p_Counter > 0 THEN
            B(p_Counter);
            p_Counter := p_Counter - 1;
        END IF;
    END A;

    PROCEDURE B(p_Counter IN OUT BINARY_INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('B(' || p_Counter || ')');
        p_Counter := p_Counter - 1;
        A(p_Counter);
    END B;
BEGIN
    B(v_TempVal);
END;
```

## Sobrecarregando subprogramas locais

Assim como os subrprogramas declarados em pacotes podem ser sobrecarregados (como já visto), isso também é verdadeiro para subprogramas locais. Por exemplo:

```
DECLARE
    -- Two overloaded local procedures
    PROCEDURE LocalProc(p_Parameter1 IN NUMBER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('In version 1, p_Parameter1 = ' ||
                             p_Parameter1);
    END LocalProc;

    PROCEDURE LocalProc(p_Parameter1 IN VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('In version 2, p_Parameter1 = ' ||
                             p_Parameter1);
    END LocalProc;
BEGIN
    -- Call version 1
    LocalProc(12345);
```

```
-- And version 2
LocalProc( 'abcdef' );
END;
```

## Subprogramas locais *versus* armazenados

Se você possuir um subprograma útil e querer chamá-lo a partir de mais de um bloco, considere que deva ser armazenado no banco de dados. Os subprogramas locais geralmente são considerados apenas a partir de uma seção específica do programa utilizadas para evitar duplicação de código dentro de um único bloco.

## Considerações sobre subprogramas e pacotes armazenados

Subprogramas e pacotes armazenados como objetos do dicionário de dados têm a vantagem de serem compartilhados entre os usuários do banco de dados, porém as dependências dos subprogramas com os objetos armazenados podem sofrer implicações. No caso de uma procedure compilada e armazenada no dicionário de dados ela é dependente dos objetos que referencia, e pode tornar-se inválida caso uma operação DDL seja realizada em um dos seus objetos dependentes - por exemplo, ao acrescentar uma coluna extra na tabela.

## Recompilação automática

Se o objeto dependente for invalidado, o mecanismo PL/SQL tentará recompilá-lo automaticamente na próxima vez que for chamado. Uma vez que procedure, no caso, não referenciam a nova coluna, a recompilação será bem-sucedida. No caso de falha, o bloco de chamada receberá um erro de compilação em tempo de execução.

## Pacotes e dependências

No caso de pacotes o corpo de pacote depende do cabeçalho do pacote, contudo o cabeçalho de pacote não depende do corpo do pacote (caso os argumentos não precisem ser alterados). Pode-se alterar o corpo de pacote sem alterar o cabeçalho, outros objetos que dependem do cabeçalho não terão de ser recompilados. Porém, se o cabeçalho for alterado, automaticamente invalida o corpo.

Obs. As visualizações de dicionário de dados `user_dependencies`, `all_dependencies` e `dba_dependencies` listam diretamente o relacionamento entre os objetos de esquema.

## Como as invalidações são determinadas

As invalidações que foram determinadas ocorreram devido ao dicionário de dados que monitora as dependências. Por exemplo, caso duas procedures P1 e P2 na qual P1 depende de P2, o que significa que recompilar P2 invalidará P1.

```
CREATE OR REPLACE PROCEDURE P2 AS
```

```

BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside P2!');
END P2;

CREATE OR REPLACE PROCEDURE P1 AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside P1!');
P2;
END P1;

```

Ao recompilar P2...

```

SQL> ALTER PROCEDURE P2 COMPILE;
P1 se torna inválida.

```

Porém, no caso de P1 estiver em um diferente banco de dados e chamada por P2 por meio de um link de banco de dados, ao recompilar P2 não invalida imediatamente P1. Pois, o link de banco de dados (loopbak), ou seja, dependencias remotas não são monitoradas pelo dicionário de dados. Neste caso, a validade é verificada no tempo de execução, P1 e P2 são comparados para examinar se P1 precisa ser recompilado. Há dois métodos diferentes de comparação - registro de data/hora e de assinatura.

- Modelo de registro de data/hora

Nesse modelo, os registros de data/hora das ultimas modificações de dois objetos são comparados. Se o objeto base tiver um registro de data/hora mais recente que o objeto dependente, o objeto dependente será recompilado. Para utilizar o parametro de inicialização `REMOTE_DEPENDENCIES_MODE` é `TIMESTAMP` (default).

- Modelo de assinatura

Quando um procedimento é criado, uma *assinatura* é armazenada no dicionário de dados além do p-code. No caso, assinatura de P2 será alterada apenas quando os parâmetros forem alterados. Na primeira vez que P1 é compilado, a assinatura de P2 é incluída (em vez do registro de data/hora). Portanto, P1 precisa ser recompilado apenas quando a assinatura de P2 mudar. É um método diferente fornecido pela PL/SQL para determinar quando os objetos dependentes remotos precisam ser recompilados.

O parametro de inicialização `REMOTE_DEPENDENCIES_MODE` deve estar configurado como `SIGNATURE` - ambos métodos também podem ser configurados através dos comandos `ALTER SYSTEM` ou `ALTER SESSION`.

Para recompilar manualmente uma procedure, utilize o comando:

```
ALTER PROCEDURE nome_de_procedure COMPILE;
```

para funções:

```
ALTER FUNCTION nome_de_função COMPILE;
```

e para pacotes, utilize:

```
ALTER PACKAGE nome_de_pacote COMPILE;
ALTER PACKAGE nome_de_pacote COMPILE SPECIFICATION;
ALTER PACKAGE nome_de_pacote COMPILE BODY;
```

Se SPECIFICATION e BODY não forem especificados, ambos são compilados.

### **Estado em tempo de execução de pacote**

O estado da execução de um pacote - a saber, variáveis empacotadas e cursor - são mantidos na memória de cada sessão que 'instanciou' o pacote e permanente até que a sessão seja fechada, assim sendo, as variáveis em um cabeçalho de pacote podem ser utilizadas como variáveis globais.

### **Privilégios e subprogramas armazenados**

Subprogramas e pacotes armazenados são objetos no dicionário de dados, e como tais são possuídos por um usuário em particular ou esquema do banco de dados. Se forem concedidos os privilégios corretos, outros usuários podem acessar esses objetos.

#### **Privilégio EXECUTE**

Para executar subprogramas e pacotes armazenados, o privilégio EXECUTE é necessário ser concedido pela instrução GRANT.

Ao conceder o privilégio EXECUTE de um subprograma pertencente ao UsuárioA para um UsuárioB, o UsuárioB poderá executar o subprograma; os objetos dependentes do subprograma e pertencentes ao UsuárioA terão preferência. Por exemplo, se o subprograma altera uma tabela Temp1 existente tanto em UsuárioA quanto UsuárioB a preferência é executar os objetos pertencentes ao proprietário do subprograma, no caso, UsuárioA.

O Oracle8i inclui um novo recurso conhecido como "direitos do chamador" que pode modificar essa situação (visto adiante).

Observação importante: Os únicos objetos disponíveis dentro de uma procedure armazenada, função, pacote ou trigger são aqueles possuídos pelo proprietário do subprograma ou explicitamente concedidos ao proprietário.

#### **Direito do chamador versus direito do definidor**

O Oracle8i introduz um tipo diferente de solução de referência externa. Em um subprograma de *direitos do chamador*, as referências externas são resolvidas com o conjunto de privilégios do chamador, não do proprietário. Uma rotina de direitos do chamador é criada utilizando a cláusula AUTHID. Ela é válida apenas em subprogramas independentes, especificações de pacote e especificações de tipo de objeto.

A sintaxe de AUTHID é:

```
CREATE [OR REPLACE] FUNCTION nome_de_função
[lista_de_parâmetros] RETURN tipo_de_retorno
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
corpo_de_função;
```

```
CREATE [OR REPLACE] PROCEDURE nome_de_procedure
[lista_de_parâmetros]
```

```
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
corpo_de_procedure;

CREATE [OR REPLACE] PACKAGE nome_da_especificação_do_pacote
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
especificação_de_procedure;
```

Se CURRENT\_USER for especificado na cláusula AUTHID, o objeto terá os direitos do chamador. Se DEFINER for especificado, então o objeto terá os direitos do definidor. Default -> direitos do definidor. O exemplo a seguir é uma procedure de direitos do chamador:

```
CREATE OR REPLACE PROCEDURE RecordFullClasses
  AUTHID CURRENT_USER AS

  -- Note that we have to preface classes with
  -- UserA, since it is owned by UserA only.
  CURSOR c_Classes IS
    SELECT department, course
      FROM UserA.classes;
  BEGIN
    FOR v_ClassRecord IN c_Classes LOOP
      -- Record all classes which don't have very much room left
      -- in temp_table.
      IF AlmostFull(v_ClassRecord.department,
                     v_ClassRecord.course) THEN
        INSERT INTO temp_table (char_col) VALUES
          (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
           ' is almost full!');
      END IF;
    END LOOP;
  END RecordFullClasses;
```

## Triggers, visualizações e direitos do chamador

Um trigger de banco de dados sempre será executado com os direitos do definidor e com o conjunto de privilégios do esquema que possui a tabela-trigger. Isso também é verdadeiro em uma função PL/SQL que é chamada a partir de uma visão. Nesse, caso a função será executada com o conjunto de privilégios do proprietário visão.

## Utilizando funções armazenadas em instruções SQL

Se uma função independente ou uma função empacotada cumprir certas restrições, ela pode ser chamada durante a execução de uma instrução SQL. Recurso inicialmente introduzido na PL/SQL 2.1 (Oracle7 versão 7.1) e aprimorada no Oracle8i.

As funções definidas pelo usuário são chamadas de mesma maneira que as funções predefinidas como TO\_CHAR, UPPER, ADD\_MONTHS. Dependendo de onde uma função definida pelo usuário é utilizada e qual versão do Oracle esta sendo executada, ela deve cumprir diferentes restrições. Essas restrições são definidas em termos de níveis de pureza.

## Níveis de pureza para as funções

Há quatro níveis de pureza que definem quais tipos de estruturas de dados a função lê ou modifica e dependendo do nível de pureza, ele estará sujeito a restrições:

- A pureza de função está diretamente relacionada com os subprogramas que ela chama. Por exemplo, se uma função chamar uma procedure armazenada que faz um UPDATE, a função não tem o nível de pureza de WNDS e assim não pode ser utilizada dentro de uma instrução SELECT.
- Qualquer função chamada a partir de uma instrução SQL não pode modificar nenhuma tabela de banco de dados (WNDS). (obs. o Oracle8i e versões superiores, uma função chamada a partir de uma instrução não-SELECT pode modificar as tabelas de banco de dados).
- A fim de ser executada remotamente (via um link de banco de dados) ou em paralelo, uma função não deve ler ou gravar o valor de variáveis empacotadas (RNPS e WNPS).
- Independentemente dos seus níveis de pureza, funções armazenadas PL/SQL não podem ser chamadas a partir de uma cláusula de restrição CHECK de um comando CREATE TABLE ou ALTER TABLE, ou ser utilizada para especificar um valor padrão para uma coluna.

Nível de pureza	Significado	Descrição
WNDS	Não grava nenhum estado de banco de dados	A função não modifica nenhuma tabela de banco de dados (utilizando as instruções de DML)
RNDS	Não lê nenhum estado de banco de dados	A função não lê nenhuma tabela de banco de dados (utilizando a instrução SELECT)
WNPS	Não grava nenhum estado de pacote	A função não modifica nenhuma variável empacotada (nenhuma variável empacotada é utilizada no lado esquerdo de um atribuição ou de uma instrução FETCH)
RNPS	Não lê nenhum estado de pacote	A função não examina nenhuma variável (nenhuma variável empacotada aparece no lado direito de uma atribuição ou como parte de uma expressão procedural ou SQL).

Além das restrições, uma função definida pelo usuário também deve cumprir requisitos para ser chamável a partir de uma instrução SQL. Observe que as funções predefinidas também devem cumprir esses requisitos.

- A função tem de ser armazenada no banco de dados, seja isoladamente seja como parte de um pacote. Ela não deve ser local a um outro bloco.
- A função pode apenas receber parâmetros IN.
- Os parâmetros formais e o retorno da função devem utilizar apenas tipos de bando de dados, não tipos da PL/SQL como BOOLEAN OU RECORD. Os tipos do banco de dados incluem NUMBER, CHAR, VARCHAR2, ROWID, LONG, RAW, LONG RAW, DATE, bem como os novos tipos introduzidos pelo Oracle 8i e Oracle9i.
- A função não deve concluir a transação autal com COMMIT ou ROLLBACK ou reverter para um ponto de salvamento anterior à execução da função.
- Também não deve emitir nenhum comando ALTER SESSION ou ALTER SYSTEM.

Como um exemplo, a função FullName recebe um número de ID de aluno como uma entrada e retorna o nome e o sobrenome concatenados:

```
CREATE OR REPLACE FUNCTION FullName (
    p_StudentID students.ID%TYPE)
RETURN VARCHAR2 IS

    v_Result VARCHAR2(100);
BEGIN
    SELECT first_name || ' ' || last_name
    INTO v_Result
    FROM students
    WHERE ID = p_StudentID;
    RETURN v_Result;
END FullName;
```

executando...

```
SELECT ID, FullName(ID) "Full Name"
    FROM students;
```

e...

```
INSERT INTO temp_table (char_col)
VALUES (FullName(10010));
```

### Chamando funções armazenadas a partir da SQL no Oracle8i

Nas versões anteriores ao Oracle8i era utilizado o pragma RESTRICT\_REFERENCES para impor níveis de pureza em tempo de compilação para as funções empacotadas. A partir do Oracle8i, se o pragma não estiver presente, o banco de dados verificará o nível de pureza de uma função em tempo de execução. Porém, utilizar o pragma pode poupar algumtempo de execução e também serve para documentar o comportamento da função. Por exemplo:

```
CREATE OR REPLACE PACKAGE StudentOps AS

    FUNCTION FullName(p_StudentID IN students.ID%TYPE)
        RETURN VARCHAR2;

    /* Returns the number of History majors. */
    FUNCTION NumHistoryMajors
        RETURN NUMBER;
END StudentOps;

CREATE OR REPLACE PACKAGE BODY StudentOps AS
    -- Packaged variable to hold the number of history majors.
    v_NumHist NUMBER;

    FUNCTION FullName(p_StudentID IN students.ID%TYPE)
        RETURN VARCHAR2 IS
        v_Result VARCHAR2(100);
    BEGIN
        SELECT first_name || ' ' || last_name
```

```

    INTO v_Result
    FROM students
    WHERE ID = p_StudentID;

    RETURN v_Result;
END FullName;

FUNCTION NumHistoryMajors RETURN NUMBER IS
    v_Result NUMBER;
BEGIN
    IF v_NumHist IS NULL THEN
        /* Determine the answer. */
        SELECT COUNT(*)
            INTO v_Result
            FROM students
            WHERE major = 'History';
        /* And save it for future use. */
        v_NumHist := v_Result;
    ELSE
        v_Result := v_NumHist;
    END IF;

    RETURN v_Result;
END NumHistoryMajors;
END StudentOps;

```

Chamando a função a partir da SQL:

```

SQL> SELECT studentOps.FullName(ID) FROM students WHERE major='History';

ou...
SQL> INSERT INTO temp_table (num_col) VALUES
(StudentOps.NumHistoryMajors);

```

### Chamando funções a partir de instruções de DML

Antes do Oracle8i, uma função chamada a partir de uma instrução de DML não poderia atualizar o banco de dados (isso é, ela deve afirmar o nível de pureza de WNDs). Entretanto, com o Oracle8i, essa restrição foi abrandada. Agora, uma função chamada a partir de uma instrução de DML não deve ler ou modificar a(s) tabela(s) que esta(ão) sendo modificada(s) por essa instrução de DML, porém ela pode atualizar outras tabelas. Por exemplo:

```

CREATE OR REPLACE FUNCTION UpdateTemp(p_ID IN students.ID%TYPE)
    RETURN students.ID%TYPE AS
BEGIN
    INSERT INTO temp_table (num_col, char_col)
        VALUES(p_ID, 'Updated!');
    RETURN p_ID;
END UpdateTemp;

SQL> UPDATE students
    SET major = 'Nutrition'
    WHERE UpdateTemp(ID) = ID;

```

Nota: Uma função chamada a partir de uma instrução de DML em paralelo não deve modificar o banco de dados, mesmo tabelas que atualmente não estão sendo modificadas.

## Fixando no pool compartilhado

O *pool compartilhado* é a parte de SGA que contém, entre outras coisas, o p-code de subprogramas compilados conforme eles são executados. Na primeira vez que um subprograma armazenado for chamado, o p-code é carregado a partir do disco no pool compartilhado. Uma vez que o objeto não é mais referenciado, ele está livre para expirar e ser removido. Os objetos são removidos do pool compartilhado utilizando um algoritmo LRU (last recently used).

O pacote DBMS\_SHARED\_POOL permite fixar objetos no pool compartilhado. Quando um objeto é *fixado*, ele nunca será removido até que você solicite isso, independentemente da quantidade armazenada no pool ou da frequência com que o objeto é acessado. Isso pode melhorar o desempenho, pois leva tempo para recarregar um pacote a partir do disco. Fixar um objeto também ajuda a minimizar a fragmentação do pool compartilhado.

DBMS\_SHARED\_POOL tem quatro procedures: DBMS\_SHARED\_POOL.KEEP, DBMS\_SHARED\_POOL.UNKEEP, DBMS\_SHARED\_POOL.SIZES e DBMS\_SHARED\_POOL.ABORTED\_REQUEST\_THRESHOLD.

### KEEP

A procedure DBMS\_SHARED\_POOL.KEEP é utilizada para fixar objetos no pool. Pacotes, triggers, sequencias, tipos de objeto e objetos Java (Oracle8i e superior) e instruções SQL podem ser fixados.

KEEP é definido com:

```
PROCEDURE KEEP (nome VARCHAR2, flag CHAR DEFAULT 'P');
```

Uma vez que o objeto foi mantido, ele não será removido até que o banco de dados é desligado ou a procedure DBMS\_SHARED\_POOL.UNKEEP é utilizada. Observe que DBMS\_SHARED\_POOL.KEEP não carrega imediatamente o pacote no pool compartilhado; em vez disso, ele será primeiramente fixado e subsequentemente carregado.

Parâmetro	Tipo	Descrição
<i>nome</i>	VARCHAR2	Nome do objeto. Isso pode ser um nome de objeto ou o identificador associado com uma instrução SQL. O identificador SQL é a concatenação dos campos address e hash_value na visão v\$sqlarea (por padrão, selecionável apenas por SYS) e retornada pelo procedimento SIZES.
<i>flag</i>	CHAR	Determina o tipo de objeto. Significados dos valores <i>flag</i> P – pacote, função ou procedure Q – Sequencia R – Trigger

		T – Tipo de objeto (Oracle8 e superior) JS – Fonte Java (Oracle8 <i>i</i> e superior) JC – Classe Java (Oracle8 <i>i</i> e superior) JR – Recurso Java (Oracle8 <i>i</i> e superior) JD – Dados de Java compartilhados (Oracle8 <i>i</i> e superior) C – Cursor SQL
--	--	--

## UNKEEP

UNKEEP é a única maneira de remover um objeto mantido do pool compartilhado sem reiniciar o banco de dados. Objetos mantidos nunca expiram automaticamente. UNKEEP é definido com

```
PROCEDURE UNKEEP (nome VARCHAR2, flag CHAR DEFAULT ‘P’);
```

Os argumentos são os mesmo de KEEP. Se o objeto especificado ainda não existe no pool compartilhado, um erro é levantado.

## SIZES

SIZES ecoará o conteúdo do pool compartilhado para a tela. Ele é definido com:

```
PROCEDURE SIZES (tamanho_mínimo NUMBER);
```

Objetos com um tamanho maior do que *tamanho\_mínimo* serão retornados. SIZES utiliza DBMS\_OUTPUT para retornar os dados, portanto certifique-se de que utilizar “set serveroutput on” na SQL\*Plus ou Server Manager antes de chamar a procedure.

## ABORTED\_REQUEST\_THRESHOLD

Quando o banco de dados determina que não há memória suficiente no pool compartilhado para satisfazer uma dada solicitação, ele começa a expirar os objetos até que haja memória suficiente. Se um número suficiente de objetos expirar, isso pode ter um impacto no desempenho em outras sessões do banco de dados. ABORTED\_REQUEST\_THRESHOLD pode ser utilizada para reparar isso. Ela é definida com:

```
PROCEDURE ABORTED_REQUEST_THRESHOLD (tamanho_limiar NUMBER);
```

Uma vez que essa procedure tenha chamada, o Oracle não começará a expirar os objetos no pool a menos que pelo menos bytes *tamanho\_limiar* sejam necessários.

## TRIGGERS DE BANCO DE DADOS

Triggers (ou gatilhos) são semelhantes as procedures ou funções, pelo fato de serem blocos identificados PL/SQL com seções declarativas, executáveis e de tratamento de exceções. Por outro lado, um trigger é executado implicitamente sempre que o evento desencadeador acontece. Um trigger também não aceita argumentos.

Os triggers têm várias utilidades, incluindo:

- Manter restrições complexas de integridade
- Fazer auditoria das informações com detalhes
- Sinalizar automaticamente a outros programas que uma ação precisa acontecer quando são feitas alterações em uma tabela

Há três tipos principais de triggers: DML, instead-of e triggers de sistema.

### Sintaxe para criação de triggers

```
CREATE [OR REPLACE] TRIGGER nome
  {BEFORE | AFTER | INSTEAD-OF} evento
  [cláusula_de_referência]
  [WHEN condição_da_trigger]
  [FOR EACH ROW]
  corpo_de_trigger;
```

### Criando triggers de DML

São acionados em uma operação INSERT, UPDATE ou DELETE de uma tabela de banco de dados. Pode ser acionado antes ou depois que uma instrução é executada e pode ser acionado uma vez por linha ou uma vez por instrução.

Uma tabela pode ter qualquer número de triggers definidos nela, incluindo mais de um dado tipo de DML.

Exemplo:

```
SQL> CREATE SEQUENCE trig_seq
  2      START WITH 1
  3      INCREMENT BY 1;
```

Sequence created.

```
SQL>
SQL> CREATE OR REPLACE PACKAGE TrigPackage AS
  2      -- Global counter for use in the triggers
  3      v_Counter NUMBER;
  4  END TrigPackage;
  5 /
```

Package created.

```

SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesBStatement
  2      BEFORE UPDATE ON classes
  3      BEGIN
  4          -- Reset the counter first.
  5          TrigPackage.v_Counter := 0;
  6
  7          INSERT INTO temp_table (num_col, char_col)
  8              VALUES (trig_seq.NEXTVAL,
  9                      'Before Statement: counter = ' || 
TrigPackage.v_Counter);
 10
 11      -- And now increment it for the next trigger.
 12      TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
 13  END ClassesBStatement;
 14 /

```

Trigger created.

```

SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesAStatement1
  2      AFTER UPDATE ON classes
  3      BEGIN
  4          INSERT INTO temp_table (num_col, char_col)
  5              VALUES (trig_seq.NEXTVAL,
  6                      'After Statement 1: counter = ' || 
TrigPackage.v_Counter);
  7
  8      -- Increment for the next trigger.
  9      TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
10  END ClassesAStatement1;
11 /

```

Trigger created.

```

SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesAStatement2
  2      AFTER UPDATE ON classes
  3      BEGIN
  4          INSERT INTO temp_table (num_col, char_col)
  5              VALUES (trig_seq.NEXTVAL,
  6                      'After Statement 2: counter = ' || 
TrigPackage.v_Counter);
  7
  8      -- Increment for the next trigger.
  9      TrigPackage.v_Counter := TrigPackage.v_Counter + 1;

```

```
10 END ClassesAStatement2;
11 /
```

Trigger created.

```
SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesBRow1
2   BEFORE UPDATE ON classes
3   FOR EACH ROW
4 BEGIN
5   INSERT INTO temp_table (num_col, char_col)
6   VALUES (trig_seq.NEXTVAL,
7           'Before Row 1: counter = ' || TrigPackage.v_Counter);
8
9   -- Increment for the next trigger.
10  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
11 END ClassesBRow1;
12 /
```

Trigger created.

```
SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesBRow2
2   BEFORE UPDATE ON classes
3   FOR EACH ROW
4 BEGIN
5   INSERT INTO temp_table (num_col, char_col)
6   VALUES (trig_seq.NEXTVAL,
7           'Before Row 2: counter = ' || TrigPackage.v_Counter);
8
9   -- Increment for the next trigger.
10  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
11 END ClassesBRow2;
12 /
```

Trigger created.

```
SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesBRow3
2   BEFORE UPDATE ON classes
3   FOR EACH ROW
4 BEGIN
5   INSERT INTO temp_table (num_col, char_col)
6   VALUES (trig_seq.NEXTVAL,
```

```

7           'Before   Row   3:   counter   =   '   ||
TrigPackage.v_Counter);
8
9      -- Increment for the next trigger.
10     TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
11 END ClassesBRow3;
12 /

```

Trigger created.

```

SQL>
SQL> CREATE OR REPLACE TRIGGER ClassesARow
2   AFTER UPDATE ON classes
3   FOR EACH ROW
4 BEGIN
5   INSERT INTO temp_table (num_col, char_col)
6   VALUES (trig_seq.NEXTVAL,
7          'After Row: counter = ' || TrigPackage.v_Counter);
8
9   -- Increment for the next trigger.
10  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
11 END ClassesARow;
12 /

```

Trigger created.

```

SQL>
SQL> DELETE FROM temp_table;

```

0 rows deleted.

```

SQL>
SQL> UPDATE classes
2   SET num_credits = 4
3   WHERE department IN ('HIS', 'CS');

```

4 rows updated.

```

SQL>
SQL> SELECT *
2   FROM temp_table
3   ORDER BY num_col;

```

NUM_COL	CHAR_COL
1	Before Statement: counter = 0

```

2 Before Row 3: counter = 1
3 Before Row 2: counter = 2
4 Before Row 1: counter = 3
5 After Row: counter = 4
6 Before Row 3: counter = 5
7 Before Row 2: counter = 6
8 Before Row 1: counter = 7
9 After Row: counter = 8
10 Before Row 3: counter = 9
11 Before Row 2: counter = 10
12 Before Row 1: counter = 11
13 After Row: counter = 12
14 Before Row 3: counter = 13
15 Before Row 2: counter = 14
16 Before Row 1: counter = 15
17 After Row: counter = 16
18 After Statement 2: counter = 17
19 After Statement 1: counter = 18

```

19 rows selected.

A instrução UPDATE na tabela afeta quatro linhas. Os triggers BEFORE e AFTER do nível de instrução são executados uma vez e os triggers BEFORE e AFTER FOR EACH ROW são executadas quatro vezes, uma para cada linha.

### **Identificadores de correlação em triggers de nível de linha**

Um trigger FOR EACH ROW é acionado uma vez por linha. Dentro do trigger, você pode acessar os dados da linha que está sendo processada atualmente. Isso é realizado através dos identificadores :NEW e :OLD. Esses identificadores são semelhantes a uma variável do tipo registro TABELA\_DA\_TRIGGER%ROWTYPE.

Devemos observar que :OLD é NULL para triggers em instruções INSERT, e :NEW é NULL para instruções DELETE.

Exemplo:

```

CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT ON students
  FOR EACH ROW
BEGIN
  SELECT student_sequence.NEXTVAL
    INTO :new.ID
    FROM dual;
END GenerateStudentID;
/

```

## Cláusula REFERENCING

Se desejar, você pode utilizar a cláusula **REFERENCING** para especificar um nome diferente para :old e :new.

Exemplo:

```
CREATE OR REPLACE TRIGGER GenerateStudentID
  BEFORE INSERT OR UPDATE ON students
  REFERENCING new AS new_student
  FOR EACH ROW
BEGIN
  SELECT student_sequence.nextval
    INTO :new_student.ID
    FROM dual;
END GenerateStudentID;
```

## A cláusula WHEN

É válida apenas para triggers FOR EACH ROW. Quando presente, a trigger será executada apenas para aquelas linhas que satisfazem a condição especificada. Ela será avaliada para cada linha processada. Os registros :new e :old também podem ser referenciados dentro da cláusula WHEN, mas os dois-pontos não são utilizados.

Exemplo:

```
CREATE OR REPLACE TRIGGER CheckCredits
  BEFORE INSERT OR UPDATE OF current_credits ON students
  FOR EACH ROW
  WHEN (new.current_credits > 20)
  BEGIN
    /* corpo de trigger aqui */
  END;
```

## Predicados de trigger: INSERTING, UPDATING e DELETING

Um mesmo trigger pode ser disparado para INSERT, UPDATE e DELETE. Dentro de um trigger desse tipo há três funções booleanas que podem ser usadas para determinar qual é a operação que está disparando o trigger.

Exemplo:

```
CREATE OR REPLACE TRIGGER LogRSChanges
  BEFORE INSERT OR DELETE OR UPDATE ON registered_students
  FOR EACH ROW
  DECLARE
    v_ChangeType CHAR(1);
```

```

BEGIN
  IF INSERTING THEN
    v_ChangeType := 'I';
  ELSIF UPDATING THEN
    v_ChangeType := 'U';
  ELSE
    v_ChangeType := 'D';
  END IF;

  INSERT INTO RS_audit
  (change_type, changed_by, timestamp,
   old_student_id, old_department, old_course, old_grade,
   new_student_id, new_department, new_course, new_grade)
  VALUES
  (v_ChangeType, USER, SYSDATE,
   :old.student_id,      :old.department,      :old.course,
   :old.grade,
   :new.student_id,      :new.department,      :new.course,
   :new.grade);
END LogRSChanges;
/

```

## Criando triggers Instead-of

Podem ser definidos apenas em VIEWS. Diferente de um trigger de DML, um trigger instead-of será executado para substituir a instrução DML que o acionou. Esses triggers devem ser do tipo FOR EACH ROW, e são utilizados para permitir que uma VIEW que não poderia ser modificada se torne atualizável.

Em geral, uma VIEW permite modificação em suas linhas se não contiver:

- Conjunto de operadores (UNION, UNION ALL, MINUS)
- Funções agregadas (SUM, AVG, COUNT,...)
- Cláusulas GROUP BY, CONNECT BY ou START WITH
- O operador DISTINCT
- Junções (Joins)

É possível modificar a tabela base de VIEWS que possuem certos tipos de JOIN. A tabela base é aquela que depois de uma junção com outra tabela, as chaves da tabela original são também as chaves na junção resultante.

Exemplo de instead-of:

```

SQL> CREATE OR REPLACE VIEW classes_rooms AS
  2    SELECT department, course, building, room_number
  3    FROM rooms, classes
  4    WHERE rooms.room_id = classes.room_id;

```

View created.

```

SQL>
SQL> SELECT * FROM classes_rooms;

DEP      COURSE  BUILDING      ROOM_NUMBER
---  -----  -----  -----
HIS        101 Building 7      201
HIS        301 Building 6      170
CS         101 Building 6      101
ECN        203 Building 6      150
CS         102 Building 6      160
MUS        410 Music Building 100
ECN        101 Building 7      300
NUT        307 Building 7      310

```

**8 rows selected.**

```

SQL>
SQL> INSERT INTO classes_rooms (department, course, building,
room_number)
2      VALUES ('MUS', 100, 'Music Building', 200);
*                                *
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a
join view

```

```

SQL>
SQL> CREATE TRIGGER ClassesRoomsInsert
2      INSTEAD OF INSERT ON classes_rooms
3      DECLARE
4          v_roomID rooms.room_id%TYPE;
5      BEGIN
6          -- First determine the room ID
7          SELECT room_id
8              INTO v_roomID
9              FROM rooms
10             WHERE building = :new.building
11             AND room_number = :new.room_number;
12
13          -- And now update the class
14          UPDATE CLASSES
15             SET room_id = v_roomID
16             WHERE department = :new.department
17             AND course = :new.course;
18      END ClassesRoomsInsert;
19      /

```

Trigger created.

```
SQL> INSERT INTO classes_rooms (department, course, building,
room_number)
2      VALUES ( 'MUS' , 100 , 'Music Building' , 200 );
```

**1 row created.**

```
SQL>
SQL> SELECT * FROM classes_rooms;
```

DEP	COURSE	BUILDING	ROOM_NUMBER
HIS	101	Building 7	201
HIS	301	Building 6	170
CS	101	Building 6	101
ECN	203	Building 6	150
CS	102	Building 6	160
MUS	410	Music Building	100
ECN	101	Building 7	300
NUT	307	Building 7	310
MUS	100	Music Building	200

9 rows selected.

## Criando triggers de sistema

São acionados em dois tipos diferentes de eventos: DLL ou banco de dados.

Sintaxe:

```
CREATE [OR REPLACE] TRIGGER [esquema.]nome_do_trigger
{BEFORE | AFTER}
{lista_de_evento_DDL / lista_de_evento_de_banco_de_dados}
ON {DATABASE | [esquema.] SCHEMA}
[cláusula_when]
corpo_de_trigger;
```

Eventos de DLL disponíveis para triggers de sistema:

Evento	Sincronização	Descrição
STARTUP	AFTER	Acionado quando uma instância é iniciada
SHUTDOWN	BEFORE	Acionado quando uma instância é desativada
SERVERERROR	AFTER	Acionado quando ocorrer um erro
LOGON	AFTER	Acionado depois que um usuário é conectado
LOGOFF	BEFORE	Acionado no início de um Logoff de usuário
CREATE	BEFORE, AFTER	Acionado quando um objeto é criado

Evento	Sincronização	Descrição
DROP	BEFORE, AFTER	Acionado quando um objeto é descartado
ALTER	BEFORE, AFTER	Acionado quando um objeto é alterado

### Triggers de banco de dados versus esquema

As palavras chave DATABASE e SCHEMA determinam o nível para um trigger, que pode ser acionado ao ocorrer um evento no banco de dados como um todo, ou apenas em determinado esquema de usuário. Se o esquema não for especificado com a palavra SCHEMA, o esquema que possui o trigger é assumido por padrão.

Exemplo:

```
connect UserA/UserA
CREATE OR REPLACE TRIGGER LogUserAConnects
    AFTER LOGON ON SCHEMA
BEGIN
    INSERT INTO example.temp_table
        VALUES (1, 'LogUserAConnects fired!');
END LogUserAConnects;
/
```

### Funções do atributo de evento

Dentro de um trigger de sistema há várias funções disponíveis para obter informações sobre o evento que o disparou.

Exemplo:

```
CREATE OR REPLACE TRIGGER LogCreations
    AFTER CREATE ON DATABASE
BEGIN
    INSERT INTO ddl_creations (user_id, object_type,
object_name,
                                object_owner, creation_date)
        VALUES (USER,
                SYS.DICTIONARY_OBJ_TYPE,
                SYS.DICTIONARY_OBJ_NAME,
                SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;
/
```

Função	Tipo de dados	Evento de sistema	Descrição
SYSEVENT	VARCHAR2(20)	Todos	Retorna o evento que acionou o trigger

INSTANCE_NUM	NUMBER	Todos	Retorna o número da instância atual
DATABASE_NAME	VARCHAR2(50)	Todos	Retorna o nome do banco de dados
SERVER_ERROR	NUMBER	SERVERERROR	Retorna o erro na posição da pilha de erros
IS_SERVERERROR	BOOLEAN	SERVERERROR	Retorna TRUE se o erro passado como parâmetro se encontra na pilha de erros
LOGIN_USER	VARCHAR2(30)	Todos	Retorna o usuário que acionou o trigger
DICTIONARY_OBJ_TPYE	VARCHAR2(30)	CREATE, DROP, ALTER	Retorna o tipo de objeto
DICTIONARY_OBJ_NAME	VARCHAR2(30)	CREATE, DROP, ALTER	Retorna o nome do objeto da operação DLL
DICTIONARY_OBJ_OWNER	VARCHAR2(30)	CREATE, DROP, ALTER	Retorna o proprietário do objeto da DLL

Exemplo:

```

CREATE TABLE error_log (
    timestamp      DATE,
    username       VARCHAR2(30),
    instance       NUMBER,
    database_name VARCHAR2(50),
    error_stack    VARCHAR2(2000)
);

CREATE OR REPLACE TRIGGER LogErrors
    AFTER SERVERERROR ON DATABASE
BEGIN
    INSERT INTO error_log
        VALUES (SYSDATE, SYS.LOGIN_USER, SYS.INSTANCE_NUM, SYS.
                DATABASE_NAME, DBMS_UTILITY FORMAT_ERROR_STACK);
END LogErrors;
/

```

## Corpos de triggers

O corpo de um trigger não pode exceder 32K. Caso isso ocorra, você pode reduzi-lo movendo alguns dos códigos em procedures ou pacotes ou utilizar a instrução CALL para invocar um procedimento armazenado.

Exemplo:

```
CREATE OR REPLACE PACKAGE LogPkg AS
    PROCEDURE LogConnect(p.UserID IN VARCHAR2);
    PROCEDURE LogDisconnect(p.UserID IN VARCHAR2);
END LogPkg;
/

CREATE OR REPLACE PACKAGE BODY LogPkg AS
    PROCEDURE LogConnect(p.UserID IN VARCHAR2) IS
        BEGIN
            INSERT INTO connect_audit (user_name, operation,
            timestamp)
            VALUES (p.USerID, 'CONNECT', SYSDATE);
        END LogConnect;

    PROCEDURE LogDisconnect(p.UserID IN VARCHAR2) IS
        BEGIN
            INSERT INTO connect_audit (user_name, operation,
            timestamp)
            VALUES (p.USerID, 'DISCONNECT', SYSDATE);
        END LogDisconnect;
END LogPkg;
/

CREATE OR REPLACE TRIGGER LogConnects
    AFTER LOGON ON DATABASE
    CALL LogPkg.LogConnect(SYS.LOGIN_USER)
/
CREATE OR REPLACE TRIGGER LogDisconnects
    BEFORE LOGOFF ON DATABASE
    CALL LogPkg.LogDisconnect(SYS.LOGIN_USER)
/
```

## Privilégios de trigger

Há cinco privilégios de sistema que podem ser atribuídos aos usuários do banco de dados, são eles: **CREATE TRIGGER**, **CREATE ANY TRIGGER**, **ALTER ANY TRIGGER**, **DROP ANY TRIGGER**, **ADMINISTER DATABASE TRIGGER**.

## Views do dicionário de dados

Certas views do dicionário de dados contêm informações sobre as triggers e seus status. São elas: USER\_TRIGGERS, ALL\_TRIGGER, DBA\_TRIGGER.

## Descartando e desativando triggers

Semelhante a outros objetos, os triggers também podem ser descartados.

```
DROP TRIGGER nome_do_trigger;
```

Entretanto, um trigger pode ser desativado sem ser excluído. Quando um trigger é desativado, ele ainda existe no dicionário de dados, porém nunca é acionado.

```
ALTER TRIGGER nome_do_trigger {DISABLE | ENABLE};
```

Todos os trigger para uma tabela particular também podem ser ativados ou desativados utilizando o comando ALTER TABLE. Exemplo:

```
ALTER TABLE students DISABLE ALL TRIGGERS;
```

```
ALTER TABLE students ENABLE ALL TRIGGERS;
```

## Tabelas que sofrem mutação

Não podemos efetuar certos DMLs na tabela associada a uma trigger FOR EACH ROW, pois esta tabela está sendo atualizada nesse momento.

Exemplo:

Suponha que queremos limitar para cinco o número de alunos de cada especialização. Poderíamos realizar isso com um trigger BEFORE INSERT OR UPDATE FOR EACH ROW em studens:

```
SQL> CREATE OR REPLACE TRIGGER LimitMajors
  2  BEFORE INSERT OR UPDATE OF major ON students
  3  FOR EACH ROW
  4  DECLARE
  5    v_MaxStudents CONSTANT NUMBER := 5;
  6    v_CurrentStudents NUMBER;
  7  BEGIN
  8    SELECT COUNT(*)
  9      INTO v_CurrentStudents
 10     FROM students
 11    WHERE major = :new.major;
 12
```

```

13      IF v_CurrentStudents + 1 > v_MaxStudents THEN
14          RAISE_APPLICATION_ERROR(-20000,
15              'Too many students in major ' || :new.major);
16      END IF;
17  END LimitMajors;
18 /

```

Trigger created.

```

SQL>
SQL> UPDATE students
  2      SET major = 'History'
  3      WHERE ID = 10003;
UPDATE students
*
ERROR at line 1:
ORA-04091: table SCOTT.STUDENTS is mutating, trigger/function
may not see it
ORA-06512: at "SCOTT.LIMITMAJORS", line 5
ORA-04088: error during execution of trigger 'SCOTT.LIMITMAJORS'

```

Students está sofrendo mutação apenas para um trigger **FOR EACH ROW**. Isso significa que não podemos consultá-lo em um trigger desse tipo, mas podemos em um trigger de nível de instrução. Entretanto, não podemos simplesmente tornar a trigger LIMITMAJORS um trigger de instrução, pois precisamos utilizar o valor de :new.major no corpo do trigger. A solução para isso é criar dois triggers – um de nível de linha e outro de nível de instrução. No trigger de linha, registramos os valores desejados em variáveis gobais para a sessão, e no trigger de instrução usamos esses valores para fazer as verificações através da DML SELECT.

```

SQL> PROMPT StudentData package...
StudentData package...
SQL> CREATE OR REPLACE PACKAGE StudentData AS
  2      TYPE t_Majors IS TABLE OF students.major%TYPE
  3          INDEX BY BINARY_INTEGER;
  4      TYPE t_IDs IS TABLE OF students.ID%TYPE
  5          INDEX BY BINARY_INTEGER;
  6
  7      v_StudentMajors t_Majors;
  8      v_StudentIDs    t_IDs;
  9      v_NumEntries    BINARY_INTEGER := 0;
10  END StudentData;
11 /

```

Package created.

SQL>

```

SQL> PROMPT RLimitMajors trigger...
RLimitMajors trigger...
SQL> CREATE OR REPLACE TRIGGER RLimitMajors
  2      BEFORE INSERT OR UPDATE OF major ON students
  3      FOR EACH ROW
  4      BEGIN
  5          StudentData.v_NumEntries := StudentData.v_NumEntries +
1;
  6          StudentData.v_StudentMajors(StudentData.v_NumEntries)
:=
  7          :new.major;
  8          StudentData.v_StudentIDs(StudentData.v_NumEntries) :=
:new.id;
  9      END RLimitMajors;
10  /

```

Trigger created.

```

SQL>
SQL> PROMPT SLimitMajors trigger...
SLimitMajors trigger...
SQL> CREATE OR REPLACE TRIGGER SLimitMajors
  2      AFTER INSERT OR UPDATE OF major ON students
  3      DECLARE
  4          v_MaxStudents      CONSTANT NUMBER := 5;
  5          v_CurrentStudents NUMBER;
  6          v_StudentID        students.ID%TYPE;
  7          v_Major            students.major%TYPE;
  8      BEGIN
  9          FOR v_LoopIndex IN 1..StudentData.v_NumEntries LOOP
10              v_StudentID      :=
StudentData.v_StudentIDs(v_LoopIndex);
11              v_Major := StudentData.v_StudentMajors(v_LoopIndex);
12
13              SELECT COUNT(*)
14                  INTO v_CurrentStudents
15                  FROM students
16                  WHERE major = v_Major;
17
18              IF v_CurrentStudents > v_MaxStudents THEN
19                  RAISE_APPLICATION_ERROR(-20000,
20                      'Too many students for major ' || v_Major ||
21                      ' because of student ' || v_StudentID);
22              END IF;
23          END LOOP;
24
25          StudentData.v_NumEntries := 0;

```

```
26 END SLimitMajors;
27 /
```

Trigger created.

```
SQL>
SQL> UPDATE students
2      SET major = 'History'
3      WHERE ID = 10003;
```

1 row updated.

```
SQL> UPDATE students
2      SET major = 'History'
3      WHERE ID = 10002;
```

1 row updated.

```
SQL>
SQL> UPDATE students
2      SET major = 'History'
3      WHERE ID = 10009;
UPDATE students
*
ERROR at line 1:
ORA-20000: Too many students for major History because of
student 10009
ORA-06512: at "SCOTT.SLIMITMAJORS", line 17
ORA-04088: error during execution of trigger
'SCOTT.SLIMITMAJORS'
```

## RECURSOS AVANÇADOS

### SQL Dinâmica nativa

Como vimos, apenas instruções DML podem ser incluídas diretamente em blocos PL/SQL. Porém, isso pode ser retificado por meio da SQL dinâmica. Em vez de ser analisada sintaticamente junto com o bloco PL/SQL, a SQL dinâmica é analisada sintaticamente e é subsequentemente executada em tempo de execução. Há duas técnicas para executar a SQL dinâmica. A primeira é o pacote DBMS\_SQL, a segunda, introduzida no Oracle8i, é a SQL dinâmica nativa através da instrução EXECUTE IMMEDIATE. Esta segunda técnica é uma parte integrante da própria linguagem PL/SQL, e consequentemente é significativamente mais simples e rápida que utilizar o pacote DBMS\_SQL.

Exemplo:

```
SQL> set serveroutput on
SQL>
SQL> DECLARE
 2   v_SQLString  VARCHAR2(200);
 3   v_PLSQLBlock VARCHAR2(200);
 4 BEGIN
 5   EXECUTE IMMEDIATE
 6     'CREATE TABLE execute_table (col1 VARCHAR(10))';
 7
 8   FOR v_Counter IN 1..10 LOOP
 9     v_SQLString :=
10       'INSERT INTO execute_table
11         VALUES (''Row '' || v_Counter || '')';
12     EXECUTE IMMEDIATE v_SQLString;
13   END LOOP;
14
15   v_PLSQLBlock :=
16     'BEGIN
17       FOR v_Rec IN (SELECT * FROM execute_table) LOOP
18         DBMS_OUTPUT.PUT_LINE(v_Rec.col1);
19       END LOOP;
20     END;';
21
22   EXECUTE IMMEDIATE v_PLSQLBlock;
23
24   EXECUTE IMMEDIATE 'DROP TABLE execute_table';
25 END;
26 /
```

Row 1  
Row 2  
Row 3

```
Row 4
Row 5
Row 6
Row 7
Row 8
Row 9
Row 10
```

PL/SQL procedure successfully completed.

**Podemos também utilizar EXECUTE IMMEDIATE com instruções utilizando variáveis de vinculação.**

Exemplo:

```
DECLARE
    v_SQLString    VARCHAR2(1000);
    v_PLSQLBlock   VARCHAR2(1000);

    CURSOR c_EconMajor IS
        SELECT *
        FROM students
        WHERE major = 'Economics';

BEGIN
    v_SQLString := 
        ' INSERT INTO CLASSES (department, course, description,
                               max_students, current_students,
                               num_credits)
          VALUES (:dep, :course, :descr, :max_s, :cur_s,
                  :num_c) ';

    EXECUTE IMMEDIATE v_SQLString USING
        'ECN', 103, 'Economics 103', 10, 0, 3;

    FOR v_StudentRec IN c_EconMajor LOOP
        EXECUTE IMMEDIATE
            ' INSERT INTO registered_students
              (student_ID, department, course, grade)
              VALUES (:id, :dep, :course, NULL)'
            USING v_StudentRec.ID, 'ECN', 103;

        v_PLSQLBlock :=
            'BEGIN
                UPDATE classes SET current_students =
                current_students + 1
                WHERE department = :d and course = :c;
            END; ';
```

```

    EXECUTE IMMEDIATE v_PLSQLBlock USING 'ECN', 103;
END LOOP;
END;
/

```

## Consultas com EXECUTE IMMEDIATE

Podemos utilizar a EXECUTE IMMEDIATE para consultas de única linha, tanto com variáveis de vinculação como sem variáveis de vinculação. Com a vinculação em volume (que veremos adiante), é possível também utilizar para consultas com múltiplas linhas.

Exemplo:

```

1  DECLARE
2      v_SQLQuery VARCHAR2(200);
3      v_Class classes%ROWTYPE;
4      v_Description classes.description%TYPE;
5  BEGIN
6      v_SQLQuery :=
7          'SELECT description ' ||
8          ' FROM classes ' ||
9          ' WHERE department = ''ECN''' ||
10         ' AND course = 203';
11     EXECUTE IMMEDIATE v_SQLQuery
12         INTO v_Description;
13     DBMS_OUTPUT.PUT_LINE('Fetched ' || v_Description);
14     v_SQLQuery :=
15         'SELECT * ' ||
16         ' FROM classes ' ||
17         ' WHERE description = :description';
18     EXECUTE IMMEDIATE v_SQLQuery
19         INTO v_Class
20         USING v_Description;
21     DBMS_OUTPUT.PUT_LINE(
22         'Fetched ' || v_Class.department || ' ' || 
v_Class.course);
23* END;
24 /
Fetched Economics 203
Fetched ECN 203

PL/SQL procedure successfully completed.

```

## Executando consultas com cursos utilizando OPEN FOR

```
SQL> CREATE OR REPLACE PACKAGE NativeDynamic AS
 2      TYPE t_RefCur IS REF CURSOR;
 3
 4      FUNCTION StudentsQuery(p_WhereClause IN VARCHAR2)
 5          RETURN t_RefCur;
 6
 7      FUNCTION StudentsQuery2(p_Major IN VARCHAR2)
 8          RETURN t_RefCur;
 9  END NativeDynamic;
10 /
```

Package created.

```
SQL>
SQL> show errors
No errors.
SQL>
SQL> CREATE OR REPLACE PACKAGE BODY NativeDynamic AS
 2      FUNCTION StudentsQuery(p_WhereClause IN VARCHAR2)
 3          RETURN t_RefCur IS
 4          v_ReturnCursor t_RefCur;
 5          v_SQLStatement VARCHAR2(500);
 6      BEGIN
 7          v_SQLStatement := 'SELECT * FROM students ' ||
p_WhereClause;
 8
 9          OPEN v_ReturnCursor FOR v_SQLStatement;
10
11      RETURN v_ReturnCursor;
12
13      END StudentsQuery;
14
15      FUNCTION StudentsQuery2(p_Major IN VARCHAR2)
16          RETURN t_RefCur IS
17          v_ReturnCursor t_RefCur;
18          v_SQLStatement VARCHAR2(500);
19          BEGIN
20              v_SQLStatement := 'SELECT * FROM students WHERE
major = :m';
21
22              OPEN v_ReturnCursor FOR v_SQLStatement USING
p_Major;
23
24          RETURN v_ReturnCursor;
25      END StudentsQuery2;
26  END NativeDynamic;
27 /
```

Package body created.

```

SQL>
SQL> show errors
No errors.

SQL> set serveroutput on
SQL>
SQL> DECLARE
 2      v_Student students%ROWTYPE;
 3      v_StudentCur NativeDynamic.t_RefCur;
 4  BEGIN
 5      v_StudentCur :=
 6          NativeDynamic.StudentsQuery('WHERE MOD(id, 2) = 0');
 7
 8      DBMS_OUTPUT.PUT_LINE('The following students have even
IDs:');
 9      LOOP
10          FETCH v_StudentCur INTO v_Student;
11          EXIT WHEN v_StudentCur%NOTFOUND;
12          DBMS_OUTPUT.PUT_LINE(' ' || v_Student.id || ':' || |
13                                         v_Student.first_name || ' ' ||
14                                         v_Student.last_name);
15      END LOOP;
16      CLOSE v_StudentCur;
17
18      v_StudentCur := NativeDynamic.StudentsQuery2('Music');
19
20      DBMS_OUTPUT.PUT_LINE(
21          'The following students are music majors:');
22      LOOP
23          FETCH v_StudentCur INTO v_Student;
24          EXIT WHEN v_StudentCur%NOTFOUND;
25          DBMS_OUTPUT.PUT_LINE(' ' || v_Student.id || ':' || |
26                                         v_Student.first_name || ' ' ||
27                                         v_Student.last_name);
28      END LOOP;
29      CLOSE v_StudentCur;
30  END;
31  /
The following students have even IDs:
10000: Scott Smith
10002: Joanne Junebug
10004: Patrick Poll
10006: Barbara Blues
10008: Ester Elegant
10010: Rita Razmataz
The following students are music majors:

```

PL/SQL procedure successfully completed.

## Vinculação em volume

O Oracle8i e versões superiores permitem em uma operação passar todas as linhas de uma coleção ao mecanismo SQL eliminando a necessidade de alternação de contexto entre SQL e PL/SQL para cada linha da coleção.

Exemplo:

```
SQL> set serveroutput on format wrapped
SQL>
SQL> DECLARE
 2      TYPE t_Numbers IS TABLE OF temp_table.num_col%TYPE;
 3      TYPE t.Strings IS TABLE OF temp_table.char_col%TYPE;
 4      v_Numbers t_Numbers := t_Numbers(1);
 5      v.Strings t.Strings := t.Strings(1);
 6
 7      PROCEDURE PrintTotalRows(p_Message IN VARCHAR2) IS
 8          v_Count NUMBER;
 9      BEGIN
10          SELECT COUNT(*)
11              INTO v_Count
12              FROM temp_table;
13          DBMS_OUTPUT.PUT_LINE(p_Message || ': Count is ' ||
v_Count);
14      END PrintTotalRows;
15
16      BEGIN
17          DELETE FROM temp_table;
18
19          v_Numbers.EXTEND(1000);
20          v.Strings.EXTEND(1000);
21          FOR v_Count IN 1..1000 LOOP
22              v_Numbers(v_Count) := v_Count;
23              v.Strings(v_Count) := 'Element #' || v_Count;
24          END LOOP;
25
26          FORALL v_Count IN 1..1000
27              INSERT INTO temp_table VALUES
28                  (v_Numbers(v_Count), v.Strings(v_Count));
29
30          PrintTotalRows('After first insert');
31
32          FORALL v_Count IN 501..1000
33              INSERT INTO temp_table VALUES
34                  (v_Numbers(v_Count), v.Strings(v_Count));
```

```

35
36     PrintTotalRows('After second insert');
37
38     FORALL v_Count IN 1..1000
39         UPDATE temp_table
40             SET char_col = 'Changed!'
41             WHERE num_col = v_Numbers(v_Count);
42
43     DBMS_OUTPUT.PUT_LINE(
44         'Update processed ' || SQL%ROWCOUNT || ' rows.');
45
46     FORALL v_Count IN 401..600
47         DELETE FROM temp_table
48             WHERE num_col = v_Numbers(v_Count);
49
50     PrintTotalRows('After delete');
51 END;
52 /

```

After first insert: Count is 1000  
After second insert: Count is 1500  
Update processed 1500 rows.  
After delete: Count is 1200

PL/SQL procedure successfully completed.

## Questões transacionais

Se houver um erro no processamento de uma das linhas em uma operação de DML em volume, apenas essa linha é revertida. As linhas anteriores ainda são processadas.

Exemplo:

```

1  DECLARE
2      TYPE t.Strings IS TABLE OF temp_table.char_col%TYPE
3          INDEX BY BINARY_INTEGER;
4      TYPE t.Numbers IS TABLE OF temp_table.num_col%TYPE
5          INDEX BY BINARY_INTEGER;
6      v.Strings t.Strings;
7      v.Numbers t.Numbers;
8  BEGIN
9      DELETE FROM temp_table;
10     FOR v_Count IN 1..10 LOOP
11         v.Strings(v_Count) := '123456789012345678901234567890';
12         v.Numbers(v_Count) := v_Count;
13     END LOOP;
14    FORALL v_Count IN 1..10
15        INSERT INTO temp_table (num_col, char_col)
16            VALUES (v.Numbers(v_Count), v.Strings(v_Count));
17        v.Strings(6) := v.Strings(6) || 'a'; --aciona 1 carácter para
estourar a capacidade da coluna CHAR_COL CHAR(60) na sexta linha

```

```

18   FORALL v_Count IN 1..10
19     UPDATE temp_table
20       SET char_col = char_col || v.Strings(v_Count)
21     WHERE num_col = v.Numbers(v_Count);
22   EXCEPTION
23     WHEN OTHERS THEN
24       DBMS_OUTPUT.PUT_LINE('Got exception: ' || SQLERRM);
25     COMMIT;
26* END;
27 /
Got exception: ORA-01401: inserted value too large for column

```

PL/SQL procedure successfully completed.

```
SQL> select * from temp_table;
```

NUM_COL	CHAR_COL
1	12345678901234567890123456789012345678901234567890
2	12345678901234567890123456789012345678901234567890
3	12345678901234567890123456789012345678901234567890
4	12345678901234567890123456789012345678901234567890
5	12345678901234567890123456789012345678901234567890
6	123456789012345678901234567890
7	123456789012345678901234567890
8	123456789012345678901234567890
9	123456789012345678901234567890
10	123456789012345678901234567890

10 rows selected.

Podemos observar que as primeiras cinco linhas antes da linha com erro foram atualizadas e confirmadas no banco de dados.

Com o Oracle9i, uma nova cláusula SAVE EXCEPTION está disponível com a instrução FORALL. Com essa cláusula quaisquer erros que ocorrem durante o processamento em lotes serão salvos e o processamento prosseguirá. Você pode utilizar o novo atributo SQL%BULK\_EXCEPTIONS, que atua como uma tabela PL/SQL para examinar as exceções como a seguir:

```

SQL> DECLARE
2   TYPE t.Strings IS TABLE OF temp_table.char_col%TYPE
3     INDEX BY BINARY_INTEGER;
4   TYPE t.Numbers IS TABLE OF temp_table.num_col%TYPE
5     INDEX BY BINARY_INTEGER;
6   v.Strings t.Strings;
7   v.Numbers t.Numbers;
8   v.NumErrors NUMBER;
9 BEGIN
10   DELETE FROM temp_table;
11   FOR v_Count IN 1..10 LOOP
12     v.Strings(v_Count) := '123456789012345678901234567890';
13     v.Numbers(v_Count) := v_Count;
14   END LOOP;
15
16   FORALL v_Count IN 1..10

```

```

17      INSERT INTO temp_table (num_col, char_col)
18          VALUES (v_Numbers(v_Count), v_Strings(v_Count));
19
20      v_Strings(6) := v_Strings(6) || 'a';
21
22      FORALL v_Count IN 1..10 SAVE EXCEPTIONS
23          UPDATE temp_table
24              SET char_col = char_col || v_Strings(v_Count)
25              WHERE num_col = v_Numbers(v_Count);
26      EXCEPTION
27          WHEN OTHERS THEN
28              DBMS_OUTPUT.PUT_LINE('Got exception: ' || SQLERRM);
29          -- Print out any errors.
30          v_NumErrors := SQL%BULK_EXCEPTIONS.COUNT;
31          DBMS_OUTPUT.PUT_LINE(
32              'Number of errors during processing: ' || v_NumErrors);
33          FOR v_Count IN 1..v_NumErrors LOOP
34              DBMS_OUTPUT.PUT_LINE('Error ' || v_Count || ', iteration ' ||
35                  SQL%BULK_EXCEPTIONS(v_Count).error_index || ' is: ' ||
36                  SQLERRM(0 - SQL%BULK_EXCEPTIONS(v_Count).error_code));
37          END LOOP;
38
39          COMMIT;
40      END;
41  /
Got exception: ORA-24381: error(s) in array DML
Number of errors during processing: 1
Error 1, iteration 6 is: ORA-01401: inserted value too large for column

PL/SQL procedure successfully completed.

SQL> select * from temp_table;

  NUM_COL CHAR_COL
-----+
1 12345678901234567890123456789012345678901234567890
2 12345678901234567890123456789012345678901234567890
3 12345678901234567890123456789012345678901234567890
4 12345678901234567890123456789012345678901234567890
5 12345678901234567890123456789012345678901234567890
6 123456789012345678901234567890
7 12345678901234567890123456789012345678901234567890
8 12345678901234567890123456789012345678901234567890
9 12345678901234567890123456789012345678901234567890
10 12345678901234567890123456789012345678901234567890

```

10 rows selected.

Podemos observar que apenas a linha problemática (linha 6) não foi atualizada.

## A cláusula BULK COLLECT

Como vimos anteriormente, a instrução FORALL é utilizada em operações em volume para INSERT, UPDATE, DELETE utilizando coleções. A cláusula equivalente para operações SELECT com cursores implícitos ou explícitos é a instrução BULK

COLLECT. Ela é utilizada como parte da cláusula SELECT INTO, FETCH INTO e RETURNING INTO.

Exemplo:

```
SQL> set serveroutput on
SQL>
SQL> DECLARE
 2      TYPE t_Numbers IS TABLE OF temp_table.num_col%TYPE;
 3      TYPE t.Strings IS TABLE OF temp_table.char_col%TYPE;
 4      v_Numbers t_Numbers := t_Numbers(1);
 5      v.Strings t.Strings := t.Strings(1);
 6      v_Numbers2 t_Numbers;
 7      v.Strings2 t.Strings;
 8
 9      CURSOR c_char IS
10          SELECT char_col
11              FROM temp_table
12              WHERE num_col > 800
13              ORDER BY num_col;
14
15  BEGIN
16      v_Numbers.EXTEND(1500);
17      v.Strings.EXTEND(1500);
18      FOR v_Count IN 1..1000 LOOP
19          v_Numbers(v_Count) := v_Count;
20          v.Strings(v_Count) := 'Element #' || v_Count;
21          IF v_Count > 500 THEN
22              v_Numbers(v_Count + 500) := v_Count;
23              v.Strings(v_Count + 500) := 'Element #' ||
v_Count;
24          END IF;
25      END LOOP;
26
27      DELETE FROM temp_table;
28      FORALL v_Count IN 1..1500
29          INSERT INTO temp_table (num_col, char_col)
30              VALUES (v_Numbers(v_Count), v.Strings(v_Count));
31
32      SELECT num_col, char_col
33          BULK COLLECT INTO v_Numbers, v.Strings
34          FROM temp_table
35          ORDER BY num_col;
36
37      DBMS_OUTPUT.PUT_LINE(
38          'First query fetched ' || v_Numbers.COUNT || '
rows');

```

```

39
40     SELECT num_col
41         BULK COLLECT INTO v_Numbers2
42     FROM temp_table;
43
44     DBMS_OUTPUT.PUT_LINE(
45         'Second query fetched ' || v_Numbers2.COUNT || ' '
46 rows');
47
48     OPEN c_char;
49     FETCH c_char BULK COLLECT INTO v_Strings2;
50     CLOSE c_char;
51
52     DBMS_OUTPUT.PUT_LINE(
53         'Cursor fetch retrieved ' || v_Strings2.COUNT || ' '
54 rows');
55
First query fetched 1500 rows
Second query fetched 1500 rows
Cursor fetch retrieved 400 rows

PL/SQL procedure successfully completed.

```

## Tipos de objeto

A partir do Oracle8, o modelo relacional foi estendido para incluir objetos. Um tipo objeto contém tanto atributos como métodos.

Exemplo:

```

SQL> CREATE OR REPLACE TYPE Point AS OBJECT (
2
3     x NUMBER,
4     y NUMBER,
5
6     MEMBER FUNCTION ToString RETURN VARCHAR2,
7     PRAGMA RESTRICT_REFERENCES(ToString, RNDS, WNDS, RNPS, WNPS),
8
9     MEMBER FUNCTION Distance(p IN Point DEFAULT Point(0,0))
10    RETURN NUMBER,
11    PRAGMA RESTRICT_REFERENCES(Distance, RNDS, WNDS, RNPS, WNPS),
12
13    MEMBER FUNCTION Plus(p IN Point) RETURN Point,
14    PRAGMA RESTRICT_REFERENCES(Plus, RNDS, WNDS, RNPS, WNPS),
15
16    MEMBER FUNCTION Times(n IN NUMBER) RETURN Point,
17    PRAGMA RESTRICT_REFERENCES(Times, RNDS, WNDS, RNPS, WNPS)
18 );

```

```

19  /
Type created.

SQL>
SQL> show errors
No errors.
SQL>
SQL> CREATE OR REPLACE TYPE BODY Point AS
 2      -- Returns a string '(x, y)'
 3      MEMBER FUNCTION ToString RETURN VARCHAR2 IS
 4          v_Result VARCHAR2(20);
 5          v_xString VARCHAR2(8) := SUBSTR(TO_CHAR(x), 1, 8);
 6          v_yString VARCHAR2(8) := SUBSTR(TO_CHAR(y), 1, 8);
 7      BEGIN
 8          v_Result := '(' || v_xString || ', ';
 9          v_Result := v_Result || v_yString || ')';
10         RETURN v_Result;
11     END ToString;
12
13     MEMBER FUNCTION Distance(p IN Point DEFAULT Point(0,0))
14         RETURN NUMBER IS
15     BEGIN
16         RETURN SQRT(POWER(x - p.x, 2) + POWER(y - p.y, 2));
17     END Distance;
18
19     MEMBER FUNCTION Plus(p IN Point) RETURN Point IS
20         v_Result Point;
21     BEGIN
22         v_Result := Point(x + p.x, y + p.y);
23         RETURN v_Result;
24     END Plus;
25
26     MEMBER FUNCTION Times(n IN NUMBER) RETURN Point IS
27         v_Result Point;
28     BEGIN
29         v_Result := Point(x * n, y * n);
30         RETURN v_Result;
31     END Times;
32 END;
33 /

```

Type body created.

```

SQL> show errors
No errors.
SQL> -- Demonstrates some points.
SQL> DECLARE
 2     v_Point1 Point := Point(1, 2);
 3     v_Point2 Point;
 4     v_Point3 Point;
 5     BEGIN
 6         v_Point2 := v_Point1.Times(4);
 7         v_Point3 := v_Point1.Plus(v_Point2);
 8         DBMS_OUTPUT.PUT_LINE('Point 2: ' || v_Point2.ToString());
 9         DBMS_OUTPUT.PUT_LINE('Point 3: ' || v_Point3.ToString());
10        DBMS_OUTPUT.PUT_LINE('Distance between origin and point 1: ' ||

```

```

11      v_Point1.Distance);
12  DBMS_OUTPUT.PUT_LINE('Distance between point 1 and point 2: ' ||
13      v_Point1.Distance(v_Point2));
14 END;
15 /
Point 2: (4, 8)
Point 3: (5, 10)
Distance between origin and point 1:
2.23606797749978969640917366873127623544
Distance between point 1 and point 2:
6.70820393249936908922752100619382870632

PL/SQL procedure successfully completed.

```

## Armazenando objetos no banco de dados

Objetos podem ser armazenados em tabelas do banco de dados e manipulados utilizando a SQL. Uma tabela pode armazenar objetos como colunas ou pode ser uma tabela de objeto, onde cada linha representa uma instância de objeto.

Exemplo:

```
SQL> CREATE TABLE point_object_tab OF Point;
```

```
Table created.
```

```
SQL> CREATE TABLE point_column_tab (
2      key VARCHAR2(20),
3      value Point);
```

```
Table created.
```

```
SQL>
SQL> set serveroutput on
SQL>
SQL> DECLARE
2      v_Point Point := Point(1, 1);
3      v_NewPoint Point;
4      v_Key point_column_tab.key%TYPE;
5      v_XCoord NUMBER;
6      v_YCoord NUMBER;
7  BEGIN
8      INSERT INTO point_object_tab VALUES (v_Point);
9      INSERT INTO point_column_tab VALUES ('My Point', v_Point);
10
11     SELECT *
12         INTO v_XCoord, v_YCoord
13         FROM point_object_tab;
14     DBMS_OUTPUT.PUT_LINE('Relational query of object table: ' ||
15         v_XCoord || ', ' || v_YCoord);
16
17     SELECT VALUE(ot)
18         INTO v_NewPoint
```

```

19      FROM point_object_tab ot;
20      DBMS_OUTPUT.PUT_LINE('object table: ' || v_NewPoint.ToString());
21
22      SELECT key, value
23          INTO v_Key, v_NewPoint
24          FROM point_column_tab;
25      DBMS_OUTPUT.PUT_LINE('column table: ' || v_NewPoint.ToString());
26
27  END;
28 /
Relational query of object table: 1, 1
object table: (1, 1)
column table: (1, 1)

PL/SQL procedure successfully completed.

```

A fim de consultar uma linha em uma tabela de objeto e obter de volta o resultado como um objeto, você deve utilizar o operador VALUE, como mostrado acima.

Podemos também utilizar as tabelas/objetos na SQL como abaixo:

```

SQL> desc point_object_tab
Name           Null?    Type
-----          -----   -----
X              NUMBER
Y              NUMBER

SQL> desc point_column_tab
Name           Null?    Type
-----          -----   -----
KEY            VARCHAR2( 20 )
VALUE          POINT

SQL> desc point
Name           Null?    Type
-----          -----   -----
X              NUMBER
Y              NUMBER

METHOD
-----
 MEMBER FUNCTION TOSTRING RETURNS VARCHAR2

METHOD
-----
 MEMBER FUNCTION DISTANCE RETURNS NUMBER
Argument Name      Type          In/Out Default?
-----          -----   -----   -----
P                POINT         IN        DEFAULT

METHOD
-----
 MEMBER FUNCTION PLUS RETURNS POINT
Argument Name      Type          In/Out Default?
-----          -----   -----   -----
P                POINT         IN

```

```

METHOD
-----
 MEMBER FUNCTION TIMES RETURNS POINT
 Argument Name          Type          In/Out Default?
----- ----- -----
 N                      NUMBER        IN

```

```
SQL> select t.x, t.y, t.tostring() AS Metodo from point_object_tab t;
```

X	Y	METODO
1	1	(1, 1)

ou

```
SQL> select p.key, p.value.x, p.value.y, p.value.toString() Metodo from point_column_tab p;
```

KEY	VALUE.X	VALUE.Y	METODO
My Point	1	1	(1, 1)

## Referências de objeto

Os objetos armazenados no banco de dados são tidos como persistentes. Um objeto persistente pode ter uma referência. Uma referência de objeto é um ponteiro para o objeto, em vez do próprio objeto. Para isso, utilizamos os operadores REF e DEREF.

Exemplo:

```

DECLARE
  v_PointRef REF Point;
  v_Point Point;
BEGIN
  DELETE FROM point_object_tab;

  INSERT INTO point_object_tab (x, y)
    VALUES (0, 0);
  INSERT INTO point_object_tab (x, y)
    VALUES (1, 1);

  SELECT REF(ot)
    INTO v_PointRef
    FROM point_object_tab ot
    WHERE x = 1 AND y = 1;

  SELECT DEREF(v_PointRef)
    INTO v_Point
    FROM dual;
  DBMS_OUTPUT.PUT_LINE('Selected reference ' ||
```

```

    v_Point.ToString();

    INSERT INTO point_object_tab ot (x, y)
      VALUES (10, 10)
      RETURNING REF(ot) INTO v_PointRef;
END;
/

```

## FUNÇÕES DE TABELA EM PIPELINE

A fim de que uma função PL/SQL retorne mais de uma linha de dados, ele deve retornar um REF CURSOR ou uma coleção de dados após esta estar montada. Uma função de tabela em pipeline é semelhante a isso, mas ela retorna os dados à medida que eles são construídos, em vez de todos de uma só vez.

Exemplo:

```

SQL> CREATE TYPE MyType AS OBJECT (
 2     field1 NUMBER,
 3     field2 VARCHAR2(50));
 4   /

```

Type created.

```

SQL>
SQL> CREATE TYPE MyTypeList AS TABLE OF MyType;
 2   /

```

Type created.

```

SQL> CREATE OR REPLACE FUNCTION PipelineMe
 2   RETURN MyTypeList PIPELINED AS
 3   v_MyType MyType;
 4 BEGIN
 5   FOR v_Count IN 1..20 LOOP
 6     v_MyType := MyType(v_Count, 'Row ' || v_Count);
 7     PIPE ROW(v_MyType);
 8   END LOOP;
 9   RETURN;
10 END PipelineMe;
11 /

```

Function created.

```

SQL>
SQL> SELECT *
 2   FROM TABLE(PipelineMe);

```

```

FIELD1 FIELD2
-----
1 Row 1
2 Row 2
3 Row 3
4 Row 4
5 Row 5
6 Row 6
7 Row 7
8 Row 8
9 Row 9
10 Row 10
11 Row 11
12 Row 12
13 Row 13
14 Row 14
15 Row 15
16 Row 16
17 Row 17
18 Row 18
19 Row 19
20 Row 20

20 rows selected.

```

### **Pacotes avançados**

Além dos recursos de linguagem que examinamos nas seções anteriores, a PL/SQL tem vários pacotes predefinidos. Esses pacotes trazem uma funcionalidade adicional à linguagem como os protocolos de comunicação e acesso ao sistema de arquivos. Vejamos brevemente alguns desses pacotes:

#### **DBMS\_SQL**

Utilizado para executar a SQL dinâmica de dentro da PL/SQL.

Exemplo:

```

CREATE OR REPLACE PROCEDURE UpdateClasses(
    p_Department IN classes.department%TYPE,
    p_NewCredits IN classes.num_credits%TYPE,
    p_RowsUpdated OUT INTEGER) AS

    v_CursorID    INTEGER;
    v_UpdateStmt  VARCHAR2(100);
BEGIN

```

```

v_CursorID := DBMS_SQL.OPEN_CURSOR;

v_UpdateStmt :=
  'UPDATE classes
   SET num_credits = :nc
   WHERE department = :dept';

DBMS_SQLPARSE(v_CursorID, v_UpdateStmt, DBMS_SQL.NATIVE);

DBMS_SQL.BIND_VARIABLE(v_CursorID, ':nc', p_NewCredits);

DBMS_SQL.BIND_VARIABLE_CHAR(v_CursorID, ':dept',
p_Department);

p_RowsUpdated := DBMS_SQL.EXECUTE(v_CursorID);

DBMS_SQL.CLOSE_CURSOR(v_CursorID);

EXCEPTION
  WHEN OTHERS THEN
    DBMS_SQL.CLOSE_CURSOR(v_CursorID);
    RAISE;
END UpdateClasses;
/

```

## **DBMS\_PIPE**

Permite enviar mensagens entre sessões que estão conectadas ao mesmo banco de dados. As sessões podem estar em diferentes programas cliente ou em diferentes máquinas.

### **Exemplo de ENVIO:**

```

SQL> DECLARE
 2   v_PipeName VARCHAR2(30) := 'MyPipe';
 3   v_Status INTEGER;
 4 BEGIN
 5   DBMS_PIPE.PACK_MESSAGE(SYSDATE);
 6   DBMS_PIPE.PACK_MESSAGE(123456);
 7   DBMS_PIPE.PACK_MESSAGE('This is a message sent from the pipe!');
 8
 9   v_Status := DBMS_PIPE.SEND_MESSAGE(v_PipeName);
10   IF v_Status != 0 THEN
11     DBMS_OUTPUT.PUT_LINE('Error ' || v_Status ||
12                           ' while sending message');
13   END IF;
14 END;
15 /

```

PL/SQL procedure successfully completed.

### **Exemplo de recebimento:**

```

SQL> set serveroutput on
SQL>
SQL> DECLARE
 2   v_PipeName VARCHAR2(30) := 'MyPipe';
 3   v_Status INTEGER;
 4   v_DateVal DATE;
 5   v_NumberVal NUMBER;
 6   v_StringVal VARCHAR2(100);
 7 BEGIN
 8   v_Status := DBMS_PIPE.RECEIVE_MESSAGE(v_PipeName);
 9   IF v_Status != 0 THEN
10     DBMS_OUTPUT.PUT_LINE('Error ' || v_Status ||
11                           ' while receiving message');
12   END IF;
13
14   DBMS_PIPE.UNPACK_MESSAGE(v_DateVal);
15   DBMS_PIPE.UNPACK_MESSAGE(v_NumberVal);
16   DBMS_PIPE.UNPACK_MESSAGE(v_StringVal);
17
18   DBMS_OUTPUT.PUT_LINE('Unpacked ' || v_DateVal);
19   DBMS_OUTPUT.PUT_LINE('Unpacked ' || v_NumberVal);
20   DBMS_OUTPUT.PUT_LINE('Unpacked ' || v_StringVal);
21 END;
22 /

```

Unpacked 16/05/2004  
 Unpacked 123456  
 Unpacked This is a message sent from the pipe!

PL/SQL procedure successfully completed.

## **DBMS\_ALERT**

Semelhante ao DBMS\_PIPE, pode ser utilizado para comunicar entre sessões conectadas ao mesmo banco de dados. A diferença é que mensagens enviadas via DBMS\_ALERT só são recebidas se a sessão que irá receber a mensagem estiver esperando no ato do envio.

Exemplo de envio:

```

SQL> set serveroutput on
SQL>
SQL> DECLARE
 2   v_AlertName VARCHAR2(30) := 'MyAlert';
 3 BEGIN
 4   -- An alert is sent by the SIGNAL procedure.
 5   DBMS_ALERT.SIGNAL(v_AlertName, 'Alert! Alert! Alert!');
 6
 7   -- It is not actually sent until we commit.
 8   COMMIT;
 9 END;
10 /

```

PL/SQL procedure successfully completed.

Exemplo de recebimento:

```
SQL> set serveroutput on
SQL>
SQL> DECLARE
 2   v_AlertName VARCHAR2(30) := 'MyAlert';
 3   v_Message VARCHAR2(100);
 4   v_Status INTEGER;
 5 BEGIN
 6   -- In order to receive an alert, we must first register interest
 7   -- in it.
 8   DBMS_ALERT.REGISTER(v_AlertName);
 9
10  -- Now that we have registered, we can wait.
11  DBMS_ALERT.WAITONE(v_AlertName, v_Message, v_Status);
12
13  IF v_Status = 0 THEN
14    DBMS_OUTPUT.PUT_LINE('Received: ' || v_Message);
15  ELSE
16    DBMS_OUTPUT.PUT_LINE('WAITONE timed out');
17  END IF;
18 END;
19 /
Received: Alert! Alert! Alert!
```

PL/SQL procedure successfully completed.

## UTL\_FILE

Permite a um programa PL/SQL ler e graver dados em arquivos do sistema operacional. O UTL\_FILE acessa arquivos por meio de um diretório e de um nome de arquivo. O diretório deve ser especificado pelo parâmetro UTL\_FILE\_DIR ou criado via CREATE DIRECTORY.

Exemplo:

```
SQL> CREATE DIRECTORY MEU_DIR AS 'C:\';
Directory created.

SQL>
SQL> DECLARE
 2   v_FileHandle UTL_FILE.FILE_TYPE;
 3 BEGIN
 4
 5   v_FileHandle := UTL_FILE.FOPEN('MEU_DIR', 'utl_file.txt', 'w');
 6
 7   -- Write some lines to the file.
 8   UTL_FILE.PUT_LINE(v_FileHandle, 'This is line 1!');
 9   FOR v_Counter IN 2..11 LOOP
10     UTL_FILE.PUTF(v_FileHandle, 'This is line %s!\n', v_Counter);
11   END LOOP;
12
```

```

13      -- And close the file.
14      UTL_FILE.FCLOSE(v_FileHandle);
15  END;
16 /

```

PL/SQL procedure successfully completed.

## UTL\_TCP

Fornece a capacidade de se comunicar fora do banco de dados através da comunicação via sockets.

Exemplo:

```

SQL> set serveroutput on
SQL> DECLARE
 2      v_Connection UTL_TCP.CONNECTION;
 3      v_NumWritten PLS_INTEGER;
 4  BEGIN
 5      -- Open the connection at port 80, which is the standard HTTP port
 6      v_Connection := UTL_TCP.OPEN_CONNECTION('www.oracle.com', 80);
 7
 8      -- Send HTTP request
 9  v_NumWritten := UTL_TCP.WRITE_LINE(v_Connection, 'GET / HTTP/1.0');
10  v_NumWritten := UTL_TCP.WRITE_LINE(v_Connection);
11
12      -- Print out the first 10 lines returned
13  BEGIN
14      FOR v_Count IN 1..10 LOOP
15          DBMS_OUTPUT.PUT_LINE(UTL_TCP.GET_LINE(v_Connection, TRUE));
16      END LOOP;
17  EXCEPTION
18      WHEN UTL_TCP.END_OF_INPUT THEN
19          NULL;
20      END;
21      UTL_TCP.CLOSE_CONNECTION(v_Connection);
22  END;
23 /
HTTP/1.1 200 OK
Content-Type: text/html
Connection: Close
Server: Oracle9iAS/9.0.4 Oracle HTTP Server OracleAS-Web-Cache-
10g/9.0.4.0.0
(H:max-age=210036660+0;age=114301)
Set-Cookie: ORACLE_SMP_CHRONOS_GL=298:1084737162:679151; path=/
Content-Length: 29770
Date: Sat, 15 May 2004 12:12:13 GMT
<html lang="en-US">
<head>

PL/SQL procedure successfully completed.

```

## **UTL\_SMTP**

Pode ser utilizada para enviar e-mails.

Exemplo:

```
SQL>
1  DECLARE
2      v_FromAddr VARCHAR2(50) := 'Oracle';
3      v_ToAddr VARCHAR2(50) := 'thyago@unimedcuiaba.com.br';
4      v_Message VARCHAR2(200);
5      v_MailHost VARCHAR2(50) := '10.10.0.5';
6      v_MailConnection UTL_SMTP.Connection;
7  BEGIN
8      v_Message :=
9          'From: ' || v_FromAddr || CHR(10) ||
10         'Subject: Hello from PL/SQL!' || CHR(10) ||
11         'This message sent to you courtesy of the UTL_SMTP package.' ;
12     v_MailConnection := UTL_SMTP.OPEN_CONNECTION(v_MailHost);
13     UTL_SMTP.HELO(v_MailConnection, v_MailHost);
14     UTL_SMTP.MAIL(v_MailConnection, v_FromAddr);
15     UTL_SMTP.RCPT(v_MailConnection, v_ToAddr);
16     UTL_SMTP.DATA(v_MailConnection, v_Message);
17     UTL_SMTP.QUIT(v_MailConnection);
18* END;
19 /
```

PL/SQL procedure successfully completed.

## **UTL\_HTTP**

Permite que um programa PL/SQL aja como um cliente que se comunica com um servidor http.

Exemplo:

```
SQL> CREATE TABLE http_results (
2      sequence_no NUMBER PRIMARY KEY,
3      piece VARCHAR2(2000));
Table created.

SQL>
SQL> DECLARE
2      v_Result UTL_HTTP.HTML_PIECES;
3      v_URL VARCHAR2(100) := 'http://www.oracle.com';
4      v_Proxy VARCHAR2(100); --não iniciado proxy=null
5  BEGIN
6      v_Result := UTL_HTTP.REQUEST_PIECES(v_URL, 10, v_Proxy);
7      FOR v_Count IN 1..10 LOOP
```

```

8      INSERT INTO http_results VALUES (v_Count, v_Result(v_Count));
9  END LOOP;
10 END;
11 /

```

PL/SQL procedure successfully completed.

```

SQL> col piece for a50 truncated
SQL> select sequence_no, piece from http_results where rownum <5;

SEQUENCE_NO PIECE
-----
1 <html lang="en-US">
2 pn.oracle.com/" target="_blank" class=legalese><im
3 input type=hidden name="p_Group" value=4>
4 ><option value="/fi/">Finland</option><option valu

```

## **UTL\_INADDR**

Fornece a capacidade de pesquisar o endereço de IP de um host com base no seu nome ou vice-versa.

Exemplo:

```

SQL> DECLARE
2   v_HostName VARCHAR2(100) := 'www.terra.com.br';
3 BEGIN
4   DBMS_OUTPUT.PUT_LINE('Address of ' || v_HostName || ' is ' || 
5   UTL_INADDR.GET_HOST_ADDRESS(v_HostName));
6   DBMS_OUTPUT.PUT_LINE('Name of local host is ' || 
7   UTL_INADDR.GET_HOST_NAME());
8
9   DBMS_OUTPUT.PUT_LINE('IP 148.87.9.44 is ' || 
10  UTL_INADDR.GET_HOST_NAME('148.87.9.44'));
11
12 END;
13 /

```

Address of www.terra.com.br is 200.176.3.142  
 Name of local host is SERVERXEON  
 IP 148.87.9.44 is bigip-www.us.oracle.com

PL/SQL procedure successfully completed.

## **DBMS\_JOB**

Permite agendar um bloco PL/SQL para ser executado automaticamente em momentos específicos. Necessita dos parâmetros JOB\_QUEUE\_PROCESSES e JOB\_QUEUE\_INTERVAL (obsoleto no Oracle9i).

Exemplo:

```

SQL> CREATE SEQUENCE temp_seq
2      START WITH 1

```

```
3      INCREMENT BY 1;
```

Sequence created.

```
SQL>
SQL> CREATE OR REPLACE PROCEDURE TempInsert AS
  2  BEGIN
  3    INSERT INTO temp_table (num_col, char_col)
  4      VALUES (temp_seq.NEXTVAL,
  5              TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS'));
  6    COMMIT;
  7  END TempInsert;
  8 /
```

Procedure created.

```
SQL>
SQL> VARIABLE v_JobNum NUMBER
SQL> BEGIN
  2  DBMS_JOB.SUBMIT(:v_JobNum, 'TempInsert;', SYSDATE,
  3                     'SYSDATE + (90/(24*60*60))');
  4  COMMIT;
  5 END;
  6 /
```

PL/SQL procedure successfully completed.

```
SQL>
SQL> PRINT v_JobNum
```

```
V_JOBNUM
-----
141
```

```
SQL>
SQL> -- This block will remove the job.
SQL> BEGIN
  2  DBMS_JOB.REMOVE(:v_JobNum);
  3  COMMIT;
  4 END;
  5 /
```

PL/SQL procedure successfully completed.

## DBMS\_LOB

Esse pacote é utilizado com interface para manipular objetos LOBs em PL/SQL.

Exemplo:

```
create table MINHAS_IMAGENS
(
    CODIGO      NUMBER(3),
    IMAGEM_INT BLOB,
    IMAGEM_EXT BFILE,
    TIPO        CHAR(1) default 'I'
);

create sequence seq_imagem;

create directory dir_imagens as 'c:\imagens';

create or replace procedure p_load_imagem(p_arquivo in
varchar2, p_tipo in char default 'I') is
    MeuBlob Blob;
    MeuArq  BFile;
begin
    if p_Tipo = 'I' then --imagens armazenadas no banco
        insert into minhas_imagens(codigo, imagem_int, tipo)
            values (seq_imagem.nextval, empty_blob(), p_Tipo)
        returning imagem_int into MeuBlob;

        MeuArq := bfilename('DIR_IMAGENS', p_arquivo);
        DBMS_LOB.fileopen(MeuArq);
        DBMS_LOB.loadfromfile(dest_lob => MeuBlob, src_lob =>
MeuArq, amount => DBMS_LOB.getlength(MeuArq));
        DBMS_LOB.fileclose(MeuArq);

    elsif p_Tipo = 'E' then --imagens externas
        insert into minhas_imagens(codigo, imagem_ext, tipo)
            values (seq_imagem.nextval, bfilename('DIR_IMAGENS',
p_arquivo), p_Tipo);

    else
        RAISE_APPLICATION_ERROR(-20000,'Tipo inválido');
    end if;

    commit work;
end p_load_imagem;
/
```

This document was created with Win2PDF available at <http://www.win2pdf.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.  
This page will not be added after purchasing Win2PDF.