Introdução à programação PL/SQL

Maio 2009

Marcus William
marcuswlima@gmail.com

Sumário

Parte II - Programação PL/SQL

Parte III - Objetos Procedurais

Parte IV - Apêndices

Apresentação

Este material tem a intenção de mostrar quais são os mínimos conhecimentos necessários para a construção dos primeiros objetos escritos em linguagem PL/SQL como procedures, functons, packages e triggers.

Os livros disponíveis no mercado voltados para programação Oracle, inclusive a documententação oficial, são abrangentes e neles contém uma grande quantidade de informações, que num primeiro momento podem dificultar o aprendizado da linguagem PL/SQL. Neste material, sem dúvida, temos apenas o essencial para a iniciação na programação PL/SQL, para que esta experiência seja rápida e direta.

O leitor deverá ter conhecimentos fundamentados em lógica de programação, e iniciais de banco de dados relacional, tendo em vista que o material mostrará como as estruturas de programação são aplicadas à programação PL/SQL quanto a sintaxe.

O material foi concebido à luz de vários anos de experiência atuando em desenvolvimento para Oracle e apresenta as informações em uma seqüência pensada adequada para o objetivo do material. Faz parte também do material um arquivo simples de texto com todos os exemplos listados na apostila para evitar redigitação dos mesmos.

Organização Trabalho

Parte I

Abordaremos a construção das primeiras instruções SQL, manipulando os dados obtidos através de funções de linhas e funções de grupo. Consultas complexas com várias tabelas serão abordadas, assim como subqueries, manipulação de dados (insert, update, delete) e controle de transação.

Parte II

Iniciaremos a programação, e será apresentado como as estruturas de programação se comportam no PL/SQL, como se dá a manipulação dos famosos cursores PL/SQL e finalizando com tratamento de erros. Todas essas atividades se darão em blocos PL/SQL anônimos que são a unidade básica da programação PL/SQL.

Parte III

Continuando a programação, abordaremos a criação e compilação dos principais objetos procedurais (procedures, function, package, triggers), conhecer as suas especificidades e suas finalidades.

Parte IV

Como apêndice, porém não menos importante, serão mostrados os conceitos para uma correta configuração de um software Oracle Client.

Autor

Marcus William, Tecnólogo em Processamento de Dados, com mais de 10 anos na área de TI, convive há mais de sete anos com produtos Oracle, autodidata no aprendizado da programação PL/SQL, foi coordenador dos programadores PL/SQL em 2000 no DETRAN-PA e atualmente chefia da Divisão de Banco de Dados do TJE-PA.

Perfil Programador PL/SQL

O programador PL/SQL é o profissional da área de TI especializado em criar e manter as construções procedurais de um SGDB ORACLE, estes escritos em linguagem PL/SQL. O PL/SQL não é um produto separado, é sim uma tecnologia integrada ao SGDB Oracle e está disponível desda versão 6.

Uma das principais finalidades da linguagem PL/SQL é a construção de Stored Procedures, que são unidades de manipulação de dados com scopo definido de ações, e estarão disponíveis aos usuários do SGDB segundo uma política de acesso e privilégios customizada. A principal finalidade das Stored Procedures é prover procedimentos armazenados no SGDB de fácil utilização, aliviando assim, a carga de processamento no cliente. Os principais tipos de Stored Procedure se dão na forma de procedure, function e package.

Para se criar ou compilar um objeto no Oracle, o programador deverá submeter ao banco o comando de criação deste objeto, comando este que conterá todos os atributos e inclusive a lógica prevista para o objeto. Se o programador tem o privilégio de criar os seus objetos, ele o fará, caso contrário essa tarefa é responsabilidade do DBA (DataBase Administrator). Uma forma de facilitar este trabalho é registrar o comando de criação em scripts para fácil recriação quando necessário.

É comum escutar “Vamos alterar a procedure”. A rigor um objeto procedural não é alterado e sim recriado ou recompilado. Se a “alteração” consiste em apenas na adição de uma linha, então todo o comando de compilação, adicionado da nova linha, deverá ser reapreciado/compilado pelo Oracle.

Existem alguns excelentes aplicativos IDE (Integrated Development Environment) que auxiliam a gerência dos scripts de construção de objetos procedurais.

Produto	Fabricante	URL
PL/SQL Developer	Allroundautomations	www.allroundautomations.com/plsqldev.html
SQL Developer	Oracle	www.oracle.com/technology/products/database/sql_developer/index.html
SQL Navigator	Quest	www.quest.com/sql_navigator

Obs.:O programa PL/SQL Developer é de prefrência do autor, não por achar mais eficiente e sim por estar acostumado.

Arquitetura Banco de Dados Oracle

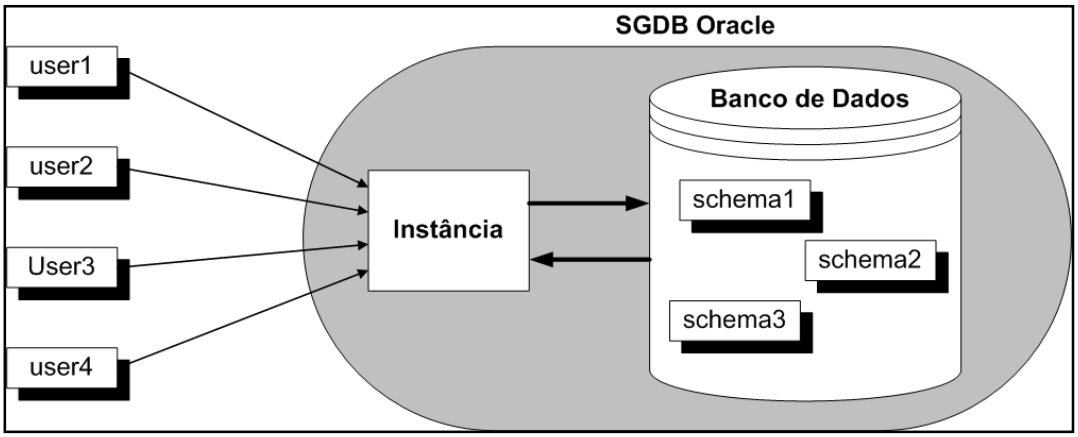
É necessário para o programador PL/SQL entender a estrutura de armazenamento de objetos disponíveis no Oracle a fim de executar suas atividades.

Vamos começar a entender a diferença entre **Banco de Dados** e **Instância**. Em um SGDB Oracle, a rigor, o termo Banco de Dados se aplica aos arquivos físicos que integram o mesmo. Assim sendo, em uma conversa com o suporte Oracle o atendente entenderá “banco de dados” como os arquivos formadores do Oracle.

Instância é o conjunto de processos de memória armazenados em forma volátil, que são responsáveis pela gerência de todas as atividades incidente e consequente no banco de dados (arquivos físicos). Quando um usuário cria uma conexão no banco, na realidade ele está se conectando a uma instância, que lhe servirá de ponte de trabalho ao banco de dados.

O **schema** é uma representação lógica organizacional que aglutina uma porção de objetos do banco de dados. Quando criamos um usuário estamos também criando um schema inicialmente vazio. Em geral, é convencionado que

um ou mais schemas contemplem os objetos de produção de um determinado sistema e que os demais usuários criados sejam utilizados apenas para fins de conexão e uso dos objetos dos schemas de produção. Para que isso aconteça, uma política de privilégios deverá ser implementada.



Toda essa arquitetura é de responsabilidade do Administrador de Banco de Dados - **DBA**. E no dia-a-dia, estas definições se confundem. Não é difícil encontrar alguém usando o termo banco de dados para uma instância ou um schema. Isso não é um pecado. O que se deve ter em mente é a real diferença entre estes conceitos.

Um programador PL/SQL deverá receber as seguintes informações do DBA para dar início às suas atividades:

- host ou endereço IP servidor de banco de dados;
- usuário de banco de dados;
- senha de acesso;
- identificação do serviço ou nome do banco de dados.

Com estas informações, o programador deverá saber configurar o software Oracle Client instalado na sua estação de trabalho a fim de acessar o banco de dados informado pelo DBA. O usuário de banco de dados informado deverá ter privilégios de compilação e recompilação de objetos procedurais seguindo a politica de privilégios imposta ao banco de dados. É boa prática, o DBA informar um banco de desenvolvimento ao invéz de um banco de produção.

Obs:. No apêndice A são apresentados os aspectos mais profundos quanto à configuração de clientes Oracle.

Parte I - Comandos SQL

1 Comandos SQL

Structured Query Language (SQL) é um conjunto de instruções com as quais os programas e usuários acessam as informações dos bancos de dados relacionais, como o Oracle. As aplicações permitem aos usuários manipulação nos dados sem o efetivo uso de instruções SQL, no entanto essas aplicações seguramente usam SQL para executar as requisições dos usuários.

O Dr. E.F. Codd em Junho de 1970 publicou o estudo "A Relational Model of Data for Large Shared Data Banks". Este modelo de banco proposto é agora aceito como definitivo pelos grandes softwares gerenciadores de banco de dados (SGDB). A linguagem SEQUEL (Structured English Query Language) foi desenvolvida pela IBM aproveitando o modelo de Codd, em seguida se transformou SQL. Em 1979 a empresa Rational Software (atualmente Oracle Corporation) lançou a primeira implementação de banco de dados relacional viável comercialmente.

Existem comitês que trabalham no sentido de padronizar como a indústria de software irá considerar o SQL em termos de sintaxe. Os principais comitês são ANSI (American National Standards Institute) e o ISO/IEC (International Organization for Standardization/ International Electrotechnical Commission). A Oracle se esforça para estar de acordo com os padrões sugeridos e participa ativamente deste comitês de padrão SQL. As versões Oracle10g(Enterprise, Standard, Express, Standard One) foram projetadas para estarem em conformidade ao padão ANSI-2003. Verifique a tabela de confirmidade em :
http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/ap_standard_sql.htm#g21788

Obs: Serão apresentados apenas os comandos SQL de consulta e manipulação de dados, os comandos de manutenção das estruturas (CREATE , ALTER) não pertencem ao escopo deste curso.

2 Implementando consultas SQL simples

Para extrairmos e consultarmos dados das tabelas de um banco relacional, usamos a instrução SELECT, que faz uma pesquisa nas estruturas relacionais do banco de dados e retorna valores na forma de linhas e colunas. Uma consulta SQL pode também ser nominada simplesmente como *query*.

2.1 Construção Básica

```
SELECT {*|coluna1 [apelido], coluna2 [apelido]}
  FROM tabela
```

Figura 2.1 – Sintaxe Select

Diretrizes:

- SQL não fazem distinção entre maiúsculas e minúsculas;
- SQL podem estar em uma ou mais linhas;
- As palavras reservadas não podem ser abreviadas;
- Normalmente as clausulas são colocadas em linhas diferentes (boa prática);
- Guias e identações são usadas para aumentar a legibilidade.

2.2 Colunas

Quanto às colunas temos duas alternativas a primeira é selecionar todas as colunas em uma única consulta SQL, a outra mais elegante é selecionar apenas as colunas interessantes, cada coluna pode ter um *alias* associado. Alis são muito úteis para cálculos matemáticos.

```
SQL>select *
  2   from departments
  3  /

DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
           10 Administration                         200        1700
           20 Marketing                              201        1800


SQL>select first_name, salary salario_mensal
  2  from employees
  3  /

FIRST_NAME           SALARIO_MENSAL
-------------------- --------------
Steven                        26400
Steven                        24000
```

Exemplo 2.1 –Select Colunas

2.3 Expressões Aritméticas (Operadores aritméticos)

É possível criar expressões com colunas do tipo *NUMBER* e *DATE* usando operadores aritméticos, na seguinte procedência **(*, /, +, -).**

```
SQL>select first_name, salary , (salary * 20) / 100 porcentagem
  2  from employees
  3  /

FIRST_NAME              SALARY PORCENTAGEM
-------------------- ---------- -----------
Steven                    26400        5280
Neena                     17000        3400
```

Exemplo 2.2 – Expressões

Para adicionarmos 1(um) dia ao valor de um coluna do tipo *DATE* devemos utilizar o operador **+** da seguinte forma *valor_data+1*, e para adicionar uma hora a seguinte sintaxe *valor_data+(1/24)*

```
SQL>select first_name, hire_date, hire_date+365 data_exame
  2  from employees
  3  /

FIRST_NAME           HIRE_DATE   DATA_EXAME
-------------------- ----------- -----------
Steven               17-JUN-1987 16-JUN-1988
Steven               17-JUN-1987 16-JUN-1988
```

Exemplo 2.3 – Expressões Data

Obs.: Um valor nulo não é o mesmo que zero ou um espaço em branco. *****QUALQUER VALOR OPERACIONADO COM NULO RESULTA NULO**.*** *Este problema é remediado pela função NVL ver(4.5.1)*

2.4 Operador de Concatenação

No Oracle o operador **||** executa contatenação de dois strings de caracteres, e poderá ser usado nas cláusulas SELECT, WHER e ORDER BY. Utilizada na cláusula SELECT de uma consulta SQL, resultará em um campo do tipo caracter.

```
SQL>select first_name||' '||last_name nome_completo
  2  from employees
  3  /

NOME_COMPLETO
----------------------------------------------
Steven King
Neena Kochhar
```

Exemplo 2.4 – Concatenação

3 Restringindo e ordenando dados

3.1 Cláusula WHERE

É possível restringir as linhas retornadas da consulta SQL utilizando a cláusula WHERE. As linhas da tabela só estarão no retorno da consulta SQL se a condição da cláusula WHERE aplicada à linha for atendida com sucesso. A cláusula WHERE é seguida de uma expressão condicional. O Oracle aplica condição às linhas da(s) tabela(s) de cláusula FROM. Cada aplicação de linha gera um valor boolean. As linhas que geram valores TRUE formarão o *dataset* de retorno

```
SQL>select first_name,job_id, department_id
  2   from employees
  3  where job_id = 'ST_CLERK'
  4  /

FIRST_NAME           JOB_ID     DEPARTMENT_ID
-------------------- ---------- -------------
Trenna               ST_CLERK              50
Curtis               ST_CLERK              50
```

Exemplo 3.1 – Cláusula WHERE

3.2 Operadores de comparação

Operadores de comparação comparam dois valores ou expressões e retornando um resultado de valor Boolean. A tabela 3.1 ilustra os operadores de comparação mais recorrentes. Geralmente os operadores são usados na cláusula WHERE.

Tabela 3.1. Operadores de Comparação.

Operador	**Significado**	**Exemplo**
=	Igual a	`SELECT * FROM EMPLOYEES WHERE LAST_NAME='SCOOT'`
>	Maior que	`SELECT LAST_NAME FROM EMPLOYEES WHERE SALARY > 2000`
>=	Maior ou igual a	`SELECT FIRST_NAME,SALARY FROM EMPLOYEES WHERE SALARY >= 2000`
<	Menor que	`SELECT LAST_NAME FROM EMPLOYEES WHERE SALARY < 2000`
<=	Menor ou igual a	`SELECT FIRST_NAME,SALARY FROM EMPLOYEES WHERE SALARY <= 2000`
<>, !=, ^=	Diferente de	`SELECT * FROM EMPLOYEES WHERE FIRST_NAME!='TIGER'`
BETWEEN ...AND...	Entre dois valores (inclusive)	`SELECT LAST_NAME FROM EMPLOYEES WHERE SALARY BETWEEN 5000 AND 10000`
IN(lista)	Vincula qualquer um de uma lista valores	`SELECT LAST_NAME,SALARY FROM EMPLOYEES WHERE FIRST_NAME IN ('Steven','Peter','Ellen','Marcus')`
LIKE	Vincula um padrão de caracter	`SELECT LAST_NAME FROM EMPLOYEES WHERE FIRST_NAME LIKE 'C%'`
IS NULL	É um valor nulo	`SELECT LAST_NAME||' '||FIRST_NAME FROM EMPLOYEES WHERE COMMISSION_PCT IS NULL`

3.3 Operadores Lógicos

Operadores lógicos são usados para combinar ou alterar o resultado de uma ou mais comparações (ver 3.2). O produto desta operação será um valor booleano e no escopo deste curso será utilizado para determinar quais linhas estarão no resultado da consulta.

Tabela 3.2. Operadores Lógicos

Operador	Significado	Exemplo

NOT	Retorna TRUE se a condição seguinte for FALSE	`SELECT *`
		`  FROM EMPLOYEES`
		`WHERE NOT (SALARY < 10000)`

AND	Retorna TRUE se a condição de componentes forem TRUE	`SELECT *`
		`  FROM EMPLOYEES`
		`WHERE SALARY > 10000`
		`  AND LAST_NAME LIKE 'H%'`

OR	Retorna TRUE *se cada* condição de componentes forem TRUE	`SELECT FIRST_NAME,SALARY`
		`  FROM EMPLOYEES`
		`WHERE LAST_NAME = 'THOMAS'`
		`  OR LAST_NAME = 'JACOB'`

3.4 Precedência

Múltiplos operadores podem formar uma expressão válida. Os operadores com maior precedência serão avaliados antes dos operadores de menor precedência, seguindo a tabela 3.3.

Tabela 3.3. Precedência de Operadores.

Operador	Propósito
*, /	Multiplicação, divisão
+, -, ||	Adição, subtração, concatenação
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparação
NOT	Negação
AND	Conjunção
OR	Disjunção

Obs.: Parênteses poderão ser utilizados para alterar a precedência de avaliação.

3.5 Order By

A ordem das linhas retornadas em um resultado de consulta é indefinida. A cláusula ORDER BY pode ser utilizada para classificar as linhas. A cláusula ORDER BY deve ser colocada após a cláusula WHERE. Por definição o resultado será exibido de forma crescente e através da palavra-chave DESC a ordem é invertida. O resultado pode também ser ordenado por várias colunas

```
SQL>select first_name,job_id, department_id
  2  from employees
  3  order by department_id, salary DESC
  4  /

FIRST_NAME           JOB_ID     DEPARTMENT_ID
-------------------- ---------- -------------
Jennifer             AD_ASST               10
Michael              MK_MAN                20
```

Exemplo 3.2 – Cláusula Order BY

Obs.: Sem o uso da cláucula ORDER BY o resultado da consulta incerto.

4 Funções de uma linha

Em instruções SQL as funções de uma linha são usadas principalmente para manipular os valores que serão apresentados. Estas aceitam um ou mais argumentos e retornam um ou único valor para cada linha do dataset gerado pela consulta. Um argumento pode ser um dos seguintes itens, constantes, valor variável, nome de coluna, expressão.

Quanto aos recursos de funções de uma linha

- Atua em cada linha retornada da consulta.
- Retorna um resultado por linha.
- Podem receber zero, um ou mais argumentos.
- Podem ser usados em cláusulas SELECT, WHERE, ORDER BY.

Obs.: Funções de Linha podem ser usadas em instruções DML´s ver(8).

4.1 Funções de Caracter

Tabela 3.4. Funções Caracter

Função	Objetivo
`Lower(string)`	Converte valores de caracteres alfabéticos para letras maiúsculas
`Upper(string)`	Converte valores de caracteres alfabéticos para letras maiúsculas
`Initcap(string)`	Converte os valores de caracteres alfabético para usar maiúscula na primeira letra de cada palavra e todas as outras letras em minúsculas
`Concat(string1, string2)`	Contatena o primeiro valor com o segundo valor. Equivalente ao operador ||
`Substr(string,m,[n])`	Retorna caracteres específicos a partir do valor de caractere começando na posição m, até n caracteres depois (Se m for nagativo, a conta inicia no final do valor de caracterres. Se n for omitido, são retornados todos caracteres até, o final da string)
`Length(char)`	Retorna o número de caracteres
`Instr(string , substring)`	Retorna a posição numária do substring
`lpad(string,n,'string')`	Preenche o valor de caracteres justificando à direita a uma largura total de n posições de caracter

Tabela 3.4. Funções Caracter - Resultados

Função	Resutado
`LOWER('SQL Course')`	`sql course`
`UPPER('SQL Course')`	`SQL COURSE`
`INITCAP('SQL Course')`	`Sql Course`
`CONCAT('Hello','World')`	`HelloWorld`
`SUBSTR('HelloWorld',1,5)`	`Hello`
`LENGTH('HelloWorld')`	`10`
`INSTR('HelloWorld','W')`	`6`
`LPAD(salary,10,'*')`	`*****24000`
`RPAD(salary,10,'*')`	`24000*****`

```
SQL>SELECT employee_id, CONCAT(first_name, last_name) NAME,
  2         job_id, LENGTH (last_name),
  3         INSTR(last_name, 'a') "Contains 'a'?"
  4  FROM   employees
  5  /

EMPLOYEE_ID NAME                 JOB_ID     LENGTH(LAST_NAME) Contains 'a'?
----------- -------------------- ---------- ----------------- -------------
        100 StevenKing           AD_PRES                    4             0
        101 NeenaKochhar         AD_VP                      7             6
        102 LexDe Haan           AD_VP                      7             5
```

Exemplo 4.1 – Funções de linha

4.2 Funções Numéricas

Tabela 3.4. Funções Numéricas

Função	Objetivo	Argumento(s)	Resultado
ROUND	Arredonda valor para determinado decimal	(45.926,2)	45.93
TRUNC	Trunca valor para determinado decimal	(45.926,2)	45.92
MOD	Retorna o restante da divisão	(1600,300)	100

```
SQL>SELECT ROUND(45.923,2), ROUND(45.923,0),
  2         ROUND(45.923,-1)
  3  FROM   DUAL;

ROUND(45.923,2) ROUND(45.923,0) ROUND(45.923,-1)
--------------- --------------- ----------------
          45,92              46               50

1 linha selecionada.

SQL>SELECT  TRUNC(45.923,2), TRUNC(45.923),
  2          TRUNC(45.923,-2)
  3  FROM   DUAL;

TRUNC(45.923,2) TRUNC(45.923) TRUNC(45.923,-2)
--------------- ------------- ----------------
          45,92            45                0

1 linha selecionada.

SQL>SELECT last_name, salary, MOD(salary, 5000)
  2  FROM   employees
  3  WHERE  job_id = 'SA_REP';

LAST_NAME                 SALARY MOD(SALARY,5000)
------------------------- ---------- ---------------
Abel                           11000            1000
Taylor                          8600            3600
Grant                           7000            2000

3 linhas selecionadas.
```

Exemplo 4.2 – Funções Numéricas

Tabela DUAL

Em todos os bancos de dados Oracle existe uma tabela chamada DUAL, aparentemente irrelevante. No entento, ela é útil quando se deseja retornar um valor pontual, sendo principalmente usada para a execução de instruções SQL que não necessitam de tabela base.

4.3 Funções de Data

Tabela 3.4. Funções Data

Função	Objetivo	Argumento(s)	Resultado
Months_between	Número de meses entre duas datas	('01-SET-95','11-JAN-94')	19.674194
add_months	Adiciona meses de calendário para a data	('11-JAN-94',6)	'11-JUL-94'
last_day	Último dia do mês	(01-SET-95')	'30-SET-95'
Round	Data de arredondamento	(25-JUL-95','mm') (25-JUL-95','yy')	'01-AGO-95' '01-JAN-95'
Trunc	Data para truncada	(25-JUL-95','mm') (25-JUL-95','yy')	'01-JUL-95' '01-JAN-95'

Devemos saber informar ao ambiente de trabalho como todas essas informações serão apresentadas, segundo as nossas necessidades. Uma das formas é através dos seguintes comandos.

```
ALTER SESSION SET NLS_DATE_LANGUAGE='PORTUGUESE';
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';
```

Figura 4.1 – Configurando o ambiente

Para manipular valores do tipo DATE em um formato diferente do padrão estabelecido pelo ambiente de trabalho, se faz necessário o uso da função TO_CHAR com os elementos de Format Model corretos.

Para o curso será utilizado o Format Model 'DD-MON-YYYY'. Verifique em http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/sql_elements004.htm#i34924 todos os Format Model disponíveis valores do tipo DATE.

4.4 Funções de Conversão

4.4.1 TO_CHAR(X[,'format_model'])

Onde X é um valor do tipo DATE ou NUMBER e 'format_model' é uma string que descreve o formato de como o argumento X será apresentado. Existem formatações específicas para o tipo DATE e outras para o tipo NUMBER.

Tabela 3.5. Principais Format Model para tipo DATE

Elemento	**Descrição**
yyyy	Ano com 4 dígitos
mm	Mês (01-12)
dd	Dia do mês (1-31)
hh24	Hora do dia (0-23)
mi	Minuto(0-59)
ss	Segundo(0-59)
day	dia semana por extenso
mon	Abreviação do mês
month	Nome extenso do mês

```
SQL>SELECT
  2      SYSDATE data1,
  3      TO_CHAR(SYSDATE,'DD/MM/YYYY hh24:mi:ss') data2,
  4      TO_CHAR(SYSDATE,'DD/Mon/YYYY') data3,
  5      TO_CHAR(SYSDATE,'fm"Belém, "DD" de "month" de "yyyy') data4
  6  FROM DUAL
  7  /

DATA1       DATA2               DATA3       DATA4
----------- ------------------- ----------- ------------------------------
27-OUT-2006 27/10/2006 19:58:09 27/Out/2006 Belém, 27 de outubro de 2006

1 linha selecionada.
```

Exemplo 4.3 – Funções TO_CHAR

Diretrizes para datas:

- O Format Model deve estar entre aspas simples e fazer distinção entre maiúsculas e minúsculas.
- O Format Model e o valor do argumento devem estar separados por vírgula.
- Para remover os espaços em branco ou suprimir os zeros à esquerda, use o modo de preenchimento *fm*.
- Qualquer string pode ser adicionada ao Format Model delimitado por aspas duplas.

Quando aos valores do tipo NUMBER, se necessário, deveremos informar ao ambiente o separador de milhar e decimal do padrão brasileiro, através do comando:

```
ALTER SESSION SET SET NLS_NUMERIC_CHARACTERS=',.';
```

Figura 4.2 – Configurando o ambiente

Tabela 3.5. Principais Format Model para tipo NUMBER

Elemento	Descrição
9	Represnta um número
0	Força que um Zero seja mostrado
G	Separador de Milhar
D	Separador de decimal
RN	Número Romano

```
SQL>select to_char(1234.5),
  2        to_char(1234.5,'99999.00'),
  3        to_char(1234.5,'99g999d00'),
  4        to_char(1234.5,'RN')
  5  from dual
  6  /

TO_CHA TO_CHAR(1 TO_CHAR(12 TO_CHAR(1234.5,
------ --------- ---------- ---------------
1234,5   1234.50   1.234,50         MCCXXXV

1 linha selecionada.
```

Exemplo 4.4 – Funções TO_CHAR

4.4.2 TO_DATE('string','formatação')

Onde 'string' é a informação que se deseja transformar para valor do tipo DATE e 'formatação' é o FORMAT MODEL que indica com como o Oracle deverá reconhecer a string apresentada no primeiro parâmetro.

```
SQL>select FIRST_NAME,HIRE_DATE
  2  from employees
  3  where HIRE_DATE = to_date('07/06/1994','dd/mm/yyyy')
  4  /

FIRST_NAME           HIRE_DATE
-------------------- -----------
Shelley              07-JUN-1994
William              07-JUN-1994

2 linhas selecionadas.

SQL>insert into job_history
  2    (EMPLOYEE_ID
  3    ,START_DATE
  4    ,END_DATE
  5    ,JOB_ID
  6    ,DEPARTMENT_ID)
  7  values
  8    (174
  9    ,to_date('05/novembro/1974','dd/month/yyyy')
 10    ,to_date('10abr1978','ddmonyyyy')
 11    ,'AD_VP'
 12    ,80)
 13  /

1 linha criada.
```

Exemplo 4.5 – Funções TO_DATE

*Obs.:Para conversão de dados, existe também o **TO_NUMBER**, no entando é mais prático se valer da conversão implícita executada pelo Oracle. Tente usar o comando **SELECT '1' + 1 FROM DUAL***

4.5 Expressões Condicionais Gerais

4.5.1 NVL

Converte valores nulos para um valor real, a função está apta a trabalhar com os tipos de dados DATE, CHARACTER e NUMBER. Os parâmetros informados devem ser do mesmo tipo de dado.

```
SQL>select salary, commission_pct, salary+commission_pct, salary+nvl(commission_pct,0)
  2  from employees
  3  where EMPLOYEE_ID in (144,149)
  4  /

    SALARY COMMISSION_PCT SALARY+COMMISSION_PCT SALARY+NVL(COMMISSION_PCT,0)
---------- -------------- --------------------- ---------------------------
     10500             ,2               10500,2                     10500,2
      2500                                                           2500

2 linhas selecionadas.
```

Exemplo 4.6 – Funções NVL

4.5.2 DECODE

```
DECODE(expr
           ,search1,result1
          [,search2,result2,...,
           ,searchN,resulN]
          [,default])
```

Figura 4.3 – DECODE

A expressão DECODE trabalha de um modo similar à lógica IF-THEN-ELSE. A expressão DECODE compara *expr* a todos *search* um por vez. Se *expr* é igual to *search* então o Oracle retorna o *result* correspondente. Se não encontrar nenhuma correspendência então o Oracle retorna *default*. Neste caso, se *defult* estiver omitida o Oracle retornará null

```
SQL>select
  2      job_id,
  3      salary,
  4      DECODE(job_id,
  5                  'IT_PROG',salary*1.1,
  6                  'ST_MAN',salary*1.2,
  7                  'MK_REP',salary*1.3,
  8            salary) realuste
  9  from employees
 10  /
```

Exemplo 4.7 – DECODE

4.5.3 CASE

```
CASE expr WHEN compare1 THEN resul1
         [WHEN compare2 THEN resul2
          WHEN compar2N THEN resulN
          ELSE resulElse
END
```

Figura 4.4 – CASE

Em uma expressão *CASE*, o Oracle pesquisa a partir da primeira cláusula WHEN no qual expr é igual a *compare* e retorna *result*. Se nenhuma das cláusulas *WHEN* for selecionada e uma cláusula *ELSE* existir, então o Oracle retornará *resultElse*.

```
SQL>select
  2      job_id,
  3      salary,
  4      CASE job_id
  5         WHEN 'IT_PROG' THEN salary*1.1
  6         WHEN 'ST_MAN'  THEN salary*1.2
  7         WHEN 'MK_REP'  THEN salary*1.3
  8         ELSE salary
  9      END
 10  from employees
 11  /
```

Exemplo 4.8 – CASE

Obs: Dê preferência ao comando CASE ao invés do DECODE. O comando CASE é mais poderoso

Obs: No link http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/functions.htm#i1482196 encontra-se a relação de todas as funções de uma linha disponíveis nas versões Oracle10g

Obs: No capítuto 15 será apresentado como criar e programar as suas própias funções

5 Mostrando dados de várias tabelas

Existem momentos em que faz necessário o uso de dados a partir de mais de uma tabela, neste caso usaremos condições especiais chamadas de JOIN's. As linhas de uma tabela podem ser relacionadas às linhas de outra tabela de acordo com os valores comuns existentes nas colunas correspondentes, que em geral são colunas de chave primária e estrangeira.

```
SELECT  table1.column, table2.column
FROM    table1, table2
WHERE   table1.column1 = table2.column2;
```

Figura 5.1 – Join

Diretrizes para Joins:

- Ao se escrever uma instrução SELECT que combine mais de uma tabela, é interessante deixar claro a que tabela o campo pertence, posicionando o nome da tabela antes do nome do campo.
- Não é obrigatório o posicionamento o nome da tabela antes do nome do campo, porém, se uma mesma coluna pertence a mais de uma tabela, deve-se prefixar a coluna com o nome da tabela.
- Para combinar *n* tabelas se fez necessário no mínimo *n-1* condições de JOIN

```
SQL>SELECT employees.employee_id, employees.last_name,
  2        employees.department_id, departments.department_id,
  3        departments.location_id
  4   FROM employees, departments
  5  WHERE employees.department_id = departments.department_id;

EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID DEPARTMENT_ID LOCATION_ID
----------- ------------------------- ------------- ------------- -----------
       1000 King                                 90            90        1700
        100 King                                 90            90        1700
```

Exemplo 5.1 – Join

É possível também adicionar condições de filtros às condições de JOIN para restringe mais ainda as linhas obtidas. No exemplo abaixo, serão apresentados o nome e o departamento do funcionário "Matos".

```
SQL>SELECT last_name, employees.department_id,
  2           department_name
  3     FROM   employees, departments
  4     WHERE  employees.department_id = departments.department_id
  5     AND    last_name = 'Matos';

LAST_NAME                 DEPARTMENT_ID DEPARTMENT_NAME
------------------------- ------------- ------------------------------
Matos                               50 Shipping

1 linha selecionada.
```

Exemplo 5.2 – Join / AND

5.1 Produto Cartesiano

Quando um JOIN for completamente inválido ou omitido, o resultado da consulta SQL será um produto cartesiano no qual serão exibidas todas combinações de linhas de todas as tabelas envolvidas na consulta. O Produto cartesiano tende a gerar um grande número de linhas e seu resultado raramente é útil. Apresentamos o produto cartesiano aqui por finalidades didáticas.

```
SQL>SELECT last_name, department_name dept_name
  2    FROM employees, departments;

LAST_NAME                 DEPT_NAME
------------------------- ------------------------------
King                      Administration
King                      Administration
Kochhar                   Administration
....
Higgins                   Contracting
Gietz                     Contracting

168 linhas selecionadas.
```

Exemplo 5.3 – Produto Cartesiano

5.2 Alias de Tabela

Para qualificar as colunas é possível utilizar *alias* de tabela ao invez do nome da tabela. Assim como os *alias* de coluna dão outro nome à coluna, os *alias* de tabela tem a mesma função. Os alias de tabela são definidos na cláusula FROM. O nome da tabela é especificado totalmente seguido do seu alias.

```
SQL>SELECT e.employee_id, e.last_name, e.department_id,
  2        d.department_id, d.location_id
  3   FROM employees e , departments d
  4  WHERE e.department_id = d.department_id;

EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID DEPARTMENT_ID LOCATION_ID
----------- ------------------------- ------------- ------------- -----------
        100 King                                 90            90        1700
        101 Kochhar                              90            90        1700
```

Exemplo 5.4 – Alias tabela

Na instrução SQL do exemplo 5.4 foram definidos os alias “e” para a tabela *employees* e alias “d” para a tabela *departments*.

Obs: Perceba quanto o uso de alias de tabela trás legibilidade à construção de consultas SQL comparando o exemplo 5.4 e o exemplo 5.1

```
SQL>SELECT
  2      e.last_name,
  3      d.department_name,
  4      l.city
  5  FROM
  6      employees   e,
  7      departments d,
  8      locations   l
  9  WHERE
 10      e.department_id = d.department_id
 11  AND d.location_id = l.location_id;

LAST_NAME                 DEPARTMENT_NAME                CITY
------------------------- ------------------------------ --------------------
King                      Executive                      Seattle
Kochhar                   Executive                      Seattle
De Haan                   Executive                      Seattle
Hunold                    IT                             Southlake
```

Exemplo 5.5 – Join em mais de uma tabela.

Diretrizes para Joins:

- Os alias de tabela não ultrapassam 30 posições;
- um alias de tabela poderá substituir o nome da tabela em todas as cláusulas do SQL.
- os alias devem ser sugestivos. Não utilizem algo com T1, T2, T3,...
- palavras reservadas não podem ser utilizadas como alias nenhum. Algo como DESC alusivo a descrição.

5.3 Outer-JOIN

Se a linha não satisfaz a condição da clausula WHERE, não aparecerá no resultado do Select. Nestas condições, estas linhas poderão aparecer se operador de *outer join* for utilizado no JOIN. O operador **(+)** deverá ser posicionado no "lado" join onde a informação é deficiente.

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id(+)
```

Exemplo 5.6 – OUTER JOIN

5.4 Self-JOIN

As vezes, é necessário a execução do um join de uma tabela com ela mesma. Desta forma a tabela aparecerá duas vezes na cláusula FROM e na cláusula WHERE existirá a restrição referente ao self-join. Neste caso o uso de alias de tabela é imperativo.

```
SQL>SELECT worker.last_name || ' works for ' || manager.last_name
  2  FROM   employees worker, employees manager
  3  WHERE  worker.manager_id = manager.employee_id ;

WORKER.LAST_NAME||'WORKSFOR'||MANAGER.LAST_NAME
-----------------------------------------------------------
Kochhar works for King
De Haan works for King
Hunold works for De Haan
```

Exemplo 5.7 – Self JOIN

6 Funções de grupo

De modo diferente das funções de uma única linha, as funções de grupo operam em conjunto de linhas para fornecer um resultado por grupo. Esses conjuntos podem ser uma tabela inteira ou a mesma dividida em grupos menores.

Tabela 3.5. Funções de Grupo.

Função	Descrição
AVG	Valor médio
COUNT	Número de linhas
MAX	Valor Máximo
MIN	Valor Mínimo
SUM	Soma de Valores

```
SQL>
SQL>SELECT AVG(salary), MAX(salary), MIN(salary), SUM(salary)
  2  FROM   employees
  3  WHERE  job_id LIKE '%REP%';

AVG(SALARY) MAX(SALARY) MIN(SALARY) SUM(SALARY)
----------- ----------- ----------- -----------
       8150       11000        6000       32600
SQL>
SQL>SELECT MIN(hire_date), MAX(hire_date)
  2  FROM   employees;

MIN(HIRE_DA MAX(HIRE_DA
----------- -----------
17-JUN-1987 29-JAN-2000
SQL>
SQL>SELECT COUNT(*)
  2  FROM   employees
  3  WHERE  department_id = 50;

  COUNT(*)
----------
         5
SQL>
SQL>SELECT COUNT(commission_pct)
  2  FROM   employees;

COUNT(COMMISSION_PCT)
---------------------
                    4
```

Exemplo 6.1 – Group by Tabela Inteira

É atravéz da cláusula GROUP BY que dividimos as linhas de uma tabela em grupo menores. Em seguida, poderá ser aplicado a esses grupos formados as funções de grupo, gerando assim informações sumárias para cada grupo.

Primeiramente deve ser determinada a identificação do grupo. A identificação do grupo pode ser uma coluna, várias colunas, uma expressão usando colunas ou várias expressões usando colunas. O Oracle considerará no grupo todas as linhas que atendenrem a cláusula WHERE caso esta exista, e então será aplicada a função de grupo ao grupo caso exista.

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id ;

SELECT   department_id dept_id, job_id, SUM(salary)
FROM     employees
GROUP BY department_id, job_id ;
```

Exemplo 6.2 – Group by criando grupos

Na figura 6.2 tempos 3(três) exemplos de consulta utilizando funções de grupo. Na primeira o resultado será agrupado pelo valor do campo *department_id*, e a cada grupo será aferida a média aritmética do campo *salary*. A quantidade de

linhas retornada no primeiro exemplo será em função da quantidade de valores existentes na coluna *department_id*. No segundo exemplo apresentamos um grupo mais sofisticado com dois campos (*department_id , job_id*) e a estes grupos será aplicado a somatória do campo *salary.*

Diretrizes para Joins:

- Usando a cláusula WHERE, linhas serão eliminadas antes de serem organizadas me grupo.
- Não é permitido o uso de alias na cláusula GROUP BY .
- Quando se deseja um campo esteja no retorno do SQL este deverá estar na cláusula GROUP BY.
- É possível criar agrupamentos de mais de um campo.
- Funções de grupo não devem ser utilizadas na cláusula WHERE e sim na cláusula HAVING.

Para se excluir um grupo inteiro criado pela cláusula GROUP BY, deveremos usar a cláusula HAVING, que executa um trabalho parecido com a cláusula WHERE que elimina as linhas, este, no entanto, elimina grupos.

```
SQL>SELECT   department_id, MAX(salary)
  2  FROM     employees
  3  GROUP BY department_id
  4  HAVING   MAX(salary)>10000 ;

DEPARTMENT_ID MAX(SALARY)
------------- -----------
           90       26400
           20       13000
           80       11000
SQL>
SQL>
SQL>SELECT   job_id, SUM(salary) PAYROLL
  2  FROM     employees
  3  WHERE    job_id NOT LIKE '%REP%'
  4  GROUP BY job_id
  5  HAVING   SUM(salary) > 13000
  6  ORDER BY SUM(salary);

JOB_ID        PAYROLL
---------- ----------
IT_PROG         19200
AD_PRES         50400
```

Exemplo 6.3 – HAVING

7 Subconsultas

Uma subconsulta é uma instrução SELECT incorporada a outra instrução SELECT. O uso de subconsultas torna possível a construção de sofisticadas instruções e são úteis quando precisamos selecionar linhas de uma tabela com uma condição que dependa dos dados na própria tabela. Também podem ser chamadas de subqueries ou consulta interna.

Tabela 7.1. Tipos de Subconsulltas

Tipo	Descrição	Operadores
Subquerie de uma única linha	O resultado da consulta interna retorna apenas uma linha	=, >, >=, <, <=, <>
Subquerie de várias linhas	O resultado da consulta interna retorna várias linhas	IN – Igual a qualquer membro da lista ANY – Compare o valor de cada valor retornado pela subconsulta. ALL – Compare o valor a todo valor retornado pela subconsulta.

```
SQL>SELECT last_name
  2  FROM   employees
  3  WHERE  salary >
  4                (SELECT salary
  5                 FROM   employees
  6                 WHERE  last_name = 'Abel')
  7  /

LAST_NAME
-------------------------
King
Kochhar
De Haan

SQL>SELECT last_name, job_id, salary
  2  FROM   employees
  3  WHERE  job_id =
  4                  (SELECT job_id
  5                   FROM   employees
  6                   WHERE  employee_id = 141)
  7  AND    salary >
  8                  (SELECT salary
  9                   FROM   employees
 10                   WHERE  employee_id = 143);

LAST_NAME                 JOB_ID         SALARY
------------------------- ---------- ----------
Rajs                      ST_CLERK         3500
Davies                    ST_CLERK         3100
```

Exemplo 7.1 – Subquerie

7.1 Operador IN

```
SQL>SELECT last_name, salary, department_id
  2     FROM   employees
  3     WHERE  salary IN (SELECT   MIN(salary)
  4                       FROM     employees
  5                       GROUP BY department_id);

LAST_NAME                     SALARY DEPARTMENT_ID
------------------------- ---------- -------------
Grant                           7000
De Haan                        17000            90
Kochhar                        17000            90

SQL>SELECT last_name, salary, department_id
  2      FROM   employees
  3      WHERE  salary IN (7000,17000,6000,8300,2500,8600,4200,4400)
  4  /

LAST_NAME                     SALARY DEPARTMENT_ID
------------------------- ---------- -------------
Kochhar                        17000            90
De Haan                        17000            90
```

Exemplo 7.2 – Operador IN

7.2 Operador ANY

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

Obs.: ***< ANY(...)*** *significa menor que o maior da lista e* ***> ANY(...)*** *significa mais do que o mínimo*

7.3 Operador ALL

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

Obs.: ***> ALL(...)*** *significa mais do que o maior valor listado e* ***< ALL(...)*** *significa manos do que o menor valor listado ca mais do que o mínimo*
Obs.: Em uma instrução com subconsulta, a consulta interna é a primeira a ser resolvida.

Obs: Em http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/toc.htm encontramos todas as possibildiades com o comando SQL disponíveis nas versões Oracle10g

8 Manipulando dados (DML)

8.1 INSERT

```
INSERT INTO tabela [ (campo1, campo2, ..., campo n) ]
values (valor_campo1, valor_campo2, ..., valor_campo n)
```

Figura 8.1 Insert

A instrução INSERT serve para adicionar linhas em uma determinada tabela. Como você pode observar a lista de campos da tabela não é obrigatória, no entanto, se você optar por supri-la deverá ter em mente a sua estrutura da tabela, pois deverá fornecê-los na mesma ordem.

```
SQL>insert into regions
  2  (region_id,region_name)
  3  values
  4  (5, 'Africa')
  5  /

1 linha criada.

SQL>
SQL>insert into countries
  2  values
  3  ('BR', 'Brasil',50)
  4  /
insert into countries
*
ERRO na linha 1:
ORA-02291: restrição de integridade (HR.COUNTR_REG_FK) violada - chave mãe não
localizada
```

Exemplo 8.1 – Insert

Obs.: encontramos todas as opções do comando INSERT disponíveis nas versões Oracle10g em *http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_9014.htm#i2163698*

8.2 UPDATE

```
UPDATE Tabela
   SET Campo1 = valor_campo1, campo2 = valor_campo2,
...
[ WHERE <condição> ]
```

Figura 8.2 Update

A instrução UPDATE altera valores de campos de uma tabela, de acordo com uma condição fornecida, se esta condição for suprida, toda a tabela será atualizada. As regras que governam a restrição de linhas nas consultas são também aplicáveis nas instruções UPDATE (Ver 3.1, 3.2, 3.3, 3.4).

```
SQL>update locations
  2  set city = 'Belém'
  3  where LOCATION_ID = 2500
  4  /

x linha criada.

SQL>
SQL>update locations
  2  set country_id = 'AG'
  3  where LOCATION_ID = 1400
  4  /
update locations
*
ERRO na linha 1:
ORA-02291: restrição de integridade (HR.LOC_C_ID_FK) violada - chave mãe não localizada
```

Exemplo 8.2 – Update

Obs.: encontramos todas as opções do comando UPDATE disponíveis nas versões Oracle10g em http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_10007.htm#i2067715

8.3 DELETE

```
DELETE {tabela} [ WHERE <condição> ]
```

Figura 8.3 Delete

A instrução DELETE exclui um ou mais registros de acordo com a condição fornecida, similarmente ao UPDATE se esta condição for suprida **todos os dados de sua tabela serão apagados!!!.** As regras que governam a restrição de linhas nas consultas são também aplicáveis nas instruções DELETE (Ver 3.1, 3.2, 3.3, 3.4).

```
SQL>delete regions
  2  where region_id = 5
  3  /

1 linha deletada.

SQL>delete locations
  2   where LOCATION_ID=2500
  3  /
delete locations
*
ERRO na linha 1:
ORA-02292: restrição de integridade (HR.DEPT_LOC_FK) violada - registro filho localizado
```

Exemplo 8.3 – Delete

Obs.: encontramos todas as opções do comando DELETE disponíveis nas versões Oracle10g em http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_8005.htm#i2117787

9 Controle de transação

Transação é uma unidade lógica de trabalho que compreende uma ou mais instruções DML executadas em função das atividades de um usuário do sistema. O conjunto de DML's que devem estar contidas em uma transação é determinado pelas regras de nogócio.

O conjunto de inserções, alterações e exclusões efetivadas pelas instruções SQL pertencentes a uma transação pode ser confirmadas (COMMIT) ou desconsideradas (ROLLBACK). Uma transação se inicia com o primeiro comando DML executado. A transação termina quando é confirmada ou desconsiderada.

Para ilustrar o conceito de transação, podemos considerar o "velho" exemplo de atividades de banco de dados para instituições finaceiras. Quando um cliente do banco transfere valores da conta poupança para a conta corrente, a transação deve consistir no mínimo de 3(três) operações.

- Decrementar da conta poupança
- Incrementar na conta corrente
- Registrar a transação

O Oracle deve prover duas situações. Se as três operações conseguiram ser bem executadas afim de fornecer o adequado balanceamento nas contas, estas deverão ser aplicadas (COMMIT) ao banco de dados. Entretanto se algum problema como saldo insuficiente, número de conta inválida or falha de hardware impedir no mínimo uma atividade da transação, então a transação inteira deverá ser desconsiderada (ROLLBACK) afim de assegurar o adequado balanceamento nas contas.

Um ***savepoint*** permite dividir uma transação em várias partes e marcar um determinado ponto da transação que permitirá ao programador um rollback total onde toda a transação será desnconsiderada ou rollback parcial onde tudo o que foi executado após o *savepoint* será desconsiderado. Todo o *savepoint* tem um nome associado a ele.

```
 1 BEGIN
 2   INSERT1;
 3   INSERT2;
 4   SAVEPOINT exemplo_transação;
 5   UPDATE1;
 6   UPDATE2;
 7   IF teste THEN
 8     ROLLBACK; (rollback total)
 9   ELSE
10     ROLLBACK TO exemplo_transação; (rollback parcial)
11   END IF;
12 END;
```

Figura 9.2 SavePoint

Os comandos COMMIT e ROLLBACK respectivamente confiram ou desconsirem as transações segundo a lógica de programação imposta.

Obs.:No link a seguir encontramos a completa referência do comando COMMIT
http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_4010.htm#i2060233

Obs.:No link a seguir encontramos a completa referência do comando ROLLBACK
http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_9021.htm#i2104635

Obs.:No link a seguir encontramos a completa referência sobre transações Oracle
http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14220/transact.htm#i6564

Parte II - Programação PL/SQL

10 Bloco PL/SQL

PL/SQL(Procedural Language/SQL) é a linguagem procedural desenvolvida pela Oracle que é utilizada para montar os blocos PL/SQL.

Um bloco PL/SQL consiste de um conjunto de instruções SQL (SELECT, INSERT, UPDATE, DELETE) ou comandos PL/SQL, e desempenha uma função lógica única, afim de resolver um problema específico ou executar um conjunto de tarefas afins. O Bloco PL/SQL também pode ser referenciado com Unidade de Programa PL/SQL

Os blocos PL/SQL são qualificados em bloco anônimo e Stored Procedure.

O bloco anônimo

- Não tem nome
- Não está armazenado no SGDB
- Geralmente está armazenada na aplicação.

Stored SubProgramas

- Utiliza a estrutura do bloco anônimo com base.
- Estão armazenados no SGDB,
- A eles é atribuído um nome que poderá ser utilizado nas aplicações ou por outros objetos do banco de dados

A estrutura de um bloco PL/SQL é constituida de três seções:

a) **SEÇÃO DE DECLAÇÃO (DECLARE)** - Nesta seção são definidos os objetos PL/SQL como variáveis, constantes, cursores e exceções definidas pelo usuário que poderão ser utilizadas dentro do bloco.
b) **SEÇÃO DE EXECUÇÕES (BEGIN..END;)** - Nesta seção contemplará a sequência de comandos PL/SQL e instruções SQL do bloco.
c) **SEÇÃO DE TRATAMENTO DE ERRO (EXCEPTION)** - Nesta seção serão tratados os erros definidos e levantados pelo próprio bloco e os erros gerados pela execução do bloco (O capítulo 12 abordará o tratamento de exeções no PL/SQL)

```
[DECLARE
  -- declarações]
BEGIN
  -- instruções e comnados
[EXCEPTION
  -- tratamentos de erro]
END;
```

Figura 10.1 – Seções de um bloco PL/SQL

Diretrizes:

- Apenas a seção de execução é obrigatória.
- As palavras chaves, DECLARE, BEGIN, EXCEOPTION não são seguidas por ponto-e-vírgula, mas END e todas as outras instruções PL/SQL requerem ponto-e-vírgula.
- Não existe bloco sem algum comando válido.
- Pode existir aninhamento de bloco, no entanto, esta funcionalidade é restrita à seção de Execução e à Seção de Tratamento de Erro.
- As lnhas da seção de execução devem ser finalizadas com ; (ponto-e-vígula)

Maiores detalhes sobre a programação PL/SQL poderão ser encontradas em http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14261/toc.htm

11 Programação PL/SQL

11.1 Comentários

Os comentários em PL/SQL são de dois tipos

a) Uma Linha: utiliza-se o delimitador --. A partir de dois hífens tudo o que for escrito até o final da linha é considerado comentário.
b) Múltiplas linhas: utiliza-se o delimitador /* para abrir e */ para fechar. Tudo e todas as linhas que estiverem entre os dois delimitadores serão ignorados na execução.

```
 1  BEGIN
 2    -- comentando apenas uma linha
 3    COMANDO1;
 4    /* comentando
 5     várias
 6     linhas  */
 7    COMANDO2;
 8    COMANDO3;  -- o resto será ignorado
 9  END;
10  /
```

Exemplo 11.1 – Uso de comentário

11.2 Declarações (seção de declaração)

Para utilizar variáveis e constantes no seu programa, você deve declará-los anteriormente. É na seção DECLARE que são declaradas as variáveis e constantes.

```
1  DECLARE
2    nVIVenda     NUMBER(16,2);
3    cNmVendedor  VARCHAR2(40);
4    dDtVenda     DATE:=SYSDATE;
5    mMultiplic   CONSTANT NUMBER:=100; --constante
6  BEGIN
7    NULL;
9  END;
10 /
```

Exemplo 11.2 – Declaração de Objetos

As declarações no exemplo 2.2 foram:

- nVIVenda do tipo numérico tamanho 16 e 2 casas decimais,
- cNmVendedor do tipo numérico de tamanho variável até 40 caracteres,
- dDTVenda do tipo data e a constante
- mMultiplic do tipo numérica com valor 100.

O escopo de uma variável é a parte do programa onde a variável pode ser acessada. Para uma variável PL/SQL, isso ocorre a partir da declaração de variáveis até o final do bloco. Variáveis declaradas em um bloco externo são acessíveis apenas neste bloco e em qualquer sub-bloco contido neste, porém variáveis declaradas no sub-bloco não são acessíveis pelo bloco externo.

Pacote DBMS_OUTPUT

Na programação PL/SQL não existe nenhuma funcionalidade de entrada ou saída. Para remediar isso, usaremos no aplicativo SQL*Plus o Supplied Package DBMS_OUTPUT que fornecerá **apenas** a capacidade de dar saídas para mensagens na tela. Isso é feito por meio de dois passos

1. Permitir a saída no SQL*Plus com o comando set serveroutput

```
SET SERVEROUTPUT {ON | OFF}
```

2. Dentro do programa PL/SQL, utilize o procedure DBMS_OUTPUT.PUT_LINE. Essa procedure adicionará o argumento informado ao buffer de saída.

Com esses passos completados, a saída impressa na tela do SQL*Plus depois que o bloco for completamente executado. Durante a execução, o buffer é preenchido pelas chamadas de DBMS_OUTPUT.PUT_LINE. O SQL*Plus não recupera o conteúdo do buffer e não o imprime até que o controle retorne para o SQL*Plus, depois que o bloco terminou a execução.

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE('Hello from PL/SQL');
  3  END;
  4  /
Hello from PL/SQL

PL/SQL procedure successfully completed.

SQL>
```

11.3 Tipos de Dados

Tipo	Descrição
VARCHAR2 [(tamanho_maximo)]	Tipo básico para dados cararter de tamanho variável com até 32.767 bytes. Não há tamanho default para as constantes e variáveis VARCHAR2
NUMBER [(precisão,escala)]	Admite número de ponto fixo e flutuante.
DATE	Tipo básico para datas e horas. Valores DATE incluem a hora do dia em segundos desde a meia-noite. A faixa é entre 01/01/4712 A.C e 31/12/9999 D.C.
CHAR [(tamanho_maximo)]	Tipo básico para dados cararter de tamanho fixo. Se você não especificar o tamanho_maximo, o tamanho default Será definido com 1.
BOOLEAN	Tipo básico para dados que armazena um dos três possíveis valores usados para cálculos lógicos: TRUE, FALSE, NULL.
BINARY_INTEGER	Inteiros com sinal entre -2.147.483.647 a 2.147.483.647
PL_INTERGER	Inteiros com sinal entre -2.147.483.647 a 2.147.483.647. Valores com PL_INTERGER requerem menos espaço e são mais rápidos que os valore NUMBER e BINARY_INTERGER

O comando **%TYPE** nos dá a possibilidade de associarmos ao tipo de uma variável o tipo de uma coluna de uma tabela, desta forma, automaticamente a variável assumirá o tipo de dado da coluna.

O comando **%ROWTYPE** criará uma estrutura de registro idêntica à estrutura de uma tabela.

```
DECLARE
  Nome_variável nome_tabela.nome_coluna%TYPE; -- variável
  Nome_registro nome_tabela%ROWTYPE;          -- registro
BEGIN
  -- instruções e comnados
END;
```

Figura 11.1 – Uso de %TYPE e %ROWTYPE

Nesta abordagem o código fica mais ligado à estrutura e não será necessária a reescrita do código quando o tipo de coluna for alterado.

11.4 Assinalar Valores

Você pode assinalar valores a uma variável de duas formas. A primeira forma utiliza o operador **:=** (sinal de dois pontos seguido do sinal de igual). Assim a variável posicionada à esquerda do operador receberá o valor da expressão posicionada à direita.

```
 1  DECLARE
 2    nSalario      NUMBER;
 3    nSalarioAtual NUMBER;
 4    aRegionName   regions.region_name%TYPE;
 5    dHoje         DATE;
 6    nAnoBi        BOOLEAN:=TRUE;
 7  BEGIN
 8    nSalario      := 400;
 9    nSalarioAtual := F_SALARY(103) * 0.10;
10    aRegionName   := 'ASIA';
11    dHoje         := SYSDATE;
12  END;
13  /
```

Exemplo 11.3 – Assinalar valores por operador

A Segunda forma de assinalar valor a uma variável é através de um resultado de SELECT que será transferido assinalado à variável.

Um SELECT que assinala valor a uma variável obrigatoriamente deverá retornar uma e somente uma linha, caso contrário, um erro de execução será disparado, NO_DATA_FOUND se não for retornada nenhuma linha e TOO_MANY_ROWS se mais de uma linha for retornada (ver 12.3)

```
 1  DECLARE
 2    bonus10 NUMBER;
 3    bonus20 NUMBER;
 4    emp_id  NUMBER:=206;
 5  BEGIN
 6    SELECT salary * 0.10
 7      INTO bonus10
 8      FROM employees
 9     WHERE employee_id = emp_id;
10
11    SELECT salary * 0.10, salary * 0.20
12      INTO bonus10, bonus20
13      FROM employees
14     WHERE employee_id = emp_id;
15
16    DBMS_OUTPUT.PUT_LINE('SALARIO COM 10% DE BONUS :'|| bonus10);
17    DBMS_OUTPUT.PUT_LINE('SALARIO COM 20% DE BONUS :'|| bonus20);
18  END;
19  /
```

Exemplo 11.4 – Assinalar valores por SELECT

11.5 Controle de Fluxo

Este conjunto de comandos permite testar uma condição e, dependendo se a condição é falsa ou verdadeira, será tomada uma determinada direção de fluxo. O controle de fluco se dá em três formas: IF-THEN, IF-THEN-ELSE,IF-THEN-ELSIF.

11.5.1 IF-THEN

É a forma mais simples. Testa a condição especificada após o IF e, caso seja verdadeira, executa o comando além do THEN. Caso não seja, executa as ações após o END IF (note que devem ser escritos separados).

```
 1  DECLARE
 2    v_first_name employees.first_name%TYPE;
 3    v_salary     employees.salary%TYPE;
 4  BEGIN
 5    SELECT first_name,salary
 6      INTO v_first_name,v_salary
 7      FROM employees
 8     WHERE employee_id = 142;
 9
10    IF v_salary > 3000 THEN
11      DBMS_OUTPUT.put_line('Salario acima de US$ 3,000');
12      DBMS_OUTPUT.put_line('Teste IF-THEN');
13    END IF;
14  END;
15  /
```

Exemplo 11.5 – IF-THEN

Obs.:Note que cada linha de ação dentro do IF deve ser terminada com ponto-e-vírgula(;) e apenas após o END IF é que se coloca o ponto-e-vírgula final do comando IF.

11.5.2 IF-THEN-ELSE

Aqui, acrescenta-se a palavra-chave ELSE para determinar o que deve ser feito caso a condição seja falsa. Dessa forma, o fluxo seguirá para os comandos após o THEN caso a condição seja verdadeira, e após o ELSE caso seja falsa.

```
 1  DECLARE
 2    v_first_name     employees.first_name%TYPE;
 3    v_commission_pct employees.commission_pct%TYPE;
 4  BEGIN
 5    SELECT first_name,commission_pct
 6      INTO v_first_name,v_commission_pct
 7      FROM employees
 8     WHERE employee_id = 174;
 9
10    IF v_commission_pct IS NULL THEN
11      DBMS_OUTPUT.put_line('Sem comissão');
12      DBMS_OUTPUT.put_line('outra ação');
13    ELSE
14      DBMS_OUTPUT.put_line('Comissão de '||v_commission_pct*100||'%');
15      DBMS_OUTPUT.put_line(' outra ação ');
16    END IF;
17  END;
18  /
```

Exemplo 11.6 – IF-THEN-ELSE

Obs.:Note que há um ponto-e-vírgula somente após cada linha de ação e após o END IF.

11.5.3 IF-THEN-ELSIF

Quando se deseja testar diversas condições utilizando um mesmo IF, utiliza-se ELSIF. Assim, pode-se após cada ELSIF, testar nova condição que, caso seja verdadeira, executará as respectivas ações.

```
 1  DECLARE
 2     Vencimentos NUMBER;
 3  BEGIN
 4    vencimentos := F_SALARY(101); -- deduções
 5    IF vencimentos <= 10000 THEN
 6      DBMS_OUTPUT.PUT_LINE('Primeira faixa');
 7    ELSIF vencimentos > 10000 AND vencimentos <= 15000 THEN
 8      DBMS_OUTPUT.PUT_LINE('Segunda faixa');
 9    ELSIF vencimentos > 15000 AND vencimentos <= 20000 THEN
10       DBMS_OUTPUT.PUT_LINE('Terceira faixa');
11    ELSE
12      DBMS_OUTPUT.PUT_LINE('Ultima faixa');
13    END IF;
14  END;
15  /
```

Exemplo 11.7 – IF-THEN-ELSIF

Mesmo utilizando diversos ELSIF's pode-se acrescentar um ELSE no final para o caso de nenhuma das condições anteriores serem satisfeitas. Mais uma vez, somente após o END IF e a cada linha de ação é que se deve colocar o ponto-e-vírgula.

11.6 Controle de Repetição

O LOOP permite que você realize repetições de determinadas ações. Na programação PL/SQL encontramos 3(três) tipo: LOOP Simples, WHILE-LOOP, FOR-LOOP.

11.6.1 LOOP Simples

Com este comando você pode realizar repetições de uma seqüência de comandos. O comando LOOP indica o inicio da área de repetição, enquanto que o END LOOP indica que o fluxo deve retornar do LOOP.

```
1  BEGIN
2    LOOP
3      DBMS_OUTPUT.PUT_LINE('Primeira ação do laço');
4      DBMS_OUTPUT.PUT_LINE('Segunca ação do laço');
5    END LOOP;
6  END;
7  /
```

Exemplo 11.8 – LOOP simples

No exemplo 11.8 como não está definida nenhuma condição de parada do laço, você já deve ter concluído que este LOOP não terá fim. Logo, para resolver este problema é necessário utilizar o comando EXIT ou EXIT WHEN. O EXIT causa uma saída incondicional do LOOP, e o EXIT WHEN permite testar uma condição e, apenas se ela for verdadeira, provocará a saida do LOOP.

```
 1  DECLARE
 2    x NUMBER;
 3  BEGIN
 4    X := 1;
 5    LOOP
 6      DBMS_OUTPUT.PUT_LINE('O valor de x eh '||X);
 7      X := X + 1;
 8      EXIT WHEN X>=5;
 9    END LOOP;
10  END;
11  /
```

Exemplo 11.9 – LOOP simples

11.6.2 WHILE-LOOP

Este comando permite testar uma condição antes de iniciar a seqüência de ações de repetição. Ao final de LOOP, é testada a condição novamente e, caso

verdadeira, continua a seqüência de ações dentro do LOOP ou sai, caso seja falsa, executando o que estiver após o END LOOP.

```
1  DECLARE
2    x NUMBER;
3  BEGIN
4    X := 1;
5    WHILE X < 5 LOOP
6      X := X + 1;
7    END LOOP;
8  END;
9  /
```

Exemplo 11.10 – WHILE - LOOP

11.6.3 FOR - LOOP

Utilize este comando sempre que você souber preveamente o número de vezes que um LOOP deve ser executado. A cada comando FOR-LOOP existe uma variável controladora que em cada interação assumirá todos os valores interiros (variando de 1 em 1) contidos entre o limite inicial e o limite final.

```
FOR contador IN [REVERSE] inicio..fim LOOP
  comando1;
  comando2;
END LOOP;

contador    Variável que terá seu valor incrementado.
[REVERSE]   Indica que se deve diminuir ao invés de aumentar O
            contador. O valor de inicio deve ser maior que o
            fim, pois o valor será decrescido a cada repetição
inicio      intervalo inicial de repetição
fim         intervalor final da repetição
```

Figura 11.2 – FOR-LOOP

```
1  DECLARE
2    Y NUMBER := 1;
3  BEGIN
4    FOR X IN 1..5 LOOP
5      Y := Y + X ;
6      DBMS_OUTPUT.PUT_LINE(x);
7    END LOOP;
8  END;
9  /
```

Exemplo 11.11 – FOR-LOOP.

Não necessidade de declarar a variável controladora, isso é feito implicitamente pelo comando FOR-LOOP. Podemos utilizar a variável controladora como uma variável normal, no entanto, não podemos assinalar valores à variável controladora. O escopo de visibilidade na variável controladora é apenas dentro do laço.

Caso o número de vezes que deva ser repetida a seqüência de ações seja fruto de um cálculo, você poderá substituir tanto o intervalo superior quanto o superior por variáveis, mas não se esqueça que esses valores devem ser sempre números inteiros.

Obs.: Você poderá utilizar o EXIT WHEN 'condição' para terminar prematuramente o FOR-LOOP.

11.7 Labels

Os labels são utilizados para melhorar a leitura do programa PL/SQL. Labels são aplicados a blocos ou LOOP's. Um label deve preceder imediatamente um bloco ou LOOP e deve ser delimitado por **<<** e **>>**. A cláusula END ou END LOOP

pode fazer referência do label. O uso de labels é vantajoso quando existem vários blocos aninhados.

```
 1  DECLARE
 2    v_dept      NUMBER(2);
 3    v_emp_count NUMBER(2);
 4  BEGIN
 5    <<seis_tentativas>>
 6    FOR cont IN 1..6 LOOP
 7      <<bloco_selecao>>
 8      BEGIN
 9        SELECT DEPARTMENT_ID
10          INTO v_dept
11          FROM DEPARTMENTS
12         WHERE DEPARTMENT_ID = cont * 10;
13        <<bloco_contador>>
14        BEGIN
15          SELECT count(*)
16            INTO v_emp_count
17            FROM EMPLOYEES
18           WHERE DEPARTMENT_ID = v_dept;
19        END bloco_contador;
20        DBMS_OUTPUT.PUT_LINE('Existe(m) '||v_emp_count||' empregados no departamento
'||v_dept);
21      END bloco_selecao;
22    END LOOP seis_tentativas;
23  END;
24  /
```

Exemplo 11.12 – Labels

11.8 Cursores

Em alguns casos necessitamos de espaços de armazenamento mais complexos que as variáveis, como uma matriz de informação resultada de uma consulta SQL, neste case se faz necessário o uso de cursores.

Os cursores em PL/SQL podem ser explícitos e implícitos. O PL/SQL declara um cursor implicitamente para toda instrução DML (UPDATE, INSERT, DELETE, SELECT...INTO), incluindo consultas que retornam apenas uma linha. As consultas que retornam mais de uma linha deverão ser declaradas explicitamente.

Cursores explícitos são indicados quando é necessário um controle no processamento do mesmo.

11.8.1 Controlando Cursores Explícitos

De acordo com a figura 11.3, quatro são os comandos que controlam o processamento de um cursor.

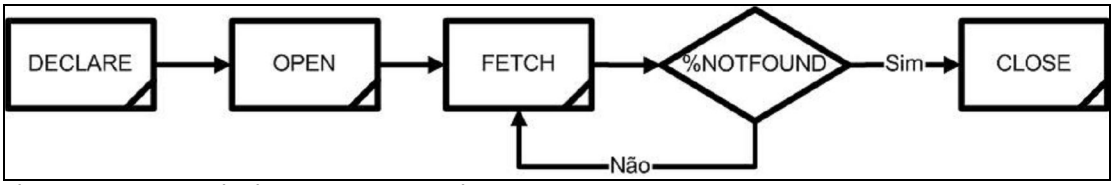


Figura 11.3 – Controle de Processamento de Cursores

11.8.2 Declarando um Cursor (DECLARE)

Quando declaramos um cursor é associado a ele um nome e a consulta SQL que será processada por este cursor. Assim como as variáveis, os cursores devem ser declarados na seção DECLARE.

O escopo de validade dos cursores é o mesmo de uma variável (ver 11.2). Cursores declarados em um bloco externo são acessíveis apenas neste bloco e em qualquer sub-bloco contido neste, porém cursores declarados no sub-bloco não são acessíveis pelo bloco externo.

```
CURSOR nome_cursor
  [(parametro1 tipo
   ,parametro2 tipo
   ,...,
   ,parametroN tipo)]
IS Instrução_SQL;
```

Figura 11.4 – Declarando um cursor

Os cursores podem ser definidos com parâmetros e para cada parâmetro devem ser escolhidos um nome e um tipo de dado(ver 11.3).

11.8.3 Abrindo um Cursor (OPEN)

O comando OPEN abre o cursor, executa a consulta associada a ele e gera o *conjunto ativo*, que consiste de todas as linhas que atendem os critérios de pesquisa da consulta associada ao cursor. Para gerenciar o conjunto ativo existe um ponteiro que registra qual linha está passível do comando FETCH. Após o OPEN o FETCH atuará sobre a primeira linha do conjunto ativo.

```
OPEN nome_cursor[(var1,var2,...)];
```

Figura 11.5 – Abrindo um cursor

11.8.4 Extraindo dados do Cursor (FETCH)

Extrair os dados do cursor é o evento onde os dados da linha atual do conjunto ativo são copiados para variáveis ou registros e a cada FETCH realizado, o ponteiro passará a apontar para a linha seguinte do conjunto ativo.

```
FETCH nome_cursor INTO [var1,var2,...|record_name];
```

Figura 11.6 – Fetch Cursor

Diretrizes

- Inclua o mesmo número de variáveis na cláusula INTO da instrução FETCH do que as colunas na instrução SELECT e certifique-se que os tipos de dados são compatíveis
- Faça a correspondência de cada variável para coincidir com as posições das colunas
- Registros podem ser utilizados. O tipo %ROWTYPE pode ser associado ao cursor ou diretamente a uma tabela. Os campos do cursor devem ser idênticos aos campos do registro usado em quantidade e tipo

11.8.5 Fechando do Cursor (CLOSE)

O comando CLOSE desativa o cursor e libera o conjunto ativo. Esta etapa permite que o cursor seja reaberto, se necessário, para gerar um outro conjunto ativo.

```
CLOSE nome_cursor;
```

Figura 11.6 – Fechando um cursor

```
 1  DECLARE
 2    V_empno NUMBER;
 3    V_ename VARCHAR2(100);
 4    CURSOR cEmplyee IS
 5     SELECT employee_id,first_name
 6       FROM EMPLOYEES;
 7    rEmployee cEmplyee%ROWTYPE;
 8    CURSOR cEmplyeeJob
 9      (p_job varchar)
10    IS
11     SELECT first_name
12       FROM EMPLOYEES
13      WHERE Job_id = p_job;
14  BEGIN
15    OPEN cEmplyee;
16    FETCH cEmplyee INTO V_empno, V_ename;
17    DBMS_OUTPUT.PUT_LINE(V_ename);
18    FETCH cEmplyee INTO V_empno, V_ename;
19    DBMS_OUTPUT.PUT_LINE(V_ename);
20    FETCH cEmplyee INTO V_empno, V_ename;
21    DBMS_OUTPUT.PUT_LINE(V_ename);
22    FETCH cEmplyee INTO rEmployee ;
23    DBMS_OUTPUT.PUT_LINE(rEmployee.first_name);
24    CLOSE cEmplyee;
25
26    OPEN cEmplyeeJob('SALESMAN');
27    FETCH cEmplyeeJob INTO V_ename;
28    DBMS_OUTPUT.PUT_LINE(V_ename);
29    FETCH cEmplyeeJob INTO V_ename;
30    DBMS_OUTPUT.PUT_LINE(V_ename);
31    CLOSE cEmplyeeJob;
32
33    OPEN cEmplyeeJob('MANAGER');
34    FETCH cEmplyeeJob INTO V_ename;
35    DBMS_OUTPUT.PUT_LINE(V_ename);
36    FETCH cEmplyeeJob INTO V_ename;
37    DBMS_OUTPUT.PUT_LINE(V_ename);
38    CLOSE cEmplyeeJob;
39  END;
40  /
```

Exemplo 11.13 – Cursores

Obs: No exemplo acima, as linhas 8 até 13 mostram a declaração de um cursor com parâmetro e nas linhas 15, 26 e 33 mostra o "open" do cursor. A linha 7 mostra um registro recebendo a estrutura de linha de um cursor (isso poderia ser feito a uma tabela) e as linhas 22 e 23 mostram o fetch para o registro e o uso do valor do registro

11.8.6 Atributos do Cursor Explícito

Quando anexados ao nome do cursor, esses atributos retornam informações úteis sobre a execução de uma instrução de manipulação de dados.

Atributo	Tipo	Descrição
%ISOPEN	Booleano	Será avaliado para TRUE se o cursor estiver aberto
%NOTFOUND	Booleano	Será avaliado para TRUE se a extração mais recente não retornar linha.
%FOUND	Booleano	Será avaliado para TRUE se a extração mais recente retornar linha.
%ROWCOUNT	Numerico	Será avaliado para o número total de linhas retornadas até o momento.

11.8.7 LOOP Simples X Cursor

Neste primeiro estilo de loop de busca, a sintaxe de loop simples é utilizada para processamento do cursor. Atributos explícitos de cursor são utilizados para controlar o número de vezes que o loop é executado.

```
 1  DECLARE
 2    CURSOR cEmpregados IS
 3      SELECT first_name FROM employees;
 4    aName employees.first_name%TYPE;
 5  BEGIN
 6    OPEN cEmpregados;
 7    LOOP
 8      FETCH cEmpregados INTO aName;
 9      EXIT WHEN cEmpregados%NOTFOUND;
10      DBMS_OUTPUT.PUT_LINE(aName);
11    END LOOP;
12    CLOSE cEmpregados;
13  END;
14  /
```

Exemplo 11.14 – Cursores Loop Simples

11.8.8 LOOP While X Cursor

O mesmo Exemplo 11.14 poderia ser escrito utilizando a sintaxe WHILE..LOOP, da seguinte maneira.

```
 1  DECLARE
 2    CURSOR cCidades IS
 3      SELECT * FROM locations;
 4    rCity locations%ROWTYPE;
 5  BEGIN
 6    OPEN cCidades;
 7    FETCH cCidades INTO rCity;
 8    WHILE cCidades%FOUND LOOP
 9      DBMS_OUTPUT.PUT_LINE(rCity.city||' - '||rCity.state_province);
10      FETCH cCidades INTO rCity;
11    END LOOP;
12    CLOSE cCidades;
13  END;
14  /
```

Exemplo 11.15 – Cursores Loop While

11.8.9 LOOP For X Cursor

Os dois exemplos de LOOP's descritos anteriormente requerem um processamento explícito de cursor por meio de instruções OPEN, FETCH, CLOSE ver (11.8.1). A programação PL/SQL fornece um tipo de LOOP mais eficiente, que trata implicitamente o processamento de cursor.

```
 1  DECLARE
 2    CURSOR cCargos IS
 3      SELECT job_title, job_id
 4        FROM jobs;
 5  BEGIN
 6    FOR rCargo IN cCargos LOOP
 7      DBMS_OUTPUT.PUT_LINE(rCargo.job_id||' - '||rCargo.job_title);
 8    END LOOP;
 9  END;
10  /
```

Exemplo 11.16 – Cursor FOR

Observações

- O registro rCargo não é declarado, sua declaração é executada implicitamente, recebe o tipo cCargos%ROWTYPE e o seu escopo é apenas o LOOP.
- O cursor cCargos é processado implicitamente, sendo desnecessário os comandos OPEN, FETCH, CLOSE.

11.8.10 LOOP For Implícitos

Além do registro, o próprio cursor pode ser implicitamente declarado. A consulta SQL geradora do conjunto ativo é apresentada em de parênteses dentro da própria instrução FOR, e neste caso, tanto o registro com o cursor são implicitamente declarados.

```
1  BEGIN
2    FOR rDepartamento IN (SELECT d.department_id, d.department_name
3                          FROM departments d) LOOP
4      DBMS_OUTPUT.PUT_LINE(rDepartamento.department_name);
5    END LOOP;
6  END;
7  /
```

Exemplo 11.17 – Cursor FOR Implícito

11.8.11 Cursores Implícitos

Existem os cursor implícitos que são criados para processar as instruções INSERT, UPDATE, DELETE, SELECT...INTO e são manipulados a revelia do programador. Neste caso apenas o atributo %ROWCOUTN é interessante para a instrução UPDATE. O cursor implícito é representado pela palavra reservada SQL.

```
1  BEGIN
2    UPDATE jobs
3       SET MAX_SALARY = MAX_SALARY+100
4     WHERE MAX_SALARY < 9000;
5
6    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT||' linhas salarios foram atualizadas');
7  END;
8  /
```

Exemplo 11.18 – Cursor Implícito

12 Tratamento de Exceção

Qualquer boa linguagem de programação deve ter a capacidade de tratar erros de run-time e, se possível, recuperar-se deles. A programação PL/SQL implementa essa funcionalidade por meio de exceções PL/SQL e tratamento de exceções.

Para se entender como o PL/SQL trabalha com os tratamentos de exeções, será necessário o entendimento dos seguintes conceitos:

- ✓ **Erro de run-time ou exeção**– situação adversa que ocasiona interrupção na execução do programa podendo ser motivada por falha na concepção do sistema, codificação equivocada, falha de hardware, ou outro motivo.
- ✓ **Exceção PL/SQL** – É o objeto de programação PL/SQL que nos permite evitar a interrupção abrupta do programa caso este seja acomedido de um erro de run-time.
- ✓ **Tratamento da exeção** – Indica qual ação ou quais ações deverão ser tomadas quando o programa for acomedido de um erro de run-time.

Quando o Oracle apresenta um erro de run-time ou exeção, este sempre será acompanhado de um cógido é uma mensagem. Todos os possíveis erros de run-time ou exceções podem ser consultados na lista de erros Oracle em http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14219/toc.htm. A cada cógido está associado além de uma mensagem, uma causa e ações a serem tomadas. Essas informações são extremamente importantes no dia-a-dia de um programador PL/SQL.

Quando criamos um programa, a rigor não sabemos, antes de colocá-lo em produção quais erros de run-time ou exceção podem ocorrer no momento da execução. No entanto, em alguns casos podemos vislumbrar ou imaginar um conjunto pequeno de erros de run-time ou exceção que poderão ocorrer na execução. A qualidade do programa se dará em função da habilidade que o programa terá para tratar os eventuais erros. Para cada erro vislumbrado ou imagnado deverá existir um tratamento espeçífico, e isso é perfeitamente programável no PL/SQL.

12.1 Tratando X Propagando

Quando um erro de run-time ou exeção ocorre, o controle do bloco PL/SQL deixa a seção de execução(BEGIN) e passa compulsivamente para a seção de exceção(EXCEPTION). Se existe algum tratamento específico para o erro de run-time (atravéz de um objeto EXCEPTION), este será **capturado e tratado** (se existir o devido tratamento) pelo bloco e a execução do bloco terminará sem apresentar o erro de run-time (isso não quer dizer que o bloco executou com sucesso todas as suas atividades). Se por outro lado, este tratamento não existe, ou no bloco não existe a seção EXCEPTION, então o bloco **propagará ou transferirá** o erro de run-time para bloco que o contém ou para o ambiente que o executou.

```
<<bloco_de_fora>>
BEGIN
  comando1;
  comando2;
  <<bloco_de_dentro>>
  BEGIN
    comando3;
    comando4;
  EXCEPTION
    tratamento1;
    tratamento2;
  END bloco_de_dentro;
  Comando5;
  Comando6;
EXCEPTION
  tratamento3;
  tratamento4;
END bloco_de_fora;
```

Figura 12.1 – Capturando X Propagando

Tabela 12.1. Tabela de Comportamento.

	Premissa p/ tratamento	**Quando Tratado**	**Quando Propagado**
<<bloco_de_dentro>>	tratamento1 e tratamento2 serem capazes de capturar todo e qualquer erro de runtime gerado por comando3 ou comando 4	O comando5 é executado, dando continuidade à execução.	O <<bloco_de_fora>> recebe o run-time e fica encaregado de propagar ou capturar e os comando5 e comando6 não são executados.
<<bloco_de_fora>>	tratamento3 e tratamento4 serem capazes de capturar todo e qualquer erro de runtime gerado por comando1, comando2 ou <<bloco_de_dentro>>	Procedimento concluído com sucesso	O ambiente que executou o bloco mostra o código de erro.

12.2 Tratamento de Exceções

É atravez do objeto **exeção PL/SQL** que temos a possibilidade de qualificar as exeções e aplicá-las o devido tratamento. É na seção de exceção que ocorrem os tratamentos de erros de run-time como ilustrado pelo exemplo 12.1.

Para cada exeção provável deverá exisitr um tratamento atravéz de uma cláusula WHEN que estará associada a uma exeção PL/SQL, seguida por uma seqüência de instruções a serem executadas quando o run-time ocorrer, que representam efetivamento o tratamento do erro de run-time.

A cláusula WHEN OTHERS, se usada, deve ser posicionada com último tratamento, e é utilizada para tratar alguma exeção que não encontrará o devido tratamento nas cláusulas WHEN. Devemos pensar a seção de exceção como um comando IF, ELSIF, ELSE. Cada WHEN corresponde a um IF ou ELSIF, e WHEN OTHERS corresponde ao ELSE.

```
 1  DECLARE
 2    v_employee_id employees.employee_id%TYPE;
 3    v_first_name  employees.first_name%TYPE:='Steven';
 4  BEGIN
 5    SELECT employee_id
 6      INTO v_employee_id
 7      FROM employees
 8     WHERE first_name = v_first_name;
 9  EXCEPTION
10    WHEN NO_DATA_FOUND THEN
11      DBMS_OUTPUT.PUT_LINE('Primeiro nome não encontrado');
12    WHEN TOO_MANY_ROWS THEN
13      DBMS_OUTPUT.PUT_LINE('Retornou mais de uma linha');
14    WHEN OTHERS THEN
15      DBMS_OUTPUT.PUT_LINE('Problemas ao executar o procedimento!!!');
15  END;
16  /
```

Exemplo 12.1 – Tratamento de Erros

No exemplo 12.1 apresenta um programa com uma instrução SELECT (linha5) e três tramatentos de exeção (linhas 10, 12 e 14). Sendo dois tratamentos específicos para determinadas exeções e o terceiro para tratar toda e qualquer exceção inesperada. Desta forma o programa sempre terminará a execução sem apresentar erro algum.

Existem 2(dois) tipos de objetos exeção PL/SQL

- Pré-definidos ou internos
- Definido pelo usuário

Diretrizes

- A palavra EXCEPTION inicia a seção de tratamento de exceções.
- São permitidos vários tratamentos de exceção
- Somente um tratamento é processado antes de se sair do bloco
- WHEN OTHERS se existir sempre deve ser o último tratamento.

12.3 Exceções PL/SQL Pré-definidas ou internas

As exeções pre-definidas são disparados automaticamente pelo programa quando este gera um erro de run-time. Do grupo de erros Oracle catalogados, existe um grupo menor para as quais foram criadas exceções pré-definidas ou internas que podem ser tratadas diretamente na seção EXCEPTION, sem a necessidade de declaração na seção DECLARE.

Os erros de run-time que têm exceção associada podem ter um tratamento específico e previlegiado através da cláusula WHEN, os demais, que formam o maior grupo, só poderão ser tratados na cláusula OTHERS. Veremos a frente com remediar isso (ver 12.6)

Na tabela 12.2 mostra os mais recorrentes e em http://download-east.oracle.com/docs/cd/B14117_01/appdev.101/b10807/07_errs.htm#i7014 encontram-se todas as exceções pré-definidas

Tabela 12.2. Exceções Pré-definidas.

Exeption	OracleError	Raise when
DUP_VAL_ON_INDEX	ORA-00001	O seu programa tentou armazenar valores duplicados em uma coluna com restrição de UNIQUE
INVALID_CURSOR	ORA-01001	O seu programa tentou um operação ilegal de cursores como fechar um cursos já fechado.
NO_DATA_FOUND	ORA-01403	Uma instrução SELECT INTO retornou nenhuma linha.
TOO_MANY_ROWS	ORA-01422	Uma instrução SELECT INTO retornou mais de uma linha.
ZERO_DIVIDE	ORA-01476	O seu programa tentou efetuar uma divisão por 0(zero)

12.4 Exceções PL/SQL definidas pelo Usuário

Para que uma exceção seja definida pelo programador, esta deve ser declarada explicitamente na seção DECLARE e são acionadas através do comando RAISE.

O escopo de validade deste tipo de exceção é o mesmo de uma variável (ver 11.2). Exceções declaradas em um bloco externo são acessíveis apenas neste bloco e em qualquer sub-bloco contido neste, porém exceções declaradas no sub-bloco não são acessíveis pelo bloco externo.

```
 1  DECLARE
 2    e_Emp_Invalido    EXCEPTION;
 3    v_sal             employees.salary%TYPE:=2000;
 4    v_employee_id     employees.employee_id%TYPE:=5;
 5  BEGIN
 6    UPDATE employees
 7       SET salary = v_sal
 8     WHERE employee_id = v_employee_id;
 9    IF SQL%NOTFOUND THEN
10      RAISE e_Emp_Invalido;
11    END IF;
12  EXCEPTION
13    WHEN e_Emp_Invalido THEN
14      DBMS_OUTPUT.PUT_LINE('Este funcionario naum existe');
15  END;
16  /
```

Exemplo 12.2 – Exceção declarada

No exemplo 12.2.o programa declara a exeção PL/SQL *e_emp_inválido* na linha 2, aciona a exeção na linha 10 e a trata na linha 13. Percebam que o tratamento dado às exeções PL/SQL definidas pelo usuário é identica ao tratamento dado às exeções PL/SQL pré-definidas.

Exeções PL/SQL definidas são uteis para tratar sutiações relativa ao negócio como *saldo insuficiente*, *cliente já cadastrado* ou *código já utilizado.*

Obs.: Se uma exceção definida pro usuário for disparada e o devido tratamento não existir na seção EXCEPTION o bloco retornará o seguinte erro "ORA-06510: PL/SQL: exceção não-manipulada definida pelo usuário"

12.5 Comando RAISE_APLLICATION_ERROR

Em alguns casos se faz necessário forçar um erro de run-time. Para isso usamos o comando RAISE_APPLICATION_ERROR, que nos permite interromper a execução de um programa, gerando erro de run-time e assinalá-lo um código e uma mensagem. Para usarmos RAISE_APPLICATION_ERROR devemos utilizar a seguinte sintaxe:

```
raise_application_error(error_number, message);
```

Figura 12.1 – Tratamento de Erros

onde `error_number` é um numero negativo e inteiro na faixa de -20999 a -20000 e `message` é a mensagem customizada pelo programador.

```
 1  DECLARE
 2    nQdt NUMBER;
 3  BEGIN
 4    SELECT COUNT(*)
 5      INTO nQdt
 6      FROM employees;
 7    IF nQdt < 100 THEN
 8      RAISE_APPLICATION_ERROR(-20000,'Ainda não Existem 100 funcionarios');
 9    END IF;
10  END;
11  /
```

Exemplo 12.3 – RAISE_APPLICATION_ERROR

Obs.: Exceções geradas por RAISE_APPLICATION_ERROR não estarão na lista erros Oracle e serão capturadas e tratadas na cláusula WHEN OTHERS.

12.6 Pragma EXCEPTION_INIT

Para tratar os erros de run-time que não tem exceção PL/SQL pré-definida (ver 12.3), deveríamos usar a cláusula OTHERS ou PRAGMA EXCEPTION_INIT. Nesta segunda abordagem, o compilador associa uma exceção declarada pelo usuário (ver 12.4) com um código de erro de run-time mapeado na lista de erros Oracle.

```
 1  DECLARE
 2    e_emp_remain EXCEPTION;
 3    PRAGMA EXCEPTION_INIT(e_emp_remain,-2292);
 4    V_department_id departments.department_id%TYPE:=60;
 5    nQdt            NUMBER;
 6  BEGIN
 7    DELETE departments
 8     WHERE department_id = V_department_id;
 9    IF SQL%NOTFOUND THEN
10      RAISE_APPLICATION_ERROR(-20001,'O depatamento '||V_department_id||' não
existe');
11    END IF;
12  EXCEPTION
13    WHEN e_emp_remain THEN
14      SELECT COUNT(*)
15        INTO nQdt
16        FROM employees
17       WHERE department_id = V_department_id;
18    DBMS_OUTPUT.PUT_LINE('Não é possivel a remoção do departamento '|| V_department_id
||'. Nele existe(m) '||nQdt||' funcionario(s)');
19  END;
20  /
```

Exemplo 12.4 – Pragma EXCEPTION_INIT

No exemplo 12.4 o comando PRAGMA EXCEPTION_INIT (linha 3) associa ORA-2292 à exeção PL/SQL *e_emp_remain* declarada na linha 2. Apartir desta associação os erros de run-time ORA-2292 poderão ser tratados pela exeção PL/SQL *e_emp_remain* (linha 13).

Esta modalidade de tratamento é interessante para os erros de chave estrangeira “ORA-02292: restrição de integridade violada - registro filho Localizado” e “ORA-02291: restrição de integridade violada - chave mãe não localizada”.

12.7 SQLCODE, SQLERRM

Quando ocorre um erro de run-time ocorre, você pode identificar o código e a mensagem do erro associado através das funções SQLCODE e SQLERRM. A obtenção do código erro e consulta na lista de erros Oracle ajuda na resolução de erros de programação.

Tabela 12.3. SQLCODE, SQLERRM

Função	Descrição
SQLCODE	Retorna o número de código de erro
SQLERRM	Retorna os dados de caracteres que contêm a mensagem associada ao número de erro

```
 1  DECLARE
 2    V_country_id countries.country_id%TYPE:='CA';
 3  BEGIN
 4    DELETE countries
 5     WHERE country_id = V_country_id;
 6  EXCEPTION
 7    WHEN OTHERS THEN
 8      DBMS_OUTPUT.PUT_LINE('Código   -> '||SQLCODE);
 9      DBMS_OUTPUT.PUT_LINE('Mensagem -> '||SQLERRM);
10  END;
11  /
```

Exemplo 12.5 – SQLCODE, SQLERRM

Parte III - Objetos Procedurais

13 Stored Subprograms

Subprogramas são compilados e armazenados no banco de dados Oracle, estão disponíveis para leitura e execução. Uma vez compilados se tornam objetos de schema na forma de stored procedure ou stores funciton, que podem ser acessados pelos usuários e aplicações conectados do banco de dados.

O comando CREATE [OR REPLACE] PROCEDURE nos permite criar uma procedure no banco de dados e o comando CREATE [OR REPLACE] FUNCTION nos permite criar uma função de banco de dados.

A utilização de subprogramas armezenados nos traz a vantagem de compartilhamento de memória, tendo se em vista que apenas uma versão do subprograma será carregada em memória e será compartilhada por vários usuários. Podemos citar também a padronização na forma de tratar os dados.

13.1 Stored Procedure

Uma PROCEDURE é um bloco PL/SQL nomeado que pode obter parâmetros (algumas vezes chamados de argumentos), e que pode ser referenciada por nome.

Para transformar um bloco PL/SQL em uma PROCEDURE basta eliminar palavra DECLARE (se existir) e adicionar **CREATE [OR REPLACE] PROCEDURE nome_procedimento (par1,par2,...,parN) IS** ao inicio. Entre o nome do procedimento e a palavra IS, podem ser especificados os parâmetros separados por vírgula.

```
CREATE [OR REPLACE] PROCEDURE nome_procedimento
  (parametro1 [MODO] tipodado
  ,parametro2 [MODO] tipodado,
  ,...
  ,parametroN [MODO] tipodado)
IS
  -- declarações
BEGIN
  -- instruções e comandos
EXCEPTON
  -- tratamentos de erro
END;
```

Figura 13.1 Sintaxe Create procedure

```
1 CREATE OR REPLACE PROCEDURE ProcNula
2 IS
3 BEGIN
4   DBMS_OUTPUT.PUT_LINE('Primeira Procedure');
5 END ProcNula;
6 /

1 BEGIN
2   ProcNula;
3 END;
/
```

Exemplo 13.1 – Primeira procedure

As PROCEDURES podem ser chamadas apartir de :

- blocos anônimos
- procedures
- triggers
- funções

É boa prática após o END de conclusão de PROCEDURE especificar o nome desta como vemos na linha 5(cinco) de exemplo 13.1.

13.1.1 Parâmetros Stored Procedures

A uma PROCEDURE pode estar assiciado vários parâmetros e para cada parâmetro, dois aspectos devem ser considerados, o tipo de parâmetro e o modo do parâmetro. Quanto ao primeiro, apenas uma definição deverá existir por parâmetro, serão considerados os tipos de dados permitidos na programação PL/SQL (VER 11.3)

Quanto ao modo do parâmetro, existem 3 casos, IN, OUT e IN OUT

Figura 13.1 Comportamento Parâmetros

IN	OUT	IN OUT
Default	Deve ser especificado	Deve ser especificado
O valor é transferido do ambiente para a procedure	O valor é transferido da procedure para o ambiente.	O valor é transferido do ambiente para a procedure e em seguida transferido da procedure para o ambiente.
Age dentro da procedure como uma constante	Age dentro da procedure como uma variável não inicializada	Age dentro da procedure como uma variável inicializada
O parâmetro pode ser uma variável inicializada, constante, expressão ou literal	Deve ser uma variável	Deve ser uma variável

```
 1  CREATE OR REPLACE PROCEDURE raise_salary
 2    (p_employee_id employees.employee_id%TYPE)
 3  IS
 4    v_salary employees.salary%TYPE;
 5  BEGIN
 6    SELECT salary
 7      INTO v_salary
 8      FROM employees
 9     WHERE employee_id = p_employee_id;
11    DBMS_OUTPUT.PUT_LINE('Salário Anterior '||v_salary);
13    v_salary := v_salary * 1.1;
15    UPDATE employees
16       SET salary = v_salary
17     WHERE employee_id = p_employee_id;
19    DBMS_OUTPUT.PUT_LINE('Salário Corrigido '||v_salary);
21  EXCEPTION
22    WHEN NO_DATA_FOUND THEN
23      DBMS_OUTPUT.PUT_LINE('Empregado '||p_employee_id||' não existe');
24  END raise_salary;
25  /

 1  BEGIN
 2    raise_salary(100);
 3  END;
 4  /
```

Exemplo 13.2 – Parâmetro IN

```
 1  CREATE OR REPLACE PROCEDURE query_emp
 2    (in_employee_id       employees.employee_id%TYPE
 3    ,out_name         OUT employees.first_name%TYPE
 4    ,out_salary       OUT employees.salary%TYPE
 5    ,out_phone_number OUT employees.phone_number%TYPE)
 6  IS
 7  BEGIN
 8    SELECT first_name, salary    , phone_number
 9      INTO out_name  , out_salary, out_ phone_number
10      FROM employees
11     WHERE employee_id = in_employee_id;
12  EXCEPTION
13    WHEN NO_DATA_FOUND THEN
14      DBMS_OUTPUT.PUT_LINE('Empregado '||in_employee_id||' não existe');
15  END query_emp;
16  /

 1  DECLARE
 2    v_name   VARCHAR2(20);
 3    v_salary NUMBER;
 4    v_phone  VARCHAR2(20);
 5  BEGIN
 6    query_emp(100,v_gname,v_gsalary,v_gphone_number);
 7    DBMS_OUTPUT.PUT_LINE('Nome: '||v_name||',salario: '||v_salary||',fone: '||
v_phone);
 8  END;
 9  /
```

Exemplo 13.3 – Parâmetro OUT

```
 1  CREATE OR REPLACE PROCEDURE format_phone
 2    (in_out_phone IN OUT VARCHAR2)
 3  IS
 4    eParametroInvalido EXCEPTION;
 5  BEGIN
 6    IF LENGTH(in_out_phone) <> 10 THEN
 7      RAISE eParametroInvalido;
 8    END IF;
 9
10    in_out_phone := '('||SUBSTR(in_out_phone,1,2)||')'||
11                        SUBSTR(in_out_phone,3,4)||'-'||
12                        SUBSTR(in_out_phone,7,4);
13  EXCEPTION
14    WHEN eParametroInvalido THEN
15      DBMS_OUTPUT.PUT_LINE('Parametro Invalido');
16  END format_phone;
17  /

 1  DECLARE
 2   v_phone VARCHAR2(15):='9132182315';
 3  BEGIN
 4    format_phone(v_phone);
 5    DBMS_OUTPUT.PUT_LINE('Telefone formatado : '||v_phone);
 6  END;
 7  /
```

Exemplo 13.4 – Parâmetro IN OUT

Nos exemplos 13.2, 13.3 e 13.4 apresentam a codificação de algumas PROCEDURES e como estas devem ser acionadas. Lembramos que PROCEDURES podem ser chamadas apartir de outras PROCEDURES, FUNCTIONS ou TRIGGERS.

Obs.: Tipo de parâmetro %TYPE e %ROWTYPE são permitidos(Ver 11.3).

13.1.2 Especificando valores de parâmetros

Para uma procedure contendo parâmetros, existem dois métodos possíveis de especificar os valores dos parâmetros.

Tabela 13.2. Especificações de valores

Método	Descrição
Posicional	Os valores dos parâmetros deverão ser listados na ordem em que mesmos foram declarados. Neste método **todos** os valores devem ser especificados.
Associação Nomeada	Os valores dos parâmetros poderão ser listados em ordem arbitrária. A associação do valor com o nome é feita através do símbolo **=>** (nome_parametro => valor).

```
SQL>CREATE OR REPLACE PROCEDURE add_dept
 2    (in_department_name departments.department_name%TYPE DEFAULT 'UNKNOWN'
 3    ,in_location_id     departments.location_id%TYPE     DEFAULT 1400)
 4  IS
 5    nMax NUMBER;
 6  BEGIN
 7    SELECT MAX(department_id)+1
 8      INTO nMax
 9      FROM departments;
10
11    INSERT INTO departments
12      (department_id, department_name   , location_id )
13    VALUES
14      (nMax         , in_department_name, in_location_id );
15  END add_dept;
16  /

 1  BEGIN
 2    add_dept;
 3    add_dept('TREINAMENTO2',1500);
 4    add_dept(IN_DEPARTMENT_NAME => 'TREINAMENTO3',in_location_id => '1500');
 5    add_dept(IN_DEPARTMENT_NAME => 'TREINAMENTO4');
 6    add_dept(in_location_id => '1500');
 7  END;
 8  /
```

Exemplo 13.5 – Especificando Parâmetros

Na especificação de parâmetro por associação nomeada, não somos obrigados a elencar todos os parâmetros, neste caso o parâmetro omitido assumirá o valor default, se este existir, caso contrário assumirá valor NULL.

13.2 Stored Function

Uma FUNCTION é um bloco PL/SQL nomeado que pode ou não obter parâmetros e que deve ser referenciada por nome. Todos os parêmetros de uma FUNCTION devem ser do modo IN e por definição uma FUNCTION deve obrigatoriamente ter um valor de retorno associado a ela.

Para transformar um bloco PL/SQL em uma FUNCTION basta eliminar palavra DECLARE (se existir) e adicionar **CREATE [OR REPLACE] FUNCTION nome_funcao (par1,par2,...,parN) RETURN tipo_de_dado IS** ao inicio do bloco. Entre o nome da função e a palavra RETURN, entre parênteses, poderão ser especificados quantos parâmetros forem necessários, separados por vírgula.

```
CREATE [OR REPLACE] FUNCTION nome_funcao
  (parametro1 tipodado
  ,parametro2 tipodado,
  ,...
  ,parametroN tipodado)
    RETURN tipo_de_dado
IS
  -- declarações
BEGIN
  -- instruções e comandos
  RETURN valor_função;
EXCEPTON
  -- tratamentos de erro
END nome_funcao;
```

Figura 13.2 Sintaxe Create Função

O comando RETURN determina o valor da função, e além disso, atua como um comando EXIT e tudo que estiver após será desconsiderado. O valor de uma pode ser codificado várias vezes, no entanto apenas um será utilizado.

```
SQL>CREATE OR REPLACE FUNCTION f_qdt_empregados
2    (in_department_id departments.department_id%TYPE)
3      RETURN NUMBER
4  IS
5    nQdt NUMBER;
6  BEGIN
7    SELECT COUNT(*)
8      INTO nQdt
9      FROM employees
10     WHERE department_id = in_department_id;
11   RETURN nQdt;
12 END f_qdt_empregados;
13 /
```

Exemplo 13.6 – Function

É boa prática após o END de conclusão da FUNCTION especificar o nome desta como vemos na linha 12(doze) de exemplo 13.6.

13.2.1 Locais permitidos para uso de Functions

Funções podem ser chamadas de qualquer lugar onde uma expressão é válida.

Tabela 13.3 Locais Possíveis

Local/Chamada	Exemplo

Assinalando valor a uma variável	`BEGIN`
	`  v_salario := f_salario(19);`
	`END;`

Em uma expressão BOOLEAN (se o tipo da função é BOOLEAN)	`IF f_pagamento_aberto(234) THEN`

Assinalando valor DEFAULT a uma variável	`DECLARE`
	`  v_salario NUMBER:= f_salario(10);`
	`BEGIN`

Passando valor como parâmetro a uma procedure.	`BEGIN`
	`  proc_reajuste(in_id_employee  => 15`
	`  ,in_salário_atual => f_salario(15)`
	`  );`
	`END;`

Lista de seleção de um comando SELCT	`SELECT DEPARTMENT_NAME,qdt_empregados(DEPARTMENT_ID)` `  FROM departments`

Condição das cláusulas WHERE	`SELECT first_name`
	`  FROM employees`
	` WHERE f_salario(employee_id) > 10500`

Cláusula VALUES do comando INSERT	`INSERT INTO employees`
	`(EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY)`
	`VALUES`
	`(1500  , 'MARCUS'  , 'WILLIAM', f_salario(1015))`

Cláusula SET de comando UPDATE	`UPDATE employees`
	`  SET SALARY = f_salario(1015)`
	`WHERE EMPLOYEE_ID = 1500`

Na tabela 13.3, observamos que as FUNCTIONS podem ser chamadas apartir de blocos PS/SQL ou apartir de instruções DML.

14 Package

As PACKAGES são objetos que agrupam logicamente elementos de programação PL/SQL. Esses elementos podem ser tipos definidos pelo usuário, variáveis, exceções PL/SQL, cursores, PROCEDURES e FUNCTIONS.

Na maioria das vezes os packages têm duas partes, a especificação e o corpo. Em alguns casos o corpo é desnecessário.

```
SQL>CREATE OR REPLACE PACKAGE pacote_sem_corpo
  2  AS
  3     minimum_balance    CONSTANT REAL := 10.00;
  4     number_processed   NUMBER;
  5     insufficient_funds EXCEPTION;
  6  END pacote_sem_corpo;
  7  /

Pacote criado.

SQL>BEGIN
  2    pacote_sem_corpo.number_processed := 100;
  3    pacote_sem_corpo.number_processed := pacote_sem_corpo.number_processed + 10;
  4    DBMS_OUTPUT.PUT_LINE(pacote_sem_corpo.number_processed);
  5
  6    IF pacote_sem_corpo.number_processed > 100 THEN
  7      RAISE pacote_sem_corpo.insufficient_funds;
  8    END IF;
  9  EXCEPTION
 10    WHEN pacote_sem_corpo.insufficient_funds THEN
 11      DBMS_OUTPUT.PUT_LINE('insufficient_funds');
 12  END;
 13  /

Procedimento PL/SQL concluído com sucesso.
```

Exemplo 14.1 – Pacote sem corpo

No exemplo 16.1 ilustra um package sem corpo, com apenas 3 elementos, uma constante, uma variável e uma exceção. Mostra também algumas maneiras de manipular esses elementos. PACKAGES sem corpo são úteis quando se deseja criar variáveis ou elementos de escopo global

Obs.: Variáveis de pacote não podem ser usadas diretamente em instruções SQL. Somente uma função pública do pacote poderá ser usada em instruções SQL e retornar o valor de uma variável, mesmo que esta variável seja também pública.

A especificação é a interface do package, e nele é declarado o conteúdo público, quais os elementos de programação que poderão ser referenciada por outros objetos do banco de dados. Os subprogramas (procedure e fuctions) deverão estar especificados com o nome e todos os seus parâmetros

No corpo, será definido o conteúdo privado do package que deverá ser referenciados e acessados apenas pelos elementos privados do package. É no corpo que se encontra a codificação completa dos subprogramas, assim como as consultas associadas aos cursores.

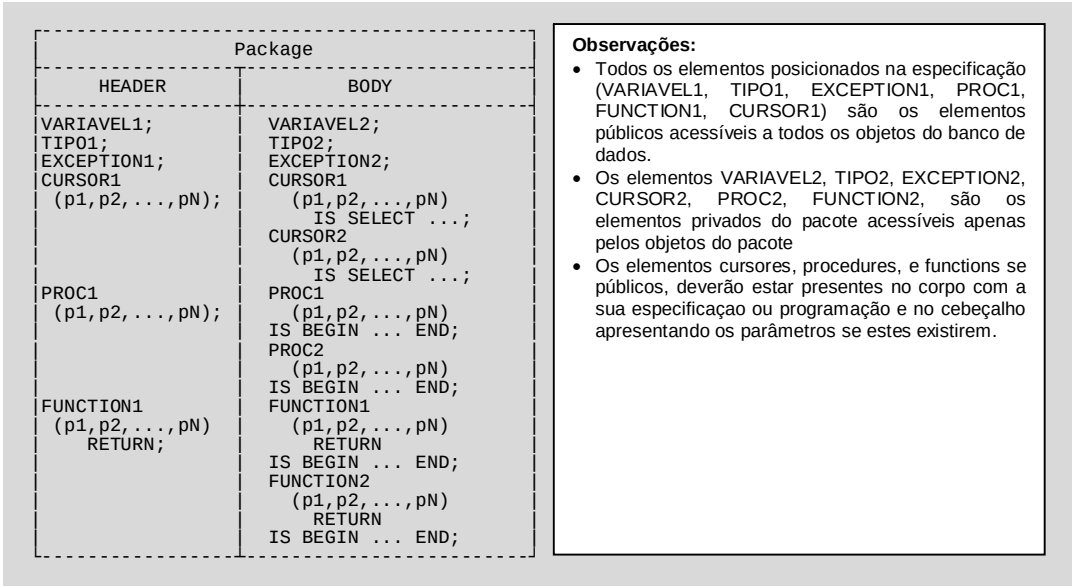


Figura 14.1 – Elementos de um package.

O corpo e a especificação são considerados objetos diferentes e devem ser criados em instruções diferentes.

```
SQL>CREATE OR REPLACE PACKAGE pck_emp
  2  IS
  3    PROCEDURE query_emp
  4      (in_employee_id       employees.employee_id%TYPE
  5      ,out_name         OUT employees.first_name%TYPE
  6      ,out_salary       OUT employees.salary%TYPE
  7      ,out_phone_number OUT employees.phone_number%TYPE);
  8  END pck_emp;
  9  /

SQL>CREATE OR REPLACE PACKAGE BODY pck_emp
  2  IS
  3    PROCEDURE query_emp
  4      (in_employee_id       employees.employee_id%TYPE
  5      ,out_name         OUT employees.first_name%TYPE
  6      ,out_salary       OUT employees.salary%TYPE
  7      ,out_phone_number OUT employees.phone_number%TYPE)
  8    IS
  9    BEGIN
 10      SELECT first_name, salary    , phone_number
 11        INTO out_name  , out_salary, out_phone_number
 12        FROM employees
 13       WHERE employee_id = in_employee_id;
 14    EXCEPTION
 15      WHEN NO_DATA_FOUND THEN
 16        DBMS_OUTPUT.PUT_LINE('Empregado '||in_employee_id||' não existe');
 17    END query_emp;
 18  END pck_emp;
 19  /

SQL>DECLARE
  2    aEmployee_id  employees.employee_id%TYPE:=206;
  3    aFirst_name   employees.first_name%TYPE;
  4    aSalary       employees.salary%TYPE;
  5    aPhone_number employees.phone_number%TYPE;
  6  BEGIN
  7    pck_emp.query_emp(aEmployee_id,aFirst_name,aSalary,aPhone_number);
  8    DBMS_OUTPUT.PUT_LINE('aFirst_name   --> '||aFirst_name);
  9    DBMS_OUTPUT.PUT_LINE('aSalary       --> '||aSalary);
 10    DBMS_OUTPUT.PUT_LINE('aPhone_number --> '||aPhone_number);
 11  END;
 12  /

aFirst_name --> William
aSalary --> 8300
aPhone_number --> 515.123.8181
```

Exemplo 14.2 – Pacote com apenas uma procedure e como utilizá-lo

```
SQL>CREATE OR REPLACE PACKAGE pck_employee
  2  IS
  3     FUNCTION emp_qtd
  4       (in_department_id  employees.department_id%TYPE)
  5         RETURN NUMBER;
  6     PROCEDURE fire_employee
  7       (in_emp_id NUMBER);
  8  END pck_employee;
  9  /

Pacote criado.

SQL>CREATE OR REPLACE PACKAGE BODY pck_employee
  2  IS
  3     nGlobal NUMBER;
  4
  5     PROCEDURE del_job
  6       (in_Employee_Id employees.Employee_Id%TYPE) IS
  7     BEGIN
  8       DELETE job_history
  9        WHERE employee_id = in_Employee_Id;
 10     END del_job;
 11
 12     FUNCTION emp_qtd
 13       (in_department_id  employees.department_id%TYPE)
 14         RETURN NUMBER
 15     IS
 16       nResultado NUMBER;
 17     BEGIN
 18       SELECT COUNT(*)
 19         INTO nResultado
 20         FROM employees
 21        WHERE department_id = in_department_id;
 22
 23       RETURN nResultado;
 24     END emp_qdt;
 25
 26     PROCEDURE fire_employee
 27       (in_emp_id NUMBER)
 28     IS
 29       nQdt NUMBER;
 30     BEGIN
 31       -- obter o departamento
 32       SELECT department_id
 33         INTO nGlobal
 34         FROM employees
 35        WHERE employee_id = in_emp_id;
 36
 37       del_job(in_emp_id);
 38
 39       -- delatar empregado
 40       DELETE employees
 41        WHERE employee_id = in_emp_id;
 42
 43       SELECT emp_qtd(nGlobal)
 44         INTO nQdt
 45         FROM DUAL;
 46
 47       DBMS_OUTPUT.put_line('Agora existem apenas '||
 48                            nQdt||
 49                            'no departamento '||nGlobal
 50                            );
 51     END fire_employee ;
 52
 53  END pck_employee;
 54  /

Corpo de Pacote criado.
```

Exemplo 14.3 – Uma procedure publica e uma function privada

Obs.: Se uma function é utilizada em uma instrução SQL (exemplo 16.3, linha 43) esta deve obrigatoriamente estar declarada na especificação do pacote.

Quando uma sessão manipula o valor de uma variável de pacote, este se torna permanente ao longo da existência desta sessão. Para cada sessão que manipula o valor de uma variável de pacote, existe uma área de memória que armazena os valores desta variável. O valor manipulado por uma sessão não interfere no valor manipulado por outra. O valor persiste para a sessão desde o momento em que é feita a primeira manipulação, até a finalização da sessão ou uma remanipulação deste valor. Essa característica também é atribuída a conjuntos ativos gerados por cursores.

15 Database Trigger

As triggers assim como as stored procedures são armazenadas do banco de dados e podem ser compostas de instruções SQL e PL/SQL. Entretanto stored procedure e triggers diferem na forma como estes são acionados. Uma stored procedure é explicitamente acionda por um usuário, aplicação ou trigger. As triggers são implicitamente disparadas pelo Oracle quando um determinado evento ocorre. O disparo da trigger independe do usuário ou aplicação que gerou o evento.

Uma database trigger é subprograma associado a uma tabela, view ou evento. A traigger pode ser acionada uma vez quando um determinado evento ocorre ou várias vezes para cada linha afetada por uma instrução INSERT, UPDATE ou DELETE. A trigger pode ser aciondad após um determindo evento para registrá-lo ou efetuar alguma atividade posterior, ou pode ser acionado antes de um evento para previnir operações indevidas ou ajustar os novos dados para que estes estejam de acordo com a regra de negócio.

Como principais motivos para o uso de database trigger, podemos citar os seguintes:

- geração automática de valores de colunas derivados;
- prevenção de transações inválidas;
- reforçar regras de negócio complexas;
- prover auditoria;
- gerar estatísticas sobre acesso às tabelas;
- prover log de transações.

É escopo deste capítulo apenas as triggers associadas a tabelas, no entanto, existem as INSTED OF triggers voltadas para DML's disparados contra objetos view e triggers de eventos de sistema voltadas para atividade de administração do banco de dados.

15.1 Elementos

Antes de se codificar uma trigger é interessante decidir, segundo as necessidades de regra de negócio, quais os elementos desta futura trigger.

Tabela 15.1. Elementos triggers

Componentes	Descrição	Valores
Tempo	Quando o trigger dispara em relação ao evento de acionamento (DML)	• BEFORE • AFTER
Evento de acionamento	Quais operações de manipulação de tabela (DML) disparam a trigger	• INSERT • UPDATE • DELETE
abrangência da trigger	Quantas vezes o corpo da trigger será executado	• de linha (for each row) • de instrução(*)
Corpo da trigger	Que ações serão executadas	Bloco PL/SQL

(*) Opção default

Quanto à quantidade de vezes que a trigger será acionda, podemos afirmar que o comportamento default das por instrução e quando o DML acionador de trigger afeta apenas uma linha, tanto o trigger de instrução quanto o trigger de linha dispararão apenas uma vez. Ao passo que, quando o DML acionado afeta

várias linhas o trigger de instrução será executado apenas uma vez enquanto que o trigger de linha será executado na mesma quantidade das linhas afetadas.

```
     SQL>CREATE OR REPLACE TRIGGER jobs_biud
       2    BEFORE INSERT OR UPDATE OR DELETE ON jobs
       3  BEGIN
       4    IF TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18' THEN
       5     RAISE_APPLICATION_ERROR(-20205,'Alterações são permitidas apenas no horário de
expediente');
       6    END IF;
       7  END jobs_biud;
```

Exemplo 15.1 – Trigger

***Obs**: O exemplo 17.1 é uma trigger que será disparada apenas uma vez quando um INSERT, UPDATE ou DELETE for efetuado na tabela **jobs**.*

15.2 Predicado Condicional

Quando programamos uma trigger para vários eventos e temos a necessidade de identifica qual evento disparou a trigger, poderemos usar os predicados condicionais que são funções booleanas que podem ser utilizadas para determinar a operação que disparou o trigger.

```
     SQL>CREATE OR REPLACE TRIGGER employees_biud
       2  BEFORE INSERT OR UPDATE OR DELETE ON employees
       3  BEGIN
       4    IF (TO_CHAR (SYSDATE,'HH24') NOT BETWEEN '08' AND '18') THEN
       5      IF DELETING THEN
       6       RAISE_APPLICATION_ERROR(-20502,'Deleções na tabela de empregados apenas no
horario normal');
       7      ELSIF INSERTING THEN
       8        RAISE_APPLICATION_ERROR(-20502,'Inserções na tabela de empregados apenas no
horario normal');
       9      ELSIF UPDATING('SALARY') THEN
      10     RAISE_APPLICATION_ERROR(-20502,'Alterações no salário apenas no horario
normal');
      11      ELSE
      12        RAISE_APPLICATION_ERROR(-20504,'Alterações nos empregados apenas no horario
normal');
      13      END IF;
      14    END IF;
      15  END employees_biud;
```

Exemplo 15.2 – Predicado

15.3 Trigger de Linha

Uma trigger de linha é disparada uma vez para cada linha afetada pela instrução DML. Uma trigger de linha é identificada pela cláusula **FOR EACH ROW**.

15.3.1 Qualificadores (:new, :old)

Em uma trigger de linha, existe uma forma de acessar os valores dos campos que estão sendo processados atualmente, atraves dos identificadores :new, :old. O compilador PL/SQL irá tratá-los como ***tabela_da_trigger%ROWTYPE***.

```
SQL>CREATE OR REPLACE TRIGGER employees_biur
  2    BEFORE  INSERT OR UPDATE ON employees
  3      FOR EACH ROW
  4      BEGIN
  5        IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP')) THEN
  6          IF :NEW.salary > 15000 THEN
  7            RAISE_APPLICATION_ERROR (-20202,'Este empregado não pode receber este
valor');
  8          END IF;
  9       END IF;
 10  END employees_biur;
```

Exemplo 15.3 – Qualificadores

Tabela 15.2 – Qualificadores

DML	:old	:new
INSERT	NULO	Valores Novos
DELETE	Valores antigos	NULO
UPDATE	Valores antigos	Valores Novos

```
SQL>create table dept_audit
  2  (userid          varchar2(30)
  3  ,timestamp       date
  4  ,tipo_dml        CHAR(1)
  5  ,old_dept_id     NUMBER
  6  ,old_name        varchar2(30)
  7  ,old_manager_id  number
  8  ,old_location_id number
  9  ,new_dept_id     NUMBER
 10  ,new_name        varchar2(30)
 11  ,new_manager_id  number
 12  ,new_location_id number
 13  )

SQL>CREATE OR REPLACE TRIGGER department_aiudr
  2    AFTER INSERT OR UPDATE OR DELETE ON departments
  3      FOR EACH ROW
  4  DECLARE
  5    v_DML dept_audit.tipo_dml%TYPE;
  6  BEGIN
  7    IF INSERTING THEN
  8      v_DML := 'I';
  9    ELSIF DELETING THEN
 10      v_DML := 'D';
 11    ELSIF UPDATING THEN
 12      v_DML := 'U';
 13    END IF;
 14
 15    INSERT INTO dept_audit
 16      (userid              , timestamp            , tipo_dml
 17      ,old_dept_id         , old_name             , old_manager_id
 18      ,old_location_id     , new_dept_id          , new_name
 19      ,new_manager_id      , new_location_id
 20      )
 21    VALUES
 22      (USER                , SYSDATE              , v_DML
 23      ,:OLD.department_id  ,:OLD.department_name  ,:OLD.manager_id
 24      ,:OLD.location_id    ,:NEW.department_id    ,:NEW.department_name
 25      ,:NEW.manager_id     ,:NEW.location_id
 26      );
 27  END;
```

Exemplo 15.4 – Auditagem por trigger

Obs.: Os qualificadores :old, :new, estão disponíveis apenas nas triggers de linha.

```
SQL>CREATE OR REPLACE TRIGGER regions_bir
  2    BEFORE INSERT ON regions
  3      FOR EACH ROW
  4  DECLARE
  5    v_region_id regions.region_id%TYPE;
  6  BEGIN
  7    SELECT MAX(region_id)+1
  8      INTO v_region_id
  9      FROM regions;
 10
 11    :NEW.region_id := v_region_id;
 12  END regions_bir;
```

Exemplo 15.5 – Autoincremento.

15.3.2 Cláusula WHEN

Nas triggers de linha, podemos restringir a ação da trigger segundo uma condição, onde a mesma será disparada apenas para as linhas que satisfaçam a condição prevista.

```
SQL>CREATE OR REPLACE TRIGGER derive_commission_pct
  2    BEFORE INSERT OR UPDATE OF salary ON employees
  3    FOR EACH ROW
  4    WHEN (NEW.job_id = 'SA_REP')
  5  BEGIN
  6    IF INSERTING THEN
  7       :NEW.commission_pct := 0;
  8    ELSIF :OLD.commission_pct IS NULL THEN
  9       :NEW.commission_pct := 0;
 10    ELSE
 11      :NEW.commission_pct := :OLD.commission_pct + 0.05;
 12    END IF;
 13  END;
 14  /
```

Exemplo 15.6 – Cláusula WHEN

No exemplo 17.6 além da cláusula condicional, percebemos na linha 2 cláusula OF seguida de um campo (salary) da tabela (employees) associada à trigger. Isso indica que a trigger só será disparada quando o update afetar a coluna indicada. Caso a cláusula OF estiver omitida a trigger do exemplo 17.6 será disparada indepedente da coluna afetada.

Parte IV – Apêndices

A – Oracle Net

É tendência, entre os envolvidos em desenvolvimento de sistemas, transferir para o DBA a responsabilidade sobre a configuração de acesso ao banco de dados nos equipamentos de trabalho. Acreditamos que cada programador deve ter a capacidade de gerenciar as configurações de acesso aos bancos de dados. Para isso o programador deverá conhecer o Oracle Net.

O Oracle Net é uma camada de software que reside no cliente e no servidor de banco de dados Oracle. É responsável por estabelecer e manter conexões entre a aplicação cliente e o servidor, bem como, intercambiar mensagens entre os mesmos, usando protocolos de comunicação disponíveis no mercado.

A.1 - Arquitetura

A figura A.1 mostra com a arquitetura no Oracle Net e como funciona o estabelecimento de uma conexão com um banco de dados Oracle. Uma aplicação primeiramente envia uma requisição de conexão ao banco correto após a leitura dos arquivos de configuração do Oracle Net (sqlnaet.ora, tnsnames.ora) No servidor existe um elemento chamado listener que gerencia e libera as conexões com o banco de dados.

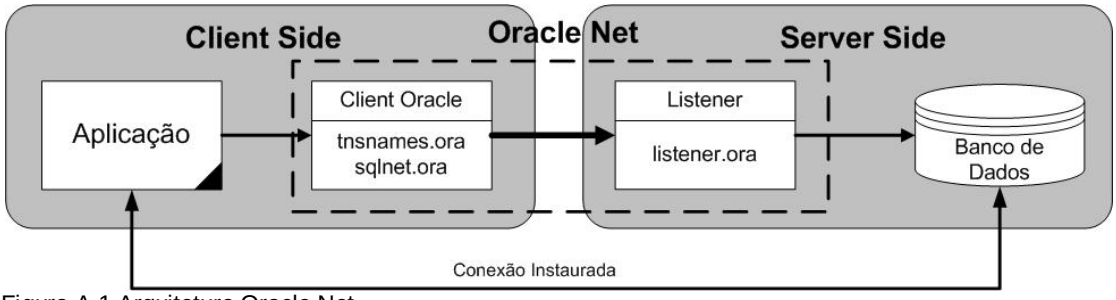


Figura A.1 Arquiteture Oracle Net

Para se estabelecer uma comunicação com um banco de dados Oracle devem ser fornecidas apenas três informações, o nome do usuário, senha do usuário e o *net service name*. O terceiro item é um *alias* que encapsula todas as informações necessárias para acessar um banco de dados Oracle. A rigor, para acessar um banco, são necessários, a identificação do equipamento (Host Name ou endereço IP) onde se encontra o banco de dados, o protocolo de comunicação que este equipamento está apto a usar, o número da porta que será usada para receber as requisições de banco de dados e o nome do banco de dados (service name) instalado neste equipamento. Todos estes aspectos são resolvidos em um *connect descriptor,* no qual todas estas informações estão contempladas.

Existem quatro formas de se obter as informações de um *connect descriptor (localnaming, Oracle Names Server, Hostnaming, Exteral Naming Server).* Neste curso, usaremos apenas localnaming.

A.2 - Configuração

É no arquivo sqlnet.ora, através do parâmetro NAMES.DIRECTOTY_PATH, que é definido a forma de obtenção dos *connect descriptor.* Os valores válidos são CDS, HOSTNAME, NDS, NIS, ONAMES, TNSNAMES. Podem existir combinações de valores para este parâmetro, no entanto para o nosso escopo de trabalho o valor TNSNAMES, que corresponte a localnaming, é obrigatório.

```
NANES.DIRECT_PATH=(TNSNAMES)
```

Exemplo A.1 – Conteúdo arquivo sqlnet.ora

Por sua vez, é no arquivo tnsnames.ora que são registrados dos os net service names disponíveis para o cliente oracle e seu respectivo *connect descriptor.*

```
 ORCL =                  # Primeiro net service Name
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = hostname)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = ORCL)
    )
  )
MARCUSDB =                # Segundo net service Name
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = MARCUS)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = MARCUS)
    )
  )
BELEM =                   # Terceiro net service Name
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 10.1.20.100)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SID = BELEM)
    )
  )
```

Exemplo A.2 – Conteúdo arquivo tnsnames.ora

Neste conteúdo exemplificativo de TNSNAMES.ORA encontramos três net service names que juntamente com seus connect descriptor dão acessos aos bancos de dados Oracle. As informações deste exemplo estão compiladas na tabela 1.1.

Tabela A.1. Informações compiladas

NetServiceName	PROTOCOL	HOST	PORT	SERVICE_NAME
ORCL	TCP	hostname	1521	ORCL
MARCUSDB	TCP	MARCUS	1521	MARCUS
BELEM	TCP	10.1.20.100	1521	BELEM

A variável de ambiente TNS_ADMIN tem como valor o local/pasta onde se encontra o arquivo tnsnames.ora que será usado nas conexões Oracle. Case esta variável não exista será o local default que são $ORACLE_HOME/network/admin para UNIX e %ORACLE_HOME%\network\admin para Windows

A.3 - Principais Problemas

Na figura A.2 estão listadas as etapas executadas quando uma conexão com o banco de dados Oracle é estabelecida. O bom entendimento destas etapas se faz necessário para se enfrentar os problemas de conexão com o banco de dados Oracle. Estão listados os principais problemas de conexão.

```
1.  A aplicação envia ao Oracle Net as seguintes informações: usuário,
    senha e NetServiceName
2.  O Oracle Net faz busca do NetServiceName informado no arquivo
    tnanames.ora para obter os dados do connect descriptor.
3.  O computador informado na diretiva HOST é acionado.
4.  Já no servidor, o listener procura nos seus arquivos de configuração
    o SERVICO_NAME ou SID informado no connect descriptor usado.
5.  O listener estabelece a conexão
```

Figura A.2 – Etapas no estabelecimento de conexão

ORA-12154: TNS: could not resolve service name
Motivos
- A leitura do *connect descriptor* não foi executada com sucesso

Ação
- Assegurar que "TNSNAMES" está listado com um dos valores do parâmetro NAMES.DIRECTORY_PATH no arquivo SQLNET.ORA(Oracle Net Profile)
- Verificar se o arquivo TNSNAMES.ORA existe e está no local adequado e acessível (variável de ambiente .TNS_ADMIN)
- Verificar se o net service name usado com identificador de conexão existe no arquivo TNSNAMES.ORA
- Assegurar que não existam erros no arquivo TNSNAMES.ORA, como parênteses não fechados

ORA-03505: TNS: Failed do resolve name
Motivos
- O net service name usado com identificador de conexão não foi encontrado no arquivo TNSNAMES.ORA

Ação
- Idem ORA-12154

ORA-12535 : Connect failed because target host or object does not exist
Motivos
- O equipamento indicado na diretiva HOST não está ativo (desligado).
- O equipamento indicado na diretiva HOST não foi alcançado (problema de rede).

Ação
- Ativar o servidor de Banco de dados.
- Correção na diretiva HOST para o endereço de IP ou HostName correto.

ORA-12535 TNS:operation timed out
Motivos
- O equipamento indicado na diretiva HOST não está ativo (desligado).
- O equipamento indicado na diretiva HOST não foi alcançado (problema de rede).

Ação
- Ativar o servidor de Banco de dados.
- Correção na diretiva HOST para o endereço de IP ou HostName correto.

ORA-12541: TNS: no listener
Motivos
- O listener do servidor está inativo impossibilitado de estabelecer novas conexões
- O equipamento indicado na diretiva HOST existe, foi alcançado, porém nele não está instalado SGDB Oracle.

Ação
- Acionar o DBA para ativar o listener.
- Correção na diretiva HOST do connect descriptor para o equipamento correto.

ORA-12514: Listener could not resolve SERVICE_NAME given in connect description
Motivos
- O valor da diretiva SERVICE_NAME está incorreto. O NetServiceName existe, o HOST foi alcançado, porém o SERVICE_NAME informado não é esperado pelo listener.
- Se o banco estiver na versão 8i ou inferior

Ação
- Confirmar com o DBA qual o valor correto para a diretiva SERVICE_NAME
- Passar a usar a diretiva SID ao invez de SERVICE_NAME

ORA-01033: ORACLE initialization or shutdown in progress
Motivos
- O banco está em manutenção

Ação
- Perguntar para o DBA o que ele está fazendo

ORA-01034: ORACLE not avaliable
Motivos
- O banco está fora do ar e indisponível

Ação
- Correr e avisar o DBA.

ORA-01017: invalid username/password; logon denied
Motivos
- Usuário ou senha inválidos

Ação
- Confirmar como DBA as informações de usuário e senha.

B – Schema HR (Human Resource)