

Sumário

01 – Introdução ao PL/SQL.....	5
Objetivo do Curso.....	6
Conceitos Básicos.....	7
O que é PL/SQL?.....	7
Por que aprender a PL/SQL?.....	7
Blocos PL/SQL.....	7
Comentários.....	8
Execução de Blocos.....	9
Ambiente de Execução.....	10
Ferramentas para execução de declarações SQL e desenvolvimento PL/SQL.....	10
SQL *Plus.....	10
PL/SQL Developer	12
TOAD.....	15
Oracle SQL Developer.....	18
Declaração de variáveis.....	23
Explorando os datatypes.....	23
Datatypes escalares (tipos de dados simples).....	23
Tipo Registro.....	26
O Escopo de uma variável.....	28
Usando tags para identificar um variável.....	29
Valores Nulos.....	29
Tratamento de exceções.....	31
Funcionamento geral.....	31
Capturando uma exceção.....	32
Exceções pré-definidas da Oracle.....	32
Erros indefinidos da Oracle.....	33
Erros definidos pelo usuário.....	34
SQLCODE e SQLERRM.....	35
O procedimento raise_application_error.....	36
Regras de escopo de exceção.....	37
Propagando as exceções.....	37
Exercícios propostos.....	38
02 – Estruturas PL/SQL.....	39
Cenário para prática.....	40
Modelo lógico de entidade e relacionamento.....	40
Modelo físico de entidade e relacionamento.....	41
Declarações IF e LOOPS.....	43
A declaração IF.....	43
A declaração IF...THEN...ELSE.....	43
A declaração IF...ELSIF.....	44
Loop Simples.....	45
Criando um loop REPEAT...UNTIL.....	45
Loop FOR.....	45
Loop WHILE.....	46
Qual loop devo usar?.....	47
Utilização de Cursores.....	48
Conceitos básicos.....	48
Cursores Explícitos.....	48
Declarando um cursor.....	48
Abrindo um Cursor.....	49
Recuperando dados de um Cursor.....	50
Fechando o Cursor.....	51

Atributos de cursores explícitos.....	52
Cursores, registros e o atributo %ROWTYPE.....	53
Cursores explícitos automatizados (LOOP Cursor FOR).....	55
Passando parâmetros para cursores.....	57
Cursores implícitos.....	57
Atributos do cursor implícito.....	58
Variáveis de cursor.....	59
Blocos anônimos, procedimentos e funções.....	61
Exercícios propostos.....	64
03 – Stored Procedures.....	65
Procedimentos armazenados (Stored Procedure).....	66
Por que usar os procedimentos?.....	66
Procedimentos versus funções.....	66
Procedimentos.....	67
Funções.....	68
Manutenção de procedimentos armazenados.....	70
Usando os parâmetros.....	70
Definições de parâmetro.....	71
Dependências de procedimentos.....	71
Segurança da invocação de procedimento.....	71
Pacotes.....	72
Vantagens do uso de pacotes.....	72
Estrutura de um pacote.....	72
A especificação do pacote.....	72
O corpo do pacote.....	73
Utilizando os subprogramas e variáveis de um pacote.....	75
Manutenção dos pacotes.....	76
Estados de um pacote.....	76
Recompilando pacotes.....	76
Triggers.....	77
O que é um Trigger?.....	77
Triggers DML.....	77
Tipos de triggers DML.....	78
Ativando e desativando triggers.....	79
Exercícios Propostos.....	80
04 – Collections.....	82
Coleções.....	83
Tabelas por índice.....	83
Declarando uma tabela por índice.....	83
Manipulando uma tabela por índice.....	84
Tabelas aninhadas.....	87
Declarando uma tabela aninhada.....	87
Manipulando tabelas aninhadas.....	88
Arrays de tamanho variável.....	91
Declarando e inicializando um VARRAY.....	92
Adicionando e removendo dados de um VARRAY.....	92
Métodos de tabela da PL/SQL.....	94
Executando declarações SELECT em uma coleção.....	95
Criando um cursor explícito a partir de uma coleção.....	98
O bulk binding.....	99
Usando BULK COLLECT.....	100
Usando FORALL.....	102
Tratamento de exceções para as coleções.....	103
Exercícios Propostos.....	105
05 – Tópicos avançados: PL/SQL.....	106

SQL Dinâmico.....	107
SQL Dinâmico em cursores.....	110
Stored Procedure com transação autônoma.....	110
Tabela Função PL/SQL (PIPELINED).....	114
Montando uma função PIPELINED.....	115
Utilizando uma função PIPELINED em uma declaração SQL.....	117
UTL_FILE: Escrita e leitura de arquivos no servidor.....	118
Procedimentos e funções de UTL_FILE.....	119
Gerando um arquivo com informações extraídas do banco de dados.....	121
Recuperando informações de um arquivo.....	122
TEXT_IO.....	122
Exercícios Propostos.....	124
06 – Tópicos avançados: SQL e funções incorporadas do Oracle Database.....	125
SQLs avançados para usar com PL/SQL.....	126
O outer join do Oracle.....	126
ROWNUM.....	127
Comando CASE no SELECT.....	128
SELECT combinado com CREATE TABLE e INSERT.....	128
CREATE TABLE AS SELECT.....	129
INSERT ... SELECT.....	129
MERGE.....	129
GROUP BY com HAVING.....	131
Recursos avançados de agrupamento: ROLLUP e CUBE.....	132
ROLLUP.....	132
CUBE.....	132
Consultas hierárquicas com CONNECT BY.....	133
Funções incorporadas do Oracle Database.....	134
07 – Dicas de performance e boas práticas em SQL.....	137
Introdução.....	138
O Otimizador Oracle.....	138
Otimizador baseado em regra (RBO).....	138
Otimizador baseado em custo (CBO).....	139
Variáveis de ligação (Bind Variables).....	139
SQL Dinâmico.....	141
O uso de índices.....	142
Colunas indexadas no ORDER BY.....	143
EXPLAIN PLAN.....	143
O AUTOTRACE do SQL*Plus.....	144
A cláusula WHERE é crucial!.....	147
Use o WHERE ao invés do HAVING para filtrar linhas.....	148
Especifique as colunas principais do índice na cláusula WHERE.....	148
Evite a cláusula OR.....	148
Cuidado com “Produto Cartesiano”.....	148
SQLs complexas.....	149
Quando usar MINUS, IN e EXISTS.....	149
Evite o SORT.....	150
EXISTS ao invés de DISTINCT.....	150
UNION e UNION ALL.....	150
Cuidados ao utilizar VIEWS.....	151
Joins em VIEWS complexas.....	151
Reciclagem de VIEWS.....	151
Database Link.....	151
Aplicativos versus Banco de Dados.....	151
Utilize o comando CASE para combinar múltiplas varreduras.....	151
Utilize blocos PL/SQL para executar operações SQL repetidas em LOOPS de sua aplicação.....	152

Use a clausula RETURNING em declarações DML:.....	153
Exercícios Propostos.....	154

01 – Introdução ao PL/SQL

1. **Objetivo do curso;**
2. **Conceitos básicos;**
3. **Declaração de variáveis;**
4. **Tratamento de exceções;**

Objetivo do Curso

O objetivo do curso é apresentar aos desenvolvedores, recursos da linguagem PL/SQL que poderão auxiliá-los na manipulação de dados armazenados no banco de dados Oracle.

Grande parte dos recursos apresentados, como cursores e coleções, serão explorados dentro de unidades de programas, que na PL/SQL são representadas por:

- Blocos anônimos;
- *Stored procedures*;
- e *Packages*;

Conceitos básicos como declaração de variáveis, tipos e tratamento de exceções, também serão mostrados e ao final do curso, será apresentado um tópico de otimização e boas práticas de SQL.

Conceitos Básicos

O que é PL/SQL?

PL/SQL (Procedural Language/Structured Query Language) é uma extensão de linguagem de procedimentos desenvolvida pela Oracle para a SQL Padrão, para fornecer um modo de executar a lógica de procedimentos no banco de dados.

A SQL em si é uma linguagem declarativa poderosa. Ela é *declarativa* pois você descreve resultados desejados, mas não o modo como eles são obtidos. Isso é bom porque você pode isolar um aplicativo dos detalhes específicos de como os dados são armazenados fisicamente. Um programador competente em SQL também consegue eliminar uma quantidade grande de trabalho de processamento no nível do servidor por meio do uso criativo da SQL.

Entretanto, existem limites para aquilo que você pode realizar com uma única consulta declarativa. O mundo real geralmente não é tão simples como desejaríamos que ele fosse. Os desenvolvedores quase sempre precisam executar várias consultas sucessivas e processar os resultados específicos de uma consulta antes de passar para a consulta seguinte. Isso cria dois problemas para um ambiente cliente/servidor:

1. A lógica de procedimentos, ou seja, a definição do processo, reside nas máquinas do cliente;
2. A necessidade de olhar os dados de uma consulta e usá-los como a base da consulta seguinte resulta em uma quantidade maior de tráfego de rede.

A PL/SQL fornece um mecanismo para os desenvolvedores adicionarem um componente de procedimento no nível de servidor. Ela foi aperfeiçoada a ponto de os desenvolvedores agora terem acesso a todos os recursos de uma linguagem de procedimentos completa no nível de servidor. Ela também forma a base da programação do conjunto da Oracle de ferramentas de desenvolvimento para cliente/servidor, como Forms & Reports.

Por que aprender a PL/SQL?

Independente da ferramenta *front-end* que está usando, você pode usar a PL/SQL para executar o processamento no servidor em vez de executá-lo no cliente. Você pode usar a PL/SQL para encapsular regras de negócios e outras lógicas complicadas. Ela fornece modularidade e abstração. Você pode usá-la nos gatilhos (*triggers*) do banco de dados para codificar restrições complexas, as quais reforçam a integridade do banco de dados, para registrar alterações e para replicar os dados.

A PL/SQL também pode ser usada com os procedimentos armazenados e as funções para fornecer maior segurança ao banco de dados. Finalmente, ela fornece um nível de independência da plataforma. O Oracle está disponível para muitas plataformas de hardware, mas a PL/SQL é igual em todas elas.

Blocos PL/SQL

A PL/SQL é chamada de linguagem *estruturada em blocos*. Um bloco PL/SQL é uma unidade sintática que pode conter código de programa, declarações de variáveis, *handlers* de erro, procedimento, funções e até mesmo outros blocos PL/SQL.

Cada unidade de programa do PL/SQL consiste em um ou mais blocos. Cada bloco pode ser completamente separado ou aninhado com outros blocos.

Um bloco PL/SQL é formado por três sessões:

1. Declarativa (opcional);
2. Executável;
3. Exceções (também opcional).

A área declarativa, indicada pela palavra chave `DECLARE`, é a parte inicial do bloco e reservada para declaração de variáveis, constantes, tipos, exceções definidas por usuários, cursores e subrotinas.

A seção executável contém os comandos SQL e PL/SQL que irão manipular dados do banco e é iniciada pela palavra chave `BEGIN`. Quando o bloco não possui a sessão de exceções, a seção executável é finalizada pela palavra chave `END`, seguida do ponto e vírgula, do contrário, é finalizada pelo início da sessão de exceções, que é indicada pela palavra chave `EXCEPTION`.

A sessão executável, iniciada pelo `BEGIN` e finalizada pelo `END` seguido do ponto e vírgula, é a única sessão obrigatória em um bloco PL/SQL.

O Bloco anônimo é o tipo mais simples de um bloco PL/SQL e possui a estrutura mostrada na figura abaixo:

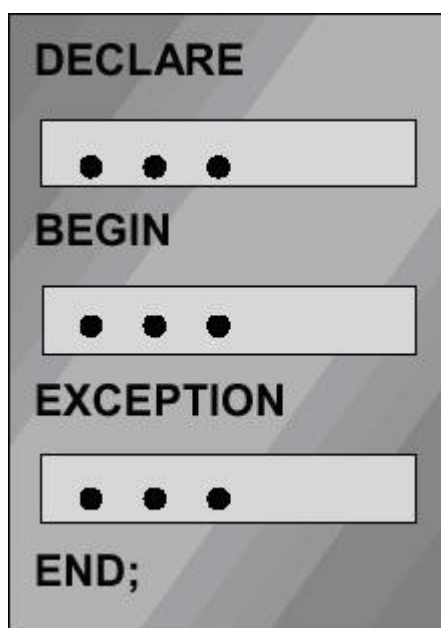


Figura 01: Estrutura de bloco anônimo do PL/SQL.

No decorrer do curso, veremos um outro tipo de bloco PL/SQL: *Stored Procedure*.

Comentários

Os comentários são utilizados para documentar cada fase do código e ajudar no *debug*.

Os comentários são informativos e não alteram a lógica ou o comportamento do programa. Quando usados de maneira eficiente, os comentários melhoram a leitura e as futuras manutenções do programa.

Em PL/SQL, os comentários podem ser indicados de duas formas:

- Usando os caracteres `--` no início da linha a ser comentada;
- ou concentrando o texto (ou código) entre os caracteres `'/*'` e `'*/'`.

Abaixo temos o exemplo de um bloco PL/SQL que não faz nada, mas demonstra o uso de comentários:

```
DECLARE  
-- Declaração de variáveis
```



```
BEGIN
  /*O programa não executa nada*/
  NULL;
EXCEPTION --Aqui inicia a sessão e exceções
  /*A área de exceção também não faz nada*/
  NULL;
END;
```

Utilizar '--' em comentários de uma linha e '/* */' em comentários de mais de uma linha, é uma **boa prática** de programação.

Execução de Blocos

Na execução de um bloco PL/SQL:

- Colocar um ponto e vírgula(;) no final de cada comando SQL ou PL/SQL;
- Quando um bloco é executado com sucesso, sem erros de compilação ou erros não tratados, a seguinte mensagem será mostrada:

PL_SQL procedure successfully completed

- Não colocar ponto e vírgula após as palavras chaves DECLARE, BEGIN e EXCEPTION;
- END e qualquer outro comando PL/SQL e SQL são terminados por ponto e vírgula;
- Comandos podem ser colocados na mesma linha, mas não é recomendado, pois dificulta a visualização do código;

Abaixo segue o exemplo de um bloco PL/SQL, para ser executado no SQL*Plus, que imprime na tela o total de produtos cadastrados no banco de dados:

```
DECLARE
  -- Declaração de variáveis
  vCount INTEGER;
BEGIN
  -- Recuperar quantidade de produtos cadastrados
  SELECT count(*)
  INTO vCount
  FROM produto;

  -- Imprimir, na tela, a quantidade de produtos cadastrados
  dbms_output.put_line('Existem '||to_char(vCount)||' cadastrados.');
```

EXCEPTION

```
  /* Se ocorrer qualquer erro, informar o usuário que não foi possível
  verificar a quantidade de produtos cadastrados */
  WHEN OTHERS THEN
    dbms_output.put_line('Não foi possível verificar a quantidade de' ||
                        'produtos cadastrados.');
```

END;
/

Ambiente de Execução

O ambiente de execução PL/SQL possui dois componentes essenciais:

- Motor PL/SQL;
- *Client do Oracle* (com uma Ferramenta de desenvolvimento).

O Motor PL/SQL é parte do Oracle Database e, portanto, ao instalar o SGBD da Oracle, o motor PL/SQL estará pronto para ser utilizado.

A ferramenta de desenvolvimento pode ser qualquer uma que permita o desenvolvimento de código PL/SQL e execução de declarações SQL.

Quando você executa uma declaração SQL em uma ferramenta, os seguintes passos devem acontecer:

1. A ferramenta transmitir a declaração SQL pela rede para o servidor de banco de dados;
2. A ferramenta aguardar uma resposta do servidor de banco de dados;
3. O servidor de banco de dados executar a consulta e transmitir os resultados de volta para a ferramenta;
4. A ferramenta exibir os resultados da declaração.

Nós já vimos como montar um bloco PL/SQL e executamos esse bloco no SQL*PLUS. Pois, bem, o SQL*PLUS é uma ferramenta para execução de declarações SQL e desenvolvimento PL/SQL.

Existem muitas outras ferramentas para desenvolvimento e execução de blocos PL/SQL e, embora não seja o foco deste treinamento, veremos a seguir algumas destas ferramentas.

Ferramentas para execução de declarações SQL e desenvolvimento PL/SQL

SQL*Plus



O SQL*Plus é uma ferramenta nativa da Oracle que acompanha todas as versões do Oracle Database e está disponível na versão gráfica (Windows) e/ou na versão de prompt. Em ambas versões, os recursos são os mesmos.

Abaixo você pode observar, através das figuras, os dois ambientes:

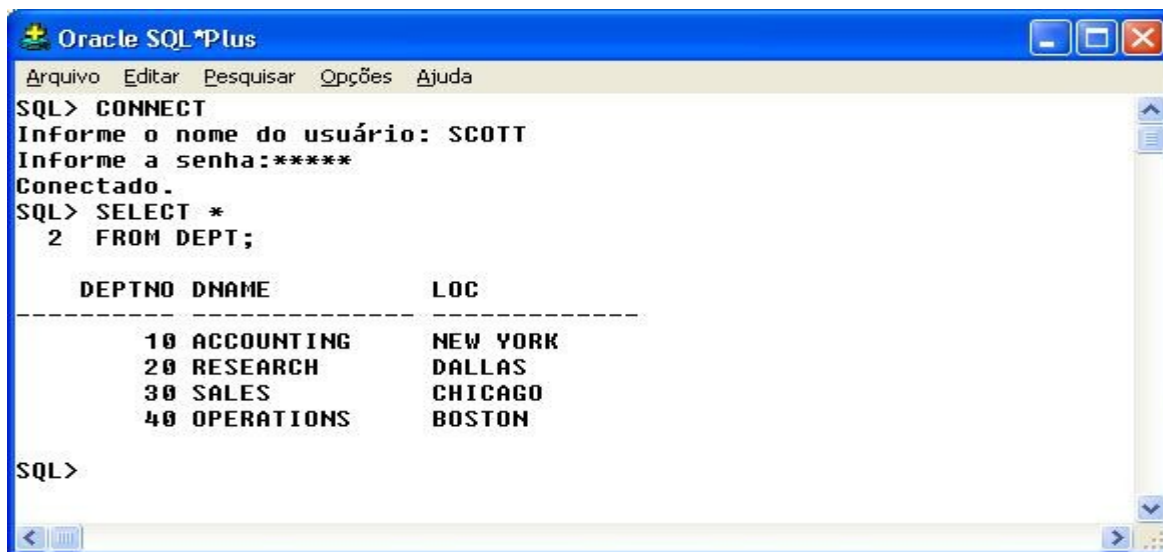


Figura 02: Ambiente gráfico do SQL*Plus

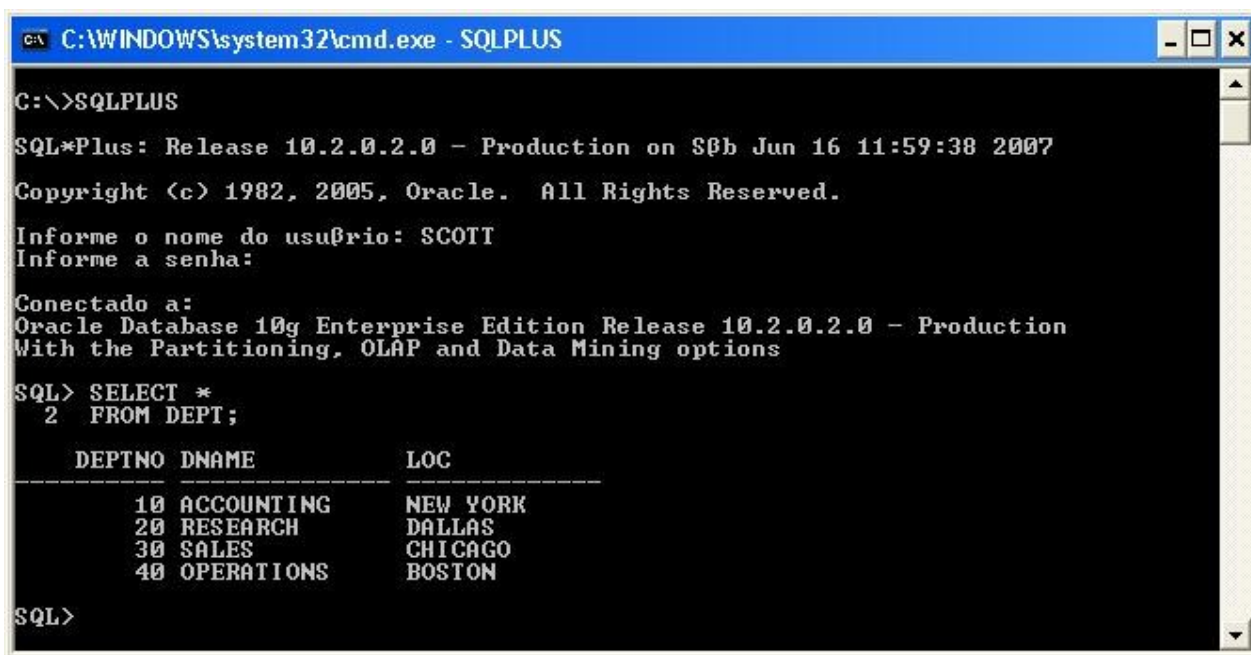


Figura 03: Ambiente de prompt do MS-DOS

Embora seja uma ferramenta poderosa, o SQL*Plus não é popular entre os desenvolvedores PL/SQL, pois não disponibiliza recursos com os quais eles estão acostumados, como por exemplo, *debugger*, *auto-complete*, exibição de resultados em *grid* e botões gráficos para manipulação e execução dos comandos.

A maior parte dos usuários do SQL*Plus, se concentra na comunidade de DBAs que utilizam a ferramenta para manutenção do banco de dados, pois é através do SQL*Plus que os *scripts* de manutenção são executados.

Mas mesmo sendo um desenvolvedor, é importante saber que sempre que existir na máquina uma instalação do *Server* ou do *Client* Oracle, o SQL*Plus estará disponível (ele pode ser muito útil se você estiver atendendo um cliente que não possui ferramentas de desenvolvimento!).

Para conhecer detalhes de uso e configuração do ambiente do SQL*Plus, consulte o site da Oracle (http://www.oracle.com/pls/db92/db92.sql_keywords?letter=A&category=sqlplus).

PL/SQL Developer



Provavelmente este é o ambiente de desenvolvimento PL/SQL mais utilizado pelos desenvolvedores e é perfeitamente justificável, pois trata-se de um ambiente completo de desenvolvimento e muito simples de usar.

PL/SQL Developer é um ambiente de desenvolvimento integrado (IDE) que foi especialmente destinado ao desenvolvimento de programas armazenados em bancos de dados Oracle, ou seja, vai muito além do desenvolvimento de blocos PL/SQL e execução de comandos SQL.

Entre os vários recursos para desenvolvedores, podemos destacar:

- **Debugger integrado:** Um debugger poderoso, muito próximo dos encontrados em ambientes RAD, como o Delphi. Nele você encontrará recursos como Breakpoints, Watches e muito mais.:

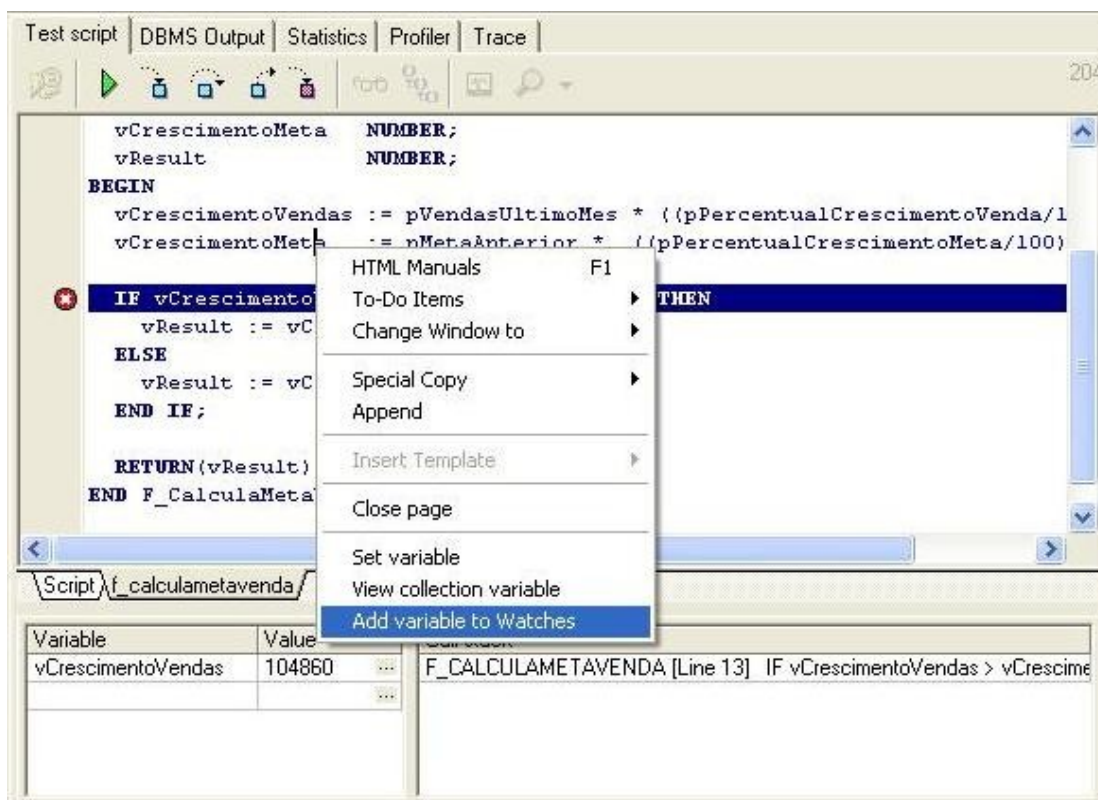
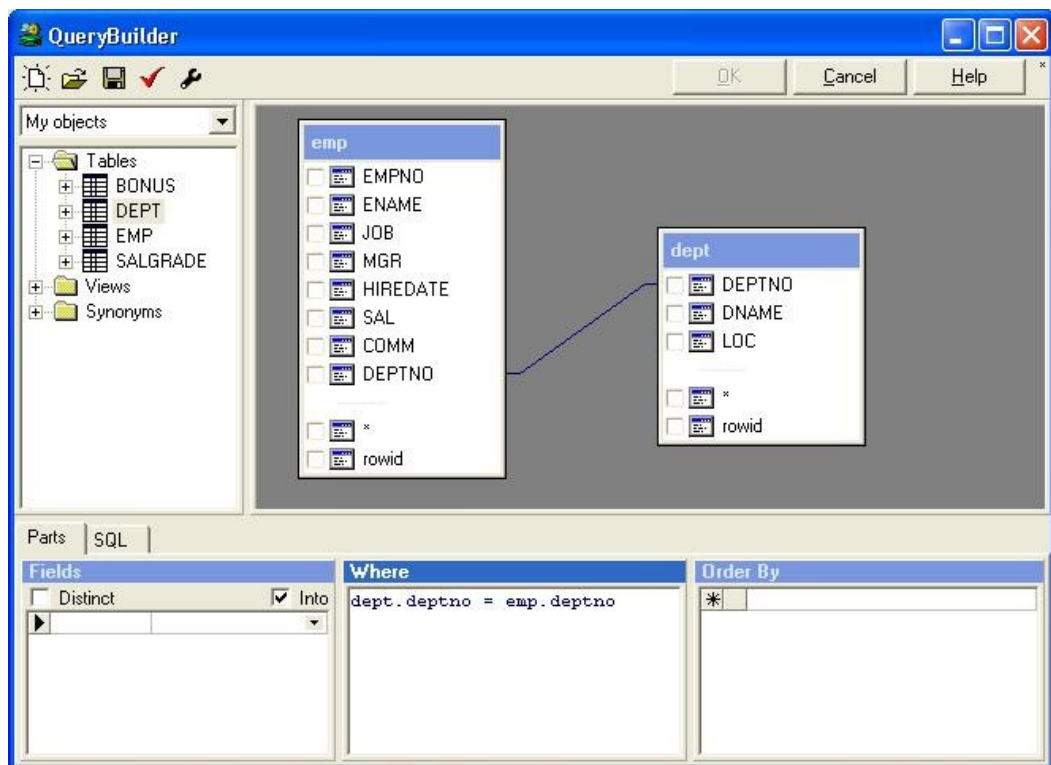


Figura 04: Debugger do PL/SQL Developer

- **Query Builder:** Uma tarefa que muito desenvolvedor detesta é relacionar tabelas, principalmente em bancos de dados onde é comum o uso de chaves primárias compostas. Com o Query Builder é muito fácil realizar esse trabalho. Se as chaves primárias e estrangeiras estiverem todas definidas, o usuário montará suas queries apenas utilizando o mouse:Figura 05: Query Builder do PL/SQL Developer



➤ **SQL Window:** Ferramenta ideal para execução de declarações de recuperação (SELECT), alteração (UPDATE), inclusão (INSERT) ou exclusão (DELETE). Os resultados dos comandos de recuperação são exibidos em *grid*:

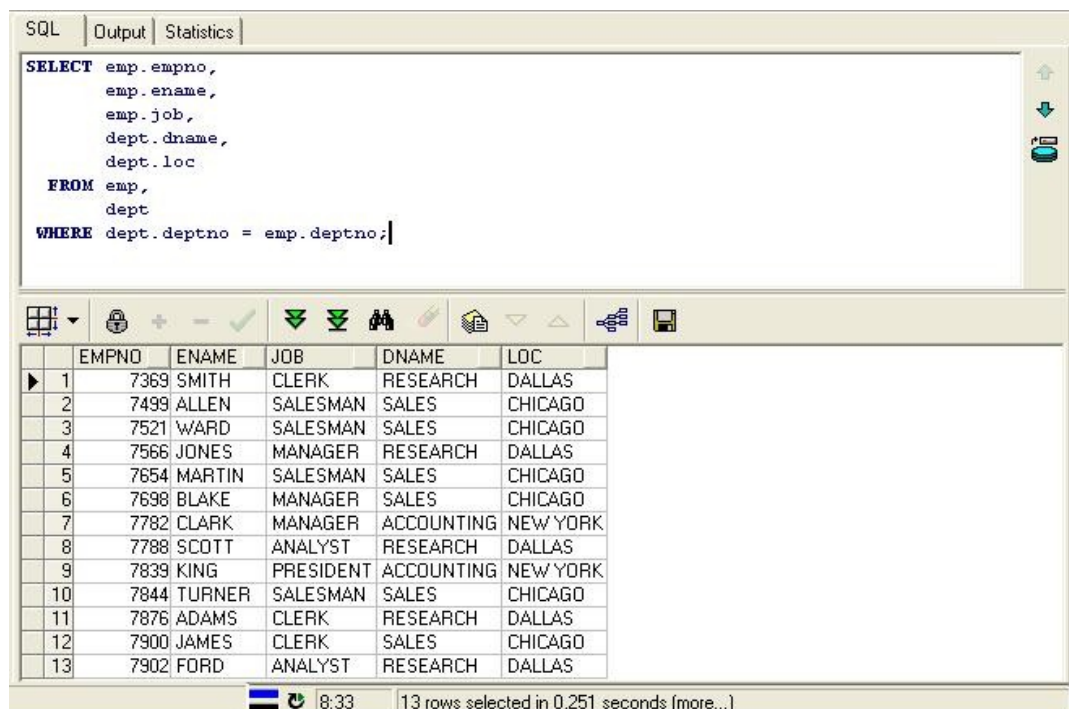


Figura 06: SQL Window do PL/SQL Developer

➤ **Command Window:** Ideal para execução de blocos anônimos, scripts e comandos de alteração de objetos (DDL). Seu funcionamento equivale ao do SQL*PLUS, com a grande vantagem

de contar com um editor integrado e recursos como o *auto-complete* e teclas de atalho para execução do script contido em seu editor. Muitos comandos do SQL*PLUS como o *SET SERVEROUTPUT ON* e o *CL SCR*, também funcionam nessa ferramenta:

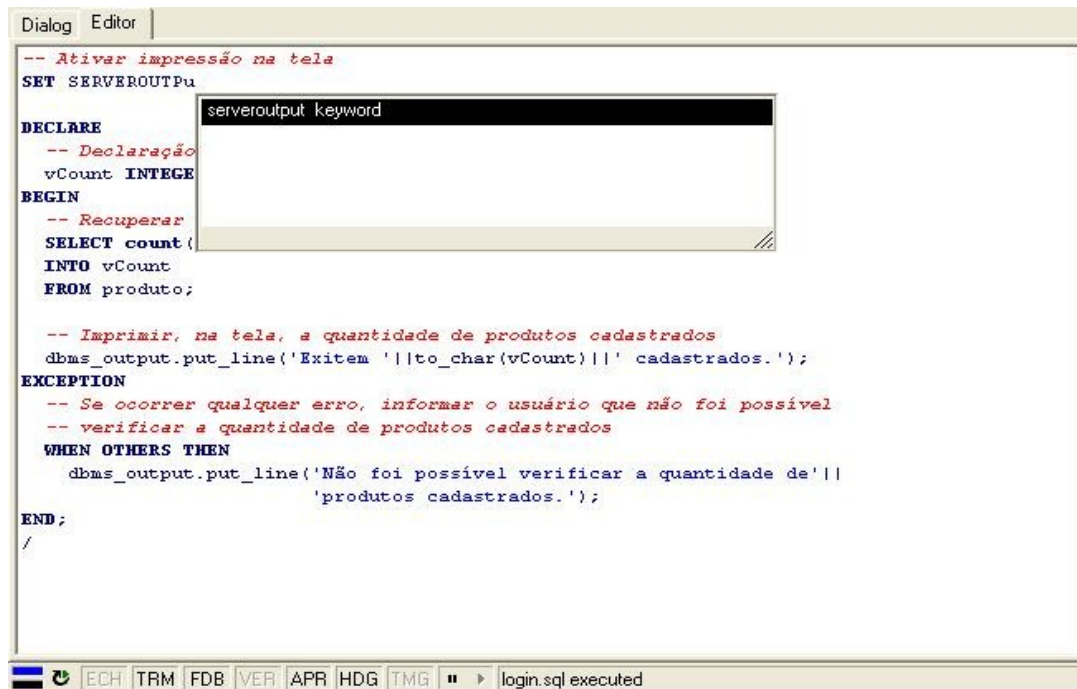


Figura 07: Editor do Command Window do PL/SQL Developer

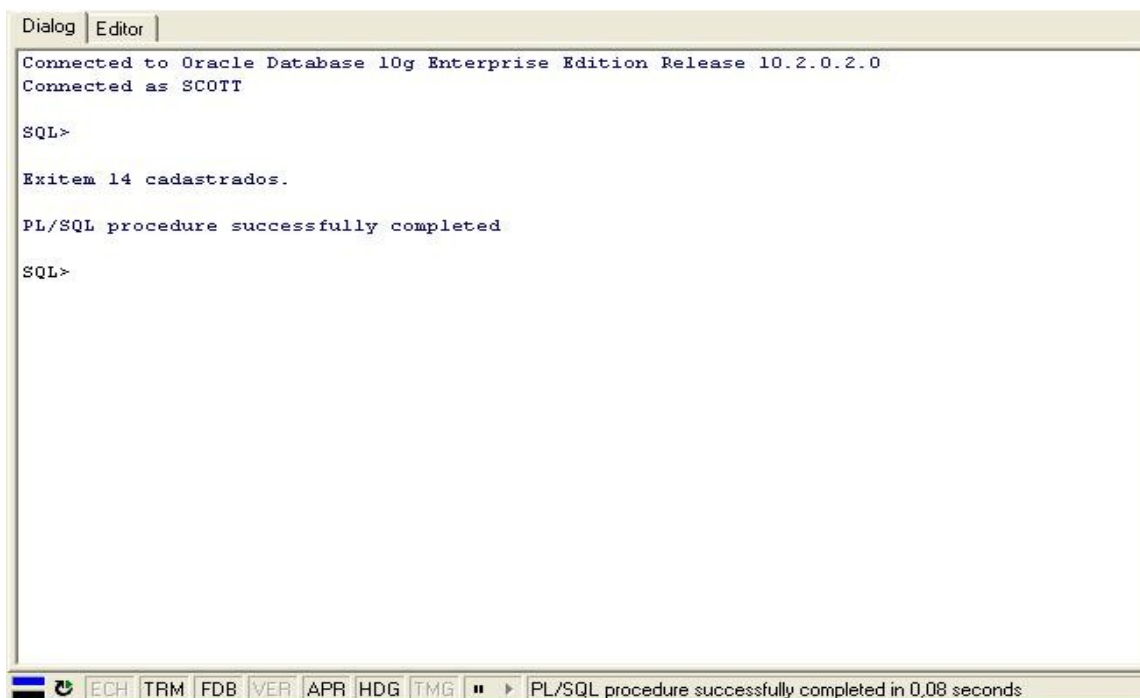


Figura 08: Janela de execução do Command Window do PL/SQL Developer

- **Object Browser:** Lista de forma hierárquica os objetos permitidos ao usuário logado,

disponibilizando atalhos que facilitam a manutenção e edição desses objetos:

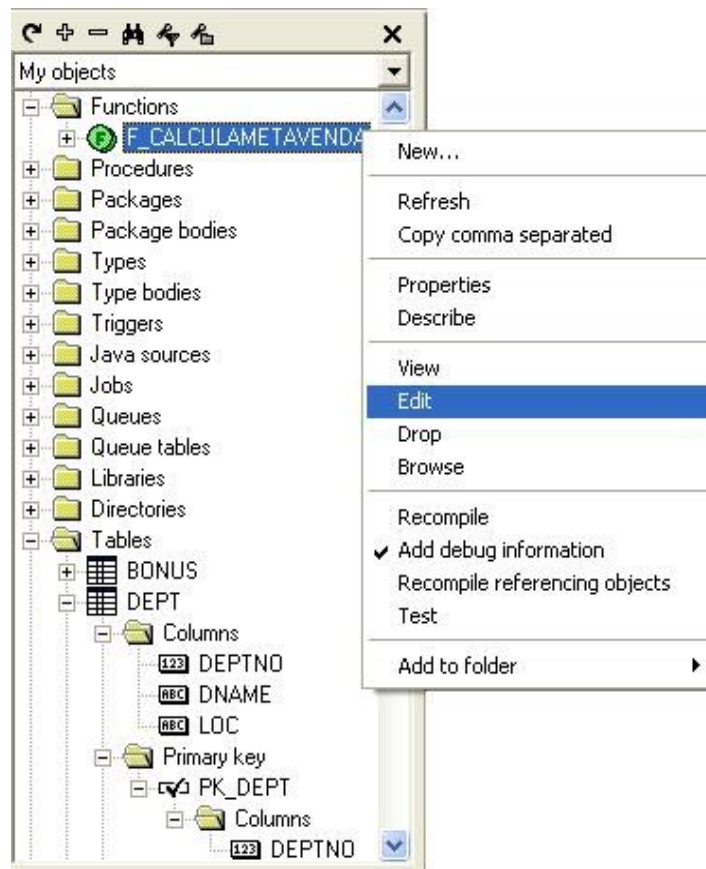


Figura 09: Object Browser do PL/SQL Developer

O PL/SQL Developer não é uma ferramenta gratuita, mas pode ser baixado para teste através do site de seu fabricante: www.allroundautomations.com

TOAD



Outra ferramenta de grande aderência ao trabalho dos desenvolvedores. Muito semelhante ao PL/SQL Developer e amplamente conhecida entre os profissionais de PL/SQL.

Possui todos recursos encontrados em seu concorrente, mudando em alguns casos, a maneira de usar.

Na figura a seguir, podemos observar que o object browser do TOAD é segmentado por tipo de objeto, mas segue a mesma idéia do PL/SQL Developer.

O seu editor também é avançado e em suas opções de menu encontramos recursos como SQL Builder (equivalente ao Query Builder do PL/SQL Developer), visualizadores de sessões e ferramentas de tuning.

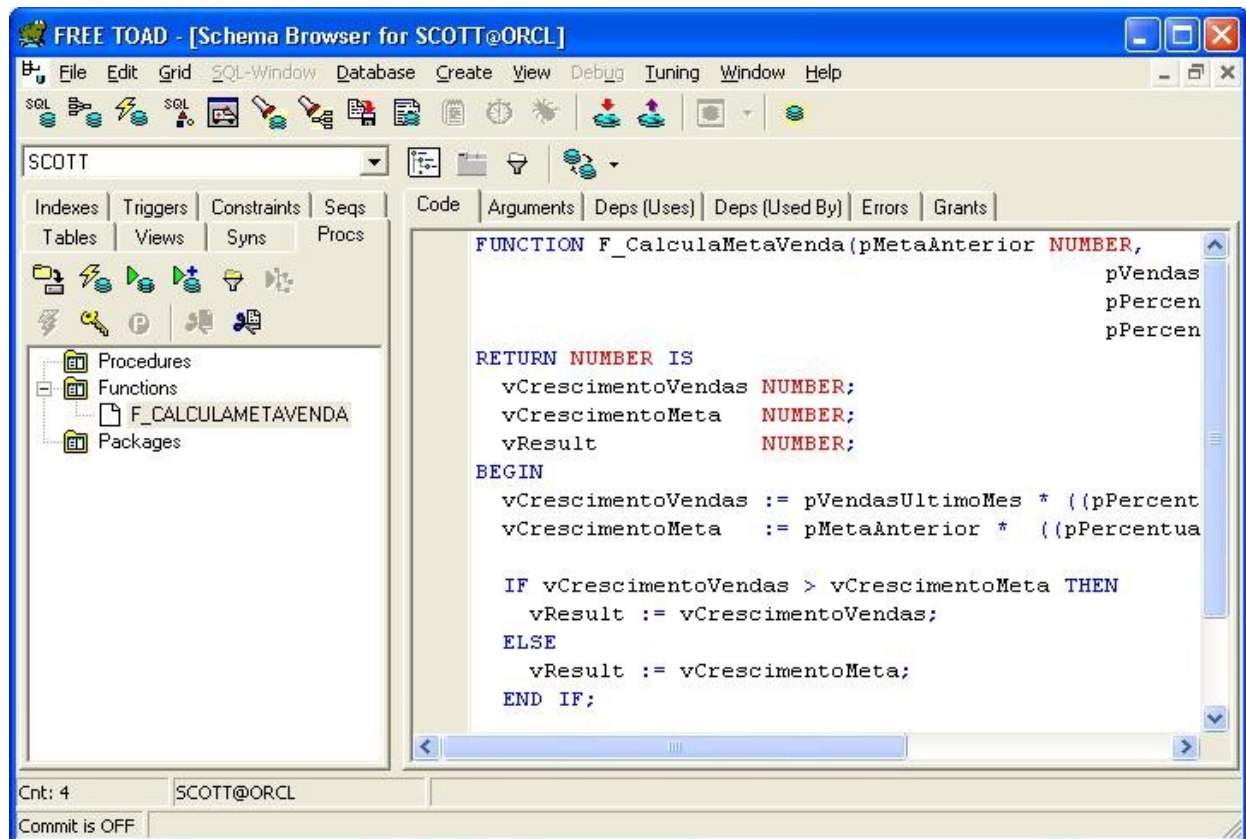


Figura 10: Edição de stored procedure no TOAD

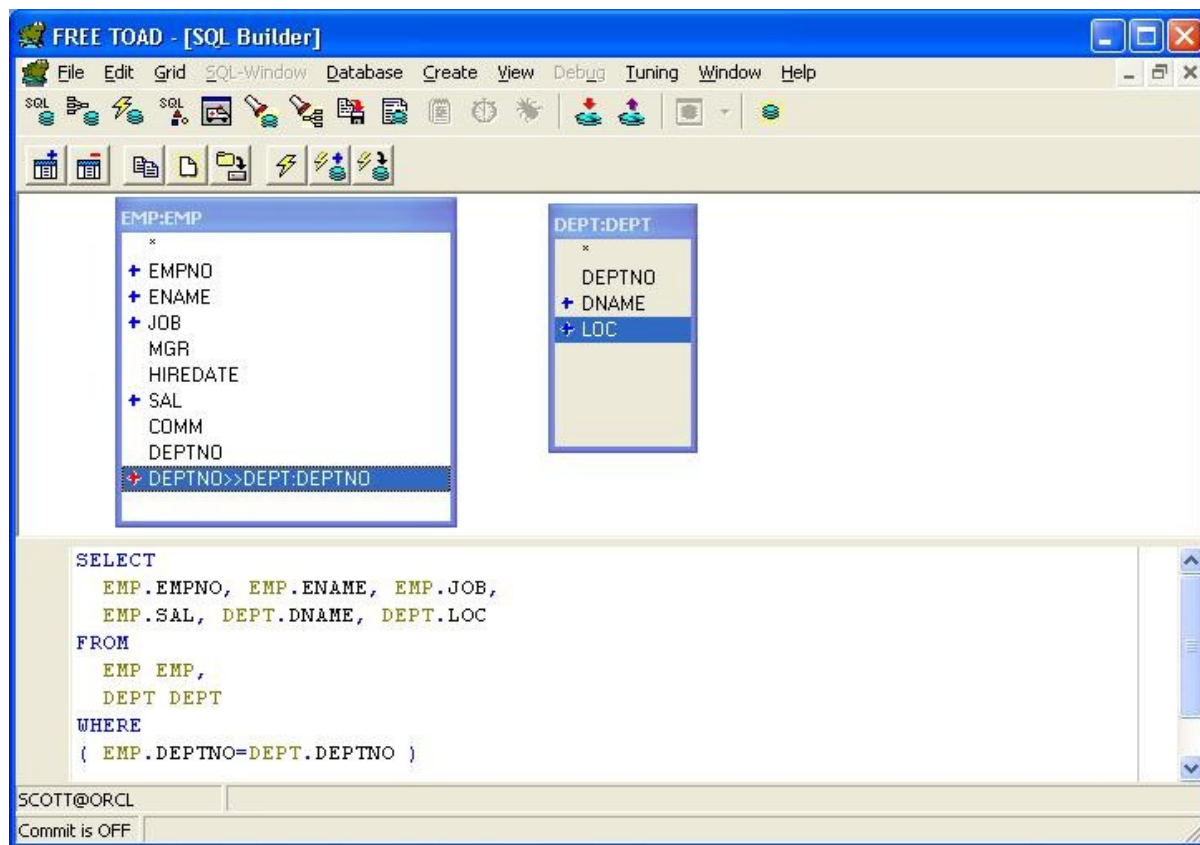
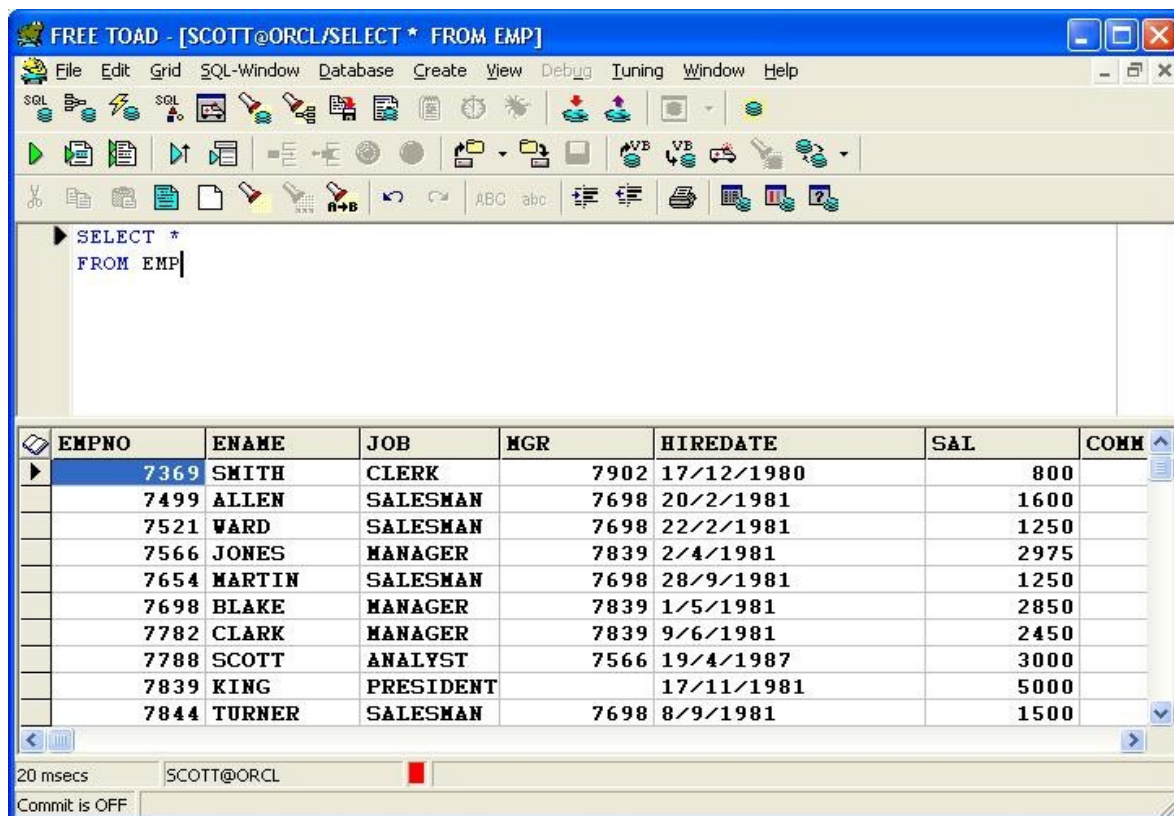


Figura 11: O SQL Builder do TOAD



The screenshot shows the TOAD SQL Edit Window with the title bar 'FREE TOAD - [SCOTT@ORCL/SELECT * FROM EMP]'. The menu bar includes File, Edit, Grid, SQL-Window, Database, Create, View, Debug, Tuning, Window, and Help. The toolbar contains various icons for file operations, SQL execution, and debugging. The SQL editor area contains the query 'SELECT * FROM EMP'. Below the editor, a table of results is displayed with columns EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, and COMM. The table contains 14 rows of data. At the bottom, the status bar shows '20 msec', 'SCOTT@ORCL', and 'Commit is OFF'.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
7369	SMITH	CLERK	7902	17/12/1980	800	
7499	ALLEN	SALESMAN	7698	20/2/1981	1600	
7521	WARD	SALESMAN	7698	22/2/1981	1250	
7566	JONES	MANAGER	7839	2/4/1981	2975	
7654	MARTIN	SALESMAN	7698	28/9/1981	1250	
7698	BLAKE	MANAGER	7839	1/5/1981	2850	
7782	CLARK	MANAGER	7839	9/6/1981	2450	
7788	SCOTT	ANALYST	7566	19/4/1987	3000	
7839	KING	PRESIDENT		17/11/1981	5000	
7844	TURNER	SALESMAN	7698	8/9/1981	1500	

Figura 12: O SQL Edit Window do TOAD

A ferramenta TOAD possui versões gratuitas e pagas, podendo ser facilmente encontrada para download na internet.

Para maiores informações, visite o site da Quest, fabricante da ferramenta: www.quest.com

Oracle SQL Developer



Embora o SQL*Plus seja a ferramenta nativa do Banco de dados Oracle para execução de comandos SQL e PL/SQL, a Oracle disponibiliza para download gratuito em seu site, uma outra ferramenta para desenvolvedores PL/SQL: O Oracle SQL Developer.

Essa ferramenta, desenvolvida em Java, disponibiliza para o desenvolvedor todos os recursos necessários para escrever seus programas PL/SQL.

Além dos recursos já vistos no PL/SQL Developer e no TOAD, como debugger, wizards para montagem de queries e object browser, o Oracle SQL Developer possui alguns recursos de destaque, como:

➤ **Trabalhar com conexões simultâneas:** No Oracle SQL Developer você consegue ativar mais de uma conexão por vez e carregar janelas em forma de abas, para cada conexão, como mostrado na figura abaixo:

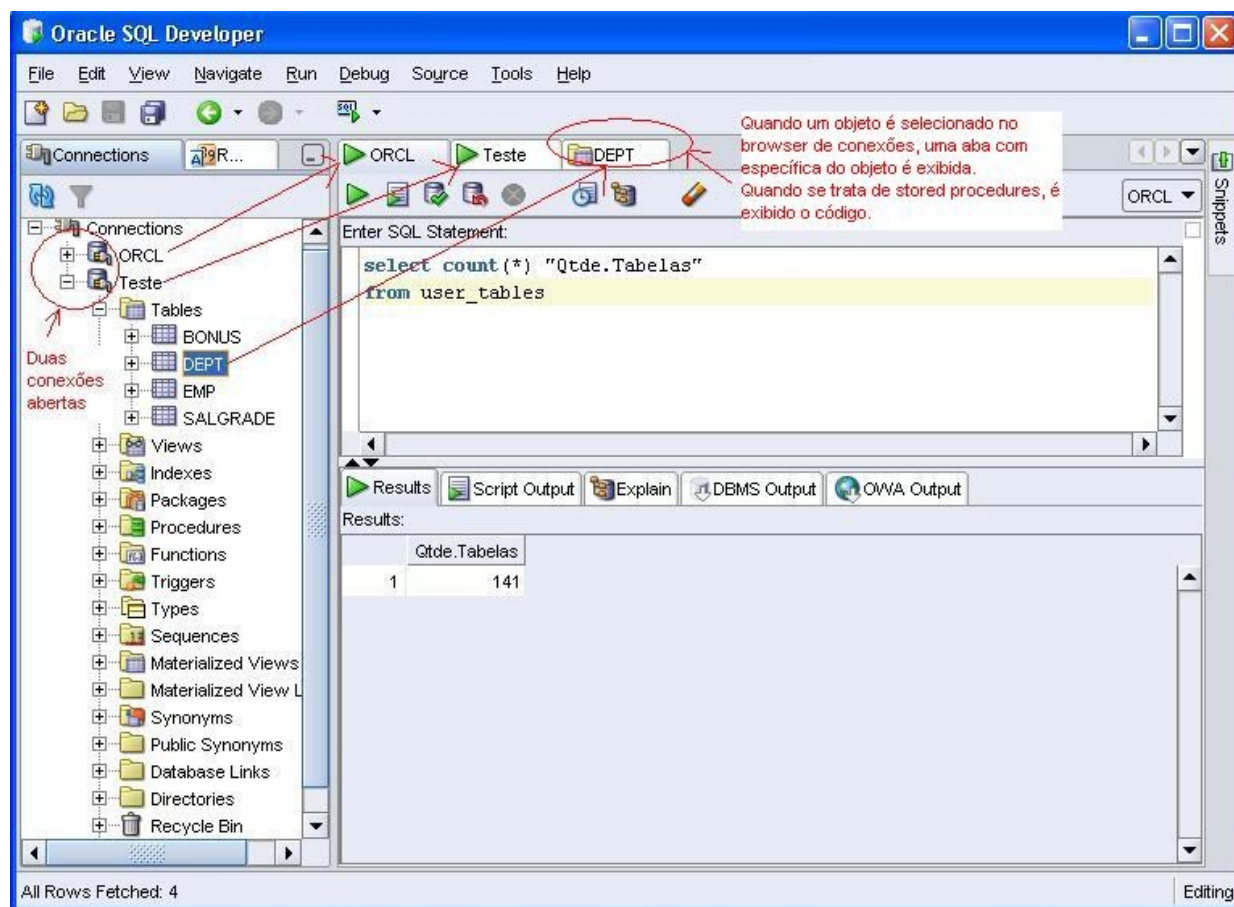


Figura 13: Várias conexões abertas no Oracle SQL Developer, visualizadas na tela principal

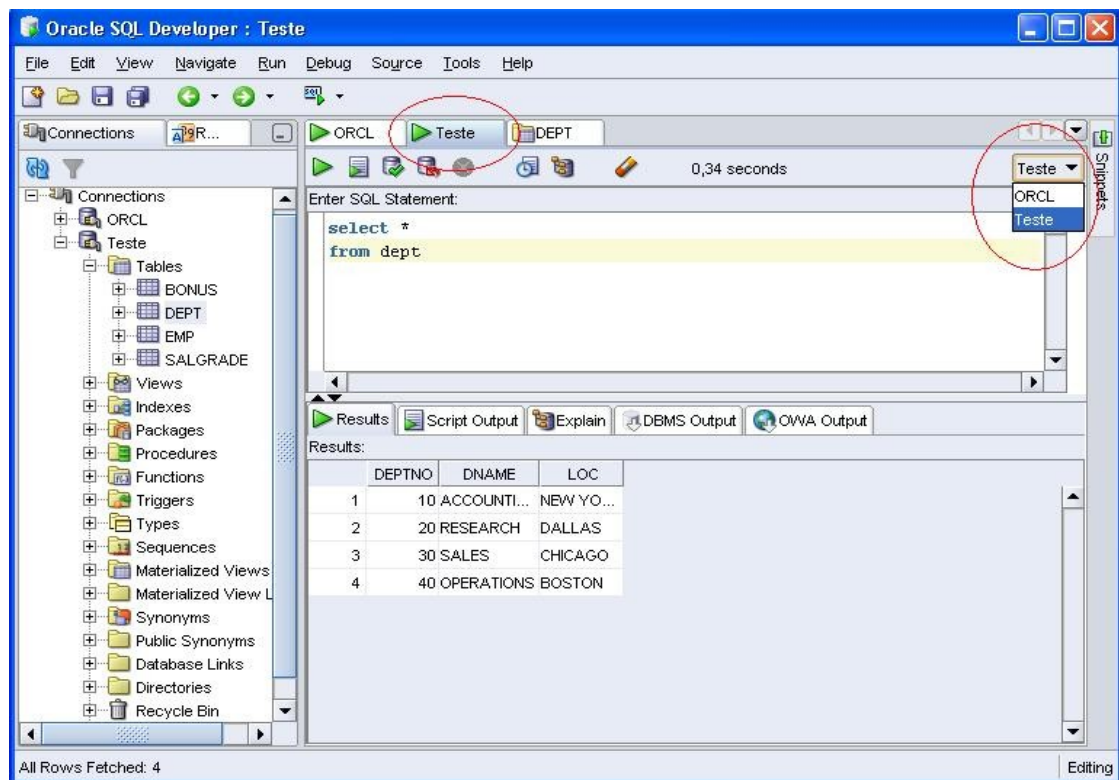


Figura 14: Alterando a conexão da aba de trabalho ativa no Oracle SQL Developer

- **Diversas formas de resultados na mesma tela, através de abas:**

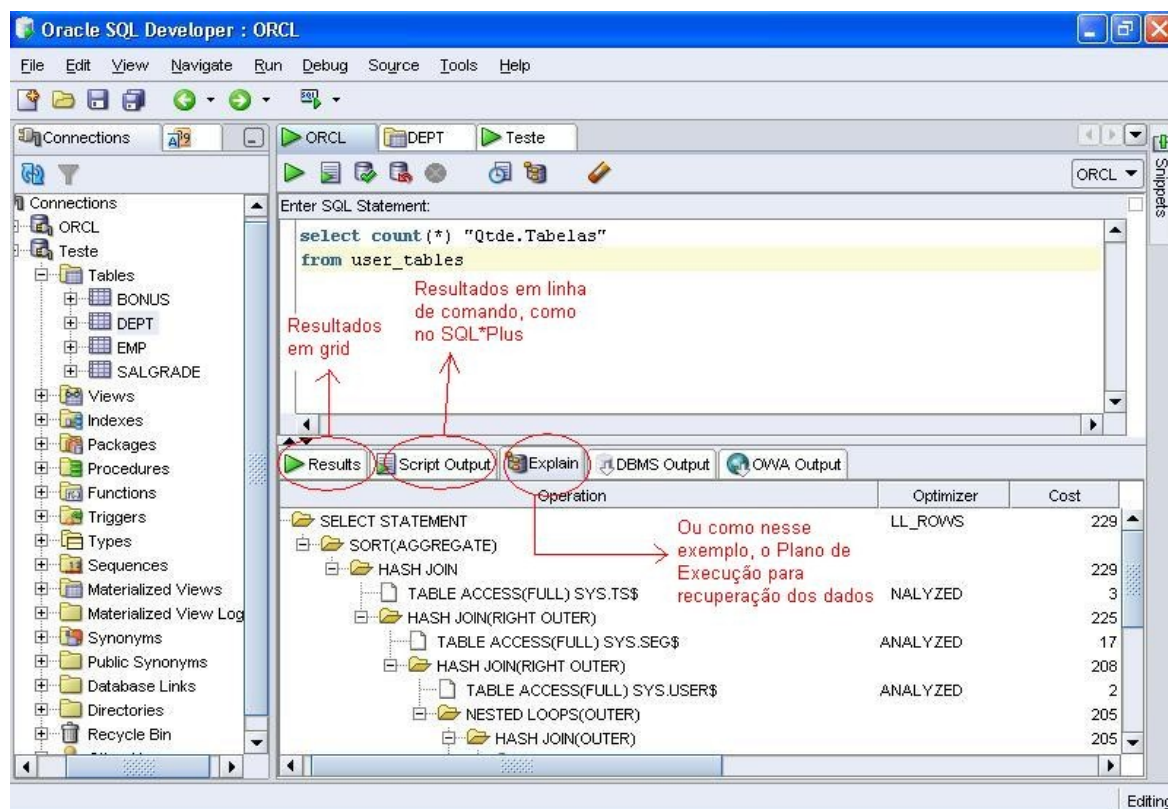


Figura 15: Resultado da instrução, apresentados de várias maneiras, através de abas

➤ **Editor com opção de ocultar blocos:** No Oracle SQL Developer você pode ocultar um bloco e para visualizá-lo novamente basta um clique para expandir sua área novamente. Caso queira visualizá-lo sem expandir, basta posicionar o cursor do mouse sobre o botão de expansão, como a seguir:

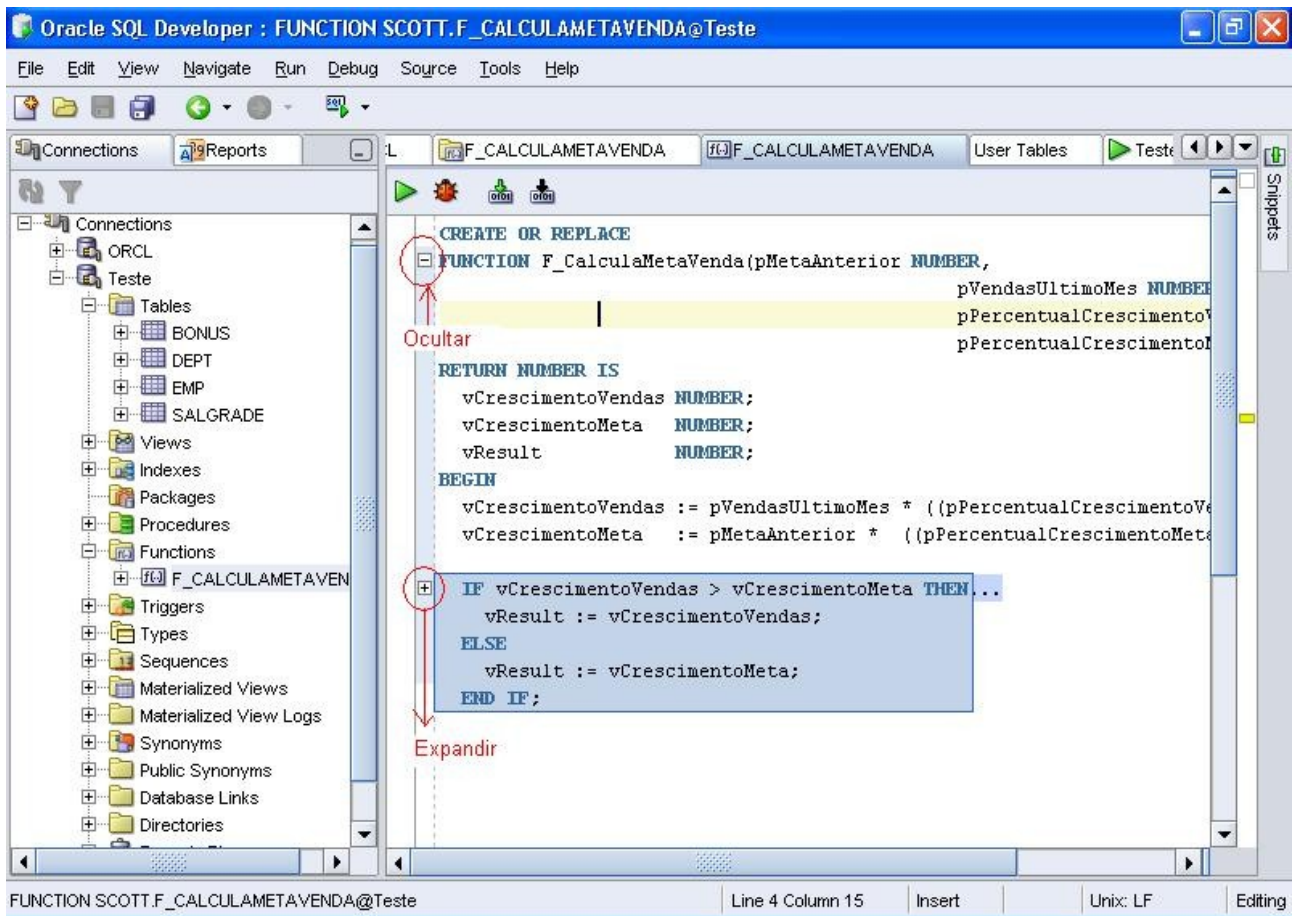


Figura 16: Editor que oculta blocos e exibe apenas para leitura com um simples movimento do mouse

O Oracle SQL Developer é uma ferramenta "pesada", provavelmente por ter sido desenvolvida na plataforma Java, por isso, é necessário uma máquina com bons recursos de memória e processamento para seu uso, do contrário, o desenvolvedor pode sofrer com a navegação da ferramenta.

A escolha da ferramenta depende do gosto do desenvolvedor. Esse treinamento pode ser praticado em qualquer uma das ferramentas apresentadas aqui ou outras que sigam os padrões para execução de SQL e PL/SQL.

Declaração de variáveis

Devemos sempre declarar as variáveis na Sessão Declarativa antes de referenciá-las nos blocos PL/SQL. Podemos atribuir um valor inicial para a variável criada, definir um valor que será constante e defini-la como NOT NULL.

Sintaxe:

```
Identificador [CONSTANT] Tipo de Dado [NOT NULL] [ := | DEFAULT expressão ]
```

onde temos:

- **Identificador:** nome da variável.
- **CONSTANT:** Contém valores que não podem ser alterados. Constantes devem ser inicializadas.
- **Tipo de Dado:** é um tipo de dado que pode ser escalar, composto, referencial ou LOB.
- **NOT NULL:** valida a variável que sempre deve conter valor. Variáveis NOT NULL também devem ser inicializadas.
- **Expressão:** é qualquer expressão PL/SQL. Pode ser uma expressão literal, outra variável ou uma expressão envolvendo operadores e funções.

Ao declarar variáveis, procure:

- a) Adotar padrões para dar nomes a variáveis. Por exemplo, vNome para representar uma variável e cNome para representar uma variável de coleção;
- b) Se usar uma variável NOT NULL, inicie com algum valor válido;
- c) Declarar uma variável que facilite a leitura e manutenção do código;
- d) Não utilizar uma variável com o mesmo nome de uma coluna de tabela referenciada no bloco PL/SQL;

Dica: você pode inicializar uma variável utilizando o operador ' := ' ou a palavra chave DEFAULT, como nos exemplos abaixo:

```
vNome varchar2(30) := 'Seu Madruga';
```

Ou

```
vNome varchar2(30) DEFAULT 'Seu Madruga';
```

Explorando os *datatypes*

A PL/SQL fornece vários *datatypes* que você pode usar e eles podem ser agrupados em várias categorias: *datatypes* escalares, *datatypes* de objeto grande, registros e ponteiros.

Neste curso exploraremos os *datatypes* escalares e os registros.

Datatypes escalares (tipos de dados simples)

Uma *variável escalar* é uma variável que não é formada de alguma combinação de outras variáveis. As variáveis escalares não têm componentes internos que você pode manipular individualmente. Elas são usadas para construir *datatypes* mais complexos, tais como os registros e *arrays*.

A tabela 1 abaixo exemplifica alguns *datatypes* escalares da PL/SQL:

Tabela 1: Exemplo de *datatypes* escalares

<i>Datatype</i>	<i>Uso</i>
VARCHAR2	Strings de caractere de comprimento variável
CHAR	Strings de caractere de comprimento fixo
NUMBER	Números fixos ou de ponto flutuante
BYNARY_INTEGER	Valores de inteiros
PLS_INTEGER	Usado para cálculos rápidos de inteiros
DATE	Datas
BOOLEAN	Valores TRUE ou FALSE
LONG	Usado para armazenar strings longas de caracteres (obsoleto)
LONG ROW	Usado para armazenar grandes quantidades de dados binários (obsoleto)

IMPORTANTE: Alguns *datatypes* da PL/SQL não possuem equivalentes no banco de dados Oracle, embora muitos coincidam. O *datatype* PLS_INTEGER, por exemplo, é exclusivo da PL/SQL e outros possuem pequenas diferenças, como LONG que possui limites inferiores ao do banco de dados.

A PL/SQL também fornece subtipos de alguns *datatypes*. Um subtipo representa um caso especial de um *datatype*, geralmente representando um intervalo menor de valores do que o tipo pai. Por exemplo, POSITIVE é um subtipo de BINARY_INTEGER que contém apenas valores positivos. Em alguns casos, os subtipos só existem para fornecer alternativas por questões de compatibilidade com padrão SQL ou com outras marcas conhecidas de banco de dados do mercado.

Abaixo segue uma tabela com exemplos de outros subtipos:

Tabela 2: Exemplo de subtipos

<i>Subtipo</i>	<i>Subtipo de</i>	<i>Uso</i>
VARCHAR	VARCHAR2	Apenas para compatibilidade. Uso não recomendado.
STRING	VARCHAR2	Apenas para compatibilidade.
DECIMAL	NUMBER	Igual a NUMBER
DEC	NUMBER	Igual a DECIMAL
DOUBLE PRECISION	NUMBER	Igual a NUMBER
NUMERIC	NUMBER	Igual a NUMBER
REAL	NUMBER	Igual a NUMBER
INTEGER	NUMBER	Equivalente a NUMBER(38)
INT	NUMBER	Igual a INTEGER
SMALLINT	NUMBER	Igual a NUMBER(38)
FLOAT	NUMBER	Igual a NUMBER
POSITIVE	BINARY_INTEGER	Permite que apenas os inteiros positivos sejam armazenados, até o máximo de 2.147.483.687. Zero não é considerado um valor positivo e, portanto, não é um valor permitido.
NATURAL	BYNARY_INTEGER	Permite que apenas os números naturais sejam armazenados, o que inclui o zero. O máximo é 2.147.483.687

A declaração de uma variável escalar é muito simples, como pode ser visto no exemplo abaixo:

```
DECLARE
  vDataNasc  DATE;
  vCodDepto  NUMBER(2) NOT NULL := 10;
  vCidade    VARCHAR2(30)       := 'ITU';
  vPI        CONSTANT NUMBER    := 3.1415;
  vBloqueado BOOLEAN            := FALSE;
BEGIN
```



```
NULL;  
END;  
/
```

IMPORTANTE: Sempre que uma variável não é inicializada, seu valor inicial será NULL.

O Atributo %TYPE

Quando declaramos uma variável PL/SQL para manipular valores que estão em tabelas, temos que garantir que esta variável seja do mesmo tipo e tenha a mesma precisão do campo que será referenciado. Se isto não acontecer, um erro PL/SQL ocorrerá durante a execução. Para evitar isto, podemos utilizar o atributo %TYPE para declarar a variável de acordo com a declaração de outra variável ou coluna de uma tabela. O atributo %TYPE é mais utilizado quando o valor armazenado na variável for um valor derivado de uma coluna de uma tabela do banco. Para usar este atributo ao invés de um tipo de dado que é obrigatório na declaração de uma variável, basta colocar o nome da tabela e a coluna desejada e logo depois, coloque o atributo %TYPE

Sintaxe:

```
Identificador Tabela.Nome_Coluna%TYPE;
```

Quando o bloco é compilado, uma atribuição do valor de uma coluna para uma variável é feita, o PL/SQL verifica se o tipo e o tamanho da variável é compatível com o tipo da coluna que está sendo usada. Então, quando utilizamos o atributo %TYPE para definir uma variável, garantimos que a variável sempre estará compatível com a coluna, mesmo que o tipo desta coluna tenha sido alterado.

Exemplos:

```
DECLARE  
  vProVlMaiorvenda produto.pro_vl_maiorvenda%TYPE;  
BEGIN  
  SELECT p.pro_vl_maiorvenda  
  INTO vProVlMaiorvenda  
  FROM produto p  
  WHERE p.pro_in_codigo = 1;  
END;  
/
```

Neste exemplo, podemos observar duas situações:

1. Não precisamos nos preocupar se o tipo da variável esta compatível com o campo da tabela;
2. Garantimos que se alguma alteração for realizada no tipo de dado da coluna, não precisaremos alterar nada na variável.

Agora imagine se nesse mesmo código o tipo da variável fosse NUMBER(9,2) e sem avisar o desenvolvedor, o responsável pelos objetos do banco alterasse o tipo da coluna pro_vl_maiorvenda para NUMBER(11,2)?

O desenvolvedor só saberia quando esse código fosse executado e o PL/SQL exibisse o erro. Não é nada bom que o seu cliente receba um erro desses durante o trabalho!

Variáveis do tipo boolean

A declaração de variáveis do tipo *boolean* não difere em nada dos demais tipos, como pudemos ver em alguns exemplos anteriores, mas existe um recurso interessante no que se refere a atribuição de valores

para variáveis do tipo *boolean*.

Essas variáveis só recebem os valores TRUE, FALSE e NULL. As expressões aritméticas sempre retornam TRUE ou FALSE e, portanto, você pode atribuir uma expressão aritmética para uma variável do tipo *boolean*.

Vejamos um exemplo:

O seguinte bloco PL/SQL está correto:

```
DECLARE
  -- Declaração de variáveis
  vValorUltimaVenda produto.pro_vl_ultimavenda%TYPE;
  vValorMaiorVenda produto.pro_vl_maiorvenda%TYPE;
  vUltimaVendaMaior BOOLEAN := FALSE;
BEGIN
  -- Comando para recuperar valor da última venda e da maior venda do produto
  SELECT p.pro_vl_ultimavenda, p.pro_vl_maiorvenda
  INTO vValorUltimaVenda, vValorMaiorVenda
  FROM produto p
  WHERE p.pro_in_codigo = 1;

  -- Condição para definir se última venda é ou não a maior
  IF vValorUltimaVenda = vValorMaiorVenda THEN
    vUltimaVendaMaior := TRUE;
  ELSE
    vUltimaVendaMaior := FALSE;
  END IF;
END;
/
```

Mas poderia ter sido escrito da seguinte maneira:

```
DECLARE
  -- Declaração de variáveis
  vValorUltimaVenda produto.pro_vl_ultimavenda%TYPE;
  vValorMaiorVenda produto.pro_vl_maiorvenda%TYPE;
  vUltimaVendaMaior BOOLEAN := FALSE;
BEGIN
  -- Comando para recuperar valor da última venda e da maior venda do produto
  SELECT p.pro_vl_ultimavenda, p.pro_vl_maiorvenda
  INTO vValorUltimaVenda, vValorMaiorVenda
  FROM produto p
  WHERE p.pro_in_codigo = 1;

  -- Atribuir valor da condição para definir se última venda é ou não a maior
  vUltimaVendaMaior := (vValorUltimaVenda = vValorMaiorVenda);
END;
/
```

Como pode ser observado, obtemos o mesmo resultado, mas com um código mais limpo.

Tipo Registro

Um registro é uma coleção de valores individuais que estão relacionados de alguma forma. Com frequência os registros são usados para representar uma linha de uma tabela, e assim o relacionamento se baseia no fato de que todos os valores vêm da mesma linha. Cada campo de um registro é exclusivo e tem seus próprios valores. Um registro como um todo não tem um valor.

Usando os registros você pode agrupar dados semelhantes em uma estrutura e depois manipular sua estrutura como uma entidade ou unidade lógica. Isso ajuda a reproduzir o código, e o mantém mais fácil de atualizar e entender.

Para usar um registro você deve defini-lo declarando um tipo de registro. Depois você deve declarar uma ou mais variáveis PL/SQL como sendo daquele tipo.

Você declara um tipo de registro na parte da declaração de um bloco, como outra variável qualquer.

O exemplo abaixo mostra como declarar e usar um tipo registro:

```
DECLARE
-- Definição de tipos
TYPE TProduto IS RECORD (
    Nome          VARCHAR2 (40) ,
    Marca         VARCHAR2 (20) ,
    ValorUltimaVenda NUMBER (9,2)
);

-- Declaração de variáveis
vProd TProduto;
BEGIN
-- Atribuir valor para o registro vProduto
SELECT p.pro_st_nome, p.pro_st_marca, p.pro_vl_ultimavenda
INTO vProd.Nome, vProd.Marca, vProd.ValorUltimaVenda
FROM produto p
WHERE p.pro_in_codigo = 1;

-- Imprimir na tela os dados recuperados
dbms_output.put_line('Nome do produto: '||vProd.Nome||chr(10)||
                    'Marca: '||vProd.Marca||chr(10)||
                    'Valor última venda: '||to_char(vProd.ValorUltimaVenda)
                    );
END;
/
```

IMPORTANTE: Para que o pacote DBMS_OUTPUT.PUT_LINE imprima o resultado desejado na tela, é necessário ativar a impressão na ferramenta. No SQL*Plus e no Command Window do PL/SQL Developer essa funcionalidade é ativada através do comando SET SERVEROUTPUT ON. No Oracle SQL Developer basta ativar o botão na aba DBMS Output, da janela Worksheet.

O Atributo %ROWTYPE

Quando uma variável de tipo de registro se baseia em uma tabela, isso significa que os campos do registro têm exatamente o mesmo nome e *datatype* das colunas da tabela especificada. Você usa o atributo %ROWTYPE para declarar um registro com base em uma tabela.

A sintaxe para declarar uma variável usando o atributo %ROWTYPE é a seguinte:

```
nome_variavel tabela%ROWTYPE
```

A maior vantagem no uso do atributo %ROWTYPE é que uma alteração na definição de tabela se reflete automaticamente no seu código PL/SQL.

IMPORTANTE: O acréscimo de uma coluna a uma tabela será transparente para o seu código PL/SQL, assim como determinados tipos de alterações no *datatype*. Entretanto, quando você remove uma coluna (*drop column*) de tabela que o seu código está usando, você precisa alterá-lo para retirar as referências daquela coluna (inclusive nas variáveis de registro).

A seguir, mostramos o exemplo de tipo de registro alterado para usar o atributo %ROWTYPE:

```
DECLARE
  -- Declaração de variáveis
  vProd produto%ROWTYPE;
BEGIN
  -- Atribuir valor para o registro vProduto
  SELECT p.*
  INTO vProd
  FROM produto p
  WHERE p.pro_in_codigo = 1;

  -- Imprimir na tela os dados recuperados
  dbms_output.put_line('Nome do produto: '||vProd.pro_st_nome||chr(10)||
                        'Marca: '||vProd.pro_st_marca||chr(10)||
                        'Valor última venda: '||to_char(vProd.pro_vl_ultimavenda)
                        );
END;
/
```

A desvantagem deste exemplo em relação ao anterior, é que neste caso todas as colunas da tabela são recuperadas para a variável *vProd*. Dependendo do número de colunas que “*” retornar, haverá desperdício de memória.

O Escopo de uma variável

Escopo de uma variável é a região de um programa no qual podemos referenciar a variável. Variáveis declaradas em blocos PL/SQL são considerados **local** para o bloco onde está declarado e **global** para todos seus sub-blocos. Se uma variável global é redeclarada em um sub-bloco, ambos permanecem no escopo. Dentro do sub-bloco, somente a variável local é visível e para referenciar a variável global devemos utilizar uma *tag*, que será vista a seguir.

Entretanto, não podemos declarar uma variável duas vezes no mesmo bloco, mas podemos declarar a mesma variável em dois blocos diferentes. As duas variáveis são distintas, ou seja, qualquer mudança em uma, não irá afetar a outra.

Um bloco não pode referenciar variáveis declaradas em outros blocos de mesmo nível porque estas variáveis não são locais e nem globais para este bloco.

Uma variável é visível no bloco no qual ela foi declarada e para todos os sub-blocos e *procedures* e *functions* aninhados. Se o bloco não acha a variável declarada localmente, ele procura na sessão declarativa do bloco pai. Um bloco pai nunca procura por uma variável declarada no bloco filho.

O escopo descrito aqui se aplica a todos objetos declarados, tais como variáveis, cursores, exceções definidas por usuário e constantes.

Veja uma representação do escopo de variáveis:

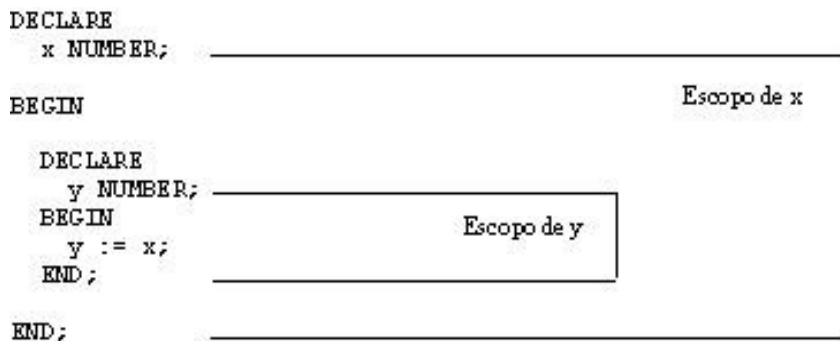


Figura 17: Escopo de uma variável

Usando tags para identificar um variável

Podemos identificar uma variável global dentro de um sub-bloco usando uma *tag* como prefixo do bloco PL/SQL. Fazendo isto, conseguimos com que uma variável declarada no bloco pai e de mesmo nome da variável declarada no bloco filho, possa ser manipulada dentro do bloco filho. Esta tag é definida através dos símbolos << tag >>, onde *tag* pode ser qualquer palavra, não podendo utilizar números, nem outros símbolos.

Vejamos um exemplo prático:

```
<<BlocoPai>>
DECLARE
  x number := 4;
BEGIN

  DECLARE
    x number := 2;
  BEGIN
    dbms_output.put_line(BlocoPai.x);
  END;

  dbms_output.put_line(x);
END;
/
```

No exemplo acima, estamos referenciando a variável *x* do bloco pai, dentro da sessão executável do bloco filho, mesmo o bloco filho tendo uma variável de mesmo nome. Isto só foi possível porque utilizamos a tag **BlocoPai**, que está pré-fixada no bloco pai.

Valores Nulos

É comum que os desenvolvedores menos experientes confundam **NULL** com 0 (zero) ou espaço em branco. ISSO É UM ERRO! É um erro grave, pois pode acarretar em falha do programa.

NULL não é um valor propriamente dito. **NULL** é igual a **NADA**!

Quando uma variável possui valor **NULL**, quer dizer que não foi atribuído valor algum a essa variável (ou atribuíram **NULL**). O mesmo vale para o preenchimento de colunas de tabelas do banco de dados Oracle. Se você executa uma declaração **SELECT** para recuperar o valor de determinada coluna e, para algumas linhas retornadas, o valor exibido é **NULL**, quer dizer que nada foi gravado naquela coluna para aquele registro.

Valores `NULL` prejudicam, diretamente, expressões aritméticas e condições *booleanas* (que retornam verdadeiro ou falso), pois o resultado de qualquer soma ou subtração ou divisão ou multiplicação entre um valor qualquer e `NULL`, será `NULL`.

Portanto, tome muito cuidado em utilizar variáveis cujo conteúdo seja `NULL`. Esse é o valor inicial de qualquer variável, antes da inicialização.

Tratamento de exceções

Eventualmente o servidor Oracle ou o aplicativo do usuário causa um erro durante o processamento em *runtime*. Tais erros podem surgir de falhas de hardware ou rede, erros lógicos de aplicativo, erros de integridade de dados e de muitas outras fontes. Esses erros são conhecidos como exceções, ou seja, esses eventos indesejados são exceções do processamento normal e esperado.

Funcionamento geral

Geralmente, quando um erro ocorre, o processamento do bloco PL/SQL é imediatamente encerrado. O processamento corrente não é concluído. A Oracle permite que você esteja preparado para esses erros e implemente lógica nos programas para lidar com os erros, permitindo que o processamento continue. Essa lógica implementada para gerenciar os erros é conhecida como *código de tratamento de exceções*. Com o tratamento de exceções da Oracle, quando um erro é detectado, o controle é passado para a parte de tratamento de exceções do programa e o processamento é completado normalmente. O tratamento de erros também fornece informações valiosas para depurar os aplicativos e para proteger melhor o aplicativo contra erros futuros.

Sem um recurso para tratamento de exceções, um programa deve verificar as exceções a cada comando, como mostrado a seguir:

```
DECLARE
  vMeta      NUMBER :=0;
  vRealizado NUMBER :=10000;
  vPercentual NUMBER :=0;
BEGIN
  -- Recuperar meta do vendedor para o mês corrente
  SELECT r.rep_vl_metamensal
  INTO vMeta
  FROM representante r
  WHERE r.rep_in_codigo = 10;

  /* Só acha percentual se meta for maior que zero
     para não gerar erro de divisão por zero
  */
  IF vMeta > 0 THEN
    vPercentual := (vRealizado / vMeta) * 100;
    dbms_output.put_line('O representante já realizou '||to_char(vPercentual)||
                        ' de sua meta.'
                        );
  ELSE
    vPercentual := 0;
    dbms_output.put_line('O representante não possui meta.');
```

```
END IF;
END;
/
```

Se o tratamento de exceções do PL/SQL fosse usado no código acima, não seria necessário utilizar a estrutura `IF...ELSE...END IF`, deixando assim o código mais limpo e evitando uma verificação obrigatória, mesmo quando a expressão não fosse gerar um erro.

Existem três tipos de exceções na PL/SQL:

- Erros predefinidos da Oracle;
- Erros não definidos da Oracle;

- Erros definidos pelo usuário.

Capturando uma exceção

Se a exceção é invocada na área de execução do bloco e a mesma for manipulada na área de exceção com sucesso, então a execução não propaga para o bloco ou para o ambiente e o bloco termina com sucesso.

Podemos capturar qualquer erro incluindo comandos dentro da área de exceção de um bloco PL/SQL. Cada manipulação consiste de uma cláusula `WHEN` que especifica uma exceção, seguida por uma sequência de comandos que serão executados quando a exceção for invocada.

Sintaxe:

EXCEPTION

```
WHEN exceção1 [OR exceção2 ...] THEN
    Comando1;
    Comando2;
    . . .
WHEN exceção3 [OR exceção4 ...] THEN
    Comando1;
    Comando2;
    . . .
WHEN OTHERS THEN
    Comando1;
    Comando2;
```

IMPORTANTE:

- A área de manipulação de exceções, inicia-se, sempre, com a palavra chave `EXCEPTION`.
- O desenvolvedor deve, sempre, definir um manipulador para cada possível exceção dentro do bloco PL/SQL;
- Quando uma exceção ocorre, o PL/SQL processa somente um manipulador antes de deixar o bloco.
- Coloque a cláusula `OTHERS` depois de todas as cláusulas de manipulação de exceção, pois o seu manipulador será executado caso nenhum outro manipulador corresponda a exceção gerada;
- Podemos ter somente uma cláusula `OTHERS` dentro de uma área de tratamento de exceções;
- Exceções não podem aparecer dentro de comandos SQL.

Exceções pré-definidas da Oracle

O servidor Oracle define vários erros com nomes padrão. Embora todo erro Oracle tenha um número, um erro deve ser referenciado pelo nome. A PL/SQL tem alguns erros e exceções comuns da Oracle predefinidos, os quais incluem o seguinte:

Tabela 3: Exceções pré-definidas da Oracle

Exceções	Descrição
no_data_found	A SELECT de linha única não retornou dados.

<i>Exceções</i>	<i>Descrição</i>
too_many_rows	A SELECT de linha única retornou mais de uma linha.
invalid_cursor	Houve a tentativa de operação ilegal de cursor.
value_error	Ocorreu um erro de aritmética, conversão, <i>truncagem</i> ou restrição.
invalid_number	A conversão de uma <i>string</i> para um número, falhou.
zero_divide	Ocorreu uma tentativa de dividir por zero.
dup_val_on_index	Houve uma tentativa de inserir, em duplicata, um valor em uma coluna (ou um conjunto de colunas) que possui um índice exclusivo (UNIQUE KEY ou PRIMARY KEY).
cursor_already_open	Houve uma tentativa de abrir um cursor que foi aberto anteriormente.
not_logged_on	Uma chamada de banco de dados foi feita sem o usuário estar conectado ao Oracle.
transaction_backed_out	Uma parte remota de uma transação teve "rollback".
login_danied	Um login no banco de dados Oracle falhou por causa de um nome de usuário e/ou senha inválidos.
program_error	A PL/SQL encontrou um problema interno.
storage_error	A PL/SQL ficou sem memória ou a memória está corrompida.
timeout_on_resource	Um timeout ocorreu enquanto o Oracle estava esperando por um recurso
rowtype_mismatch	Uma variável de cursor não é incompatível com a linha de cursor
others	Uma declaração catchall que detecta um erro que não foi detectado nas exceções anteriores

Exemplo:

```
BEGIN
. . .
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    Comando1;
    Comando2;
    . . .
  WHEN TOO_MANY_ROWS THEN
    Comando1;
    Comando2;
    . . .
  WHEN OTHERS THEN
    Comando1;
    Comando2;
END;
/
```

Erros indefinidos da Oracle

Conseguimos capturar erros não definidos pelo Servidor Oracle declarando-os primeiro ou usando o manipulador OTHERS. A exceção declarada é invocada implicitamente. No PL/SQL, o `PRAGMA EXCEPTION_INIT()` "diz" para o compilador para associar uma exceção com um número de erro Oracle. Isto permite referenciar em qualquer exceção interna pelo nome e escrever um manipulador específico para isto.

IMPORTANTE: PRAGMA é uma palavra reservada e uma diretiva de compilação que não é processada quando um bloco é executado. Ele direciona o compilador do PL/SQL para interpretar todas

ocorrências de nomes de exceção dentro de um bloco como número de erro do servidor Oracle.

Exemplo:

```
DECLARE
  -- Definição de exceções
  eResumoDependente EXCEPTION;

  -- Associar exceção definida ao erro do Oracle
  PRAGMA EXCEPTION_INIT (eResumoDependente, -2292);
BEGIN
  -- Excluir cliente
  DELETE cliente
  WHERE cli_in_codigo = 10;
  COMMIT;
EXCEPTION
  -- Tratar erro de dependência entre a tabela RESUMO_MENSAL_VENDA
  WHEN eResumoDependente THEN
    ROLLBACK;
    dbms_output.put_line('O Cliente 10 não pode ser excluído, pois existe resumo
de vendas vinculado.');
```

(Para o servidor Oracle, o erro -2292 é uma violação de integridade)

Erros definidos pelo usuário

Um usuário pode levantar explicitamente uma exceção usando o comando `RAISE`. Esse procedimento deve ser usado apenas quando a Oracle não levanta sua própria exceção ou quando o processamento não é desejado ou é impossível de ser completado.

As etapas para levantar e tratar um erro definido pelo usuário, são os seguintes:

1. Declarar o nome para a exceção de usuário dentro da seção de declaração do bloco;
2. Levantar a exceção explicitamente dentro da parte executável do bloco, usando o comando `RAISE`;
3. Referenciar a exceção declarada com uma rotina de tratamento de erro.

O exemplo seguinte ilustra o uso da exceção definida pelo usuário:

```
DECLARE
  -- Declarar exceção
  eRegistroInexistente exception;
BEGIN
  -- Excluir cliente
  DELETE cliente c
  WHERE c.cli_in_codigo = 1000;

  -- Se cliente não existe, gerar exceção
  IF SQL%NOTFOUND THEN
    raise eRegistroInexistente;
  END IF;

  -- Validar exclusão
```

```
COMMIT;
EXCEPTION
  -- Se registro não existe, informa usuário
  WHEN eRegistroInexistente THEN

    ROLLBACK;
    dbms_output.put_line('Cliente 1000 não existe!' || chr(10) ||
                        'Nenhum registro foi excluído.'
                        );
END;
/
```

SQLCODE e SQLERRM

Quando uma exceção ocorre, podemos identificar o código do erro ou a mensagem de erro usando duas funções. Baseados nos valores do código ou mensagem, podemos decidir qual ação tomar baseada no erro.

As funções são:

- **SQLCODE**: Retorna um valor numérico para o código de erro.
- **SQLERRM**: Retorna um caractere contendo a mensagem associada com o número do erro.

O uso dessas funções pode ser muito útil quando a exceção é detectada com a cláusula **WHEN OTHERS**, a qual é usada para detectar exceções não previstas ou desconhecidas.

Até a versão do Oracle Database 8i, você não pode usar **SQLCODE** e **SQLERRM** diretamente em uma declaração SQL. Em vez disso, você deve atribuir seus valores às variáveis locais e depois usar essas variáveis na declaração SQL (nas versões seguintes, isso já é possível).

A função **SQLCODE** retorna o código de erro da exceção.

SQLERRM, retorna a mensagem de erro correspondente o código de erro da exceção.

Quando nenhuma exceção foi levantada, o valor de **SQLCODE** é 0 (zero) e quando a exceção foi declarada pelo usuário, o valor é 1;

Abaixo, segue um exemplo de uso das funções **SQLCODE** e **SQLERRM**:

```
DECLARE
  -- Declaração de variáveis
  vCodeError    NUMBER := 0;
  vMessageError VARCHAR2(255) := '';

  -- Declarar exceção
  eRegistroInexistente exception;
BEGIN
  -- Excluir cliente
  DELETE cliente c
  WHERE c.cli_in_codigo = 1000;

  -- Se cliente não existe, gerar exceção
  IF SQL%NOTFOUND THEN
    raise eRegistroInexistente;
```

```
END IF;

-- Validar exclusão
COMMIT;
EXCEPTION
-- Se registro não existe, informa usuário
WHEN eRegistroInexistente THEN

    ROLLBACK;
    dbms_output.put_line('Cliente 1000 não existe!' || chr(10) ||
                        'Nenhum registro foi excluído.'
                        );

WHEN OTHERS THEN
    ROLLBACK;
    vCodeError := SQLCODE;
    vMessageError := SQLERRM;

    -- Imprime na tela o código do erro e a mensagem
    dbms_output.put_line('Cód.Erro: ' || to_char(vCodeError) || chr(10) ||
                        'Mensagem: ' || vMessageError
                        );
END;
/
```

Como pode ser visto no exemplo acima, caso não exista o registro a ser excluído, o bloco levanta a exceção `eRegistroInexistente` e se ocorrer qualquer outra exceção, é executado um `ROLLBACK` e o código do erro, mais sua descrição, são exibidos na tela.

O procedimento `raise_application_error`

Usando o procedimento `RAISE_APPLICATION_ERROR`, podemos comunicar uma exceção pré-definida retornando um código de erro e uma mensagem de erro não padrão. Com isto, podemos retornar erros para a aplicação e evitar exceções não manipuladas.

Sintaxe:

```
raise_application_error(num_erro, mensagem[, {TRUE | FALSE }]);
```

Onde:

- `num_erro` é número pré-definido pelo usuário para exceções entre -20000 e -20999;
- `mensagem` é uma mensagem pré-definida pelo usuário para a exceção. Pode ser um string de até 2048 bytes;
- `TRUE|FALSE` é um parâmetro booleano opcional. Se `TRUE`, o erro é colocado na fila dos próximos erros e se for `FALSE`, o padrão, o erro substitui todos os erros precedentes.

Exemplo:

```
DECLARE
-- Declarar exceção
eRegistroInexistente exception;
BEGIN
-- Excluir cliente
DELETE cliente c
WHERE c.cli_in_codigo = 1000;
```

```
-- Se cliente não existe, gerar exceção
IF SQL%NOTFOUND THEN
    raise eRegistroInexistente;
END IF;

-- Validar exclusão
COMMIT;
EXCEPTION
-- Se registro não existe, informa usuário
WHEN eRegistroInexistente THEN

    ROLLBACK;
    raise_application_error(-20100, 'Cliente 1000 não existe!' || chr(10) ||
                                'Nenhum registro foi excluído.'
                                );

WHEN OTHERS THEN

    ROLLBACK;
    raise_application_error(-20101, 'Ocorreu um erro não identificado' || chr(10) ||
                                'ao tentar excluir o cliente 1000.');
```

END;

/

IMPORTANTE: O procedimento `RAISE_APPLICATION_ERROR` pode ser invocado do bloco principal (não é exclusividade da área de exceção)

Regras de escopo de exceção

Você precisa conhecer as orientações abaixo sobre o escopo das declarações:

- Uma exceção não pode ser declarada duas vezes no mesmo bloco, mas a mesma exceção pode ser declarada em dois blocos diferentes;
- As exceções são locais ao bloco onde elas foram declaradas e globais a todos os sub-blocos do bloco. Os blocos incluídos (bloco pai) não podem referenciar as exceções que foram declaradas em nenhum de seus sub-blocos;
- As declarações globais podem ser declaradas de novo no nível do sub-bloco local. Se isso ocorrer, a declaração local tem precedência sobre a declaração global.

Propagando as exceções

Quando um erro é encontrado, a PL/SQL procura o tratamento de exceção apropriado no bloco atual. Se nenhum tratamento estiver presente, então a PL/SQL *propaga* o erro para o bloco incluído (bloco pai). Se nenhum tratamento for encontrado lá, o erro é propagado para os blocos incluídos até que um tratamento seja encontrado. Esse processo pode continuar até o ambiente *host* receber e lidar com o erro. Por exemplo, se o SQL*Plus recebeu um erro não tratado de um bloco PL/SQL, o SQL*Plus lida com esse erro exibindo o código de erro e a mensagem na tela do usuário.

Exercícios propostos

1. Escreva um bloco PL/SQL que recupere a quantidade de registros de uma tabela qualquer e imprima essa quantidade na tela, usando o SQL*Plus. Salve esse bloco em um arquivo do sistema operacional e execute-o novamente no SQL*Plus, usando o comando @.
2. Edit o bloco escrito no exercício anterior no *Command Window* do PL/SQL Developer e execute-o.
3. Escreva um bloco PL/SQL que declare uma variável e exiba o valor. Depois, adicione um bloco aninhado que declara uma variável de mesmo nome e exiba seu valor.
4. Escreva um bloco que declare uma variável do tipo date para guardar a data de nascimento de uma pessoa. Atribua uma data para essa variável, calcule a idade da pessoa com base na data atual (SYSDATE) e imprima essa idade em anos. Dica: Uma subtração entre datas, em PL/SQL, retorna a diferença em dias.
5. Altere o bloco escrito no exercício quatro, eliminando a variável criada para armazenar a data de nascimento e criando um tipo que guarde: o nome da pessoa, a data de nascimento e a idade. Defina uma variável do tipo criado. Logo no início do bloco, atribua o nome da pessoa e a data de nascimento para a variável. Após calcular a idade da pessoa, atribua o resultado para o campo de idade da variável. Por fim, imprima na tela o nome da pessoa, a data de nascimento e a sua idade.
6. Escreva um bloco PL/SQL que recupere dados de uma tabela qualquer e armazene seu retorno em uma variável do tipo registro. O bloco deve conter uma área de exceção para tratar os seguintes erros:
 - a) Muitas linhas retornadas pela instrução SQL;
 - b) Nenhuma linha retornada pela instrução SQL;
 - c) Outros erros.

02 – Estruturas PL/SQL

1. **Cenário para prática**
2. **Declarações IF e LOOPS;**
3. **Utilização de cursores;**
4. **Blocos anônimos, procedimentos e funções.**

Cenário para prática

Até agora vimos muitas explicações e exemplos, mas praticamos pouco.

Nos próximos capítulos precisaremos exercitar o que for visto e, de preferência, num ambiente de teste que simule algum sistema, mesmo que simplório.

Para facilitar isso, usaremos um modelo de dados de um sistema de vendas bem simples, que não chega a normalização ideal, porém, atende perfeitamente às nossas necessidades.

Modelo lógico de entidade e relacionamento

Abaixo segue o nosso modelo lógico de entidade e relacionamento:

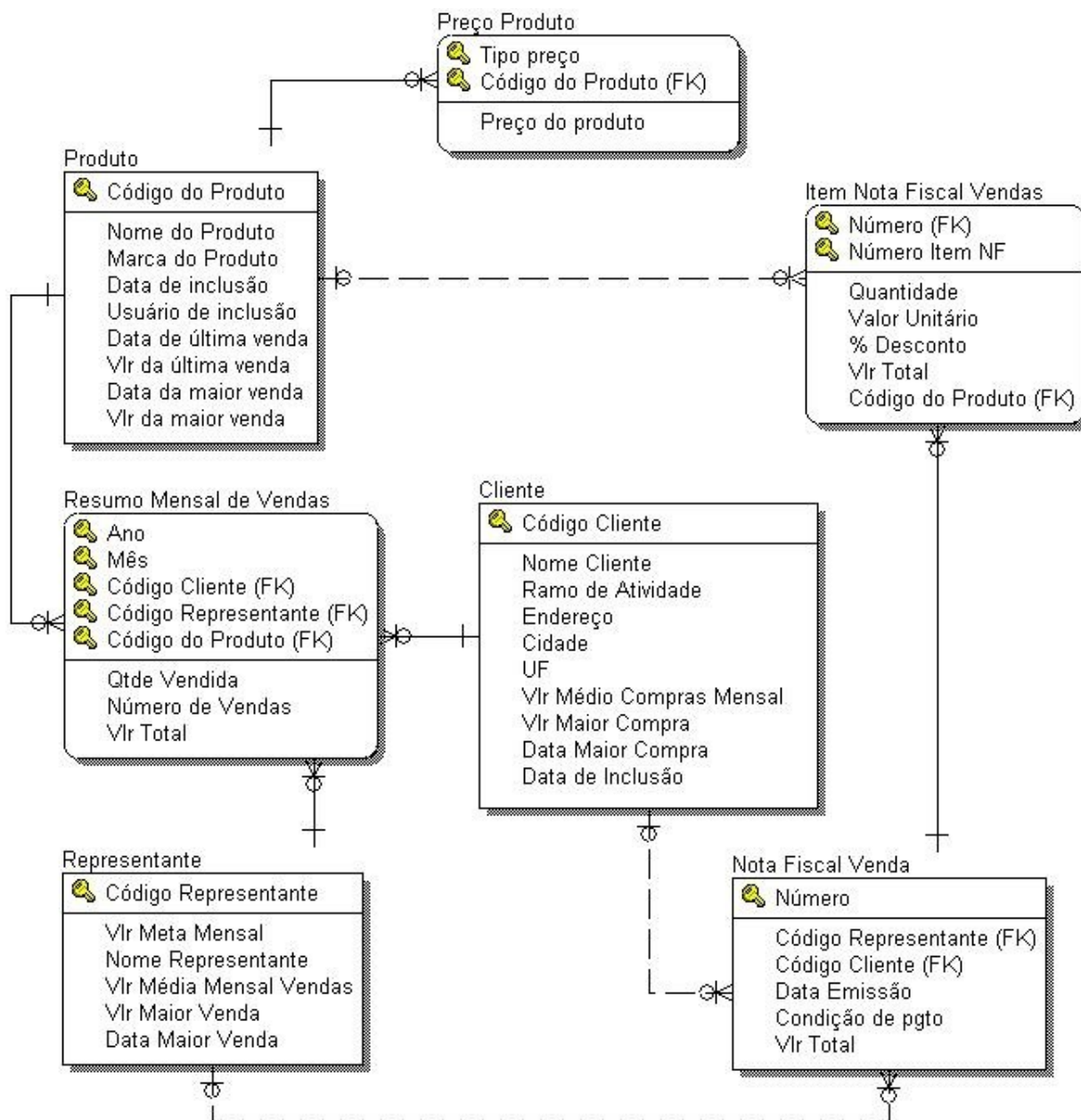


Figura 18: Modelo Lógico de Entidade Relacionamento do sistema de vendas

Modelo físico de entidade e relacionamento

Para entendermos como as entidades se relacionam, o modelo lógico é a melhor opção, pois fornece uma representação mais limpa, porém, quando precisamos de detalhes de implementação física, como o tipo de uma coluna para elaborar uma instrução SQL, a melhor representação é o modelo físico de entidade e relacionamento, como na figura a seguir:

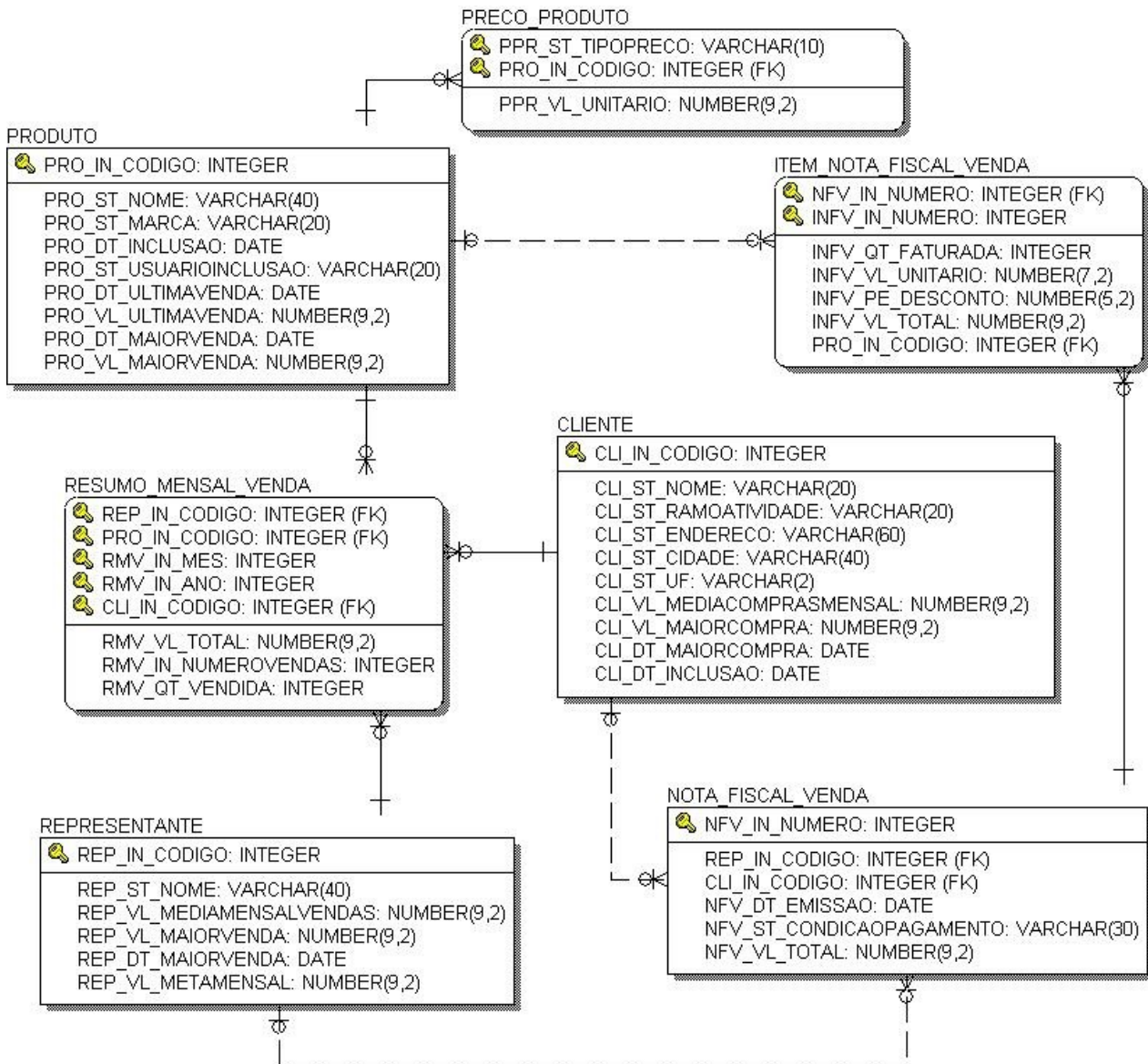


Figura 19: Modelo Físico de Entidade Relacionamento do sistema de vendas

De agora em diante, procuraremos elaborar os exercícios com base neste modelo.

A criação dos objetos e a criação da massa de dados encontra-se no arquivo TreinamentoPLSQL.sql.

Crie a tablespace USERS, caso ainda não exista, e crie um *schema* (USER) com o seu nome para ser o proprietário dos objetos.

Declarações IF e LOOPS

A declaração IF

A declaração `IF` permite avaliar uma ou mais condições, como em outras linguagens.

A sintaxe de uma declaração `IF` no PL/SQL é a seguinte:

```
IF <condição> THEN
    <comandos a serem executados>
END IF;
```

Nessa sintaxe, *condição* é a condição que você quer verificar. Se a condição for avaliada como `TRUE`, ou seja, verdadeira, os comandos de execução serão executados.

Como exemplo, abaixo temos um bloco PL/SQL que gera uma exceção se não existirem clientes cadastrados:

```
DECLARE
    vQtdeClientes INTEGER;
BEGIN
    -- Recuperar quantidade de clientes
    SELECT COUNT(*)
    INTO vQtdeClientes
    FROM cliente;

    -- Se não existir clientes gera uma exceção
    IF vQtdeClientes = 0 THEN
        raise_application_error(-20100, 'Não existem clientes cadastrados');
    END IF;
END;
/
```

A declaração IF...THEN...ELSE

No exemplo anterior você não se importou com os resultados quando existem clientes. A declaração `IF...THEN...ELSE` permite processar uma série de declarações abaixo de `ELSE` se a condição for falsa.

A sintaxe da declaração `IF...THEN...ELSE` é:

```
IF <condição> THEN
    <comandos a serem executados para condição verdadeira>
ELSE
    <comandos a serem executados para condição verdadeira>
END IF;
```

Nessa sintaxe, se a condição for `FALSE`, é executado os comandos abaixo de `ELSE`.

Para exemplificar o uso da declaração `IF...THEN...ELSE`, vamos modificar o exemplo anterior para gerar uma exceção quando não existir clientes ou imprimir na tela a quantidade de clientes cadastrados quando existir:

```
DECLARE
    vQtdeClientes INTEGER;
BEGIN
    -- Recuperar quantidade de clientes
    SELECT COUNT(*)
    INTO vQtdeClientes
    FROM cliente;
```

```
-- Se não existir clientes gera uma exceção
IF vQtdeClientes = 0 THEN
    raise_application_error(-20100,'Não existem clientes cadastrados');
ELSE
    dbms_output.put_line('Existem '||to_char(vQtdeClientes)|| ' cadastrados.');
```

END IF;

END;

/

Assim como em outras linguagens, você pode aninhar as declarações `IF` e `IF...THEN...ELSE`, como mostrado abaixo:

```
IF <condição_01> THEN
    IF <condição_02> THEN
        ...
        ...
    ELSE
        IF <condição_03> THEN
            ...
            ...
        END IF;
        ...
    END IF;
    ...
    ...
END IF;
```

A declaração IF...ELSIF

Você pode imaginar as declarações `IF` aninhadas como executando um `AND` lógico mas `ELSIF` como executando um `OR` lógico. O recurso bom no uso de `ELSIF` no lugar de `IF` aninhado é que fica muito mais fácil seguir a lógica da declaração `ELSIF` porque você pode identificar facilmente quais declarações ocorrem em quais condições lógicas, como pode ser visto abaixo:

```
IF <condição_01> THEN
    IF <condição_02> THEN
        ...
        ...
    ELSIF <condição_03> THEN
        ...
        ...
    ELSIF <condição_04> THEN
        ...
        ...
    ELSE
        ...
    END IF;
END IF;
```

Como podemos observar, ao final de uma sequência de `ELSIF`, podemos ter um `ELSE`. Isso quer dizer que se nenhuma das condições `IF` ou `ELSIF` for atendida, os comandos abaixo de `ELSE` serão executados.

Loop Simples

De todos os loops, esse é o mais simples de usar e entender. Sua sintaxe é a seguinte:

```
LOOP
    <<Declarações>>
END LOOP;
```

Como vemos na sintaxe, esse loop não tem uma cláusula de saída, ou seja, ele é infinito. Para abandonar o loop podemos utilizar duas declarações: `EXIT` ou `EXIT WHEN`.

Exemplo de loop simples:

```
BEGIN
  LOOP
    NULL;
    EXIT;
  END LOOP;
END;
/
```

O exemplo acima não faz nada, mas é suficiente para exemplificar o funcionamento de um loop simples.

Criando um loop REPEAT...UNTIL

Em PL/SQL não existe um loop do tipo `REPEAT...UNTIL` incorporado, entretanto, você pode simular um usando o loop simples e as declarações `EXIT` ou `EXIT WHEN`.

A sintaxe de um loop `REPEAT...UNTIL`, em PL/SQL, seria da seguinte forma:

```
LOOP;
  <Declarações>
  IF <condicao> THEN
    EXIT;
  END IF;
END LOOP;
```

Alternativamente você pode usar o método mais comum, que é:

```
LOOP;
  <Declarações>
  EXIT WHEN <condição>;
END LOOP;
```

Loop FOR

Loop `FOR` têm a mesma estrutura de um loop simples. A diferença é que possui um comando de controle antes da palavra chave `LOOP`, para determinar o número de iterações que o PL/SQL irá realizar.

Sintaxe:

```
FOR contador IN [REVERSE] limite_inferior..limite_superior LOOP
  Comando1;
  Comando2;
END LOOP;
```

Onde:

Contador é uma declaração inteira implícita cujo valor é incrementado ou decrementado(se a palavra

chave `REVERSE` for usada) automaticamente em 1 em cada iteração do loop, até o limite inferior ou limite superior ser alcançado. Não é preciso declarar nenhum contador, ele é declarado implicitamente como um inteiro.

IMPORTANTE: A sequência de comandos é executado cada vez que o contador é incrementado, como determinado pelos dois limites inferior e superior.

Os limites inferior e superior podem ser literais, variáveis ou expressões, mas devem ser valores inteiros. Se o limite inferior for um valor maior que o limite superior, a sequência de comandos não será executada.

Exemplo de um loop que vai de 1 à 5 e imprime na tela o número de cada loop:

```
BEGIN
  FOR LoopCount IN 1..5 LOOP
    dbms_output.put_line('Loop ' || to_char(LoopCount));
  END LOOP;
END;
/
```

IMPORTANTE: O Oracle não fornece opções para passar por um loop com um incremento que não seja um e o valor do contador não pode ser alterado durante o loop. Você pode escrever loops que são executados com um incremento diferente executando as declarações apenas quando determinada condição é verdadeira. O exemplo abaixo demonstra como incrementar por um valor de 2:

```
BEGIN
  FOR vLoopCount IN 1..6 LOOP
    IF MOD(vLoopCount,2) = 0 THEN
      dbms_output.put_line('Contador é igual a ' || to_char(vLoopCount));
    END IF;
  END LOOP;
END;
/
```

Quando executar esse bloco, o resultado deve ser o seguinte:

```
Contador é igual a 2
Contador é igual a 4
Contador é igual a 6
```

Este exemplo mostra apenas uma das várias maneiras pelas quais você pode incrementar um loop. A função `MOD`, neste caso, simplesmente testa para ter certeza de que o número é divisível igualmente por um valor de 2. Você pode alterar isso, facilmente, para 3, 5 ou o que quiser incrementar. Para decremento basta adicionar a palavra-chave `REVERSE`.

Loop WHILE

Podemos usar o Loop `WHILE` para repetir uma sequência de comandos enquanto a condição de controle for `TRUE`. A condição é avaliada no início de cada iteração. O loop termina quando a condição for `FALSE`. Se a condição for `FALSE` no início do Loop, então nenhuma iteração será realizada.

Sintaxe:

```
WHILE condição LOOP
```

```
Comando1;  
Comando2;  
END LOOP;
```

Se a variável utilizada na condição não mudar durante o corpo do loop, então a condição irá permanecer `TRUE` e o loop nunca terminará.

Se a condição retornar `NULL`, o loop será encerrado e o controle passará para o próximo comando.

Abaixo temos o exemplo de um loop que nunca será executado:

```
DECLARE  
vCalc NUMBER := 0;  
BEGIN  
  WHILE vCalc >= 10 LOOP  
    vCalc := vCalc + 1;  
    dbms_output.put_line('O valor de vCalc é '||vCalc);  
  END LOOP;  
END;  
/
```

E abaixo uma versão corrigida do exemplo anterior, para que o loop seja executado:

```
DECLARE  
vCalc NUMBER := 0;  
BEGIN  
  WHILE vCalc <= 10 LOOP  
    vCalc := vCalc + 1;  
    dbms_output.put_line('O valor de vCalc é '||vCalc);  
  END LOOP;  
END;  
/
```

Qual loop devo usar?

Todas essas opções de loops podem causar confusão! Como foi visto nos exemplos, você pode usar as declarações `FOR`, `WHILE` e `LOOP` para criar a mesma saída. Entretanto, a Tabela 5 mostra algumas orientações gerais sobre quando usar qual tipo de loop.

Tabela 4: Tipo de Loop a ser usado

Loop	Quando usar
LOOP (Simples)	Você pode usar o LOOP simples se quiser criar um loop do tipo REPEAT...UNTIL. O LOOP simples é perfeito para executar essa tarefa
FOR	Use sempre o loop FOR se você souber especificamente quantas vezes o loop deve ser executado. Se tiver de codificar uma declaração EXIT ou EXIT WHEN em um loop FOR, você pode reconsiderar seu código e usar um loop simples ou uma abordagem diferente.
WHILE	Use este loop quando você pode nem querer executar o loop uma vez. Embora você possa duplicar esse resultado em um loop FOR usando EXIT ou EXIT WHEN, essa situação é mais adequada para o loop WHILE. O loop WHILE é o loop mais usado porque ele fornece mais flexibilidade

Utilização de Cursores

Os cursores da PL/SQL fornecem um modo pelo qual seu programa pode selecionar várias linhas de dados do banco de dados e depois processar cada linha individualmente. Especificamente, um *cursor* é um nome atribuído pela Oracle para cada declaração SQL que é processada. Esse nome fornece à Oracle um meio de orientar e controlar todas as fases do processamento da SQL.

Conceitos básicos

Existem dois tipos de cursores:

- **Cursor Implícito:** são cursores declarados pelo PL/SQL implicitamente para todos comandos DML e comandos `SELECT` no PL/SQL, independente da quantidade de registros processadas. Ele precisa fazer isso para gerenciar o processamento da declaração SQL.
- **Cursor Explícito:** Cursores definidos pelo usuário para manipular registros recuperados por declarações `SELECT`.

Cursores Explícitos

Utilizamos cursores explícitos para processar individualmente cada linha retornada por um comando `SELECT`. O conjunto de linhas retornadas pelo comando `SELECT` é chamado de *conjunto ativo*.

Um programa PL/SQL abre o cursor, processa as linhas retornadas pela consulta e depois fecha este cursor. O cursor marca a posição corrente dentro do conjunto ativo.

Usando cursores explícitos, o programador pode manipular cada registro recuperado por uma declaração `SELECT`.

Abaixo vemos o ciclo de vida de um cursor:

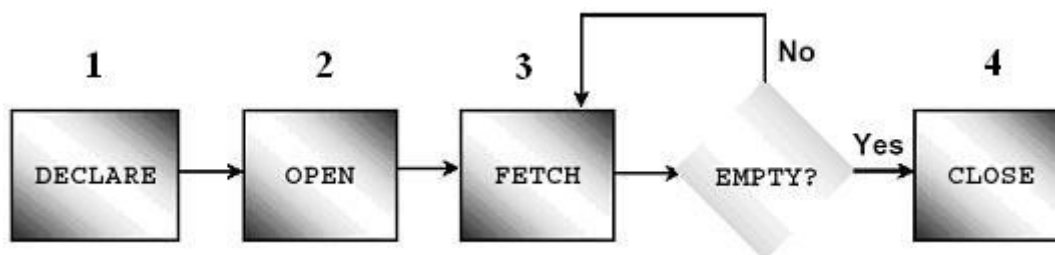


Figura 20: Ciclo de vida de um cursor

1. Declare o cursor nomeando o mesmo e defina a estrutura da consulta que será realizada por ele;
2. Através do comando `OPEN`, o cursor é aberto e executa a consulta que recuperará o conjunto ativo do banco de dados;
3. Com o comando `FETCH`, capturamos os dados do registro corrente para utilizarmos em nosso programa. A cada comando `FETCH`, devemos testar se ainda existe registro no cursor e abandonar o `LOOP` do cursor através do comando `EXIT`, caso não exista (mais adiante veremos como verificar isso);
4. Quando a manipulação dos dados do cursor for finalizada, ao abandonar o `LOOP` do cursor, devemos fechá-lo através do comando `CLOSE`, que libera as linhas do cursor;

Declarando um cursor

Declaramos um cursor utilizando o comando **CURSOR**. Podemos referencia variáveis na query, mas elas precisam ser declaradas antes do comando CURSOR.

Sintaxe:

```
CURSOR nome_cursor IS
```

```
Comando_select;
```

Onde:

nome_cursor é um indentificador PL/SQL.

Comando_select é um comando SQL sem a cláusula INTO.

Observações:

- Não colocar a cláusula INTO na declaração de um cursor, pois ela aparecerá depois no comando FETCH.
- Um cursor pode ser qualquer comando SELECT, incluindo joins e outros.

Exemplo:

```
DECLARE
CURSOR cs_representante IS
  SELECT r.rep_in_codigo, r.rep_st_nome
  FROM representante r;
CURSOR cs_notafiscal IS
  SELECT *
  FROM nota_fiscal_venda;
BEGIN
  NULL;
END;
/
```

No exemplo acima, declaramos dois cursores: *cs_representante* e *cs_notafiscal*. Em *cs_representante* será recuperado o código e o nome de todos os representantes. No cursor *cs_notafiscal*, será recuperado todos os dados de todas as notas fiscais..

Abrindo um Cursor

O comando **OPEN** executa a consulta associada ao cursor e posiciona o cursor antes da primeira linha do conjunto ativo.

Sintaxe:

```
OPEN nome_cursor;
```

Onde:

- *nome_cursor* é o nome previamente declarado do cursor.
- **OPEN** é um comando executável que realiza as seguintes operações:
 - Aloca dinamicamente área de memória para processamento;
 - Faz o **PARSE** do comando **SELECT**;

- Coloca valores nas variáveis de input obtendo seus endereços de memória;
- Identifica o conjunto de linhas que a consulta retornou. (As linhas retornadas não são colocadas nas variáveis quando o comando `OPEN` é executado, isto acontece quando o comando `FETCH` é utilizado);
- Posiciona o ponteiro antes da primeira linha do conjunto ativo.

IMPORTANTE: Se a consulta não retornar linhas quando o cursor for aberto, o PL/SQL não retorna nenhuma exceção. Entretanto, podemos testar o status do cursor depois de cada comando `FETCH` usando o atributo de cursor `SQL%ROWCOUNT`, que será estudado mais adiante.

Recuperando dados de um Cursor

O comando `FETCH` recupera linhas da consulta associada ao cursor. Para cada comando `FETCH` executado, o cursor avança para a próxima linha no conjunto ativo.

Sintaxe:

```
FETCH nome_cursor INTO [variável1, variável2, ...] / nome_registro;
```

Onde:

- *nome_cursor* é o nome do cursor declarado na sessão `DECLARE`;
- *variáveln* é o nome da variável declarada para armazenar os valores retornados no `FETCH`;
- *nome_registro* é o nome do registro no qual os dados retornados são armazenados. Esta variável pode ser declarada usando o atributo `%ROWTYPE` ou uma variável de tipo registro;

IMPORTANTE:

- A cláusula `INTO` deve conter:
 - uma variável para cada coluna retornada pelo comando `FETCH`, na ordem correspondente das colunas retornadas e com o mesmo tipo retornado;
 - Ou um Tipo Registro com o mesmo número de colunas do registro retornado (e do mesmo tipo das colunas do cursor);
 - Ou uma variável declarada com `%ROWTYPE` baseada no próprio cursor.
- Valide se o cursor contém linhas. Se quando executar o comando `FETCH`, o cursor estiver vazio, não teremos nenhuma linha para processar e nenhum erro será exibido;
- O comando `FETCH` só pode ser executado se o Cursor estiver aberto.

Exemplo:

```
DECLARE  
  -- Declaração de variáveis  
  vCodigoRep representante.rep_in_codigo%type;  
  vNomeRep   representante.rep_st_nome%type;  
  
  -- Declaração de cursores  
  CURSOR cs_representante is  
    SELECT rep_in_codigo, rep_st_nome  
    FROM representante;  
BEGIN
```

```
-- Abre cursor
OPEN cs_representante;

-- Executa um loop com 10 ciclos
FOR i IN 1..10 LOOP
    -- Extrai dados o registro corrente do cursor e avança para o próximo
    FETCH cs_representante INTO vCodigoRep , vNomeRep;
    -- Imprime dados extraídos na tela
    dbms_output.put_line(vCodigoRep||' - '||vNomeRep);
END LOOP;
END;
/
```

No exemplo acima, o programa imprime na tela 10 representantes, um a um.

Fechando o Cursor

O comando `CLOSE` desabilita o cursor e, conseqüentemente, o conjunto ativo torna-se indefinido. Sempre que o processamento do comando `SELECT` estiver completo, feche o cursor, isto permite que o cursor seja reaberto quando preciso.

Sintaxe:

```
CLOSE nome_cursor;
```

Se for aplicado o comando `FETCH` em um cursor fechado, a exceção `INVALID_CURSOR` ocorrerá.

O comando `CLOSE` libera a área de memória alocada para o cursor. É possível terminar um bloco PL/SQL sem precisar fechar o cursor, mas é importante que se torne um hábito fechá-lo, pois assim, estaremos poupando recursos da máquina.

IMPORTANTE: Existe um limite máximo de cursores abertos por usuário que é determinado pelo parâmetro do Oracle `OPEN_CURSORS`.

Exemplo:

```
DECLARE
    -- Declaração de variáveis
    vCodigoRep representante.rep_in_codigo%type;
    vNomeRep    representante.rep_st_nome%type;

    -- Declaração de cursores
    CURSOR cs_representante IS
        SELECT rep_in_codigo, rep_st_nome
        FROM representante;
BEGIN
    -- Abre cursor
    OPEN cs_representante;

    -- Executa um loop com 10 ciclos
    FOR i IN 1..10 LOOP
        -- Extrai dados o registro corrente do cursor e avança para o próximo
        FETCH cs_representante INTO vCodigoRep , vNomeRep;
        -- Imprime dados extraídos na tela
        dbms_output.put_line(vCodigoRep||' - '||vNomeRep);
    END LOOP;
END;
```

```
END LOOP;  
  
-- Fechar cursor  
CLOSE cs_representante;  
END;  
/
```

Atributos de cursores explícitos

A PL/SQL disponibiliza quatro atributos de cursor que são muito úteis, como mostrado na tabela a seguir:

Tabela 5: Atributos de cursor explícito

Exceções	Descrição
%rowcount	Mostra o número de linhas do cursor
%found	Retorna TRUE se o mais recente FETCH retornar uma linha.
%notfound	Retorna TRUE se o mais recente FETCH não retornar uma linha.
%isopen	Retorna TRUE se o cursor estiver aberto.

Um atributo de cursor explícito não pode ser referenciado diretamente de uma instrução SQL. É necessário atribuir o seu retorno para uma variável.

Para extrair o valor de um atributo, ele deve ser precedido do nome do cursor, como no exemplo abaixo:

```
DECLARE  
-- Declaração de variáveis  
vCodigoRep representante.rep_in_codigo%type;  
vNomeRep representante.rep_st_nome%type;  
  
-- Declaração de cursores  
CURSOR cs_representante IS  
SELECT rep_in_codigo, rep_st_nome  
FROM representante;  
BEGIN  
-- Abre cursor se ainda não estiver aberto  
IF NOT cs_representante%ISOPEN THEN  
OPEN cs_representante;  
END IF;  
  
-- Executa um loop com 10 ciclos  
FOR i IN 1..10 LOOP  
-- Extrai dados o registro corrente do cursor e avança para o próximo  
FETCH cs_representante INTO vCodigoRep , vNomeRep;  
-- Imprime dados extraídos na tela  
dbms_output.put_line(vCodigoRep||' - '||vNomeRep);  
END LOOP;  
  
-- Fechar cursor  
CLOSE cs_representante;  
END;  
/
```

No exemplo acima, verificamos se o cursor já estava aberto, usando para isso o atributo %ISOPEN.

Além de utilizar o atributo %ISOPEN para verificar se o cursor está aberto, é recomendado o uso do atributo %NOTFOUND para determinar quando sair do LOOP. Este atributo retorna FALSE se o último FETCH

retornar uma linha e `TRUE` se não retornar (Antes do primeiro `FETCH`, o valor do atributo é `NULL`).

Abaixo, temos um exemplo do uso do atributo `%NOTFOUND`:

```
DECLARE
-- Declaração de variáveis
vCodigoRep representante.rep_in_codigo%type;
vNomeRep    representante.rep_st_nome%type;

-- Declaração de cursores
CURSOR cs_representante IS
    SELECT rep_in_codigo, rep_st_nome
    FROM representante;
BEGIN
-- Abre cursor se ainda não estiver aberto
IF NOT cs_representante%ISOPEN THEN
    OPEN cs_representante;
END IF;

-- Executa um loop com 10 ciclos
LOOP
-- Extrai dados o registro corrente do cursor e avança para o próximo
    FETCH cs_representante INTO vCodigoRep , vNomeRep;

-- Sai do Loop quando não houver mais registros para processar
    EXIT WHEN cs_representante%NOTFOUND;

-- Imprime dados extraídos na tela
    dbms_output.put_line(vCodigoRep||' - '||vNomeRep);
END LOOP;

-- Fechar cursor
CLOSE cs_representante;
END;
/
```

O atributo `%ROWCOUNT` também pode ser de grande utilidade, mas lembre-se que:

- Antes do primeiro `FETCH` (mesmo com o cursor já aberto), seu valor é 0 (zero);
- Se o cursor não estiver aberto e for referenciado pelo atributo `%ROWCOUNT`, ocorrerá a exceção `INVALID_CURSOR`.

Para processar diversas linhas de um cursor explícito, definimos um `LOOP` para realizar um `fetch` em cada iteração. Eventualmente, todas as linhas do cursor são processadas e quando um `FETCH` não retornar mais linhas, o atributo `%NOTFOUND` passa a ser `TRUE`. Use os atributos de cursores explícitos para testar o sucesso de cada `FETCH`, antes de fazer qualquer referência no cursor. Um `LOOP` infinito irá acontecer se nenhum critério de saída for utilizado.

Cursores, registros e o atributo `%ROWTYPE`

Ao invés de declararmos uma variável para cada coluna retornada por um cursor, podemos definir registros com a mesma estrutura do cursor e declarar variáveis do tipo desses registros.

O mais prático é definir essa variável de registro utilizando o atributo `%ROWTYPE`, mas é claro que

cada caso deve ser estudado com cautela.

É conveniente utilizar o atributo %ROWTYPE, pois as linhas do cursor serão carregadas neste registro e seus valores carregados diretamente nos campos deste registro.

A seguir temos dois exemplos. No primeiro mostramos a utilização do Tipo registro e no segundo, o uso do atributo %ROWTYPE:

Exemplo do uso de Tipo Registro

```
DECLARE
-- Declaração de tipo registro
TYPE TRepresentante IS RECORD(
    Codigo representante.rep_in_codigo%type,
    Nome    representante.rep_st_nome%type
);
-- Declaração de variáveis
rRep TRepresentante;
-- Declaração de cursores
CURSOR cs_representante is
    SELECT rep_in_codigo, rep_st_nome
    FROM representante;
BEGIN
-- Abre cursor se ainda não estiver aberto
IF NOT cs_representante%ISOPEN THEN
    OPEN cs_representante;
END IF;

-- Executa um loop com 10 ciclos
LOOP
-- Extrai dados o registro corrente do cursor e avança para o próximo
FETCH cs_representante INTO rRep;

-- Sai do Loop quando não houver mais registros para processar
EXIT WHEN cs_representante%NOTFOUND;

-- Imprime dados extraídos na tela
dbms_output.put_line(rRep.Codigo||' - '||rRep.Nome);
END LOOP;

-- Fechar cursor
CLOSE cs_representante;
END;
/
```

No exemplo acima, embora seja melhor do que definir uma variável para cada coluna do cursor, se a SELECT do cursor for alterada para retornar mais colunas ou menos, o TYPE TRepresentante também precisará ser alterado.

Por esse motivo, o uso do atributo %ROWTYPE é muito mais simples, como podemos ver no segundo exemplo:

```
DECLARE
-- Declaração de cursores
CURSOR cs_representante is
    SELECT rep_in_codigo, rep_st_nome
    FROM representante;
```

```
-- Declaração de variáveis
rRep cs_representante%ROWTYPE;
BEGIN
-- Abre cursor se ainda não estiver aberto
IF NOT cs_representante%ISOPEN THEN
    OPEN cs_representante;
END IF;

-- Executa um loop com 10 ciclos
LOOP
    -- Extrai dados o registro corrente do cursor e avança para o próximo
    FETCH cs_representante INTO rRep;

    -- Sai do Loop quando não houver mais registros para processar
    EXIT WHEN cs_representante%NOTFOUND;

    -- Imprime dados extraídos na tela
    dbms_output.put_line(rRep.rep_in_codigo||' - '||rRep.rep_st_nome);
END LOOP;

-- Fechar cursor
CLOSE cs_representante;
END;
/
```

Dessa maneira, qualquer alteração na estrutura do cursor, refletirá na variável que `rRep`.

Cursor explícitos automatizados (LOOP Cursor FOR)

Os `LOOPs Cursor FOR` são ideais quando você quer fazer o `LOOP` em todos os registros retornados pelo cursor. Com o `LOOP Cursor FOR` você não deve declarar o registro que controla o `LOOP`. Da mesma forma, você não deve usar o `LOOP Cursor FOR` quando as operações do cursor precisarem ser tratadas manualmente.

Um `LOOP Cursor FOR` faz as seguintes coisas implicitamente:

1. Declara o índice do `LOOP`;
2. Abre o cursor;
3. Faz o `FETCH` da linha seguinte a partir do cursor para cada iteração do `LOOP`;
4. Fecha o cursor quando todas as linhas são processadas ou quando o `LOOP` é encerrado.

Um `LOOP Cursor FOR` processa linhas em um cursor explícito. Isto é uma facilidade por que o cursor é aberto, linhas são carregadas em cada iteração do `LOOP`, o `LOOP` é finalizado quando a última linha é processada e o cursor é fechado, automaticamente.

Sintaxe:

```
FOR nome_registro IN nome_cursor LOOP
    Comando1;
    Comando2;
    Comando3;
END LOOP;
```

Onde:

- *nome_registro* é o nome do registro implícito declarado;
- *nome_cursor* é o nome do cursor previamente declarado pelo usuário.

Exemplo:

```
DECLARE
-- Declaração de cursores
CURSOR cs_representante IS
    SELECT rep_in_codigo, rep_st_nome
    FROM representante;
BEGIN
-- Inicia o loop no conjunto ativo do cursor
FOR rRep IN cs_Representante LOOP
    -- Imprime dados extraídos na tela
    dbms_output.put_line(rRep.rep_in_codigo||' - '||rRep.rep_st_nome);
END LOOP;
END;
/
```

IMPORTANTE:

- Não declare o registro que controla o LOOP por que ele é declarado implicitamente;
- Não use LOOP Cursor FOR quando as operações do cursor tiverem que ser manipuladas explicitamente.

LOOP Cursor FOR usando Sub-Consultas

Quando usamos uma Sub-consulta em um LOOP Cursor FOR, não precisamos declarar um cursor.

Veja abaixo:

```
BEGIN
-- Inicia o loop no conjunto ativo do cursor
FOR rRep IN (SELECT rep_in_codigo, rep_st_nome
             FROM representante)
LOOP
    -- Imprime dados extraídos na tela
    dbms_output.put_line(rRep.rep_in_codigo||' - '||rRep.rep_st_nome);
END LOOP;
END;
/
```

Embora pareça simples, o uso desse recurso pode não ser favorável a formatação do código e, conseqüentemente, à sua manutenção.

No exemplo acima, montamos um LOOP Cursor FOR com base em uma declaração SELECT que acessa apenas uma tabela e retorna duas colunas.

Agora, imagine se fosse uma SQL com relacionamento entre várias tabelas, com chaves primárias compostas, retornando dezenas de colunas, etc., etc.. Com certeza o código não estaria assim, tão amigável.

Por isso, procure declarar os cursores, SEMPRE, considerando ser uma **boa prática de programação**.

Passando parâmetros para cursores

Podemos passar parâmetros para um cursor previamente declarado. Isto significa que podemos abrir e fechar um cursor dentro de um bloco PL/SQL, retornando diferentes linhas no cursor em cada ocasião. Para cada execução, o cursor anterior é fechado e reaberto com um novo conjunto de parâmetros.

Esse recurso pode ser utilizado em cursores manipulados manualmente ou em cursores automatizados.

Sintaxe:

```
CURSOR nome_cursor [(nome_parametro tipo, ...)] IS  
Comando Select;
```

Onde:

- *nome_cursor* é o nome do cursor declarado pelo usuário.
- *nome_parametro* é o nome do parâmetro.
- *tipo* é o tipo do parâmetro.

Para cada parâmetro declarado no cursor, deverá existir um correspondente quando o comando OPEN for usado. Estes parâmetros, possuem os mesmos tipos de dados que as variáveis escalares, a única diferença é que não fornecemos seus tamanhos.

Exemplo:

```
DECLARE  
-- Declaração de cursores  
CURSOR cs_representante(pMenorMedia NUMBER, pMaiorMedia NUMBER) IS  
    SELECT rep_in_codigo, rep_st_nome  
    FROM representante  
    WHERE rep_vl_mediamensalvendas BETWEEN pMenorMedia AND pMaiorMedia;  
BEGIN  
-- Abre cursor para representantes com média entre 40000 e 80000  
dbms_output.put_line('Representantes com média entre 40000 e 80000');  
FOR rRep IN cs_Representante(40000,80000) LOOP  
    /* Imprime na tela os vendedores cuja média de vendas mensal  
       Está no intervalo de 40000 e 80000  
    */  
    dbms_output.put_line(rRep.rep_in_codigo||' - '||rRep.rep_st_nome);  
END LOOP;  
  
-- Abre cursor para representantes com média entre 80001 e 100000  
dbms_output.put_line('Representantes com média entre 80001 e 100000');  
FOR rRep IN cs_Representante(80001,100000) LOOP  
    /* Imprime na tela os vendedores cuja média de vendas mensal  
       Está no intervalo de 80001 e 100000  
    */  
    dbms_output.put_line(rRep.rep_in_codigo||' - '||rRep.rep_st_nome);  
END LOOP;  
END;  
/
```

Cursores implícitos

Como já mencionado antes, o Oracle cria e abre um cursor para cada declaração SQL que não faz parte de um cursor declarado explicitamente. O cursor implícito mais recente pode ser chamado de cursor SQL. Você **não** pode usar os comandos `OPEN`, `CLOSE` e `FETCH` com um cursor implícito. Entretanto, você pode usar os atributos de cursor para acessar as informações sobre a declaração SQL executada mais recentemente por meio do cursor SQL.

Atributos do cursor implícito

Assim como os cursores explícitos, os cursores implícitos também usam os atributos. Os atributos do cursor implícito são os mesmo do cursor explícito, ou seja:

Tabela 6: Atributos de cursor implícito

Exceções	Descrição
<code>%rowcount</code>	Mostra o número de linhas do cursor
<code>%found</code>	Retorna TRUE se o mais recente <code>FETCH</code> retornar uma linha.
<code>%notfound</code>	Retorna TRUE se o mais recente <code>FETCH</code> não retornar uma linha.
<code>%isopen</code>	Retorna TRUE se o cursor estiver aberto.

Como os cursores implícitos não têm nome, você deve substituir o nome do cursor pela palavra SQL junto ao atributo.

O cursor implícito contém as informações sobre o processamento da última declaração SQL (`INSERT`, `UPDATE`, `DELETE` e `SELECT INTO`) que não foi associada a um cursor explícito.

Você pode usar os atributos do cursor implícito nas declarações PL/SQL, não nas declarações SQL.

Quando estudamos as exceções, vimos o uso de um atributo de cursor implícito em um de nossos exemplos, mas vamos lembrá-lo agora que entendemos o seu funcionamento por completo:

```
DECLARE
  -- Declarar exceção
  eRegistroInexistente exception;
BEGIN
  -- Excluir cliente
  DELETE cliente c
  WHERE c.cli_in_codigo = 1000;

  -- Se cliente não existe, gerar exceção
  IF SQL%NOTFOUND THEN
    raise eRegistroInexistente;
  END IF;

  -- Validar exclusão
  COMMIT;
EXCEPTION
  -- Se registro não existe, informa usuário
  WHEN eRegistroInexistente THEN

    ROLLBACK;
    dbms_output.put_line('Cliente 1000 não existe!' || chr(10) ||
                        'Nenhum registro foi excluído.'
                        );
END;
```

Nesse exemplo utilizamos o atributo `%NOTFOUND` para verificarmos quantos registros foram afetados

pela comando `DELETE`. Note que o atributo é precedido pela palavra `SQL`, que refere-se a instrução `SQL` mais recente (o `DELETE`)

Variáveis de cursor

Como vimos nos tópicos anteriores, o cursor da PL/SQL é uma área nomeada do banco de dados. Uma variável de cursor, por definição, é uma referência àquela área nomeada. Uma variável de cursor é como um ponteiro de uma linguagem de programação como C. As variáveis de cursor indicam a área de trabalho de uma consulta, na qual o conjunto de resultados da consulta é armazenado. Uma variável de cursor também é dinâmica por natureza porque ela não está vinculada a uma consulta específica. A Oracle conserva essa área de trabalho enquanto um ponteiro de cursor está apontando para ela. Você pode usar uma variável de cursor para qualquer consulta de tipo compatível.

Um dos recursos mais significativos da variável de cursor é que a Oracle permite passar uma variável de cursor como um argumento para um procedimento ou uma chamada de função.

Para criar uma variável de cursor, você primeiro deve criar um tipo de cursor referenciado e depois declara uma variável de cursor daquele tipo.

A sintaxe para definir um tipo cursor é a seguinte:

```
TYPE cursor_type_name IS REF CURSOR RETURN return_type;
```

Nessa sintaxe, `REF` quer dizer referência, `cursor_type_name` é o nome do tipo do cursor e `return_type` é a especificação de dados do tipo de cursor de retorno. A cláusula `RETURN` é opcional.

As variáveis de cursores podem ser muito úteis em situação em que o ambiente aplicativo executa um procedimento PL/SQL (Como procedure ou function) e passa como argumento um cursor e/ou recebe um argumento do tipo cursor.

Vejamos um exemplo simples de como podemos utilizar uma variável do tipo cursor:

```
DECLARE
-- Declaração de tipos
TYPE TCursor IS REF CURSOR;

-- Declaração de variáveis
vCursor TCursor;

-- Sub-rotinas
PROCEDURE GerarXML(pCursor TCursor) IS
BEGIN
    NULL;
    /* implementação para gerar o XML a partir do cursor*/
END;
BEGIN
-- Abre o cursor atribuindo para a variável criada
OPEN vCursor FOR
    SELECT rep_in_codigo, rep_st_nome
    FROM representante;

-- Executa a rotina que recebe o cursor e converte seu conteúdo para XML
GerarXML(vCursor);

-- Fecha o cursor
CLOSE vCursor;
END;
/
```

IMPORTANTE: Nas próximas lições, estudaremos os procedimentos armazenados, ou seja, *procedures* e *functions*, mas como você pode observar no exemplo acima, podemos criar funções e *procedures* dentro de qualquer bloco PL/SQL. São as chamadas *subrotinas*.

Blocos anônimos, procedimentos e funções

Em PL/SQL temos três tipos de blocos: Blocos Anônimos, procedimentos (*Procedures*) e funções (*Functions*).

Um bloco anônimo, como já vimos (e como o próprio nome sugere), não é nomeado e esta é a única diferença entre ele e os procedimentos e funções.

Um procedimento é um conjunto de declarações SQL e PL/SQL (como um bloco anônimo) agrupadas logicamente, que executam uma tarefa específica. Ele é um programa em miniatura.

Abaixo temos uma figura que mostra a diferença de nomeação entre bloco anônimo, *procedure* e *function*:

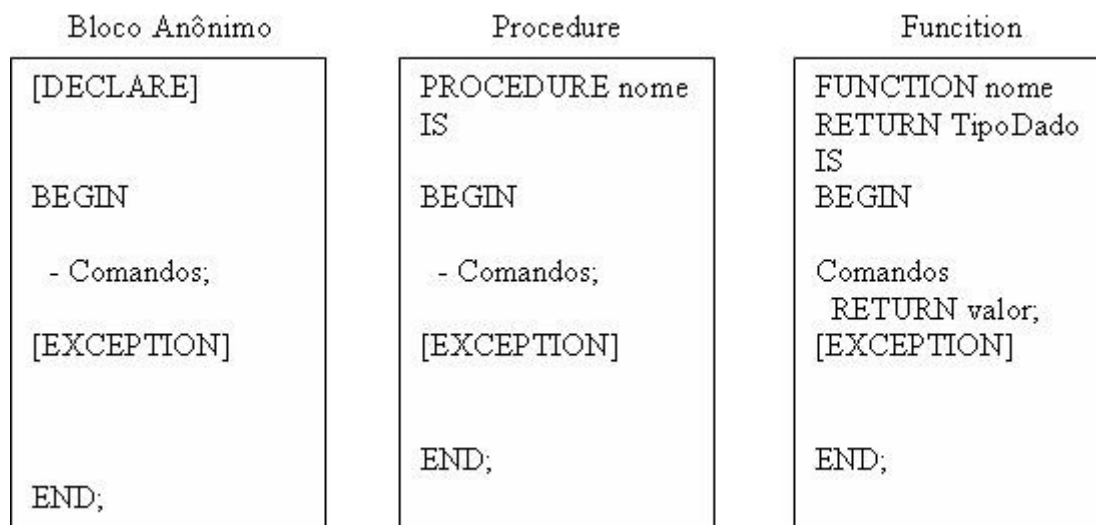


Figura 21: Bloco anônimo, procedure e function

Procedimentos e funções podem ser implementadas de três formas:

1. Como subrotina de um bloco anônimo;
2. Como procedimento independente e armazenado no banco de dados;
3. Como uma rotina de um pacote armazenado no banco de dados.

Nesta lição focaremos as subrotinas em blocos anônimos.

A principal vantagem de usar subrotinas é que você pode modularizar seu programa, centralizando códigos que seriam repetidos. Desta forma, caso tal código precisasse de manutenção, estaria implementado em apenas um ponto do programa.

Abaixo segue um exemplo simples, onde uma *procedure* e uma função são subrotinas de bloco anônimo.

```
DECLARE
-- Declaração de cursores
CURSOR cs_cliente is
SELECT c.*
FROM cliente c;
```

```

-- declaração de variáveis
rCli cs_cliente%ROWTYPE;

/*****
/*                               */
/*                               */
*****/

-- função definir se o cliente compra o seu potencial máximo
FUNCTION ClienteCompraPotencial (pMediaCompraMensal NUMBER,
                                pMaiorCompra          NUMBER)
RETURN BOOLEAN IS
    -- Declaração de variáveis
    vPercentualPotencial NUMBER;
    vResult              BOOLEAN;
BEGIN
    -- Definir quanto a média mensal de compras representa da maior compra
    vPercentualPotencial := (pMediaCompraMensal / pMaiorCompra) * 100;

    -- Resultado só é verdadeiro se percentual for igual ou maior a 80
    vResult := vPercentualPotencial >= 80;

    RETURN(vResult);
END;

-- procedimento para exibir mensagem na tela
PROCEDURE Exibir(pMensagem VARCHAR2) IS
BEGIN
    dbms_output.put_line(pMensagem);
END;
BEGIN
    -- Inicia o loop no conjunto ativo do cursor
    FOR rCli IN cs_Cliente LOOP
        IF ClienteCompraPotencial(rCli.cli_vl_mediacomprasmensal,
                                rCli.cli_vl_maiorcompra
                                ) THEN
            -- Imprime dados extraídos na tela
            Exibir(rCli.cli_in_codigo||' - '||rCli.cli_st_nome);
        END IF;
    END LOOP;
END;
/

```

No exemplo acima utilizamos a função ClienteCompraPotencial para verificar se o cliente está comprando aquilo que tem potencial (em valor) e o procedimento Exibir para imprimir uma mensagem na tela.

Imagine que nosso bloco anônimo fosse muito maior e que, em vários pontos precisasse verificar se o Cliente compra o potencial que tem! Ao invés de reescrever a regra para esse cálculo em todos esses pontos, basta chamar a função que passaria a ser o único ponto de manutenção dessa regra.

Outra coisa que deve ser observada é que os procedimentos e funções recebem parâmetros, como vimos nos cursores .

IMPORTANTE: Ao invocar um procedimento que possui parâmetros de entrada, procure informar o

valor de cada parâmetro na ordem definida, pois não há como o Oracle “adivinhar” os valores se estiverem “embaralhados”. É possível informar parâmetros em ordem diferente da definida, porém é mais trabalhoso, pois você precisará informar além do valor, o nome do parâmetro ao que se refere, como abaixo:

```
DECLARE
```

```
...
```

```
BEGIN
```

```
...
```

```
    ClienteCompraPotencial(pMaiorCompra =>100000, pMediaCompraMensal => 80000)
```

```
END;
```

```
/
```

Exercícios propostos

1. Crie um bloco anônimo com um cursor automatizado que carregue todos os clientes de um determinado ramo de atividade, mediante o uso de parâmetros. Abra um loop para cada ramo de atividade possível (HIPERMERCADO, SUPERMERCADO, MERCADO, MERCEARIA). Antes de iniciar cada loop, imprima na tela o ramo de atividade. Durante o loop, se a data da maior compra do cliente for do mês corrente, imprima na tela a seguinte mensagem

'A maior compra do cliente x foi realizada no mês corrente'.

Onde x é código do cliente.
2. Crie um bloco anônimo com um cursor não automatizado que carregue todos os produtos e imprima na tela:
 - *'Novo record de venda do produto x é n',* se o valor da última venda for maior ou igual ao valor da maior venda, onde x é o código do produto e n é o valor da maior venda;
 - *'O record de venda do produto x não foi alcançado',* se o valor da última venda não for maior ou igual ao valor da maior venda;
3. Crie um bloco anônimo com um cursor que carregue todos os representantes e imprima na tela:
 - *'A meta mensal de vendas do representante x está abaixo de sua média mensal',* se a meta mensal de vendas do representante for menor do que sua média mensal;
 - *'O representante x tem potencial maior do que sua meta mensal.',* se a meta mensal for maior do que sua média e menor do que sua maior venda;
 - *'O representante x atingiu todo o potencial de vendas!'*, se a meta mensal for maior do que sua média e maior do que sua maior venda, ou seja, se não atender as duas primeiras condições. (utilize `IF...ELSIF...ELSE`);

onde x é o código do representante.
4. Construa um bloco anônimo que:
 - a) Inicialize uma variável do tipo DATE com a data do primeiro dia do mês atual;
 - b) Execute um loop que simule um REPEAT...UNTIL;
 - c) Incremente a variável em um dia, a cada ciclo do loop e imprima na tela o valor obtido;
 - d) Abandone o loop quando a data não avançar o mês atual.
5. Reproduza o exercício anterior declarando no bloco anônimo, uma função para incrementar a data a ser impressa na tela.

03 – Stored Procedures

1. **Procedimentos armazenados;**
2. **Pacotes;**

Procedimentos armazenados (*Stored Procedure*)

Um procedimento armazenado é um procedimento que foi compilado e armazenado dentro do banco de dados. Depois de armazenado o procedimento é um objeto de esquema, ou seja, um objeto específico do banco de dados.

Por que usar os procedimentos?

Os procedimentos são criados para solucionar um problema ou tarefa específicos. Os procedimentos PL/SQL oferecem as seguintes vantagens:

- Na PL/SQL você pode personalizar um procedimento segundo seus requisitos específicos;
- Os procedimentos são modulares, o que significa que eles deixam você dividir um programa em unidades gerenciáveis e bem definidas;
- Como os procedimentos são armazenados em um banco de dados, eles podem ser reutilizados. Após um procedimento ter sido validado, ele pode ser usado repetidamente sem ser recompilado ou distribuído pela rede;
- Os procedimentos aumentam a segurança do banco de dados. Você pode restringir o acesso ao banco de dados permitindo que os usuários acessem os dados apenas por meio dos procedimentos armazenados;
- Os procedimentos aproveitam os recursos de memória compartilhados.

Uma das coisas mais úteis que você pode fazer com o seu conhecimento da PL/SQL talvez seja usá-la para escrever *stored procedures*. A encapsulação do código que você escreveu anteriormente em uma função armazenada permite que você a compile uma vez e armazene no banco de dados para uso futuro. Da próxima vez que quiser executar aquele bloco PL/SQL (toda *stored procedure* é um bloco PL/SQL), você só precisa invocar a função.

As duas maiores diferenças entre um bloco anônimo e uma *stored procedure* são:

1. Blocos anônimos não são armazenados no banco de dados e *stored procedures* são. Quando o desenvolvedor cria um bloco anônimo, é responsabilidade dele guardar o seu código, ou seja, se o bloco for desenvolvido, executado e não for salvo em algum arquivo de sistema operacional (ou objeto de banco de dados criado para isso), seu código será perdido. Uma *stored procedure*, para ser usada, deve ser compilada e quando é compilada o Oracle armazena seu código no dicionário de dados, permitindo assim a sua recuperação para posterior manutenção;
2. *Stored Procedure* possui um cabeçalho para nomeá-lo, declarar variáveis de input e output e tipo de retorno; blocos anônimos não.

A estrutura de um procedimento armazenada é exatamente igual à procedimentos implementados dentro de blocos PL/SQL e o que define que ele será armazenada é o uso do comando `CREATE OR REPLACE` antes de seu nome.

Procedimentos versus funções

Os procedimentos e as funções são sub programas PL/SQL que são armazenados no banco de dados. A diferença significativa entre os dois são apenas os tipos de saída dos dois objetos gerados. Uma função retorna um valor simples, enquanto que um procedimento é usado para executar processamento complicado quando você quer ter de volta uma quantidade substancial de informações.

A seguir veremos um pouco mais sobre procedimentos e funções.

Procedimentos

Uma *procedure* nada mais é do que um bloco PL/SQL nomeado sem retorno definido (somente funções possuem um retorno definido). A grande vantagem sobre um bloco PL/SQL anônimo é que pode ser compilado e armazenado no banco de dados como um objeto de schema. Graças a essa característica, as *procedures* são de fácil manutenção, o código é reutilizável e permitem que trabalhem com módulos de programa.

A sintaxe básica de uma *procedure* é:

```
CREATE [OR REPLACE] PROCEDURE [schema.]nome_da_procedure
[(parâmetro1 [modo1] tipodedado1,
  parâmetro2 [modo2] tipodedado2,
  ...)]
IS|AS
[Bloco PL/SQL]
```

Onde:

- *replace* indica que caso a *procedure* exista ela será eliminada e substituída pela nova versão criada pelo comando;
- *nome_da_procedure* indica o nome da *procedure*;
- *parâmetro* indica o nome da variável PL/SQL que é passada na chamada da *procedure* ou o nome da variável que retornará os valores da *procedure* ou ambos. O que irá conter em *parâmetro* depende de *modo*;
- *modo* Indica que o *parâmetro* é de entrada (IN), saída (OUT) ou ambos (IN OUT). É importante notar que *IN* é o modo *default*, ou seja, se não dissermos nada o modo do nosso *parâmetro* será, automaticamente, *IN*;
- *tipodedadoN* indica o tipo de dado do *parâmetro*. Pode ser qualquer tipo de dado do SQL ou do PL/SQL. Pode usar referências como *%TYPE*, *%ROWTYPE* ou qualquer tipo de dado escalar ou composto. Também é possível definir um valor *default*. Atenção: não é possível fazer qualquer restrição ao tamanho do tipo de dado neste ponto.
- *is/as*: Por convenção usamos *IS* na criação de *procedures* armazenadas e *AS* quando estivermos criando pacotes.
- *Bloco PL/SQL* inicia com uma cláusula *BEGIN* e termina com *END* ou *END nome_da_procedure* onde as ações serão executadas.

Abaixo, temos um exemplo de uma *procedure* armazenada que recupera a maior vendas de um produto dentro de um período e atualiza essa informação no cadastro do produto, se for maior do que o valor já existente :

```
CREATE OR REPLACE PROCEDURE pr_AtualizaMaiorVendaProduto (pProduto
produto.pro_in_codigo%TYPE,
                                                    pDataInicial DATE,
                                                    pDataFinal DATE
) IS
vValorMaiorVenda_old NUMBER := 0;
vValorMaiorVenda_new NUMBER := 0;
BEGIN
-- Recupera o valor da maior venda registrada no produto
BEGIN
SELECT pro_vl_maiorvenda
```

```

    INTO vValorMaiorVenda_old
    FROM produto
    WHERE pro_in_codigo = pProduto;
EXCEPTION
    WHEN no_data_found THEN
        raise_application_error(-20100, 'Produto não cadastrado!');
    WHEN OTHERS THEN
        raise_application_error(-20100, 'Erro ao tentar recuperar dados do
produto!');
END;

-- Recupera o valor da maior venda do produto no período
SELECT MAX(i.infv_vl_total)
INTO vValorMaiorVenda_new
FROM nota_fiscal_venda n,
     item_nota_fiscal_venda i
WHERE n.nfv_dt_emissao BETWEEN pDataInicial AND pDataFinal
AND n.nfv_in_numero = i.nfv_in_numero
AND i.pro_in_codigo = pProduto;

-- Se a maior venda do período for maior do que a já cadastrada, atualiza
IF vValorMaiorVenda_new > vValorMaiorVenda_old THEN
    UPDATE produto
    SET pro_vl_maiorvenda = vValorMaiorVenda_new
    WHERE pro_in_codigo = pProduto;
END IF;
END;
/

```

Esse procedimento armazenado pode ser executado diretamente do prompt de uma ferramenta de comandos, como abaixo:

```
SQL> execute pr_AtualizaMaiorVendaProduto(20,SYSDATE,last_day(SYSDATE));
```

Ou a partir de um bloco PL/SQL, como abaixo:

```

BEGIN
    pr_AtualizaMaiorVendaProduto(20,SYSDATE,last_day(SYSDATE));
END;
/

```

Funções

Como vimos antes, as funções são semelhantes aos procedimentos da PL/SQL, exceto pelas seguintes diferenças:

- As funções retornam um valor;
- As funções são usadas como parte de uma expressão.

A sintaxe básica de uma função é:

```

CREATE [OR REPLACE] FUNCTION [schema.]nome_da_funcao
[(parâmetro1 [modo1] tipodedado1,

```

```

        parâmetro2 [modo2] tipodedado2,
        ...)]
RETURN TipoRetorno IS|AS
[Bloco PL/SQL]

```

Onde:

- *replace* indica que caso a função exista ela será eliminada e substituída pela nova versão criada pelo comando;
- *nome_da_funcao* indica o nome da função;
- *parâmetro* indica o nome da variável PL/SQL que é passada na chamada da função ou o nome da variável que retornará os valores da função ou ambos. O que irá conter em parâmetro depende de *modo*;
- *modo* Indica que o parâmetro é de entrada (IN), saída (OUT) ou ambos (IN OUT). É importante notar que IN é o modo *default*, ou seja, se não dissermos nada o modo do nosso parâmetro será, automaticamente, IN;
- *tipodedadoN* indica o tipo de dado do parâmetro. Pode ser qualquer tipo de dado do SQL ou do PL/SQL. Pode usar referências como %TYPE, %ROWTYPE ou qualquer tipo de dado escalar ou composto. Também é possível definir um valor *default*. Atenção: não é possível fazer qualquer restrição ao tamanho do tipo de dado neste ponto.
- *is/as*: Por convenção usamos IS na criação de funções armazenadas e AS quando estivermos criando pacotes.
- *Bloco PL/SQL* inicia com uma cláusula BEGIN e termina com END ou END *nome_da_funcao* onde as ações serão executadas.

Vamos escrever uma função armazenada que recebe o código do cliente e retorna a quantidade de notas faturadas para ele, dentro de um período:

```

CREATE OR REPLACE FUNCTION fn_QtdeNFV(pCodigoCliente NUMBER,
                                         pDataInicial DATE,
                                         pDataFinal DATE
                                         )
RETURN INTEGER IS
    -- Declaração de variáveis
    vResult INTEGER :=0;
BEGIN
    -- Recupera quantidade de notas do cliente dentro do período
    SELECT COUNT(*)
    INTO vResult
    FROM nota_fiscal_venda nfv
    WHERE nfv.cli_in_codigo = pCodigoCliente
    AND nfv.nfv_dt_emissao BETWEEN pDataInicial AND pDataFinal;

    RETURN(vResult);
END;
/

```

Como a função acima é armazenada, podemos usá-la das seguintes formas:

1. Atribuindo seu retorno diretamente para uma variável:

```
DECLARE
```

```
vQtdeNFV NUMBER;  
BEGIN  
vQtdeNFV := fn_QtdeNFV(10, ADD_MONTHS(SYSDATE,-1), SYSDATE);  
END;  
/
```

Nesse exemplo, atribuímos para a variável *vQtdeNFV*, o retorno da função *fQtdeNFV*, tendo como período os últimos trinta dias (*SYSDATE* é igual a data atual e *ADD_MONTHS* adiciona o número de meses informados)

2. Recuperando seu valor através de uma instrução SQL:

```
SELECT fn_QtdeNFV(10, ADD_MONTHS(SYSDATE,-1), SYSDATE) QtdeNFV  
FROM dual;
```

A tabela *DUAL* é uma tabela especial do Oracle que sempre existe, sempre tem exatamente uma linha e sempre tem exatamente uma coluna. Ela é a tabela perfeita para ser usada quando se experimentam as funções. A seleção da função na tabela *DUAL* faz com que o resultado da função seja exibido.

Se a declaração *SELECT* acima estivesse dentro de um bloco PL/SQL, o retorno da função *fQtdeNFV* poderia ser atribuído à uma variável através da cláusula *INTO*.

IMPORTANTE: Evite utilizar comandos DML (*INSERT*, *UPDATE* e *DELETE*) em funções, pois se elas forem invocadas de uma declaração *SELECT* você receberá o seguinte erro:

```
ORA-14551: não é possível executar uma operação DML dentro de uma consulta
```

Manutenção de procedimentos armazenados

Para recompilar explicitamente um procedimento armazenado, use o comando *ALTER PROCEDURE*, como no exemplo abaixo.

```
ALTER PROCEDURE pr_AtualizaMaiorVendaProduto COMPILE;
```

(ou *ALTER FUNCTION* para funções armazenadas)

O comando acima, como pode ser observado, não altera a declaração ou definição do procedimento. Para isso, você deve usar o comando *CREATE OR REPLACE [PROCEDURE | FUNCTION]*, junto com todo o código do procedimento (da mesma forma que é feito quando se cria o procedimento).

Para excluir um procedimento armazenado, você deve utilizar o comando *DROP [PROCEDURE | FUNCTION]*. A sintaxe do comando é:

```
DROP PROCEDURE | FUNCTION nome_do_procedimento;
```

Onde:

- *DROP* é o comando de exclusão;
- *PROCEDURE* ou *FUNCTION* defini o tipo de procedimento a ser excluído;
- *nome_do_procedimento* é o nome do procedimento armazenado.

Usando os parâmetros

Os procedimentos usam os *parâmetros* para passar as informações. Quando um parâmetro está sendo passado para um procedimento, ele é conhecido como um *parâmetro real*. Os parâmetros declarados

como internos a um procedimento são conhecidos como parâmetros *internos* ou *formais*.

O parâmetro real e seu parâmetro formal correspondente devem pertencer a *datatypes* compatíveis. Por exemplo, a PL/SQL não pode converter um parâmetro real com um *datatype* `DATE` para um parâmetro formal com um *datatype* `LONG`. Nesse caso, o Oracle retornaria uma mensagem de erro. Essa questão de compatibilidade também se aplica aos valores de retorno.

Definições de parâmetro

Quando você invoca um procedimento, deve passar um valor para cada um dos parâmetros do procedimento. Se você passar valores para o parâmetro, eles são posicionais e devem aparecer na mesma ordem em que aparecem na declaração do procedimento. Se você passar nomes de argumentos, eles podem aparecer em qualquer ordem. Você pode ter uma combinação entre valores e nomes nos valores do argumento. Quando esse for o caso, os valores identificados na ordem devem preceder os nomes de argumento.

Dependências de procedimentos

Um dos recursos inerentes da PL/SQL é que ela verifica o banco com base nos objetos de dados para ter certeza de que as operações de um procedimento, uma função ou um pacote são possíveis, os quais o usuário tem acesso. Por exemplo, se você tiver um procedimento que exija acesso a diversas tabelas e visões, a PL/SQL verifica durante o tempo de compilação se aquelas tabelas e visões estão presentes e disponíveis para o usuário. Diz-se que o procedimento é *dependente* dessas tabelas e visões.

IMPORTANTE: A PL/SQL recompila automaticamente todos os objetos dependentes quando você recompila explicitamente o objeto pai. Essa recompilação automática dos objetos dependentes acontece quando o objeto dependente é chamado. Assim sendo, não é recomendado recompilar um módulo pai de um sistema de produção, pois isso faz com que todos os objetos dependentes sejam recompilados e, conseqüentemente, pode causar problemas de desempenho para o seu sistema de produção.

Segurança da invocação de procedimento

Ao executar um procedimento armazenado, o Oracle verifica se você tem os privilégios necessários para tanto. Se você não tiver as permissões de execução apropriadas, ocorre um erro.

Uma vez tendo direito de execução de um procedimento, ao executá-lo, o Oracle não verifica se você tem direitos sobre os objetos referenciados pelo procedimento. O Oracle verifica se o proprietário (OWNER) do procedimento tem permissões e isso durante a compilação do procedimento.

Desta forma, a PL/SQL disponibiliza através dos procedimentos armazenados, um ótimo recurso para controle de permissão. Você pode ter uma aplicação onde, apenas um usuário tem direito de incluir, alterar e excluir registros. Dentro do *schema* desse usuário você desenvolve rotinas que executa essas operações e então, transmite o direito de executar os procedimentos aos usuário que desejar. Esses usuário só conseguirão alterar os registros através da rotina e você poderá aplicar regras internas no procedimento, como consistências e geração de logs.

Pacotes

Um *pacote* é uma coleção encapsulada de objetos de esquema relacionados.

Esses objetos podem incluir procedimentos, funções, variáveis, constantes, cursores, tipos e exceções. Um pacote é compilado e depois armazenado no dicionário de dados do banco de dados como um objeto de esquema.

Um uso comum dos pacotes é para reunir todos os procedimentos, funções e outros objetos relacionados que executam tarefas semelhantes. Por exemplo, você pode agrupar todos os objetos utilizados para gerar um resumo de vendas em um único pacote.

Os pacotes contêm *subprogramas armazenados*, ou programas isolados, os quais são chamados subprogramas do pacote. Esses subprogramas podem ser chamados de outro programa armazenado, triggers, programas ou de qualquer um dos programas interativos da Oracle, tais como O SQL*Plus. Ao contrário dos subprogramas armazenados, o pacote em si não pode ser chamado ou aninhado, e os parâmetros não podem ser passados para ele.

Vantagens do uso de pacotes

Os pacotes oferecem as seguintes vantagens:

- Eles permitem organizar o desenvolvimento do seu aplicativo de forma mais eficiente com os módulos. Cada pacote é facilmente entendido e as interfaces entre os pacotes são simples, claras e bem definidas;
- Eles permitem conceder os privilégios de forma eficiente;
- As variáveis *public* e os cursores de um pacote persistem durante a sessão, ou seja, todos os cursores e procedimentos que são executados nesse ambiente podem compartilhar deles;
- Eles permitem executar a sobrecarga nos procedimentos e funções;
- Eles melhoram o desempenho carregando vários objetos ao mesmo tempo. Assim sendo, as chamadas subseqüentes a subprogramas relacionados no pacote não requerem entrada/saída;
- Eles promovem a reutilização do código por meio do uso das bibliotecas que contêm os procedimentos armazenados e as funções, eliminando assim a codificação redundante.

Estrutura de um pacote

Geralmente um pacote tem dois componentes:

1. *Especificação*: Declara os tipos, as variáveis, as constantes, as exceções, os cursores e os subprogramas que estão disponíveis para uso.
2. *Corpo*: Define totalmente as funções e os procedimentos e, assim, implementa a especificação.

A especificação do pacote

A especificação do pacote contém declarações *public*. Isso significa que os objetos declarados no pacote são acessíveis de qualquer parte do pacote e são disponíveis à programas externos. Assim sendo, todas as informações que o aplicativo precisa para executar um programa armazenado estão contidas na especificação do pacote.

A sintaxe do comando de especificação é o seguinte:

```
CREATE [OR REPLACE] PACKAGE package_name  
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}  
[package body object declaration]
```



```
END [package_name];
```

Nessa sintaxe, as palavras-chave e os parâmetros são os seguintes:

- *package_name* é o nome do pacote que o criador define. É importante dar-lhe um nome que seja significativo e represente o conteúdo do pacote;
- *AUTHID* representa o tipo de direitos que você quer invocar quando o pacote for executado. Os direitos podem ser aquele de *CURRENT_USER* ou aquele do pacote *DEFINER* (Criador);
- *package body object declaration* é o lugar no qual você lista os objetos que serão criados dentro do pacote.

Abaixo temos um exemplo de uma declaração de pacote:

```
CREATE OR REPLACE PACKAGE pck_cliente AS

-- Author   : JOSINEI
-- Created  : 27/6/2007 17:45:36
-- Purpose  : -- atualiza dados cliente

/*-----*/
/*          Public function and procedure declarations          */
/*-----*/

-- Procedimento para atualizar valor médio de compras do cliente
PROCEDURE AtualizarMediaComprasMensal (pCodigoCliente NUMBER) ;

-- Procedimento para atualizar dados de maior compra do cliente
PROCEDURE AtualizarMaiorCompra (pCodigoCliente NUMBER) ;

-- Função que retorna o endereço do cliente
FUNCTION EnderecoCliente (pCodigoCliente NUMBER) RETURN VARCHAR2;

END pck_cliente;
/
```

O corpo do pacote

O corpo de um pacote contém a definição dos objetos *public* que você declara na especificação. O corpo também contém as outras declarações de que são privadas do pacote e que estão acessíveis apenas para os objetos do corpo do pacote. Os objetos privados declarados no corpo do pacote não são acessíveis para outros objetos fora do pacote. Ao contrário da especificação do pacote, a parte de declaração do corpo do pacote pode conter corpos de subrotinas.

IMPORTANTE: Se a especificação declarar apenas constantes e variáveis, o corpo do pacote não é necessário.

A sintaxe para criar o corpo (*body*) do pacote é o seguinte:

```
CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
[package body object declaration]

END [package_name];
```

Nessa sintaxe, as palavras-chave e os parâmetros são os seguintes:

- *package_name* é o nome do pacote definido pelo criador. É importante dar-lhe um nome significativo e que represente o conteúdo do corpo do pacote. Esse nome deve ser igual ao nome

dado a package durante a criação da especificação;

➤ *package body object declaration* é o lugar no qual você lista os objetos que serão criados dentro do pacote.

Abaixo segue um exemplo de corpo de pacote que implementa as rotinas públicas que declaramos na especificação do pacote pck_cliente:

```
CREATE OR REPLACE PACKAGE BODY pck_cliente AS
-- Declaração de variáveis privadas
rCli cliente%ROWTYPE;

-- Procedimento para atualizar valor médio de compras do cliente
PROCEDURE AtualizarMediaComprasMensal(pCodigoCliente NUMBER) IS
BEGIN
    -- Recupera a média a partir da soma mensal
    SELECT AVG(rmv_vl_total)
    INTO rCli.cli_vl_mediacomprasmensal
    FROM (SELECT rmv_in_mes,
                 rmv_in_ano,
                 sum(rmv_vl_total) rmv_vl_total
          FROM resumo_mensal_venda
          WHERE cli_in_codigo = pCodigoCliente
          GROUP BY rmv_in_mes,
                 rmv_in_ano
         );

    -- Atualiza Média Compras Mensal
    UPDATE cliente
    SET cli_vl_mediacomprasmensal = rCli.cli_vl_mediacomprasmensal
    WHERE cli_in_codigo = pCodigoCliente;

    -- Finaliza transação confirmando alterações
    COMMIT;
END;

-- Procedimento para atualizar dados de maior compra do cliente
PROCEDURE AtualizarMaiorCompra(pCodigoCliente NUMBER) IS
BEGIN
    -- Recupera a maior compra do cliente
    SELECT MAX(nfv_vl_total)
    INTO rCli.cli_vl_maiorcompra
    FROM nota_fiscal_venda
    WHERE cli_in_codigo = pCodigoCliente;

    -- Atualiza Maior Compra do cliente
    UPDATE cliente
    SET cli_vl_maiorcompra = rCli.cli_vl_maiorcompra
    WHERE cli_in_codigo = pCodigoCliente;

    -- Finaliza transação confirmando alterações
    COMMIT;
END;

-- Função que retorna o endereço do cliente
FUNCTION EnderecoCliente(pCodigoCliente NUMBER) RETURN VARCHAR2 IS
vEnderecoCliente VARCHAR2(256);
```

```
BEGIN
  -- Recupera o endereço concatenado do cliente
  SELECT cli_st_endereco || ' - ' || cli_st_cidade || ' - ' || cli_st_uf
  INTO vEnderecoCliente
  FROM cliente
  WHERE cli_in_codigo = pCodigoCliente;

  -- Retorna o endereço concatenado
  RETURN (vEnderecoCliente);
END;
END pck_cliente;
/
```

Observando o corpo desse pacote, vemos que todos procedimentos recebem o código do cliente como parâmetro (e todos eles são públicos se observarmos o exemplo de especificação). Poderíamos ter definido na especificação, uma variável para receber esse código e então, em todos os procedimentos utilizaríamos o seu valor, que seria inicializado antes de executar os procedimentos, como veremos a seguir.

Utilizando os subprogramas e variáveis de um pacote

Quando um pacote é invocado, o Oracle executa três etapas para executá-lo:

1. Verifica o acesso de usuário – Confirma se o usuário tem a concessão do privilégio de sistema EXECUTE;
2. Verifica a validade do pacote – Verifica no dicionário de dados e determina se o sub-programa é válido. Se o objeto é inválido, ele é automaticamente recompilado antes de ser executado;
3. Executa – O sub-programa invocado é executado.

Para referenciar os sub-programas e objetos do pacote, você deve usar a notação de ponto, como abaixo:

- ➔ *package_name.type_name* para referenciar um tipo definido na especificação de um pacote;
- ➔ *package_name.variable_name* para referenciar uma variável declarada na especificação de um pacote;
- ➔ *package_name.subprogram_name* para invocar um procedimento declarado na especificação do pacote e implementado no corpo do pacote.

Onde:

- *package_name* é o nome do pacote;
- *type_name* é o nome de um tipo;
- *variable_name* é o nome de uma variável;
- *subprogram_name* é o nome de um procedimento interno do pacote.

Poderíamos invocar um procedimento do pacote que utilizamos para exemplificar a especificação e o corpo de um pacote, da seguinte maneira:

```
DECLARE
```

```
vEndereco VARCHAR2(256);  
BEGIN  
  pck_cliente.AtualizarMediaComprasMensal(10);  
  
  vEndereco := pck_cliente.EnderecoCliente(10);  
END;  
/
```

Manutenção dos pacotes

Estados de um pacote

O pacote é considerado inválido se o seu código-fonte ou qualquer objeto que ele referencia foi excluído, alterado ou substituído desde que a especificação do pacote foi recompilada pela última vez. Quando um pacote se torna inválido, o Oracle também torna inválido todo objeto que referencia o pacote.

Recompilando pacotes

Para recompilar um pacote você deve usar o comando `ALTER PACKAGE` com a palavra-chave `compile`. Essa recompilação explícita elimina a necessidade de qualquer recompilação implícita no *runtime* e evita todos os erros associados de compilação no após modificações no pacote.

Quando recompila um pacote, você também recompila todos os objetos definidos dentro do pacote. A recompilação não altera a definição do pacote ou de nenhum de seus objetos.

Os seguintes exemplos recompilam apenas o corpo de um pacote. A segunda declaração recompila todo o pacote, incluindo o corpo e a especificação:

```
ALTER PACKAGE pck_cliente COMPILE BODY;  
  
ALTER PACKAGE pck_cliente COMPILE PACKAGE;
```

Você pode compilar todos os pacotes do schema usando o utilitário `dbms_utility` da Oracle.

```
SQL> execute dbms_utility.compile_schema(SCHEMA => 'JSILVA');
```

Dependência de pacote

Durante o processo de recompilação de um pacote, o Oracle invalida todos os objetos dependentes. Esses objetos incluem procedimentos armazenados ou pacotes que chamam ou referenciam objetos declarados na especificação recompilada. Quando o programa de outro usuário chama ou referencia um objeto dependente antes de ser recompilado, o Oracle o recompila automaticamente no *runtime*.

Durante a recompilação do pacote, o Oracle determina se os objetos dos quais o corpo do pacote depende são válidos. Se algum desses objetos for inválido, o Oracle os recompila antes de recompilar o corpo do pacote. Se a recompilação for bem-sucedida, então o corpo do pacote se torna válido. Se forem detectados erros as mensagens de erro apropriadas são geradas e o corpo do pacote permanece inválido.

Triggers

O que é um *Trigger*?

Um *trigger* é um bloco PL/SQL que é associado a um evento específico, armazenado em um banco de dados e executado sempre que o evento ocorrer.

No Oracle Database é possível criar *triggers* de tipos diferentes, como por exemplo:

- *Triggers* DML (Data Manipulation Language): *Triggers* tradicionais para tratamento de operações INSERT, UPDATE e DELETE, que o Oracle Database suporta há muitos anos.
- *Instead-of*: Introduzidos na versão 8 do Oracle Database como um modo de possibilitar a atualização de determinados tipos de *views*.
- Data Definition Language (DDL): Disponível a partir do Oracle Database 8i, é muito utilizado para auditoria de alterações nos objetos de um *schema*.
- Banco de Dados: A exemplo das *triggers* DDL, também foi disponibilizada na versão 8i e é utilizada nos eventos de banco de dados, tais como inicialização e fechamento do banco, *logon*, *logoff* e etc..

Embora os *triggers* sejam classificados em tipos diferentes, a estrutura entre esses tipos são semelhantes.

Neste treinamento, estudaremos os *triggers* DML.

Triggers DML

Os *triggers* DML são os *triggers* tradicionais que podem ser definidos em uma tabela e são executados, ou “disparados”, em resposta aos seguintes eventos:

- Inclusão de uma linha em uma tabela
- Atualização de uma linha em uma tabela
- Exclusão de uma linha em uma tabela

Uma definição de *trigger* DML consiste das seguintes partes básicas:

- Evento que dispara o *trigger*
- Tabela no qual o evento ocorrerá
- Condição opcional que controla a hora em que o *trigger* é executado
- Bloco PL/SQL contendo o código a ser executado quando o *trigger* é disparado

Assim como *functions*, *procedures* e *packages*, um *trigger* é armazenado no banco de dados, ou seja, é um objeto do banco e é sempre executado quando o evento para o qual ele é definido ocorre. Não importa se o evento é disparado por alguma digitação em uma declaração SQL usando SQL*Plus, executando um programa *client-server* ou qualquer outra ferramenta pela qual seja possível manipular dados da tabela.

Por isso, uma lógica para manipular o dado incluído, atualizado ou alterado em uma tabela, sempre será executado se estiver no *trigger*.

Entre os usos mais comuns de *triggers* DML, podemos citar a geração de *primary key* (PK) e o registro de *log* de alteração.

Abaixo segue um exemplo muito comum de *trigger* que gera um *log* de alteração:

```
CREATE OR REPLACE TRIGGER trg_log_preco_prod_iu
  AFTER UPDATE ON preco_produto
  FOR EACH ROW
BEGIN
  INSERT INTO log_preco_produto
    (lpp_st_tipopreco
    ,lpp_in_codigo
    ,lpp_vl_unitario_anterior
    ,lpp_vl_unitario_novo
    ,lpp_dt_alteracao
    ,lpp_st_usuario)
  VALUES
    (:OLD.ppr_st_tipopreco
    ,:OLD.pro_in_codigo
    ,:OLD.ppr_vl_unitario
    ,:NEW.ppr_vl_unitario
    ,SYSDATE
    ,USER) ;
END trg_log_preco_prod_iu;
/
```

Tipos de *triggers* DML

Os *triggers* DML podem ser classificados de duas maneiras diferentes: quando são disparados (com relação à declaração SQL) ou se eles disparam ou não para cada linha afetada pela declaração SQL de disparo (nível do disparo). Combinando essas classificações, temos quatro tipos de *triggers* DML.

Quando os *triggers* são disparados?

Existem duas opções de “quando” o *trigger* deve ser disparado: antes e depois.

Os *before triggers* são executados antes do disparo da declaração SQL e os *after triggers* são executados depois do disparo da declaração SQL.

Quais são os níveis do *trigger*?

Existem dois possíveis níveis para um *trigger* DML:

- Nível de linha: É o mais comum e é executado uma vez para cada linha afetada pela SQL que a disparou. Apenas *triggers* em nível de linha têm acesso aos valores de dados dos registros afetados (novos e antigos)
- Nível de declaração: Pouco usada, é executado apenas uma vez diante da execução da SQL.

E os eventos do *trigger* DML?

Podemos definir três eventos para um *trigger* DML:

- INSERT
- UPDATE
- DELETE

Se combinarmos esses eventos aos níveis e o momento de execução (quando), teremos **12 tipos de *triggers* DML**.

IMPORTANTE: Os *triggers* definidos de forma idêntica são executados sem ordem determinada. Se você escrever diversos *triggers* que são disparados antes de uma linha ser atualizada, por exemplo, deve garantir que a integridade do banco de dados não dependa da ordem de execução.

Dicas:

- *Use os triggers no nível de linha antes da atualização para a implantação de regras de negócio complexas e cálculos complexos, pois provavelmente você deseja fazer isso antes que a linha seja inserida;*
 - *Use os triggers no nível de linha após a atualização para logging das alterações;*
 - *Use os triggers no nível de declaração antes da atualização para implementar regras de segurança que não dependam dos valores dos registros afetados;*
 - *Não implemente regras de negócio diretamente nos triggers. Prefira implementar regras em stored procedures e apenas executá-las a partir das triggers;*
 - ***NUNCA** utilize triggers para implementar integridade referencial. Para isso existe as constraints no banco de dados.*
-

Ativando e desativando *triggers*

Os *triggers* podem ser temporariamente desativados sem precisar excluí-los e depois recriá-los. Isso pode ser útil quando você precisar fazer algum processamento especial, como por exemplo, uma carga de dados. O comando ALTER TRIGGER é usado para ativar e desativar *triggers*. Veja o exemplo abaixo:

```
-- Desabilitar trigger
ALTER TRIGGER rep_trg_iu DISABLE;

-- Habilitar trigger
ALTER TRIGGER rep_trg_iu ENABLE;
```

Dica: Executar carga de dados com *triggers* desabilitados pode representar um grande ganho de desempenho, mas cada caso deve ser analisado, pois qualquer regra implementada nos *triggers* desabilitados, serão ignoradas.

Exercícios Propostos

1. Desenvolva uma função armazenada que receba como argumento o número de uma nota fiscal e retorne o valor total dos itens (coluna `INFV_VL_TOTAL` da tabela de itens). Nomeie essa função como `fnValorTotalItensNotaFiscal` e teste utilizando uma declaração SQL simples na tabela `DUAL`.

2. Desenvolva um procedimento armazenado que busque, para cada nota fiscal cadastrada, o valor total de seus itens e valide se o seu valor atual (coluna `NFV_VL_TOTAL`) corresponde ao total dos itens e, caso não corresponda, atualize esse valor. Utilize a função criada no primeiro exercício para recuperar o valor total dos itens. Nomeie esse procedimento com `prAtualizarValorNotaFiscal`.

3. Crie um pacote chamado `pck_NotaFiscalVenda` e implemente nele a função criada no primeiro exercício e o procedimento criado no segundo exercício. Tanto a função quanto o procedimento, devem ser publicados na especificação do pacote. Adicione nesse pacote, um procedimento que valide se o valor total do item está certo e, caso não esteja, corrija. O valor correto do item é obtido multiplicando a quantidade (`INFV_QT_FATURADA`) pelo valor unitário (`INFV_VL_UNITARIO`) e subtraindo o desconto (`INFV_PE_DESCONTO`). Esse novo procedimento não deve ser público e o procedimento `prAtualizarValorNotaFiscal` deve executá-lo antes de atualizar o valor total da nota.

4. Crie uma procedure que receba dois argumentos: um ano e um mês. Esse procedimento deve recuperar as notas fiscais do período informado (ano/mês) e atualizar a tabela de resumo mensal de vendas para cada produto/cliente/representante, sendo que, se o período já existir, deve ser atualizado. Nomeie a procedure como `prAtualizarVendasMensal`.

5. Crie um pacote chamado `pck_cliente` que contenha:

- Um procedimento público que receba como argumento o código do cliente e atualize:
 - O valor da maior compra (coluna `CLI_VL_MAIORCOMPRA`), obtida na tabela de nota fiscal de vendas;
 - A data da maior compra (coluna `CLI_DT_MAIORCOMPRA`), obtida na tabela de nota fiscal de vendas;
 - A média de compras mensal (coluna `CLI_VL_MEDIACOMPRASMENSAL`), obtida na tabela de resumo mensal de vendas.
- Uma função pública que receba como argumento o código do cliente e retorne o código do produto com maior frequência de compra (através dos itens de nota fiscal ou da tabela de resumo mensal);
- Uma função pública que receba como argumento o código do cliente e retorne o código do produto com maior representatividade financeira nas compras do cliente.

6. Crie um pacote chamado `pck_representante` que contenha:

- Um procedimento público que receba como argumento o código do representante e atualize:

- O valor da maior venda (coluna REP_VL_MAIORVENDA), obtida na tabela de nota fiscal de vendas;
- A data da maior venda (coluna REP_DT_MAIORVENDA), obtida na tabela de nota fiscal de vendas;
- A média mensal de venda (coluna REP_VL_MEDIAMENSALVENDAS), obtida na tabela de resumo mensal de vendas;
- Uma função privada do pacote que calcule a meta mensal de vendas, tirando a média de vendas dos últimos três meses adicionados de 5%;
- Um procedimento público que atualize a meta mensal do representante (coluna REP_VL_METAMENSAL) utilizando a função criada.

7. Desenvolva um pacote chamado pck_produto, com procedimentos e funções necessários para atualizar:

- Data e valor de última venda de cada produto;
- Data e valor de maior venda de cada produto.

04 – Collections

1. **Tabelas por índice;**
2. **Tabelas aninhadas;**
3. **Arrays;**
4. **Collection como fonte de dados (SELECT no Array);**
5. **Bulk Binding**

Coleções

As coleções da PL/SQL permitem agrupar muitas ocorrências de um objeto. A PL/SQL disponibiliza três tipos de coleções:

- Tabelas por índice;
- Tabelas aninhadas;
- Arrays de tamanho variado (varray).

Se você conhece as outras linguagens de programação, pense em uma coleção como algo semelhante a um *array* (Um vetor). Um array é uma série de elementos repetidos, todos do mesmo tipo. Você vai aprender que alguns tipos de coleção da Oracle são mais parecidos com arrays tradicionais do que outros.

Tabelas por índice

As tabelas por índice são um dos três tipos de coleção suportados pela PL/SQL. Na verdade, elas são o tipo de coleção original. Nas primeiras versões da PL/SQL, as tabelas por índices eram o único tipo de coleção disponível.

Uma tabela por índice é uma tabela de elementos mantida em memória, onde cada elemento é indexado por um valor de inteiro. As tabelas por índice funcionam da mesma forma que os arrays, com algumas diferenças:

- Uma tabela por índice pode ser populada esparsamente;
- Você não define um tamanho máximo para uma tabela por índice.

Declarando uma tabela por índice

Para declarar uma tabela por índice você deve:

1. Definir um tipo, especificando um datatype para a coleção e outro para o índice da tabela, sendo que, o datatype da coleção pode ser um tipo escalar, tal como um `NUMBER` ou `VARCHAR2`, ou ele pode ser um tipo composto, tal como um registro e o datatype do índice da tabela deve sempre ser `BINARY_INTEGER`.
2. Declarar uma variável do tipo definido.

A sintaxe para declarar um tipo para tabela de índice é o seguinte:

```
TYPE type_name IS TABLE OF data_type [NOT NULL] INDEX BY BINARY_INTEGER;
```

Onde:

- *type_name* é o nome do tipo que você está declarando (Ele será usado para definir o tipo das variáveis);
- *data_type* é o tipo de dado da coleção, ou seja, cada elemento da tabela armazena um valor desse tipo;
- *NOT NULL* proíbe que uma entrada da tabela contenha valor nulo.

Abaixo, segue um exemplo de definição de tipos para coleção e declaração de variáveis desses tipos:

```
DECLARE  
-- Definição de tipos
```

```

TYPE TArrayNumerico IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
TYPE TRepresentante IS TABLE OF representante%ROWTYPE INDEX BY BINARY_INTEGER;

-- Declaração de variáveis
cOrdemRepresentante TArrayNumerico;
cRepresentante TRepresentante;
BEGIN
  NULL;
END;
/

```

No exemplo acima, a variável *cOrdemRepresentante* é como um array de uma única coluna do tipo number e a variável *cRepresentante* é como uma matriz de duas dimensões, ou seja, um array contendo todas as colunas da tabela representante.

Manipulando uma tabela por índice

Inserindo entradas em uma tabela por índice

Os elementos de uma tabela por índice são identificados exclusivamente por um valor de inteiro, ou índice. Sempre que referencia um valor da tabela, você deve fornecer o índice daquele valor. Para inserir os valores em uma tabela PL/SQL, você usa uma declaração de atribuição como esta:

```
table_var(index) := value;
```

Onde:

- *table_var* é o nome da variável do tipo tabela por índice;
- *index* é o valor de inteiro que indica o índice da entrada e pode ser qualquer número entre 1 e 2.147.483.647, não precisando ser consecutivos, ou seja, se você precisasse colocar apenas duas entradas em uma tabela, poderia usar os índices 1 e 2 ou poderia usar 1 e 2.147.483.647, ocupando em ambas situações, duas entradas apenas (A PL/SQL não reserva espaço para as entradas que não são usadas);
- *value* é o valor a ser atribuído para a entrada.

No exemplo abaixo, definimos um tipo de tabela por índice para conter o nome dos clientes. A partir desse tipo, declaramos uma variável que conterá o nome de cada um dos clientes:

```

DECLARE
-- declaração de cursores
CURSOR cs_cliente IS
  SELECT c.*
  FROM cliente c;

-- declaração de tipos
TYPE TNomeCliente IS TABLE OF VARCHAR2(1024) INDEX BY BINARY_INTEGER;

-- declaração de variáveis
cCli TNomeCliente;
vCount BINARY_INTEGER := 0;
BEGIN
-- Percorrer todos os clientes do cadastro

```

```
FOR csc IN cs_cliente LOOP

    -- Incrementa contador
    vCount := vCount + 1;

    -- Atribuição de valores
    cCli(vCount) := csc.cli_st_nome;
END LOOP;
END;
/
```

No exemplo acima, utilizamos um contador para gerar o índice, mas poderíamos utilizar o próprio código do cliente como índice, uma vez que ele é numérico e o índice não precisa ser utilizado na sequência, como no exemplo abaixo:

```
DECLARE
    -- declaração de cursores
    CURSOR cs_cliente IS
        SELECT c.*
        FROM cliente c;

    -- declaração de tipos
    TYPE TNomeCliente IS TABLE OF VARCHAR2(1024) INDEX BY BINARY_INTEGER;

    -- declaração de variáveis
    cCli TNomeCliente;
BEGIN
    -- Percorrer todos os clientes do cadastro
    FOR csc IN cs_cliente LOOP
        -- Atribuição de valores
        cCli(csc.cli_in_codigo) := csc.cli_st_nome;
    END LOOP;
END;
/
```

Referenciando valores em uma tabela por índice

Para referenciar uma entrada específica em uma tabela por índice, você especifica um valor de índice, usando a mesma sintaxe do tipo array que usou ao inserir os dados. Por exemplo, para avaliar se o cliente da entrada 2 é novo, basta escrever uma declaração `IF` como esta:

```
IF cCli(2).Novo THEN
    ...
    ...
END IF;
```

ou

```
IF cCli(2).Novo = TRUE THEN
    ...
    ...
END IF;
```

(Referencie um valor do tipo `BOOLEAN` como preferir!)

Como vimos anteriormente, as tabelas por índice são populadas esparsamente e por isso pode haver

valores de índices para os quais não há entrada. Se tentar referenciar um deles, você tem um exceção `NO_DATA_FOUND` e, conseqüentemente, poderá tratar essa exceção. Caso não queira utilizar a área de `EXCEPTION` para tratar essa exceção, você pode utilizar o método `EXISTS` da coleção, como mostrado abaixo:

```
IF cCli.EXISTS(15) THEN
    ...
    ...
END IF;
```

Um método é uma função ou um procedimento que é anexado a um objeto. Neste caso, a tabela `cCli` é o objeto e a função `EXISTS` é o método.

Alterando entradas de tabela

Você atualiza uma tabela PL/SQL de modo semelhante ao que você faz para inserir. Se você já inseriu um número de linha 10 na tabela `cCli` (usada no nosso exemplo), então você pode atualizar aquela mesma linha com uma declaração como esta:

```
cCli(10).Nome := 'Grupo Carrefour';
```

Essa declaração atualiza o campo *Nome* do registro na entrada da tabela de número 10, com o texto 'Grupo Carrefour'.

Excluindo entradas de tabela

Você pode excluir entradas de uma tabela invocando o método `DELETE`, que pode excluir uma entrada, um intervalo de entradas ou todas as entradas, da tabela. A sintaxe para o método `DELETE` é a seguinte:

```
table_name.DELETE[(primeira_entrada[,ultima_entrada])];
```

onde:

- *table_name* é o nome da variável de tabela;
- *primeira_entrada* é o índice da entrada que você quer excluir ou o índice da primeira entrada de um intervalo de entradas que você quer excluir;
- *ultima_entrada* é o último índice de um intervalo de entradas que você quer excluir.

IMPORTANTE: A invocação do método `DELETE` sem a passagem de nenhuma argumento, faz com que todos os dados sejam excluídos da tabela.

Então:

- Para limpar toda tabela `cCli`, basta executar a declaração:

```
cCli.DELETE;
```

- Para apagar apenas a entrada 8 da tabela `cCli`, basta executar a declaração:

```
cCli.DELETE(8);
```

- E para apagar as entradas de 6 a 10 da tabela `cCli`, execute a declaração:

```
cCli.DELETE(6,10);
```

Tabelas aninhadas

As tabelas aninhadas surgiram no Oracle 8. No banco de dados, uma tabela aninhada é uma coleção não ordenada de linhas. Pense em um sistema de notas fiscais, onde cada nota contenha um número de itens. Tradicionalmente, os criadores de banco de dados o implementariam usando duas tabelas de banco de dados, uma para as notas e uma segunda para os itens. Cada registro de item conteria um vínculo de chave estrangeira com sua “nota mãe”. A partir do Oracle 8, é possível que cada nota tenha sua própria tabela de item e que aquela tabela seja armazenada como uma coluna.

A PL/SQL suporta as tabelas aninhadas porque o banco de dados Oracle as suporta. As tabelas aninhadas são declaradas e usadas de forma semelhante às tabelas por índice, mas existem algumas diferenças que você deve conhecer:

- Quando você recupera uma tabela aninhada do banco de dados, as entradas são indexadas consecutivamente. Mesmo ao construir uma tabela aninhada dentro da PL/SQL, você não pode pular arbitrariamente os valores de índice como faria com as tabelas por índice;
- As tabelas aninhadas não suportam os datatypes específicos da PL/SQL;
- Você precisa usar um método construtor para inicializar uma tabela aninhada;
- O intervalo de índices das tabelas aninhadas é de -2.147.483.647 até 2.147.483.647 (Os índices das tabelas por índice não podem ser 0 ou números negativos).

IMPORTANTE: Quando as tabelas aninhadas estão no banco de dados, as linhas não têm nenhuma ordem determinada associada a elas. Quando você lê uma tabela aninhada na PL/SQL, cada linha recebe um índice. Assim, de certa forma, naquele ponto há uma certa ordem envolvida. Entretanto, a ordem não é preservada. Se você selecionar a mesma tabela aninhada duas vezes, você pode ter uma ordem de linha diferente a cada vez.

CUIDADO: Até a versão 8i do Oracle Database, tabela aninhada só estava disponível na distribuição Enterprise do Oracle Database, por isso, certifique-se que seu cliente possui esse recurso antes de utilizá-lo.

Se as tabelas aninhadas fossem criadas primeiro, a Oracle nunca teria desenvolvido o tipo por índice. Entretanto, ambas estão disponíveis e você deve selecionar entre elas. Se você precisar de um array de um datatype específico da PL/SQL, tal como BOOLEAN, NATURAL ou INTEGER, então sua única escolha é uma tabela por índice. A outra coisa que deve ser considerada é que as tabelas aninhadas requerem que os seus índices sejam consecutivos, tal como 1,2,3 e assim por diante.

Declarando uma tabela aninhada

Você declara uma tabela aninhada usando as duas mesmas etapas que usa para declarar uma tabela por índice. Em primeiro lugar, você declara um tipo, e depois declara uma ou mais variáveis daquele tipo.

A sintaxe para criar o tipo é a seguinte:

```
TYPE type_name IS TABLE OF data_type [NOT NULL];
```

onde:

- *type_name* é o nome do tipo que você está declarando;
- *data_type* é o datatype da coleção;
- *NOT NULL* proíbe que uma entrada de tabela seja nula.

IMPORTANTE: O datatype de uma coleção NÃO pode ser `BOOLEAN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `REF CURSOR`, `TABLE` e `VARRAY`;

Como pode ser observado, a declaração de um tipo de uma tabela aninhada é semelhante àquela usada para uma tabela por índice, mas a cláusula `INDEX BY` não é usada. A presença de uma cláusula `INDEX BY` é que indica para o Oracle que você está utilizando uma tabela por índice.

Depois de declarar um tipo de tabela aninhada, você pode usar aquele tipo para declarar as variáveis de tabela.

Manipulando tabelas aninhadas

Inicializando uma tabela aninhada

Quando você declara uma variável para uma tabela aninhada, você tem uma variável que não contém nada. Ela é considerada nula. Para torná-la uma tabela, você precisa chamar uma função construtora para criar aquela tabela e, depois, armazenar o resultado daquela função na variável que você está usando. Por exemplo, digamos que você tenha usado as seguintes declarações para criar uma tabela aninhada chamada `cRep`:

```
TYPE TRepresentante IS TABLE OF representante%ROWTYPE;  
  
cRep TRepresentante;
```

Para usar realmente `cRep` como uma tabela aninhada, você precisa chamar uma função construtora para inicializá-la. A função construtora sempre toma seu nome do nome do tipo usado para declarar a tabela; portanto, nesse caso, a função construtora se chamaria `TRepresentante`. Esse conceito vem do *mundo orientado a objetos*, onde um construtor é a função que realmente aloca memória para um objeto. Você pode chamar a função construtora sem passar nenhum argumento e criar uma tabela vazia, como no exemplo seguinte:

```
cRep := TRepresentante();
```

Você também pode chamar a função construtora e passar valores para uma ou mais entradas. Entretanto, os valores listados no construtor devem ser do mesmo tipo da tabela. Isso é um pouco difícil de fazer com os registros. O exemplo seguinte mostra como declarar `cRep` como uma tabela de registros de representante e depois a inicializa com dois representantes:

```
DECLARE  
  -- Definição de tipos  
  TYPE TRepresentante IS TABLE OF representante%ROWTYPE;  
  
  -- Declaração de variáveis  
  cRep TRepresentante;  
  rRep1 representante%ROWTYPE;  
  rRep2 representante%ROWTYPE;  
BEGIN  
  -- Atribuindo dados do primeiro representante  
  rRep1.rep_in_codigo := 10;  
  rRep1.rep_st_nome := 'Seu Madruga';  
  
  -- Atribuindo dados do segundo representante  
  rRep2.rep_in_codigo := 20;  
  rRep2.rep_st_nome := 'Chaves';
```



```
-- Inicializando a coleção de representante com duas entradas: rRep1 e rRep2
cRep := TRepresentante(rRep1, rRep2);
END;
/
```

O segredo nesse exemplo é que os argumentos para o construtor são rRep1 e rRep2.

Ambos são registros do tipo representante%ROWTYPE e coincidem com o tipo de elemento da tabela. Obviamente é um pouco trabalhoso definir as coisas dessa forma.

Para adicionar entradas a uma tabela além daquelas que você criou com o construtor, você precisa estender a tabela como discutimos na seção seguinte.

Estendendo uma tabela aninhada

Para estender uma tabela aninhada de modo a adicionar mais entradas a ela, use o método *extend*. O método *extend* permite que você adicione uma ou várias entradas. Ele também permite que você clone uma entrada existente uma ou mais vezes. A sintaxe do método *extend* é a seguinte:

```
colecacao.EXTEND[(quantidade_entradas[,entrada_clone])];
```

onde:

- *colecacao* é o nome da tabela aninhada;
- *quantidade_entradas* é o número de entradas novas que você quer adicionar;
- *entrada_clone* é uma variável ou constante que indica qual entrada você quer clonar.

A seguir temos um exemplo que utiliza o método *extend* para estender a coleção de representantes:

```
DECLARE
-- Declaração de cursores
CURSOR cs_representante IS
SELECT r.*
FROM representante r;

-- Definição de tipos
TYPE TRepresentante IS TABLE OF representante%ROWTYPE;

-- Declaração de variáveis
cRep TRepresentante;
vRepCount PLS_INTEGER :=0;
BEGIN
-- Inicializa a coleção criando uma entrada vazia
cRep := TRepresentante();

-- Adiciona uma entrada na coleção para cada representante do cadastro
FOR csr IN cs_representante LOOP
vRepCount := vRepCount + 1;
cRep.EXTEND;
cRep(vRepCount).rep_in_codigo := csr.rep_in_codigo;
cRep(vRepCount).rep_st_nome := csr.rep_st_nome;
cRep(vRepCount).rep_vl_maiorvenda := csr.rep_vl_maiorvenda;
END LOOP;
```

```

-- Clona a primeira entrada 5 vezes
cRep.EXTEND(5,1);

-- Exibe todos os registros da coleção
FOR i IN 1..(vRepCount + 5) LOOP
    dbms_output.put_line('Representante: '||cRep(i).rep_in_codigo    ||chr(13)||
                        'Nome.....: '||cRep(i).rep_st_nome      ||chr(13)||
                        'Maior Venda...: '||cRep(i).rep_vl_maiorvenda||chr(13)
                        );
END LOOP;
END;
/

```

Removendo entradas de uma tabela aninhada

Você pode remover entradas de uma tabela aninhada usando o método *delete*, assim como faz com as tabelas por índice. O exemplo abaixo exclui a entrada 10 da coleção cRep:

```
cRep.DELETE(10);
```

Você pode usar as entradas depois de excluí-las. As outras entradas da tabela não são renumeradas.

Outro método para remover as linhas de uma tabela aninhada é invocar o método *trim*. O método *trim* remove um número especificado de entradas do final da tabela.

Sintaxe:

```
colecacao.TRIM[(quantidade_entradas)];
```

Onde:

- *colecacao* é o nome da tabela aninhada;
- *quantidade_entradas* é o número de entradas a serem removidas do final. O padrão é 1.

O método *trim* se aplica apenas às tabelas aninhadas e aos arrays de tamanho variado. Ele não pode ser aplicado às tabelas por índice.

Para exemplificar a remoção de entradas de uma coleção, vamos alterar o exemplo anterior para, após exibir as entradas, remover as cinco que foram clonadas usando o método *trim* e apagar a entrada 1 usando o método *delete*:

```

DECLARE
-- Declaração de cursores
CURSOR cs_representante IS
    SELECT r.*
    FROM representante r;

-- Definição de tipos
TYPE TRepresentante IS TABLE OF representante%ROWTYPE;

-- Declaração de variáveis
cRep    TRepresentante;
vRepCount PLS_INTEGER :=0;
BEGIN
-- Inicializa a coleção criando uma entrada vazia
cRep := TRepresentante();

-- Adiciona uma entrada na coleção para cada representante do cadastro
FOR csr IN cs_representante LOOP

```

```

vRepCount := vRepCount + 1;
cRep.EXTEND;
cRep(vRepCount).rep_in_codigo := csr.rep_in_codigo;
cRep(vRepCount).rep_st_nome := csr.rep_st_nome;
cRep(vRepCount).rep_vl_maiorvenda := csr.rep_vl_maiorvenda;
END LOOP;

-- Clona a primeira entrada 5 vezes
cRep.EXTEND(5,1);

-- Exibe todos os registros da coleção
FOR i IN 1..(vRepCount + 5) LOOP
    dbms_output.put_line('Representante: '||cRep(i).rep_in_codigo ||chr(13)||
                        'Nome.....: '||cRep(i).rep_st_nome ||chr(13)||
                        'Maior Venda...: '||cRep(i).rep_vl_maiorvenda||chr(13)
                        );
END LOOP;

dbms_output.put_line(cRep.COUNT||' entradas encontradas!');

-- Remove as 5 entradas clonadas
cRep.TRIM(5);

-- Exclui a primeira entrada
cRep.DELETE(1);

-- Exibe a nova contagem de entradas
dbms_output.put_line('Agora restam '||cRep.COUNT||' entradas!');

-- Exibe as entradas que restaram
FOR i IN 1..(vRepCount + 5) LOOP
    IF cRep.EXISTS(i) THEN
        dbms_output.put_line('Representante: '||cRep(i).rep_in_codigo ||chr(13)||
                            'Nome.....: '||cRep(i).rep_st_nome ||chr(13)||
                            'Maior Venda...: '||cRep(i).rep_vl_maiorvenda||chr(13)
                            );
    END IF;
END LOOP;
END;
/

```

Nesse exemplo, o método *trim* é usado para remover os cinco clones que criamos no exemplo anterior. Logo em seguida, o método *delete* é chamado para excluir também a primeira entrada. Depois do uso do método *delete*, utilizamos o método *count* para exibir o número de entradas. Até a versão 8.1.5 a PL/SQL só reconhecia a exclusão das entradas após referenciar a contagem da coleção, ou seja, o uso do método *count* servia para desviar de um bug do Oracle. Por fim, as entradas restante são exibidas.

Arrays de tamanho variável

Assim como as tabelas aninhadas, os *arrays* de tamanho variável ou *varrays* também começaram a existir a partir da versão Oracle8. Os *arrays* são semelhantes às tabelas aninhadas, mas eles possuem um tamanho máximo fixo. Eles diferem das tabelas aninhadas porque quando você armazena um varray em uma coluna de banco de dados, a ordem dos elementos é preservada.

CUIDADO: Até a versão 8i do Oracle Database, array de tamanho variável só estava disponível na distribuição Enterprise do Oracle Database, por isso, certifique-se que seu cliente possui esse recurso antes de utilizá-lo.

Declarando e inicializando um VARRAY

Para criar um *varray*, você usa a palavra-chave `VARRAY` em uma declaração de tipo para criar um tipo de *array*. Depois você pode usar aquele tipo para declarar uma ou mais variáveis. A sintaxe para declarar um tipo *varray* é a seguinte:

```
TYPE nome_tipo IS VARRAY (size) OF tipo_entrada [NOT NULL];
```

Onde:

- *nome_tipo* é o nome do tipo de array;
- *size* é o número de elementos que você quer que o array contenha;
- *tipo_entrada* é o tipo de dados dos elementos do array;
- `NOT NULL` proíbe que as entradas do array sejam nulas.

Os *varray* precisam ser inicializados assim como as tabelas aninhadas. Antes de usar um `VARRAY`, você precisa chamar seu construtor. Você pode passar os valores para o construtor e aqueles valores são usados para criar elementos de array, ou você pode invocar o construtor sem nenhum parâmetro para criar um array vazio.

No exemplo abaixo, inicializamos uma variável `VARRAY` chamada `vStrings`, incluindo duas entradas:

```
DECLARE  
  -- Definição de tipos  
  TYPE TStrings IS VARRAY(1000000) OF VARCHAR2(1024);  
  
  -- Declaração de variáveis  
  vStrings TStrings;  
BEGIN  
  -- Inicializa a coleção criando uma entrada vazia  
  vStrings := TStrings('Linha 1', 'Linha 2');  
  
  -- Exibe todos os registros da coleção  
  FOR i IN 1..2 LOOP  
    dbms_output.put_line(vStrings(i));  
  END LOOP;  
END;  
/
```

Neste exemplo, criamos um tipo `VARRAY` chamado `TStrings` e declaramos a variável `vStrings` como sendo do tipo `TStrings`. Na seção de execução do bloco, inicializamos `vStrings` com duas linhas, contendo na primeira entrada o texto 'Linha 1' e na segunda entrada, o texto 'Linha 2'. No final, temos um `LOOP` com dois ciclos que imprimem na tela as duas entradas.

Adicionando e removendo dados de um VARRAY

Depois de inicializar um `VARRAY`, você pode adicionar e remover dados dele, assim como faz com uma tabela aninhada. Se você quiser adicionar ao *array* mais elementos do que você criou quando o inicializou, pode chamar o método *extend*. Entretanto, você só pode estender um *array* até o tamanho máximo especificado na definição do tipo *array*.

Para remover entradas de um VARRAY, basta utilizar os métodos *delete* ou *trim*, como fizemos na manipulação de tabelas aninhadas.

O exemplo abaixo mostra o conteúdo da tabela `produto` sendo lido em um VARRAY:

```
DECLARE
-- Declaração de cursores
CURSOR cs_produto IS
    SELECT p.pro_in_codigo,
           p.pro_st_nome,
           p.pro_st_marca,
           p.pro_dt_ultimavenda
    FROM produto p;

-- Definição de tipos
TYPE TProduto IS VARRAY(100) OF cs_produto%ROWTYPE;

-- Declaração de variáveis
cProd  TProduto;
vIndice PLS_INTEGER := 0;
BEGIN
-- Inicializar coleção
cProd := TProduto();

-- Abrir cursor
OPEN cs_produto;

-- Para cada produto, atribuir código, nome, marca e data de ultima venda
LOOP
    -- Incrementar indice e estender coleção
    vIndice := vIndice + 1;
    cProd.EXTEND;

    -- Recuperar dados do cursor e atribuir para coleção
    FETCH cs_produto INTO cProd(vIndice);

    -- Se não encontrar mais nada no cursor, elimina a última entrada
    -- gerada na coleção, decrementa o índice da coleção e abandona o LOOP
    IF cs_produto%NOTFOUND THEN
        cProd.TRIM(1);
        vIndice := vIndice - 1;
        EXIT;
    END IF;

END LOOP;

-- Fechar cursor
CLOSE cs_produto;

-- Imprimir cada entrada da coleção na tela
FOR i IN 1..vIndice LOOP
    dbms_output.put_line('Cód.Prod.....: '||cProd(i).pro_in_codigo||chr(13)||
                          'Nome.....: '||cProd(i).pro_st_nome||chr(13)||
                          'Marca.....: '||cProd(i).pro_st_marca||chr(13)||
                          'Última Venda: '||cProd(i).pro_dt_ultimavenda||chr(13)
    );
END FOR;
```

```
END LOOP;
END;
/
```

Neste exemplo, para cada produto retornado no cursor, utilizamos o método *extend* para incluir uma entrada na coleção e, no final, quando o comando `FETCH` não encontra mais nenhum registro no cursor, utilizamos o método *trim* para remover a última entrada da coleção que foi gerada pelo próprio comando `FETCH`.

Observe que *extend* não pode ser usado para aumentar o *array* além do tamanho máximo especificado de 100 entradas.

Métodos de tabela da PL/SQL

Nós já estudamos alguns métodos de coleção nas seções anteriores. A PL/SQL fornece vários outros métodos incorporados para uso com as tabelas. Veja a tabela a seguir:

Tabela 6: Métodos de tabela da PL/SQL

Método	Descrição	Exemplo
count	O método <i>count</i> retorna o número de entradas da tabela.	<code>vRecordCount := cCli.count</code>
exist	Essa função recebe o número da entrada como argumento e retorna o valor <code>TRUE</code> se a entrada existir. Caso contrário, ela retorna <code>FALSE</code> .	<code>IF cCli.exists(15) THEN</code> ... <code>END IF;</code>
limit	Esse método retorna o número máximo de elementos que uma coleção pode conter. Somente arrays de tamanho variável têm um limite superior. Quando usado com as tabelas aninhadas e com as tabelas por índice, o método <i>limit</i> retorna <code>NULL</code> .	<code>IF vIndice > cli_array.limit THEN</code> ... <code>ELSE</code> <code>cli_array(vIndex) := 10;</code> <code>END IF;</code>
first	Esse método retorna o valor de índice mais baixo usado em uma coleção.	<code>vMenorIndiceValido := cCli.first;</code>
last	Essa função retorna o valor do índice mais alto usado na tabela.	<code>vMaiorIndiceValido := cCli.last;</code>
next	O método <i>next</i> retorna o valor do próximo índice válido que seja maior do que um valor que você fornece.	<code>vProximoIndice := cCli.Next(vIndiceCorrente)</code>
prior	O método <i>prior</i> retorna o valor do índice válido mais alto que precede o valor do índice que você fornecer.	<code>vIndiceAnterior := cCli.Prior(vIndiceCorrente)</code>
delete	O método <i>delete</i> permite excluir entrada de uma coleção. Você pode excluir todas as entradas ou uma entrada específica ou um intervalo de entradas. O métodos <i>delete</i> é um procedimento e não retorna um valor.	-- Excluindo todas as entradas <code>cCli.delete;</code> -- Excluindo apenas uma entrada <code>cCli.delete(8);</code> -- Excluindo um intervalo de entradas <code>cCli.delete(6,10);</code>
trim	O método <i>trim</i> permite excluir entradas do final de uma coleção. Você pode usar <i>trim</i> apenas em uma	-- cortando uma entrada <code>cli_array.trim;</code>

Método	Descrição	Exemplo
	entrada ou em diversas entradas. O método <i>trim</i> é um procedimento e não retorna um valor. Este método só pode ser usado com arrays de tamanho variado e com as tabelas aninhadas.	-- cortando 3 entradas <code>cli_array.trim(3);</code>
extend	O método <i>extend</i> permite adicionar entradas no final de uma coleção. Você pode adicionar uma entrada ou várias entradas, especificando o número a ser adicionado como parâmetro. O método <i>extend</i> também pode ser usada para clonar entradas existentes. Você clona uma entrada passando o número da entrada como um segundo parâmetro para <i>extend</i> . O método <i>extend</i> é um procedimento e não retorna um valor. Assim como <i>trim</i> , <i>extend</i> só pode ser usado com os arrays de tamanho variável e com as tabelas aninhadas.	-- Adicionando uma entrada <code>cli_array.extend;</code> -- Adicionando três entradas <code>cli_array.extend(3);</code> -- Adicionando uma nova entrada clone da 3 <code>cli_array.extend(1,3);</code>

Executando declarações SELECT em uma coleção

Em PL/SQL você consegue executar uma declaração *SELECT* em uma coleção, como se ela fosse uma tabela. Isso pode ser muito útil se você precisar filtrar registros de uma coleção ou executar loops constantes no mesmo cursor, por exemplo. Mas para realizar uma operação como essa, você precisa:

1. Criar um objeto armazenado de banco de dados que contenha as colunas da coleção;
2. Criar um tipo (*type*) armazenado no banco de dados que seja uma tabela do objeto criado;

A criação de um objeto como esse pode se parecer muito com a definição de tipo registro (*record*), como no exemplo abaixo:

```
CREATE OR REPLACE TYPE ObjCliente AS OBJECT
(
  cli_in_codigo INTEGER,
  cli_st_nome   VARCHAR2(20),
  cli_ch_novo   CHAR(1)
)
/
```

Estando o objeto criado, você deve criar um tipo tabela desse objeto. O comando para isso é quase igual a definição de um tipo tabela dentro de um bloco. A única diferença é o uso da palavra *CREATE* (ou *CREATE OR REPLACE*). Veja:

```
CREATE OR REPLACE TYPE TCliente IS TABLE OF ObjCliente;
/
```

Por fim, dentro do seu bloco, você só precisa declarar uma variável do tipo criado e usá-la como as coleções que já vimos até aqui:

```
DECLARE
-- Declaração de variáveis
```

```
cCli    TCliente;  
BEGIN  
    NULL;  
END;  
/
```

Depois que você carregar os dados de sua coleção, você pode executar uma declaração `SELECT` nela. Para que PL/SQL visualizar a coleção como uma tabela, é necessário *moldar* a coleção como sendo do seu tipo, através da função `CAST` e depois, moldar resultado desse `CAST` como uma tabela. Veja o exemplo abaixo:

```
-- Imprimir total de clientes novos  
SELECT COUNT (*)  
INTO vCount  
FROM TABLE (CAST (cCli AS TCliente))
```

Onde:

- `vCount` é uma variável do tipo inteira previamente declarada;
- `CAST` é a função que informa ao PL/SQL que `cCli` deve ser visto como um tipo `TCliente`;
- e `TABLE` faz com que a PL/SQL veja a variável do tipo `TCliente`, como uma tabela.

CUIDADO: Quando usamos um tipo tabela aninhada armazenado para manipular coleções, nos deparamos com um problema de inicialização da coleção. Não basta inicializar a coleção usando o construtor do seu tipo, como vimos na seção de tabelas aninhadas. Além disso, a cada entrada estendida, é necessário inicializar a entrada com valores nulos antes de manipulá-la, usando uma variável do mesmo tipo do elemento da coleção ou usando o próprio tipo para passar os valores nulos, como no exemplo abaixo:

```
cCli (1) := ObjCliente (NULL, NULL, NULL) ;
```

Nesse exemplo, a entrada 1 da coleção cliente é inicializada com nulo para cada um dos seus três campos, usando para isso o objeto `ObjCliente`, usado para definir o tipo de cada elemento do tipo `TCliente`, que por sua vez foi usado para declarar a variável `cCli`.

Para entendermos melhor, vejamos o seguinte exemplo:

1. Criação do objeto armazenado:

```
CREATE OR REPLACE TYPE ObjCliente AS OBJECT  
(Codigo INTEGER,  
Nome VARCHAR2 (20) ,  
Novo CHAR (1)  
)  
/
```

2. Criação do tipo armazenado, sendo cada elemento do tipo do objeto criado no passo 1:

```
CREATE OR REPLACE TYPE TCliente IS TABLE OF ObjCliente;  
/
```

3. Criação do bloco que recupera todos os clientes, através de um cursor, e inclui cada cliente retornado na coleção, alimentando o código do cliente, nome do cliente e, com base na data de

inclusão, se o cliente é novo ou não. Após alimentar a coleção, imprimimos cada um dos clientes informando se é um cliente novo ou antigo. Por fim, executamos uma declaração `SELECT` para imprimir a quantidade de clientes novos e outra para imprimir a quantidade de clientes antigos:

```
DECLARE
-- Declaração de cursores
CURSOR cs_cliente IS
    SELECT c.*
    FROM cliente c;

-- Declaração de variáveis
cCli      TCliente;
vIndice   PLS_INTEGER := 0;
vCount    PLS_INTEGER := 0;
BEGIN
-- Inicializa a coleção criando uma entrada vazia
cCli := TCliente();

-- Adiciona uma entrada na coleção para cada representante do cadastro
FOR csc IN cs_cliente LOOP
    -- Estender a coleção em 1 entrada
    cCli.EXTEND;

    -- Definir valor do índice
    vIndice := cCli.LAST;

    -- Inicializar a entrada estendida
    cCli(vIndice) := ObjCliente(NULL, NULL, NULL);

    -- Atualizar os dados da entrada
    cCli(vIndice).Codigo := csc.cli_in_codigo;
    cCli(vIndice).Nome   := csc.cli_st_nome;
    IF csc.cli_dt_inclusao BETWEEN add_months(trunc(SYSDATE), -12) AND SYSDATE THEN
        cCli(vIndice).Novo := 'S';
    ELSE
        cCli(vIndice).Novo := 'N';
    END IF;
END LOOP;

-- Imprimir código e nome de cada cliente informando se é novo ou antigo
FOR i IN 1..vIndice LOOP
    CASE cCli(i).Novo
    WHEN 'S' THEN
        dbms_output.put_line('Cliente '||cCli(i).Nome||' ('||cCli(i).Codigo||') é Novo!');
    WHEN 'N' THEN
        dbms_output.put_line('Cliente '||cCli(i).Nome||' ('||cCli(i).Codigo||') é Antigo!');
    END CASE;
END LOOP;

-- Quebrar linha
dbms_output.put_line('');

-- Imprimir total de clientes novos
SELECT COUNT(*)
INTO vCount
```

```

FROM TABLE(CAST(cCli AS TCliente)) cn
WHERE cn.novo = 'S';

dbms_output.put_line('Total de Clientes Novos: '||vCount);

-- Imprimir total de clientes antigos
SELECT COUNT(*)
INTO vCount
FROM TABLE(CAST(cCli AS TCliente)) cn
WHERE cn.novo = 'N';

dbms_output.put_line('Total de Clientes Antigos: '||vCount);
END;
/

```

O uso desse recurso pode representar ganhos consideráveis em aplicações críticas, pois os dados são recuperados uma única vez e pode ser utilizado em memória quantas vezes for necessário, além de possibilitar uma manipulação pontual de cada registro através dos métodos de tabela, já estudados.

Criando um cursor explícito a partir de uma coleção

Outra maneira de utilizar uma coleção em uma instrução SQL, é atribuindo seu retorno para uma variável do tipo `REF CURSOR` e manipulá-la como um curso explícito.

Para isso basta definirmos um tipo `REF CURSOR` e declararmos uma variável desse tipo. Depois é só abrir o cursor referenciando a variável definida, usando para isso a declaração `SELECT` feita na coleção.

O exemplo abaixo é semelhante ao anterior, porém, após alimentarmos a coleção, abrimos um cursor baseado nessa coleção restringindo seu retorno aos clientes novos. Em seguida iniciamos um LOOP no cursor e imprimimos cada um dos clientes na tela.

```

DECLARE
/***** Declaração de cursores *****/
CURSOR cs_cliente IS
    SELECT c.*
    FROM cliente c;

/***** Definição de tipos *****/
TYPE TCursor IS REF CURSOR;
TYPE TRegistroCliente IS RECORD(
    Codigo INTEGER,
    Nome VARCHAR2(20),
    Novo CHAR(1)
);

/***** Declaração de variáveis *****/
-- variáveis de cursores
cs_ClienteNovo TCursor;
-- coleções
cCli          TCliente;
-- registros
rCliente      TRegistroCliente;
-- esclares
vIndice       PLS_INTEGER := 0;
BEGIN
-- Inicializa a coleção criando uma entrada vazia
cCli := TCliente();

```

```

-- Adiciona uma entrada na coleção para cada representante do cadastro
FOR csc IN cs_cliente LOOP
    -- Estender a coleção em 1 entrada
    cCli.EXTEND;

    -- Definir valor do índice
    vIndice := cCli.LAST;

    -- Inicializar a entrada estendida
    cCli(vIndice) := ObjCliente(NULL, NULL, NULL);

    -- Atualizar os dados da entrada
    cCli(vIndice).Codigo := csc.cli_in_codigo;
    cCli(vIndice).Nome := csc.cli_st_nome;
    IF csc.cli_dt_inclusao BETWEEN add_months(trunc(SYSDATE), -12) AND SYSDATE THEN
        cCli(vIndice).Novo := 'S';
    ELSE
        cCli(vIndice).Novo := 'N';
    END IF;
END LOOP;

-- Abri o cursor com uma declaração SELECT na coleção cCli
OPEN cs_ClienteNovo FOR SELECT Codigo, Nome, Novo
                        FROM TABLE(CAST(cCli AS TCliente)) cn
                        WHERE cn.novo = 'S';

-- Imprimir código e nome de cada cliente novo
LOOP
    -- Recupera registro do cursor aberto a partir da coleção
    FETCH cs_ClienteNovo INTO rCliente;

    -- Sai do loop quando não encontrar registro no cursor
    EXIT WHEN cs_ClienteNovo%NOTFOUND;

    dbms_output.put_line('Cliente ' || rCliente.Nome ||
                        ' (' || rCliente.Codigo || ') é Novo!'
                        );
END LOOP;

-- Fecha cursor
CLOSE cs_ClienteNovo;
END;
/

```

IMPORTANTE: Tipos de coleções locais não são permitidos em instruções SQL, apenas tipos armazenados.

O bulk binding

O bulk binding da PL/SQL é um recurso que surgiu na versão Oracle8i. O bulk binding permite codificar as declarações SQL que operam em todas as entradas de uma coleção, sem ter de fazer o LOOP em toda a coleção usando o código PL/SQL. Vários dos exemplos dados até aqui, usaram um LOOP FOR de cursor para carregar os dados de uma tabela de banco de dados para uma tabela ou array da PL/SQL. A mudança da SQL (FETCH) para a PL/SQL (para adicionar os dados ao array) é chamada de

mudança de contexto e consome muita *overhead*. Você pode usar o recurso de `bulk binding` para evitar grande parte daquela *overhead*.

Duas novas palavras-chave suportam o binding: `BULK COLLECT` e `FORALL`.

`BULK COLLECT` é usada com as declarações `SELECT` para colocar todos os dados em uma coleção.

`FORALL` é usada com as declarações `INSERT`, `UPDATE` e `DELETE` para executar aquelas declarações uma vez para cada elemento de uma coleção.

Usando BULK COLLECT

Você pode usar as palavras-chave `BULK COLLECT` para ter os resultados de uma declaração `SELECT` colocados diretamente em uma coleção. Você pode usar `BULK COLLECT` com as declarações `SELECT INTO` e também com as declarações `FETCH`. Por exemplo, se `RepCodigos` e `RepNomes` fossem ambas tabelas aninhadas, você emitiria a seguinte declaração `SELECT` para gerar nelas uma entrada para cada representante existente na tabela `representante`:

```
SELECT rep_in_codigo, rep_st_nome
BULK COLLECT INTO RepCodigos, RepNomes
FROM representante;
```

Se você tivesse um cursor chamado `cs_representante` que retornasse os mesmos dados, você poderia escrever `BULK COLLECT` na declaração `FETCH` da seguinte maneira:

```
OPEN cs_representante;
FETCH cs_representante BULK COLLECT INTO RepCodigos, RepNomes;
CLOSE cs_representante;
```

Até o Oracle 8i, por algum motivo, a Oracle não permitia que você usasse `BULK COLLECT` em uma coleção de registros. Assim sendo, se você selecionasse dez colunas, precisaria declarar dez coleções, uma para cada coluna.

A partir da versão 9r2 do Oracle Database, isso mudou, e passou a ser aceito o uso de coleções de registro. Dessa forma, se tivéssemos uma variável chamada `cRep` que fosse de um tipo registro que contivesse uma coluna para código e outra para nome, poderíamos reescrever nossos dois exemplos acima, como abaixo:

- Na declaração `SELECT`:

```
SELECT rep_in_codigo, rep_st_nome
BULK COLLECT INTO cRep
FROM representante;
```

- No `FETCH` do cursor:

```
OPEN cs_representante;
FETCH cs_representante BULK COLLECT INTO cRep;
CLOSE cs_representante;
```

Para exemplificar melhor, vamos reescrever o exemplo que classifica o cliente como novo ou antigo:

```
DECLARE
-- Declaração de cursores
CURSOR cs_cliente IS
SELECT ObjCliente(
    c.cli_in_codigo,
    c.cli_st_nome,
```

```

        (CASE
            WHEN (c.cli_dt_inclusao BETWEEN add_months(trunc(SYSDATE),-12)
                AND SYSDATE) THEN 'S'
            ELSE 'N'
            END
        )
    )
FROM cliente c;

-- Declaração de variáveis
cCli    TCliente;
vCount  PLS_INTEGER := 0;
BEGIN
    -- Inicializa a coleção criando uma entrada vazia
    cCli := TCliente();

    OPEN cs_cliente;
    FETCH cs_cliente BULK COLLECT INTO cCli;
    CLOSE cs_cliente;

    -- Imprimir código e nome de cada cliente informando se é novo ou antigo
    FOR i IN 1..cCli.COUNT LOOP
        CASE cCli(i).Novo
            WHEN 'S' THEN
                dbms_output.put_line('Cliente '||cCli(i).Nome||' ('||cCli(i).Codigo||') é
Novo!');
            WHEN 'N' THEN
                dbms_output.put_line('Cliente '||cCli(i).Nome||' ('||cCli(i).Codigo||') é
Antigo!');
            END CASE;
        END LOOP;

    -- Quebrar linha
    dbms_output.put_line('');

    -- Imprimir total de clientes novos
    SELECT COUNT(*)
    INTO vCount
    FROM TABLE(CAST(cCli AS TCliente)) cn
    WHERE cn.novo = 'S';

    dbms_output.put_line('Total de Clientes Novos: '||vCount);

    -- Imprimir total de clientes antigos
    SELECT COUNT(*)
    INTO vCount
    FROM TABLE(CAST(cCli AS TCliente)) cn
    WHERE cn.novo = 'N';

    dbms_output.put_line('Total de Clientes Antigos: '||vCount);
END;
/

```

Este bloco PL/SQL apresenta apenas duas alterações em relação ao bloco que usamos para exemplificar o uso de declaração `SELECT` com coleções:

1. A declaração `SELECT` do cursor `cs_cliente`, com o uso da instrução `CASE`, já retorna se o cliente é novo ou não e as colunas retornadas na declaração são moldadas para o objeto `ObjCliente` (uma espécie de `CAST`);
2. O código que executava um `LOOP` no cursor e atribuía os dados de cada cliente para a coleção, foi substituído por três linhas que:
 - a) Abre o cursor (`OPEN cs_cliente`);
 - b) Alimenta a coleção com todos os dados do cursor (`FETCH cs_cliente BULK COLLECT INTO cCli`);
 - c) Fecha o cursor (`CLOSE cs_cliente`).

Desta forma, ganhamos em performance e deixamos o código mais enxuto.

Usando FORALL

A palavra-chave `FORALL` permite que você baseie uma declaração DML (Data Manipulation Language), ou seja, `INSERT`, `UPDATE` ou `DELETE`, no conteúdo de uma coleção. Quando `FORALL` é usada, a declaração é executada uma vez para cada entrada da coleção, mas apenas uma mudança de contexto é feita da PL/SQL para a SQL. O desempenho resultante é muito mais rápido do que aquilo que você teria se codificasse um `LOOP` na PL/SQL.

Para exemplificar o uso do `FORALL`, vamos alterar o exemplo do `BULK COLLECT` para passar o nome de todos os clientes para maiúsculas:

```
DECLARE
-- Declaração de cursores
CURSOR cs_cliente IS
    SELECT c.cli_in_codigo,
           c.cli_st_nome
    FROM cliente c;

-- Definição de tipo
TYPE TClienteID IS TABLE OF cliente.cli_in_codigo%TYPE;
TYPE TClienteNome IS TABLE OF cliente.cli_st_nome%TYPE;

-- Declaração de variáveis
cCliID TClienteID;
cCliNome TClienteNome;
vCount PLS_INTEGER := 0;
BEGIN
-- Carregar coleções com dados do cursor
OPEN cs_cliente;
FETCH cs_cliente BULK COLLECT INTO cCliID, cCliNome;
CLOSE cs_cliente;

-- Alterar todos dos nomes da tabela de cliente para maiúscula
FORALL x IN cCliID.FIRST..cCliID.LAST
    UPDATE cliente
    SET cli_st_nome = upper(cCliNome(x))
    WHERE cli_in_codigo = cCliID(x);

-- Capturar quantidade de linhas atualizadas
vCount := SQL%ROWCOUNT;

-- Informar quantidade de registros atualizados
```

```
dbms_output.put_line(vCount || ' registro(s) atualizado(s).');  
  
-- Efetivar alterações no banco  
COMMIT;  
END;  
/
```

Observe que neste exemplo, definimos duas coleções cujos tipos estão definidos no próprio bloco, ao invés de estarem armazenados no banco de dados. Uma coleção é para conter o código do cliente e a outra para conter o nome. Isso é necessário porque na declaração `UPDATE` do recurso `FORALL`, não conseguimos referenciar o campo de um registro usado num `bulk binding`. Logo não conseguiríamos utilizar o campo código do objeto `ObjCliente` usado para definir os elementos do tipo `TCliente` (como vínhamos fazendo até aqui).

Se você tentar referenciar um campo de registro ao utilizar `FORALL`, receberá o seguinte erro de retorno:

PLS-00436: restrição de implementação: não é possível fazer referência a campos da tabela de registros `BULK In-BIND`

Tratamento de exceções para as coleções

Algumas exceções da PL/SQL estão relacionadas diretamente às coleções. Elas estão listadas na tabelas abaixo:

Tabela 7: Exceções relacionadas às coleções

Exceção	Causa
<code>COLLECTION_IS_NULL</code>	Você tentou usar a coleção antes de inicializá-la com sua função construtora.
<code>NO_DATA_FOUND</code>	Você tentou acessar o valor de uma entrada em uma coleção e aquela entrada não existe.
<code>SUBSCRIPT_BEYOND_COUNT</code>	Você usou um subscript (o índice) que excede o número de elementos atuais da coleção.
<code>SUBSCRIPT_OUTSIDE_LIMIT</code>	Você usou um subscript (o índice) com um varray que era maior do que o máximo suportado pela declaração de tipo do array.
<code>VALUE_ERROR</code>	Você usou um subscript (o índice) que não pode ser convertido para um inteiro.

Ao escrever código que lida com coleções, você pode detectar essas exceções ou gravar código para evitá-las. Você pode evitar `NO_DATA_FOUND`, por exemplo, testando a validade de cada entrada com o método `exists` antes de tentar acessar o valor da entrada. O seguinte trecho de código mostra como isso é feito:

```
-- Se o elemento 10 existe  
IF cCli.EXISTS(10) THEN  
    ...  
    ...  
-- Se o elemento 10 não existe  
ELSE  
    ...  
    ...  
END IF;
```

Você pode evitar os erros de *subscript* com código cuidadoso. Se você está trabalhando com

`VARRAY`, saiba antes o número de elementos que você declarou para aquele `VARRAY` conter. Se você está trabalhando com uma tabela aninhada, e não tem mais certeza do tamanho, use o método `count` para verificar o tamanho da tabela.

Exercícios Propostos

1. Adicione a coluna `cli_ch_novo` na tabela `cliente`, usando o seguinte comando:

```
ALTER TABLE cliente
ADD (cli_ch_novo CHAR(1) DEFAULT 'S' NOT NULL
     CONSTRAINT ck_cli_ch_novo CHECK(cli_ch_novo IN('S','N'))
);
```

Estando a tabela alterada, edite o pacote `pck_cliente` e crie um procedimento público, que:

- Recupere os dados dos clientes através de um cursor;
 - Monte uma coleção (ou quantas achar necessário) com os dados do cliente e mais uma coluna que identifique se o cliente é novo caso a data de inclusão não tiver mais do que um ano ou antigo, se a data de inclusão teve mais do que um ano;
 - Atualize o cadastro de cada cliente da coleção, informando 'S' na coluna `cli_ch_novo` quando o cliente for novo e 'N', quando o cliente for antigo.
2. Na package `pck_representante`, crie uma função pública que:
- Receba como argumento o código do representante, a data inicial do período e a data final;
 - Alimente uma coleção com o total de cada venda realizada pelo representante no período informado (tabela `nota_fiscal_venda`) calculando uma coluna extra com a comissão do representante para cada nota fiscal;
 - Totalize as comissões do representante e retorne esse total.
- A comissão deve ser 5% do valor total da nota, mais:
- 1% se o cliente for novo;
 - 1% se a venda for á vista ou 0,5% se a venda for em 3 vezes sem juros.
3. Adicione uma função na package `pck_produto` que receba o código do produto e retorne a quantidade de vezes que foi praticado um valor menor do que o preço de vendas cadastrado. Para isso, carregue todas as vendas do produto em uma coleção, registrando para cada elemento da coleção o valor unitário praticado e o valor de vendas cadastrado. Execute uma declaração `SELECT` nessa coleção fazendo a contagem de vezes que o valor unitário praticado é maior do que o valor de venda cadastrado. Lembre-se que o valor unitário praticado está na tabela `item_nota_fiscal_venda` e o valor de venda cadastrado está na tabela `preco_produto`, que contém os preços de venda e de compra.

05 – Tópicos avançados: PL/SQL

1. **PL/SQL:**
 - **SQL Dinâmico**
 - ***Stored Procedure* com transação autônoma**
 - **Função *PIPELINED***
 - **UTL_FILE: Escrita e leitura de arquivos no servidor**

SQL Dinâmico

SQL Dinâmico é a SQL ou PL/SQL que é gerada por um programa quando ele é executado. Você usa a SQL dinâmica quando precisa escrever software genérico. Por exemplo, se você desenvolver um gerador de relatórios não tem como saber com antecedência quais são os relatórios que as pessoas criarão usando esse gerador. Neste caso, você precisa tornar o seu código flexível e genérico o suficiente para permitir que os usuários executem qualquer consulta que queiram.

O SQL dinâmico é um recurso para fornecer a flexibilidade no desenvolvimento de código PL/SQ e SQLs genéricos. Esse código é gerado pelo software no *runtime* e depois executado.

ATENÇÃO: SQL dinâmico deve ser utilizado com responsabilidade, pois ele é prejudicial para a performance do banco de dados. O SQL dinâmico não é compartilhado na memória do banco de dados.

Nos exemplos que vimos até aqui, trabalhamos com SQL estático (também chamado de nativo), onde é necessário saber com antecedência como deve ser seu SQL.

Exemplo: Se quisermos que nossa package `pck_cliente` tenha uma função que retorne a média geral de compra de clientes, permitindo que o usuário decida se quer a média de:

- um ramo de atividade
- uma cidade
- de um ramo em uma cidade
- ou geral,

Teríamos que criar uma código como o que segue abaixo:

(Na especificação da package)

```
-- Função que retorna a média de compra (geral,  
-- por ramo de atividade, cidade ou ramo + cidade)  
FUNCTION GetMediaCompraMensal  
    (pRamo cliente.cli_st_ramoatividade%TYPE  
    ,pCidade cliente.cli_st_cidade%TYPE)  
RETURN NUMBER;
```

(No corpo da package)

```
-- Função que retorna a media de compra (geral,  
-- por ramo de atividade, cidade ou ramo + cidade)  
FUNCTION GetMediaCompraMensal  
    (pRamo cliente.cli_st_ramoatividade%TYPE  
    ,pCidade cliente.cli_st_cidade%TYPE)  
RETURN NUMBER IS  
    vlResult cliente.cli_vl_mediacomprasmensal%TYPE := 0;  
BEGIN  
  
    -- Se for por ramo de atividade e cidade  
    IF (pRamo IS NOT NULL) AND (pCidade IS NOT NULL) THEN  
        SELECT AVG(c.cli_vl_mediacomprasmensal)  
        INTO vlResult  
        FROM cliente c
```

```

WHERE c.cli_st_amoatividade = pRamo
AND c.cli_st_cidade = pCidade;

-- Se for apenas por ramo de atividade
ELSIF (pRamo IS NOT NULL) AND (pCidade IS NULL) THEN
SELECT AVG(c.cli_vl_mediacomprasmensal)
INTO vlResult
FROM cliente c
WHERE c.cli_st_amoatividade = pRamo;

-- Se for apenas por cidade
ELSIF (pRamo IS NULL) AND (pCidade IS NOT NULL) THEN
SELECT AVG(c.cli_vl_mediacomprasmensal)
INTO vlResult
FROM cliente c
WHERE c.cli_st_cidade = pCidade;

-- Se for geral
ELSIF (pRamo IS NULL) AND (pCidade IS NULL) THEN
SELECT AVG(c.cli_vl_mediacomprasmensal)
INTO vlResult
FROM cliente c;

END IF;

RETURN(vlResult);
EXCEPTION
WHEN no_data_found THEN
RETURN(0);
WHEN OTHERS THEN
raise_application_error
(-20100, 'Não foi possível recuperar média de compra para: '||
chr(13)||'Ramo: '||pRamo||
chr(13)||'Cidade: '||pCidade||
chr(13)||
chr(13)||'Erro: '||sqlerrm);

END GetMediaCompraMensal;

```

A solução acima atende a necessidade que descrevemos, conforme podemos conferir com a instrução SQL abaixo:

SELECT c.cli_in_codigo	"Cod. Cliente"
, c.cli_st_nome	"Nome Cliente"
, c.cli_vl_mediacomprasmensal	"Media Mensal Compra"
, c.cli_st_amoatividade	"Ramo Atividade"
, pck_cliente.GetMediaCompraMensal	
(c.cli_st_amoatividade	
, NULL)	"Media Ramo"
, c.cli_st_cidade	"Cidade"
, pck_cliente.GetMediaCompraMensal	
(NULL	
, c.cli_st_cidade)	"Media Cidade"
, pck_cliente.GetMediaCompraMensal	
(c.cli_st_amoatividade	

```

        ,c.cli_st_cidade)                "Media Ramo/Cidade"
FROM cliente c;

```

Mas observe na função que, nas quatro possibilidades de retorno da função, repetimos a mesma SQL trocando apenas suas restrições (WHERE ... AND).

A mesma função poderia ser reescrita utilizando SQL dinâmico com um SQL genérico, como vemos abaixo:

```

FUNCTION GetMediaCompraMensal
    (pRamo    cliente.cli_st_amoatividade%TYPE
    ,pCidade  cliente.cli_st_cidade%TYPE)
RETURN NUMBER IS
    vlResult    cliente.cli_vl_mediacomprasmensal%TYPE := 0;
    vlSQL       LONG;
    vlWhereOrAnd VARCHAR2(6) := 'WHERE ';
BEGIN

    vlSQL := 'SELECT AVG(c.cli_vl_mediacomprasmensal) ' ||
            'FROM cliente c ';

    -- Filtrar Ramo
    IF (pRamo IS NOT NULL) THEN
        vlSQL := vlSQL ||
            'WHERE c.cli_st_amoatividade = ' || pRamo || ' ';

        vlWhereOrAnd := 'AND ';
    END IF;

    -- Filtrar Cidade
    IF (pCidade IS NOT NULL) THEN
        vlSQL := vlSQL ||
            vlWhereOrAnd || 'c.cli_st_cidade = ' || pCidade || ' ';
    END IF;

    EXECUTE IMMEDIATE vlSQL
    INTO vlResult;

    RETURN(vlResult);
EXCEPTION
    WHEN no_data_found THEN
        RETURN(0);
    WHEN OTHERS THEN
        raise_application_error
            (-20100, 'Não foi possível recuperar média de compra para: ' ||
                chr(13) || 'Ramo: ' || pRamo ||
                chr(13) || 'Cidade: ' || pCidade ||
                chr(13) ||
                chr(13) || 'Erro: ' || sqlerrm);

END GetMediaCompraMensal;

```

No exemplo acima, ao invés de concatenarmos os parâmetros *pRamo* e *pCidade* na SQL, poderíamos ter usado a cláusula **USING** do comando **EXECUTE IMMEDIATE** para informar as variáveis de

ligação, mas para isso, seria preciso garantir que todas as variáveis de ligação (*bind variables*) informadas estariam presentes na SQL executada (em nosso exemplo, nem sempre isso aconteceria).

IMPORTANTE: O 'EXECUTE IMMEDIATE' foi introduzido na versão 8i do Oracle Database. Nas versões anteriores, a única maneira de gerar SQL Dinâmico era através do pacote DBMS_SQL e sua utilização não era das mais fáceis.

SQL Dinâmico em cursores

Também podemos criar cursores com SQL dinâmico, concatenando valores ou usando variáveis de ligação, como no exemplo abaixo:

```
DECLARE
    vlCursor    SYS_REFCURSOR;
    vlRamo      cliente.cli_st_amoatividade%TYPE := 'SUPERMERCADO';
    vlCliente    cliente%ROWTYPE;
BEGIN
    OPEN vlCursor FOR 'SELECT * '||
                      'FROM cliente c '||
                      'WHERE c.cli_st_amoatividade = :1' USING vlRamo;

    LOOP
        FETCH vlCursor INTO vlCliente;
        EXIT WHEN vlCursor%NOTFOUND;

        dbms_output.put_line('Cliente '||vlCliente.cli_st_nome);
    END LOOP;
    CLOSE vlCursor;
END;
```

Stored Procedure com transação autônoma

Pode existir situações onde uma programa invoque outro programa e algo do segundo programa precise de um COMMIT independente do que aconteça no primeiro. Por sua vez, o primeiro programa não pode ser afetado de forma alguma pelo COMMIT executado no segundo.

Pelo que vimos até o momento, isso não seria possível, pois um COMMIT no segundo programa também valeria para tudo que foi executado no primeiro até aquele momento.

Resumindo, o que precisamos é que o segundo programa tenha uma transação independente (autônoma). Para que isso seja possível, o Oracle disponibiliza o PRAGMA AUTONOMOUS_TRANSACTION. Com esse recurso, você poderá implementar programas PL/SQL que possuem transação independente, ou seja, qualquer COMMIT ou ROLLBACK que ocorrer dentro desse programa, só valerá para ele.

ATENÇÃO: Use o PRAGMA AUTONOMOUS_TRANSACTION com responsabilidade, pois ele pode causar comportamentos indesejados no seu sistema e dificultar a manutenção.

Vejamos um exemplo simples de como utilizar esse recurso:

Crie as seguintes tabelas:

```

CREATE TABLE atualizacao_resumo_vendas
(arv_dt_atualizacao DATE NOT NULL
,arv_st_usuario VARCHAR2(30) NOT NULL
,arv_st_conclusao VARCHAR2(10) DEFAULT 'EXECUTANDO' NOT NULL
  CONSTRAINT ck_arv_st_conclusao CHECK(arv_st_conclusao
IN('EXECUTANDO','OK','FALHOU'))
)
/

CREATE TABLE temp_atualizacao_resumo
(tar_dt_atualizacao DATE NOT NULL
,tar_st_usuario VARCHAR2(30) NOT NULL
,tar_st_conclusao VARCHAR2(10) DEFAULT 'EXECUTANDO' NOT NULL
  CONSTRAINT ck_tar_st_conclusao CHECK(tar_st_conclusao
IN('EXECUTANDO','OK','FALHOU'))
);

```

Implemente a seguinte package:

```

CREATE OR REPLACE PACKAGE pck_vendas IS

  -- Author   : JOSINEIS
  -- Created  : 15/1/2011 14:53:47
  -- Purpose  : pacote para manipulação de vendas

  /* Procedure para atualizar o resumo mensal de vendas */
  PROCEDURE AtualizaResumoMensal;

END pck_vendas;
/

CREATE OR REPLACE PACKAGE BODY pck_vendas IS

  /* Registrar log de atualização do resumo mensal de vendas */
  PROCEDURE RegistrarAtualizacaoResumo
    (pSituacaoConclusao atualizacao_resumo_vendas.arv_st_conclusao%TYPE) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    BEGIN
      INSERT INTO
atualizacao_resumo_vendas(arv_dt_atualizacao,arv_st_usuario,arv_st_conclusao)
      VALUES(SYSDATE,USER,pSituacaoConclusao);

      COMMIT;
    EXCEPTION
      WHEN dup_val_on_index THEN
        NULL;
      WHEN OTHERS THEN
        raise_application_error
          (-20100, 'Não foi possível registrar a atualização do resumo mensal
de vendas' ||
            chr(13) || 'Erro: ' || SQLERRM);
    END;

```

```

END RegistrarAtualizacaoResumo;

/* Procedure para atualizar o resumo mensal de vendas */
PROCEDURE AtualizaResumoMensal IS

    -- Declaração de cursores
    CURSOR cs_vendas IS
        SELECT to_char(n.nfv_dt_emissao,'yyyy') ano,
               to_char(n.nfv_dt_emissao,'mm') mes,
               i.pro_in_codigo,
               n.cli_in_codigo,
               n.rep_in_codigo,
               SUM(i.infv_vl_total) vl_total,
               COUNT(i.infv_in_numero) numerovendas,
               SUM(i.infv_qt_faturada) qt_faturada
        FROM nota_fiscal_venda n,
             item_nota_fiscal_venda i
        WHERE n.nfv_in_numero = i.nfv_in_numero
        GROUP BY to_char(n.nfv_dt_emissao,'yyyy'),
               to_char(n.nfv_dt_emissao,'mm'),
               i.pro_in_codigo,
               n.cli_in_codigo,
               n.rep_in_codigo;

BEGIN

    RegistrarAtualizacaoResumo(pSituacaoConclusao => 'EXECUTANDO');

    -- Inicia Loop nos registros de venda
    FOR csv IN cs_vendas LOOP
        BEGIN
            INSERT INTO
                resumo_mensal_venda(rmv_in_ano,rmv_in_mes,pro_in_codigo,cli_in_codigo,rep_in_codigo,rmv_vl_total,rmv_in_numerovendas,rmv_qt_vendida)
            VALUES (csv.ano,csv.mes,csv.pro_in_codigo,csv.cli_in_codigo,csv.rep_in_codigo,csv.vl_total,csv.numerovendas,csv.qt_faturada);
        EXCEPTION
            WHEN dup_val_on_index THEN
                UPDATE resumo_mensal_venda r
                SET r.rmv_vl_total = csv.vl_total,
                    r.rmv_in_numerovendas = csv.numerovendas,
                    r.rmv_qt_vendida = csv.qt_faturada
                WHERE r.rmv_in_ano = csv.ano
                AND r.rmv_in_mes = csv.mes
                AND r.pro_in_codigo = csv.pro_in_codigo
                AND r.cli_in_codigo = csv.cli_in_codigo
                AND r.rep_in_codigo = csv.rep_in_codigo;
            WHEN OTHERS THEN
                RegistrarAtualizacaoResumo(pSituacaoConclusao => 'FALHOU');
                ROLLBACK;
                raise_application_error
                    (-20100,'Não foi possível atualizar o seguinte resumo mensal de
vendas: ' ||
                    chr(13) || 'Ano.....: ' || to_char(csv.ano) ||
                    chr(13) || 'Mês.....: ' || to_char(csv.mes) ||
                    chr(13) || 'Produto.....: ' || to_char(csv.pro_in_codigo) ||
                    chr(13) || 'Cliente.....: ' || to_char(csv.cli_in_codigo) ||

```



```
        chr(13)||'Representante: '||to_char(csv.rep_in_codigo)||
        chr(13)||
        chr(13)||'Erro: '||SQLERRM);
    END;
END LOOP;

RegistrarAtualizacaoResumo(pSituacaoConclusao => 'OK');

END AtualizaResumoMensal;

END pck_vendas;
/
```

Abra duas janelas SQL e:

1. Na primeira execute o seguinte bloco (**NÃO EXECUTE COMMIT**):

```
BEGIN
    pck_vendas.AtualizaResumoMensal;

    INSERT INTO temp_atualizacao_resumo
        (tar_dt_atualizacao
        ,tar_st_usuario
        ,tar_st_conclusao)
    VALUES
        (SYSDATE
        ,USER
        , 'OK');
END;
/
```

2. Em seguida, execute as seguintes consultas na mesma janela do bloco anônimo:

```
SELECT * FROM atualizacao_resumo_vendas
/
SELECT * FROM temp_atualizacao_resumo
/
```

Todos os INSERTs realizados na operação serão vistos, pois como estamos na mesma transação do bloco, também visualizamos o que ainda não sofreu COMMIT, ou seja:

- Na primeira consulta: Um registro com a situação 'EXECUTANDO' e outro com a situação 'OK'. Esses registros foram incluídos pela procedure `pck_vendas.RegistrarAtualizacaoResumo`, que possui transação autônoma;
 - Na segunda consulta: Um registro com a situação 'OK', incluído pelo `INSERT` do nosso bloco anônimo e que ainda não sofreu `COMMIT`;
3. Na segunda janela de SQL, execute as duas consultas novamente e veja que:
 - A primeira consulta exibe o mesmo resultado da outra janela SQL (registros incluídos pela procedure com transação autônoma);
 - A segunda consulta não exibe o registro incluído no bloco anônimo (ainda sem `COMMIT`);
 4. Volte na janela do bloco anônimo e execute o `COMMIT`;

5. Na segunda janela de SQL, repita as duas consultas e veja que agora a segunda consulta mostra o registro incluído pelo bloco anônimo (e finalmente submetido ao `COMMIT`);

Como pode ser observado, desde a execução do bloco anônimo, os registros da tabela `ATUALIZACAO_RESUMO_VENDAS`, incluídos e submetidos ao `COMMIT` pela procedure `pck_vendas.RegistrarAtualizacaoResumo` (que possui transação autônoma), eram visíveis nas duas janelas de SQL.

Em contrapartida, os registros da tabela `TEMP_ATUALIZACAO_RESUMO`, cujo `INSERT` acontecia no bloco anônimo, só passaram a ser visíveis na outra janela de SQL após o `COMMIT`;

IMPORTANTE: Somente a partir da versão 9i do Oracle Database é que *stored* proceures com transação autônoma ganharam suporte para transações entre bancos de dados distribuídos.

Tabela Função PL/SQL (PIPELINED)

Uma tabela função PL/SQL, também chamada de *função pipelined*, é um recurso pouco conhecido do Oracle 9i.

Função pipelined é um recurso PL/SQL para gerar registros em memória, como se fosse uma tabela virtual. A grande vantagem dessa modalidade é a habilidade de devolver linhas dinamicamente a partir da geração da mesma, ou seja, quando incluímos um registro nessa *tabela virtual* (através do comando `PIPE ROW`), ele a envia de volta ao cliente, antes de gerar a próxima, ao contrário de um array PL/SQL, que constrói todos os registros em *batch* antes de retorná-los ao cliente. Isso quer dizer que o recurso de memória é pouco utilizado, pois não precisa “segurar” a informação.

Dessa forma você consegue eliminar a necessidade de armazenar dados que são apenas para auxiliá-lo na recuperação de outros dados, minimizando o uso do recurso de memória.

Um uso comum para esse recurso é para fazer carga de uma dimensão de tempo dentro de um Datawarehouse, onde você precisa gerar várias linhas de referências aos anos e isso não tem uma origem exata.

Suponha que você em seu ambiente de trabalho se depare com um problema onde você precise executar uma declaração `SELECT` que retorne o valor total de venda para cada mês de um determinado período, por exemplo, de 01/2006 a 12/2006:

<i>Período</i>	<i>Valor</i>
01/2006	0
02/2006	80000
03/2006	20000
04/2006	50000
05/2006	60000
06/2006	15000
07/2006	100000
08/2006	35000
09/2006	40000
10/2006	70000
11/2006	0
12/2006	87000

Observe que nos meses de janeiro e novembro os valores são zerados, provavelmente porque não foram realizadas vendas. Uma solução possível é criar uma tabela na base de dados com apenas um

campo, contendo todos os períodos (mês/ano) para relacionar com a tabela que contém o total de vendas.

Suponhamos que tivéssemos uma tabela chamada `período_resumo` que contivesse todos os períodos de que precisamos, como abaixo:

<code>per_in_mes</code>	<code>per_in_ano</code>
01	2006
02	2006
03	2006
04	2006
05	2006
06	2006
07	2006
08	2006
09	2006
10	2006
11	2006
12	2006

Considerando que o total de vendas por período será obtido pela tabela `resumo_mensal_vendas`, teríamos que relacionar essas duas tabelas da seguinte maneira para obtermos o resultado que desejamos.

```
SELECT p.per_in_mes,  
       p.per_in_ano,  
       SUM(r.rmv_vl_total) rmv_vl_total  
FROM período_resumo p,  
     resumo_mensal_venda r  
WHERE p.per_in_mes BETWEEN 1 AND 12  
AND p.per_in_ano = 2006  
AND p.per_in_mes = r.rmv_in_mes(+)  
AND p.per_in_ano = r.rmv_in_ano(+)  
GROUP BY p.per_in_mes,  
         p.per_in_ano  
ORDER BY p.per_in_mes,  
         p.per_in_ano;
```

O resultado esperado é alcançado utilizando somente instruções SQL, mas precisamos manter dados persistentes de período só para isso.

O Oracle é um banco de dados Objeto-Relacional e, sendo assim, possui extensões para suporte de classes e objetos sendo esse o recurso necessário para a criação de uma “PL/SQL table function”, ou seja, uma função em PL/SQL que retorne um conjunto de dados complexos que possa ser tratado como uma tabela substituindo assim dados por um “objeto” não persistente que seja acessível na linguagem SQL.

Com esse recurso, podemos montar a “tabela” de períodos apenas quando precisarmos utilizá-la, não precisando armazenar esses dados.

Montando uma função PIPELINED

Se a função retornará um conjunto de dados, primeiramente precisaremos criar no banco, essa estrutura que será retornada, ou seja, não basta criar a função. Para trabalhar com o `PIPELINED` é necessário:

1. Criar um objeto de banco de dados armazenado, contendo as colunas das quais você precisará;

2. Criar uma coleção onde cada elemento é do tipo do objeto criado no primeiro passo;
3. Criar uma função do tipo PIPELINED que retorna a coleção criada no segundo passo.

Vamos usar o nosso exemplo como períodos para exemplificar esse processo:

1. Criação do objeto armazenado no banco para possibilitar registro de mês e ano:

```
CREATE OR REPLACE TYPE ObjPeriodo AS OBJECT (  
    mes INTEGER,  
    ano INTEGER  
);  
/
```

2. Criação da coleção de períodos (mês/ano):

```
CREATE OR REPLACE TYPE TPeriodo AS TABLE OF ObjPeriodo;  
/
```

3. Criação da função PIPELINED que retorna os períodos desejados:

```
CREATE OR REPLACE PACKAGE pck_util IS  
    --Declaração da sua função  
    FUNCTION fnPeriodo(pDataInicial DATE, pDataFinal DATE)  
        RETURN TPeriodo PARALLEL_ENABLE PIPELINED;  
  
END pck_util;  
/  
  
CREATE OR REPLACE PACKAGE BODY pck_util IS  
    -- Função que retorna coleção de período  
    FUNCTION fnPeriodo(pDataInicial DATE, pDataFinal DATE)  
    RETURN TPeriodo PARALLEL_ENABLE PIPELINED IS  
        -- Declaração de variáveis  
        vQtdeMeses INTEGER := 0;  
        vDataInicial DATE := pDataInicial;  
        vDataFinal DATE := pDataFinal;  
        vDataAtual DATE := add_months(pDataInicial,-1);  
        vResult ObjPeriodo := ObjPeriodo(NULL,NULL);  
    BEGIN  
        -- Acha a quantidade de meses entre as duas datas,  
        -- incluindo 1 para considerar o último mês  
        vQtdeMeses := months_between(vDataFinal,vDataInicial);  
  
        -- Inicia uma loop que vai do mês/ano inicial até o mês/ano final  
        FOR i IN 1..vQtdeMeses LOOP  
            -- Incrementa um mês na data atual  
            vDataAtual := add_months(vDataAtual,1);  
  
            -- Defini os valores do registro atual  
            vResult.Mes := to_number(to_char(vDataAtual,'MM'));  
            vResult.Ano := to_number(to_char(vDataAtual,'YYYY'));  
  
            -- Adiciona registro na coleção  
            PIPE ROW (vResult);  
        END LOOP;  
  
        RETURN;  
    END fnPeriodo;
```

```
END pck_util;  
/
```

Neste exemplo, primeiramente, foi criado um tipo de objeto chamado `ObjPeriodo` que contém dois atributos: mês e ano. Depois, criamos uma coleção chamada `TPeriodo`, que conterá linhas do tipo `ObjPeriodo`.

Após criar o objeto e a coleção, foi criado um pacote chamado `pck_util`, que contém a função `fnPeriodo`, que como podemos observar, retorna um objeto `TPeriodo` e recebe como parâmetros uma data do período inicial e uma data do período final.

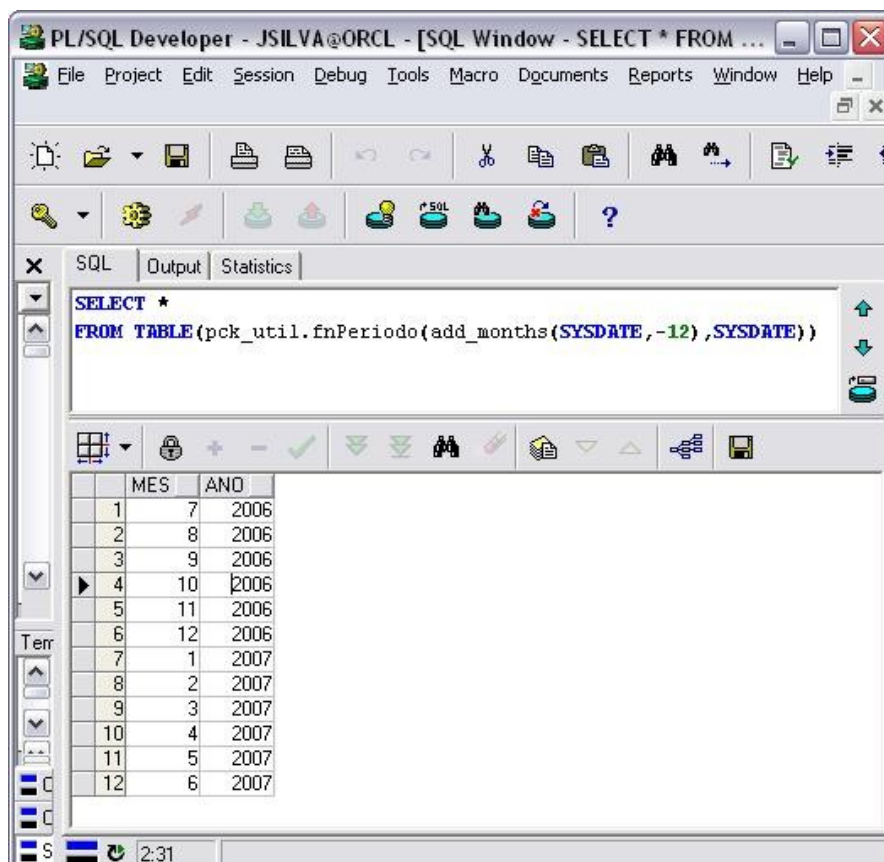
A cláusula `PIPELINED` da função é o que permite a utilização do comando `PIPE ROW` que é o responsável por incluir o registro na coleção que será retornada. `PARALLEL_ENABLE` permite que a função seja executada em sessões filhas de operações paralelas, portanto, é opcional mas necessária se você utiliza `PARALLEL QUERY`.

Utilizando uma função `PIPELINED` em uma declaração SQL

Para utilizar uma função `PIPELINED` em uma instrução SQL é tão simples quanto usar as tabelas do banco de dados. A única diferença, é que precisamos fazer um *typecasting* da função, como no exemplo abaixo:

```
SELECT *  
FROM TABLE(pck_util.fnPeriodo(add_months(SYSDATE,-12),SYSDATE))
```

Neste exemplo, temos como retorno os últimos 12 períodos mensais:



	MES	ANO
1	7	2006
2	8	2006
3	9	2006
4	10	2006
5	11	2006
6	12	2006
7	1	2007
8	2	2007
9	3	2007
10	4	2007
11	5	2007
12	6	2007

Figura 22: Resultado do uso de uma função PIPELINED

Mas como podemos utilizar esse recurso para resolver o nosso problema inicial, onde precisávamos recuperar o valor total de venda para cada mês no período de 01/2006 a 12/2006?

A nossa declaração `SELECT` que dependia da tabela `periodo_resumo`, não precisa mais, pois podemos reescrevê-la da seguinte forma:

```
SELECT p.mes,
       p.ano,
       SUM(nvl(r.rmv_vl_total,0)) rmv_vl_total
FROM TABLE(pck_util.fnPeriodo(to_date('01/01/2006','dd/mm/yyyy'),
                                   to_date('01/12/2006','dd/mm/yyyy')
                                   )
       ) p,
       resumo_mensal_venda r
WHERE p.mes BETWEEN 1 AND 12
AND p.ano = 2006
AND p.mes = r.rmv_in_mes(+)
AND p.ano = r.rmv_in_ano(+)
GROUP BY p.mes,
         p.ano
ORDER BY p.mes,
         p.ano;
```

Ou se preferir:

```
SELECT p.mes,
       p.ano,
       (SELECT SUM(nvl(r.rmv_vl_total,0))
        FROM resumo_mensal_venda r
        WHERE p.mes = r.rmv_in_mes AND p.ano = r.rmv_in_ano
       ) rmv_vl_total
FROM TABLE(pck_util.fnPeriodo(to_date('01/01/2006','dd/mm/yyyy'),
                                   to_date('01/12/2006','dd/mm/yyyy')
                                   )
       ) p
```

UTL_FILE: Escrita e leitura de arquivos no servidor

A linguagem PL/SQL em si não possui mecanismo para executar saída para arquivo ou tela. Entretanto, a Oracle fornece alguns pacotes incorporados que permitem executar a E/S e que podem ser chamados da PL/SQL.

O pacote `UTL_FILE` é o mais conhecido e mais simples de se utilizar e veremos a seguir como extrair informações do banco de dados e gerar um arquivo com esses dados no sistema operacional do servidor, bem como ler um arquivo do sistema operacional e manipular seus dados.

Existem dois pré-requisitos para usar o pacote `UTL_FILE`:

- Você deve ter privilégios de executar o pacote;
- Seu administrador de banco de dados deve definir um parâmetro de inicialização de banco de dados chamado `UTL_FILE_DIR` ou criar um `DIRECTORY` no banco de dados;

Usando UTL_FILE, o processo para leitura ou gravação de um arquivo é o seguinte:

1. Declarar uma variável de handle de arquivo para usar na identificação do arquivo quando você fizer chamadas para as diversas rotinas do UTL_FILE.
2. Declarar uma string do tipo VARCHAR2 para agir como um buffer para ler o arquivo uma linha de cada vez;
3. Abrir o arquivo, especificando se fará leitura ou escrita;
4. Manipular a linha, seja uma leitura ou uma escrita (existem sub-rotinas específicas de UTL_FILE para cada caso);
5. Fechar o arquivo.

Procedimentos e funções de UTL_FILE

Na tabela 8 estão os procedimentos e funções que encontramos no pacote UTL_FILE.

Tabela 8: Procedimentos e funções de UTL_FILE

<i>Procedure / Function</i>	<i>Descrição</i>
FCLOSE	Fecha um arquivo.
Exceções levantadas	
Exceção	Descrição
UTL_FILE.INVALID_FILEHANDLE	Você passou um handle de arquivo que não representava um arquivo aberto.
UTL_FILE.WRITE_ERROR	O sistema operacional não pode gravar no arquivo.
UTL_FILE.INTERNAL_ERROR	Um erro interno ocorreu.
FCLOSE_ALL	Fecha todos os arquivos.
Exceções levantadas	
As mesmas da função FCLOSE.	
FFLUSH	Descarrega todos os dados em <i>buffer</i> para serem gravados em disco imediatamente.
Exceções levantadas	
Exceção	Descrição
UTL_FILE.INVALID_FILEHANDLE	Você usou um handle de arquivo inválido. Provavelmente você esqueceu de abrir o arquivo.
UTL_FILE.INVALID_OPERATION	Você tentou gravar em um arquivo que não estava aberto para gravação (modos W e A).
UTL_FILE.WRITE_ERROR	Ocorreu um erro de sistema operacional, tal como um erro de disco cheio, ao tentar gravar em um arquivo
UTL_FILE.INTERNAL_ERROR	Ocorreu um erro interno.
FOPEN	Abre um arquivo.
Exceções levantadas	
Exceção	Descrição
UTL_FILE.INVALID_PATH	O diretório não é válido
UTL_FILE.INVALID_MODE	Um modo inválido foi especificado. O modem open pode ser R (ler), W

Procedure / Function	Descrição
	(escrever) ou A (para anexar a um arquivo existente).
UTL_FILE.INVALID_OPERATION	O arquivo não pode ser aberto por algum outro motivo, como por exemplo, falta de permissão de acesso do proprietário do software do Oracle. Na maioria das vezes, é problema de permissão de acesso no sistema operacional.
UTL_FILE.INTERNAL_ERROR	Um erro interno ocorreu.

GET_LINE Lê uma linha de um arquivo e avança para a próxima.

Exceções levantadas

Exceção	Descrição
UTL_FILE.INVALID_FILEHANDLE	Você passou um handle de arquivo inválido. Possivelmente, você esqueceu de abrir o arquivo primeiro.
UTL_FILE.INVALID_OPERATION	O arquivo não está aberto para leitura (modo R) ou existem problemas com as permissões do arquivo.
UTL_FILE.VALUE_ERROR	O buffer não é suficientemente longo para conter a linha que está sendo lida do arquivo. O tamanho do buffer é aumentado.
UTL_FILE.NO_DATA_FOUND	O final do arquivo foi atingido.
UTL_FILE.INTERNAL_ERROR	Ocorreu um erro interno do sistema UTL_FILE.
UTL_FILE.READ_ERROR	Ocorreu um erro do sistema operacional durante a leitura do arquivo.

IS_OPEN Verifica se um arquivo está aberto.

NEW_LINE Grava um caractere *newline* em um arquivo.

Exceções levantadas

As mesmas da função FFLUSH.

PUT Grava uma *string* de caracteres em um arquivo, mas não coloca uma *newline* depois dela.

Exceções levantadas

As mesmas da função FFLUSH.

PUT_LINE Grava uma linha em um arquivo.

Exceções levantadas

As mesmas da função FFLUSH.

PUTF Formata e grava saída. Essa é uma imitação bruta do procedimento printf() da linguagem C.

Exceções levantadas

As mesmas da função FFLUSH.

Gerando um arquivo com informações extraídas do banco de dados

Vamos criar na *package* pck_vendas, uma *procedure* que recupera o cadastro de produto do banco de dados e gerar um arquivo no servidor no formato CSV. Abaixo segue o código da nova procedure:

```
/* Procedure que exporta o cadastro de produtos para um arquivo CSV*/
PROCEDURE ExpProdutoCSV IS
    vlFile utl_file.file_type;

    CURSOR cs_produto IS
        SELECT p.pro_in_codigo
            ,p.pro_st_nome
            ,p.pro_st_marca
            ,(SELECT pp.ppr_vl_unitario
              FROM preco_produto pp
              WHERE pp.pro_in_codigo = p.pro_in_codigo
              AND pp.ppr_st_tipopreco = 'VENDAS'
            ) vl_preco_venda
            ,(SELECT pp.ppr_vl_unitario
              FROM preco_produto pp
              WHERE pp.pro_in_codigo = p.pro_in_codigo
              AND pp.ppr_st_tipopreco = 'COMPRAS'
            ) vl_preco_compra
        FROM produto p;
BEGIN
    -- Abrir arquivo
    vlFile := utl_file.fopen('INTERFACE', 'CadProd-' || USER || '.csv', 'W');

    -- Gravar cabeçalho
    utl_file.put_line(vlFile
        , '"Codigo", '
        || '"Nome", '
        || '"Marca", '
        || '"Preco de Venda", '
        || '"Preco de Compra"');

    -- Gravar produtos no arquivo
    FOR csp IN cs_produto LOOP
        utl_file.put_line(vlFile
            , '"' || csp.pro_in_codigo || '"', '
            || '"' || csp.pro_st_nome || '"', '
            || '"' || csp.pro_st_marca || '"', '
            || '"' || csp.vl_preco_venda || '"', '
            || '"' || csp.vl_preco_compra || '"');

    END LOOP;

    -- Fechar arquivo
    utl_file.fclose(vlFile);
END ExpProdutoCSV;
```

IMPORTANTE:

- Não esqueça de declarar a procedure na especificação da *package*;
- Certifique-se que tem acesso e as permissões necessárias ao diretório informado (INTERFACE, em nosso exemplo).

- Embora o nosso exemplo não implemente o tratamento de exceções, uma *procedure* para geração de arquivos bem codificada, deve conter esse tratamento. Para isso, veja a tabela 8, onde as possíveis exceções de cada sub-rotina de `UTL_FILE`, estão listadas.

Agora, execute a nova *procedure* e confira o arquivo gerado no diretório que você especificou.

Recuperando informações de um arquivo

Para testarmos a leitura de arquivos com `UTL_FILE`, implementaremos mais uma *procedure* na *package* `pck_vendas`. Essa *procedure* abrirá o arquivo que geramos no exemplo anterior e exibirá cada linha através do pacote `DBMS_OUTPUT`. Abaixo segue o código:

```
/* Procedure que importa o cadastro de produto de um arquivo CSV */
PROCEDURE ImpProdutoCSV IS
    vlFile    utl_file.file_type;
    vlLinha   VARCHAR2(1024);
BEGIN
    -- Abrir arquivo
    vlFile := utl_file.fopen('INTERFACE', 'CadProd-' || USER || '.csv', 'R');

    -- Recuperar cada linha do arquivo e exibir linha através do dbms_output
    LOOP
        BEGIN
            utl_file.get_line(vlFile, vlLinha);
            dbms_output.put_line(vlLinha);
        EXCEPTION
            WHEN no_data_found THEN
                EXIT;
        END;
    END LOOP;

    -- Fechar arquivo
    utl_file.fclose(vlFile);
END ImpProdutoCSV;
```

IMPORTANTE:

- Certifique-se que o arquivo a ser lido está no diretório utilizado na *procedure* (`INTERFACE`, em nosso exemplo);
- Não esqueça de declarar a *procedure* na especificação da *package*;
- Para executar a *procedure*, ligue a exibição de mensagens em sua sessão de SQL: `SET SERVEROUTPUT ON`;

Agora é só executar e ver o resultado na tela.

TEXT_IO

Embora não faça parte do escopo deste treinamento, vale citar a existência do pacote `TEXT_IO`.

Esse pacote é semelhante ao `UTL_FILE`, mas ele permite que você leia e grave arquivos no cliente em vez do servidor. `TEXT_IO` não faz parte do software do banco de dados da Oracle. Ele vem com o Oracle Developer (Developer 2000).

Exercícios Propostos

1. Crie um pacote chamado `pck_vendas` que contenha:
 - Uma função `PIPELINED` que receba um intervalo de datas e baseado nessas datas, monte uma *tabela virtual* que repita todos os períodos mensais para cada representante cadastrado. Nomeie essa função como `fnMensalPorRepresentante`; Utilize essa função em uma declaração `SELECT` que retorne a venda total mensal por representante em um intervalo de datas;
 - Uma função `PIPELINED` que receba um intervalo de datas e baseado nessas datas, monte uma *tabela virtual* que repita todos os períodos mensais para cada cliente cadastrado. Nomeie essa função como `fnMensalPorCliente`; Utilize essa função em uma declaração `SELECT` que retorne a venda total mensal por cliente em um intervalo de datas;
 - Uma função `PIPELINED` que receba um intervalo de datas e baseado nessas datas, monte uma *tabela virtual* que repita todos os períodos mensais para cada produto cadastrado. Nomeie essa função como `fnMensalPorProduto`; Utilize essa função em uma declaração `SELECT` que retorne a venda total mensal por produto em um intervalo de datas;
 - Uma função `PIPELINED` que receba um intervalo de datas e baseado nessas datas, monte uma *tabela virtual* que repita todos os períodos mensais para cada Cliente/Produto cadastrados. A idéia é obter quanto cada cliente comprou de cada produto em cada período. Nomeie essa função como `fnMensalClienteProduto`; Utilize essa função em uma declaração `SELECT` que retorne a venda total mensal por produto em um intervalo de datas;

06 – Tópicos avançados: SQL e funções incorporadas do Oracle Database

1. **SQLs avançados para usar com PL/SQL:**
 - *O Outer Join* do Oracle
 - ROWNUM
 - Comando CASE no SELECT
 - SELECT combinado com CREATE TABLE e INSERT
 - MERGE
 - GROUP BY com HAVING
 - Recursos avançados de agrupamento: ROLLUP e CUBE
 - Consultas hierárquicas com CONNECT BY

2. **Funções úteis do Oracle Database**

SQLs avançados para usar com PL/SQL

O *outer join* do Oracle

Um *OUTER JOIN* é um *join* simples (chamado de *INNER JOIN*) com uma "opção estendida" de trazer também os dados que não satisfazem a condição do *join*, ou seja, em um *join* entre duas tabelas, você tem a opção de trazer os dados, mesmo que não exista um registro correspondente em uma das tabelas.

Para exemplificar, vejamos a seguinte situação: Elaborar uma SQL que traga os produtos da marca "Brahma" com o resumo mensal de quantidade vendida (ano e mês), existindo ou não o resumo de vendas do produto. A SQL abaixo, que está no padrão ANSI92 (também chamada de SQL92), nos dá essa solução:

```
SELECT p.pro_in_codigo
      ,p.pro_st_nome
      ,rmv.rmv_in_ano
      ,rmv.rmv_in_mes
      ,sum(rmv.rmv_qt_vendida) qt_vendida
FROM produto p LEFT OUTER JOIN resumo_mensal_venda rmv
      ON (p.pro_in_codigo = rmv.pro_in_codigo)
WHERE p.pro_st_marca = 'Brahma'
GROUP BY p.pro_in_codigo
      ,p.pro_st_nome
      ,p.pro_st_marca
      ,rmv.rmv_in_ano
      ,rmv.rmv_in_mes
ORDER BY p.pro_in_codigo
      ,rmv.rmv_in_ano DESC
      ,rmv.rmv_in_mes DESC;
```

O *LEFT* indica que a tabela da esquerda (produto) é a base para o *OUTER JOIN*, ou seja, todos os registros da tabela *PRODUTO* que atenderem a cláusula *WHERE* devem ser retornados, mesmo que não exista resumo de vendas correspondente.

Até aí, tudo bem! Esse é padrão de *outer join* definido pela ANSI92 e, portanto, vale para qualquer banco de dados.

O problema é que esse padrão veio depois que os SGBDs do mercado já haviam implementado suas próprias soluções para os *outer joins*. No SQL Server, por exemplo, para montar o *outer join*, bastava incluir o caracter "*" junto do operador "=" do lado da tabela base da consulta, ou seja, o nosso exemplo teria as cláusulas *FROM* e *WHERE* um pouco diferente:

```
FROM produto p, resumo_mensal_venda rmv
WHERE p.pro_in_codigo *= rmv.pro_in_codigo
AND p.pro_st_marca = 'Brahma'
```

A leitura na implementação do SQL Server é bem próxima do padrão ANSI, pois o "*" está do lado da tabela base que no nosso exemplo é *LEFT*.

No entanto o padrão ANSI92 pode gerar algumas confusões para quem se habituou com a implementação de *outer join* da Oracle.

Diferente do seu concorrente, a implementação da Oracle indica a tabela que pode não haver dados correspondentes, ao invés da tabela base.

Essa indicação é feita pelo operador "(+)" e é adicionado na associação das tabelas, mas à direita da coluna da tabela que pode não ter registros, como no exemplo abaixo:

```
SELECT p.pro_in_codigo
      ,p.pro_st_nome
      ,rmv.rmv_in_ano
      ,rmv.rmv_in_mes
      ,sum(rmv.rmv_qt_vendida) qt_vendida
FROM produto p, resumo_mensal_venda rmv
WHERE p.pro_in_codigo = rmv.pro_in_codigo(+)
AND p.pro_st_marca = 'Brahma'
GROUP BY p.pro_in_codigo
      ,p.pro_st_nome
      ,p.pro_st_marca
      ,rmv.rmv_in_ano
      ,rmv.rmv_in_mes
ORDER BY p.pro_in_codigo
      ,rmv.rmv_in_ano DESC
      ,rmv.rmv_in_mes DESC;
```

Entre os profissionais Oracle, a implementação da Oracle é a mais usada porque:

1. O padrão veio depois e implementou uma lógica “contrária” a já praticada;
2. A sinalização do *outer join* usando apenas “(+)” ao invés de um texto em língua inglesa, gera SQLs mais limpos;
3. A popularidade do banco de dados Oracle leva esses profissionais a ignorar esse padrão.

Dicas:

- Se pretende trabalhar com diversos bancos de dados, utilize o padrão ANSI92;
 - Se trabalhará exclusivamente com banco de dados Oracle, siga a implementação Oracle, pois do contrário terá problemas com outros profissionais dedicados a Oracle;
 - Se está desenvolvendo ou evoluindo sistemas de um cliente que já possui implementações em produção, verifique qual padrão utilizam e dê sequência.
-

ROWNUM

Existem situações onde você executa um SELECT que retorna *n* registros, mas precisa apenas de alguns desses registros, não importando qual ou ainda incluir no retorno um número sequencial para cada linha retornada.

O Oracle Database disponibiliza a “pseudo coluna” **ROWNUM** que pode ser utilizada nesses casos. Veja os exemplos abaixo:

ROWNUM sequenciando as linhas retornadas

```
SELECT ROWNUM
      ,pp.*
FROM preco_produto pp;
```

ROWNUM limitando a quantidade de linhas retornadas: O exemplo abaixo retorna as 10 maiores vendas mensais

```
SELECT ROWNUM
      ,v.*
```

```

FROM( SELECT p.pro_in_codigo
        ,p.pro_st_nome
        ,p.pro_st_marca
        ,c.cli_st_nome
        ,rmv.rmv_in_ano
        ,rmv.rmv_in_mes
        ,rmv.rmv_qt_vendida
        ,rmv.rmv_vl_total
      FROM produto p
        ,resumo_mensal_venda rmv
        ,cliente c
     WHERE p.pro_in_codigo = rmv.pro_in_codigo
     AND rmv.cli_in_codigo = c.cli_in_codigo
     ORDER BY rmv.rmv_qt_vendida DESC
              ,rmv.rmv_in_ano DESC
              ,rmv.rmv_in_mes DESC
              ,c.cli_st_nome
        ) v
WHERE ROWNUM <= 10;

```

Comando CASE no SELECT

Por muito tempo, o comando `CASE` foi exclusividade da linguagem PL/SQL e para reproduzi-lo em SQLs utilizamos a função `decode()`, mas nas últimas versões do Oracle Database, passamos a contar com esse comando também nas SQLs.

O seu uso em SQLs é simples e recomendado, pois é melhor para o desempenho do SQL do que a função `decode` (embora seja um ganho mínimo).

Abaixo segue um exemplo de uso do `CASE` em SQLs:

```

SELECT nfv.nfv_in_numero
      ,nfv.nfv_dt_emissao
      ,r.rep_st_nome
      ,c.cli_st_nome
      ,c.cli_st_cidade
      ,(CASE
        WHEN nfv.nfv_st_condicaopagamento = 'AVISTA' THEN
          'A Vista'
        WHEN nfv.nfv_st_condicaopagamento LIKE '%COMJUROS' THEN
          'Parcelado com juros'
        WHEN nfv.nfv_st_condicaopagamento LIKE '%SEMJUROS' THEN
          'Parcelado sem juros'
        END
      ) nfv_st_condicaopagamento
FROM nota_fiscal_venda nfv
   ,cliente c
   ,representante r
WHERE nfv.cli_in_codigo = c.cli_in_codigo
AND nfv.rep_in_codigo = r.rep_in_codigo;

```

SELECT combinado com CREATE TABLE e INSERT

CREATE TABLE AS SELECT

É possível criar uma tabela a partir de outra, combinando os comandos `CREATE TABLE` e `SELECT`.

Essa combinação é um comando DDL (Definition Data Language) e como tal só pode ser executado em um bloco PL/SQL através de SQL Dinâmico.

Não é uma prática recomendada, mas pode ser útil em alguma situação onde você precise, por exemplo, criar um *backup* de uma tabela antes de executar o processamento.

Outro uso bem comum é quando você precisa criar uma cópia de uma tabela sem dados, como no exemplo abaixo:

```
CREATE TABLE simula_novo_preco_produto AS
SELECT *
FROM preco_produto
WHERE 1=2;
```

IMPORTANTE: Para executar esse comando, precisará dos privilégios de criação de tabela.

INSERT ... SELECT

Se você precisa executar uma sequência de `INSERTs`, originados de uma mesma fonte, sem a necessidade de tratar cada linha inclusa, você pode utilizar a combinação dos comandos `INSERT` e `SELECT`.

Com essa combinação, em um único comando, você conseguirá executar um `INSERT` para todos os registros retornados pelo `SELECT`, como no exemplo abaixo:

```
INSERT INTO simula_novo_preco_produto
(pro_in_codigo, ppr_st_tipopreco, ppr_vl_unitario)
SELECT pp.pro_in_codigo
      , pp.ppr_st_tipopreco
      , decode(pp.ppr_st_tipopreco
              , 'COMPRAS' , pp.ppr_vl_unitario
              , 'VENDAS'  , pp.ppr_vl_unitario + (pp.ppr_vl_unitario * 0.10))
FROM preco_produto pp;
COMMIT;
```

IMPORTANTE: Uma execução envolvendo um volume muito alto de dados pode resultar em um problema de desempenho e a solução pode não ser a ideal. Avalie com responsabilidade o uso desse recurso.

MERGE

Esta declaração SQL permite que você obtenha dados a partir de uma fonte (tabela, view ou consulta SQL) para manipulação em outra tabela, combinar várias operações DML em um único comando, (como por exemplo, um `UPDATE` e um `INSERT`).

Imagine uma situação onde você executa uma consulta em uma tabela e, a partir dos dados obtidos precise atualizar um registro em outra tabela, mas se o registro não existir, você o incluirá... Vamos criar essa situação. Inclua novos registros na tabela `PRODUTO`, conforme SQL abaixo:

```
INSERT INTO produto
(pro_st_nome, pro_in_codigo, pro_st_marca, pro_dt_inclusao
```

```

,pro_st_usuarioinclusao,pro_dt_ultimavenda,pro_vl_ultimavenda
,pro_dt_maiorvenda)
VALUES
  ('Cerveja Malzbier',11,'Brahma',SYSDATE
  ,USER,NULL,NULL,NULL)
/
INSERT INTO produto
  (pro_st_nome,pro_in_codigo,pro_st_marca,pro_dt_inclusao
  ,pro_st_usuarioinclusao,pro_dt_ultimavenda,pro_vl_ultimavenda
  ,pro_dt_maiorvenda)
VALUES
  ('Cerveja Malzbier',12,'Schincariol',SYSDATE
  ,USER,NULL,NULL,NULL)
/
COMMIT
/

```

Agora, precisamos atualizar o preço de venda dos produtos da marca “Brahma” para R\$ 1,22, mas se não existir preço cadastrado para o produto, incluiremos.

O código abaixo resolveria isso:

```

DECLARE

eRegistroInexistente EXCEPTION;

CURSOR cs_produto IS
  SELECT p.pro_in_codigo
  FROM produto p
  WHERE p.pro_st_marca = 'Brahma';

vNovoValorVenda CONSTANT preco_produto.ppr_vl_unitario%TYPE := 1.22;

BEGIN
  FOR csp IN cs_produto LOOP

    BEGIN
      UPDATE preco_produto pp
      SET pp.ppr_vl_unitario = vNovoValorVenda
      WHERE pp.pro_in_codigo = csp.pro_in_codigo
      AND pp.ppr_st_tipopreco = 'VENDAS';

      IF SQL%NOTFOUND THEN
        RAISE eRegistroInexistente;
      END IF;

    EXCEPTION
      WHEN eRegistroInexistente THEN
        INSERT INTO preco_produto (pro_in_codigo,ppr_st_tipopreco,ppr_vl_unitario)
        VALUES (csp.pro_in_codigo,'VENDAS',vNovoValorVenda);
      WHEN OTHERS THEN
        raise_application_error
          (-20100, 'Não foi possível atualizar o preço do produto '
          ||to_char(csp.pro_in_codigo));

    END;

  END LOOP;

  COMMIT;
END;

```

/

Mas é possível executar a mesma alteração utilizando apenas o comando MERGE.

No exemplo abaixo repetimos a operação para alterar o preço de venda dos produtos da marca “Schincariol” (para R\$ 1.12) utilizando esse comando:

```
DECLARE
    vNovoValorVenda CONSTANT preco_produto.ppr_vl_unitario%TYPE := 1.12;
BEGIN
    MERGE INTO preco_produto pp
    USING (SELECT p.pro_in_codigo
          FROM produto p
          WHERE p.pro_st_marca = 'Schincariol') p
    ON (pp.pro_in_codigo = p.pro_in_codigo)
    WHEN MATCHED THEN
        UPDATE SET pp.ppr_vl_unitario = vNovoValorVenda
        WHERE pp.ppr_st_tipopreco = 'VENDAS'
    WHEN NOT MATCHED THEN
        INSERT (pro_in_codigo, ppr_st_tipopreco, ppr_vl_unitario)
        VALUES (p.pro_in_codigo, 'VENDAS', vNovoValorVenda);

    COMMIT;
END;
/
```

GROUP BY com HAVING

Você já deve saber que o GROUP BY é utilizado para aplicar funções de grupo e dessa forma obter totais, médias, valores máximos e mínimos.

Existem situações onde você precisa aplicar uma restrição de retorno no valor agrupado (à soma ou à média, por exemplo).

Se você tentar aplicar essa restrição na cláusula WHERE receberá um erro, pois nela as restrições são em nível de linha e não de valores agrupados.

Se você quiser recuperar do banco de dados o total de vendas do ano por marca de produto, não poderá usar a SQL abaixo (tente):

```
SELECT p.pro_st_marca
       ,rmv.rmv_in_ano
       ,sum(rmv.rmv_qt_vendida) rmv_qt_vendida
       ,sum(rmv.rmv_vl_total) rmv_vl_total
FROM produto p
     ,resumo_mensal_venda rmv
WHERE p.pro_in_codigo = rmv.pro_in_codigo
AND sum(rmv.rmv_vl_total) > 50000
GROUP BY p.pro_st_marca
       ,rmv.rmv_in_ano
```

Se tentou, deve ter recebido o erro “ORA-00934: a função de grupo não é permitida aqui” do Oracle.

Para aplicar uma restrição em valor agrupado, você deve utilizar a cláusula HAVING após o conjunto

do GROUP BY. Ele funciona como uma cláusula WHERE, mas em nível de grupo. Teste o exemplo abaixo:

```
SELECT p.pro_st_marca
      ,rmv.rmv_in_ano
      ,sum(rmv.rmv_qt_vendida) rmv_qt_vendida
      ,sum(rmv.rmv_vl_total) rmv_vl_total
FROM produto p
      ,resumo_mensal_venda rmv
WHERE p.pro_in_codigo = rmv.pro_in_codigo
GROUP BY p.pro_st_marca
      ,rmv.rmv_in_ano
HAVING sum(rmv.rmv_vl_total) > 50000;
```

Em nosso exemplo restringimos a soma do valor total (SUM()), mas poderia ser qualquer outra função de grupo, como AVG() ou MAX(), por exemplo.

Recursos avançados de agrupamento: ROLLUP e CUBE

ROLLUP e CUBE são extensões do GROUP BY, normalmente utilizadas em sistemas de apoio a decisão, como *Datawarehouse*.

A seguir veremos como utilizar cada um deles.

ROLLUP

Com o ROLLUP você pode criar sub-totais com base nos valores de grupo.

Vamos alterar o nosso exemplo anterior (do GROUP BY com HAVING) para obtermos sub-totais por mês e ano:

```
SELECT rmv.rmv_in_ano
      ,rmv.rmv_in_mes
      ,p.pro_st_marca
      ,sum(rmv.rmv_qt_vendida) rmv_qt_vendida
      ,sum(rmv.rmv_vl_total) rmv_vl_total
FROM produto p
      ,resumo_mensal_venda rmv
WHERE p.pro_in_codigo = rmv.pro_in_codigo
GROUP BY ROLLUP (rmv.rmv_in_ano
                ,rmv.rmv_in_mes
                ,p.pro_st_marca);
```

Observe que além dos sub-totais, um total geral foi acrescentado no final. Caso não queira o total geral, deixe a coluna que representa o primeiro nível (no caso `rmv_in_ano`) fora do ROLLUP (logo após a palavra-chave GROUP BY).

CUBE

Outro recurso avançado para manipular resultados agrupados e que também é comum para sistemas de apoio a decisão é o CUBE.

O CUBE funciona como o ROLLUP, mas o resultado é mais detalhado. Ele apresenta totais para todas as combinações de totais do seu GROUP BY. Altere sua consulta para aplicar o CUBE ao invés do ROLLUP e observe o resultado:

```
SELECT rmv.rmv_in_ano
      ,rmv.rmv_in_mes
      ,p.pro_st_marca
      ,sum(rmv.rmv_qt_vendida) rmv_qt_vendida
      ,sum(rmv.rmv_vl_total) rmv_vl_total
FROM produto p
      ,resumo_mensal_venda rmv
WHERE p.pro_in_codigo = rmv.pro_in_codigo
GROUP BY CUBE (rmv.rmv_in_ano
              ,rmv.rmv_in_mes
              ,p.pro_st_marca);
```

SUGESTÃO DE PESQUISA

Existem muitos outros recursos que facilitam a manipulação de dados agrupados garantindo um bom desempenho, entre eles as **funções analíticas**. Pesquise sobre esse recurso e faça seus testes.

Consultas hierárquicas com CONNECT BY

O `CONNECT BY` é um recurso do Oracle para recuperarmos uma consulta em ordem hierárquica, como por exemplo, em um cadastro de funcionários onde um funcionário pode ser gerente de outro.

Para estudarmos esse recurso, vamos alterar a nossa tabela de representante para informar o gerente de cada representante.

Abaixo segue o *script* para preparar a tabela:

```
-- incluir coluna
ALTER TABLE representante
  ADD (rep_in_codigo_gerente INTEGER)
/
-- registrar o gerente de cada representante
UPDATE representante r
SET r.rep_in_codigo_gerente = 40
WHERE r.rep_in_codigo = 10
/
UPDATE representante r
SET r.rep_in_codigo_gerente = 50
WHERE r.rep_in_codigo IN (20,30)
/
COMMIT
/
```

Com a execução desse *script*, temos dois gerentes: 40 e 50, sendo que o 40 é gerente do representante 10 e o 50 é gerente dos representantes 20 e 30.

Agora, queremos listar os representantes em ordem hierárquica, ou seja, o gerente e em seguida seus subordinados. A SQL abaixo utiliza o `CONNECT BY` para isso:

```
SELECT r.rep_in_codigo      "Codigo"
      ,r.rep_st_nome        "Nome"
      ,r.rep_in_codigo_gerente "Cod.Gerente"
```

```

        ,r.rep_vl_metamensal      "Meta Mensal Vendas"
        ,r.rep_vl_mediamensalvendas "Media Mensal Vendas"
        ,r.rep_dt_maiorvenda      "Valor Maior Venda"
        ,r.rep_dt_maiorvenda      "Data Maior Venda"
FROM representante r
START WITH r.rep_in_codigo_gerente IS NULL
CONNECT BY PRIOR rep_in_codigo = rep_in_codigo_gerente;

```

Entendendo a SQL:

- Na cláusula **START WITH** você defini o início de sua hierarquia, que em nosso caso são os representantes que estão no “topo da pirâmide”, ou seja, não possuem gerente.
- Na cláusula **CONNECT BY PRIOR** você indica a associação do nível mais baixo com o mais alto. Se traduzíssemos ao pé da letra, seria algo como “Conecte este representante abaixo do representante indicado como gerente”.

Com o **CONNECT BY** você também pode visualizar em que nível da hierarquia cada registro se encontra. Para isso, existe uma “pseudo coluna” chamada **LEVEL**.

Você pode utilizar **LEVEL** na lista de colunas:

```

SELECT LEVEL
        ,r.rep_in_codigo      "Codigo"
        ,r.rep_st_nome        "Nome"
        ,r.rep_in_codigo_gerente "Cod.Gerente"
        ,r.rep_vl_metamensal   "Meta Mensal Vendas"
        ,r.rep_vl_mediamensalvendas "Media Mensal Vendas"
        ,r.rep_dt_maiorvenda   "Valor Maior Venda"
        ,r.rep_dt_maiorvenda   "Data Maior Venda"
FROM representante r
START WITH r.rep_in_codigo_gerente IS NULL
CONNECT BY PRIOR rep_in_codigo = rep_in_codigo_gerente;

```

LEVEL também pode ser utilizado como restrição da consulta. Veja como fica nossa consulta se quisermos apenas visualizar os gerentes:

```

SELECT LEVEL
        ,r.rep_in_codigo      "Codigo"
        ,r.rep_st_nome        "Nome"
        ,r.rep_in_codigo_gerente "Cod.Gerente"
        ,r.rep_vl_metamensal   "Meta Mensal Vendas"
        ,r.rep_vl_mediamensalvendas "Media Mensal Vendas"
        ,r.rep_dt_maiorvenda   "Valor Maior Venda"
        ,r.rep_dt_maiorvenda   "Data Maior Venda"
FROM representante r
WHERE LEVEL = 1
START WITH r.rep_in_codigo_gerente IS NULL
CONNECT BY PRIOR rep_in_codigo = rep_in_codigo_gerente;

```

Funções incorporadas do Oracle Database

Existem muitas funções incorporadas do Oracle Database que podem otimizar o seu trabalho, tanto em tempo de desenvolvimento quanto desempenho do que você produzir.

Em geral essas funções podem ser usadas em códigos PL/SQL e declarações SQL.

É impossível memorizar todas essas funções, mas a seguir, listamos algumas das mais comuns e

mais úteis:

Tabela 9: Funções de caractere incorporadas do Oracle

Função	Descrição
CHR	Retorna um caractere quando recebe seu valor ASCII.
ASCII	Retorna o código ASCII do caractere.
INITCAP	Retorna uma <i>string</i> na qual a primeira letra de cada palavra é colocada em maiúscula e todos os caracteres restantes, em minúsculas.
INSTR	Retorna a localização de uma <i>string</i> dentro de outra <i>string</i> .
LENGTH	Retorna o comprimento de uma <i>string</i> de caracteres. Retorna NULL quando o valor é NULL.
LOWER	Converte toda a <i>string</i> de caracteres para minúsculas.
LPAD	Preenche uma <i>string</i> no lado esquerdo com qualquer <i>string</i> especificada.
LTRIM	Corta uma <i>string</i> de caracteres do lado esquerdo.
REPLACE	Substitui toda ocorrência de uma <i>string</i> por outra <i>string</i> .
RPAD	Preenche uma <i>string</i> no lado direito de toda <i>string</i> especificada.
RTRIM	Corta uma <i>string</i> de caracteres no lado direito.
SUBSTR	Retorna uma parte de uma <i>string</i> de dentro de uma <i>string</i> .
TRIM	Combina a funcionalidade das funções LTRIM e RTRIM.
TRANSLATE	Igual a REPLACE, exceto que opera no nível de caractere, em vez de operar no nível de <i>string</i> .
UPPER	Converte toda a <i>string</i> de caracteres para maiúsculas.

Tabela 10: Funções numéricas incorporadas do Oracle

Função	Descrição
ABS	Retorna o valor absoluto de um número
CEIL	Retorna o valor que representa o menor inteiro que é maior do que ou igual a um número especificado. Muito usado para arredondar valores para baixo.
EXP	Retorna a exponenciação de e elevado à potência de algum número onde e= 2,7182818...
LN	Retorna o logaritmo natural de algum número x.
MOD	Retorna o resto de algum número x dividido por algum número y.
ROUND	Retorna x arredondado para y casas.
SQRT	Retorna a raiz quadrada de algum número x. X nunca deve ser negativo.
TRUNC	Retorna algum número x, truncado para y casas. Não arredonda, apenas corta na localização especificada.

Tabela 11: Funções de data incorporadas do Oracle

Função	Descrição
ADD_MONTHS	Adiciona um mês à data especificada. Ela não adiciona 30 ou 31 dias, mas simplesmente adiciona um ao mês. Se o número de meses informado for negativo, subtrai os meses.
LAST_DAY	Retorna o último dia de determinado mês.
MONTHS_BETWEEN	Calcula os meses entre duas datas. Retorna um inteiro quando ambas as datas são os últimos dias do mês. Caso contrário, ela retorna a parte fracionária de um mês de 31 dias.

Função	Descrição
NEXT_DAY	Retorna a data do primeiro dia da semana especificado em uma <i>string</i> após a data inicial.
SYSDATE	Simplesmente retorna a data e hora do sistema no formato tipo DATE.
TRUNC	Trunca até o parâmetro de data especificado, tal como dia, mês e assim por diante. Normalmente utilizado sem parâmetros, trunca na data, retirando hora, minutos e segundos.

Tabela 12: Funções de conversão incorporadas do Oracle

Função	Descrição
TO_CHAR	Converte DATES e NUMBERS em uma <i>string</i> VARCHAR2.
TO_DATE	Converte uma <i>string</i> CHAR ou VARCHAR2 em um valor DATE.
TO_NUMBER	Converte uma <i>string</i> CHAR ou VARCHAR2 em um valor NUMBER.

Tabela 13: Funções diversas incorporadas do Oracle

Função	Descrição
DECODE	Age como uma declaração IF...THEN...ELSE de uma lista de valores.
GRATEST	Toma uma lista de valores ou expressões e retorna o maior valor avaliado.
NVL	Verifica se o valor passado como primeiro parâmetro é nulo e sendo, retorna o segundo parâmetro como substituto. (simula a atribuição de valor default)
USER	Retorna o nome do usuário atual em uma <i>string</i> VARCHAR2.
USERENV	Retorna as informações sobre o seu ambiente de trabalho atual.

Dicas

- Sempre utilize TO_DATE quando estiver referenciando uma data em formato *string*. Um código bem feito segue essa prática informando a data (em formato *string*) e um segundo parâmetro com informando em que formato encontra a data a ser convertida (esse parâmetro também é um *string*);
- Ao referenciar variáveis que não podem conter valor nulo, utilize a função NVL();
- Para exibir uma data armazenada no banco de dados ou retornada por SYSDATE, em um formato específico, utilize a função TO_CHAR;
- Utilize a função CHR() para formatar suas mensagens de sistema com quebra de linha (chr(13));
- Se pretende utilizar um valor alfanumérico em uma expressão matemática, converta seu valor utilizando TO_NUMBER();
- Antes de implementar uma função ou uma lógica para tratar ou converter um dado, pesquise se já não existe o que precisa entre as funções incorporadas do Oracle Database.

07 – Dicas de performance e boas práticas em SQL

1. **Otimizador do Oracle;**
2. **Variáveis de ligação (Bind Variables);**
3. **SQL Dinâmico;**
4. **EXPLAIN PLAN;**
5. **SQLs Complexas;**
6. **SORT (Ordenação nas SQLs);**
7. **etc...**

Introdução

Problemas com aplicações de baixa performance podem estar frequentemente relacionados a consultas SQL mal estruturadas ou a uma modelagem de banco de dados ineficiente.

A metodologia de *tuning* da Oracle, recomenda que, antes de analisar configurações do banco de dados, seja analisadas e ajustadas as instruções SQL que apresentarem desempenho insatisfatório.

A otimização de uma instrução SQL constitui em determinar a melhor estratégia para executá-la no banco de dados. O otimizador do Oracle escolhe, por exemplo, se usará um índice ou não para uma consulta específica e que técnicas de *join* usar na junção de múltiplas tabelas. Estas decisões têm um impacto muito grande na performance de um SQL e por isso a otimização de uma instrução é essencial para qualquer aplicação e de extrema importância para a performance de um banco de dados relacional.

É muito importante que os desenvolvedores conheçam o otimizador do Oracle como também os conceitos básicos relativos à *tuning*. Tal conhecimento irá ajudar a escrever consultas muito mais eficientes e rápidas.

Além de conhecer o otimizador do Oracle, é imprescindível que o desenvolvedor conheça a aplicação e os dados dela. Antes de sair escrevendo uma consulta SQL, procure entender o processo do qual essa instrução fará parte. Qual a finalidade dessa instrução? Informações idênticas podem ser encontradas em diferentes fontes de dados. Se você estiver familiarizado com estas fontes, poderá identificar a fonte que proporcionará uma recuperação mais rápido, ou seja, uma consulta em menor tempo.

O Otimizador Oracle

O otimizador determina a maneira mais eficiente de se rodar uma declaração SQL. Para executar qualquer SQL o Oracle tem que montar um *plano de execução*. O plano de execução de uma consulta é uma descrição de como o Oracle irá implementar a recuperação dos dados para satisfazer a uma determinada declaração SQL.

Até a versão 9i, o Oracle possuía dois otimizadores:

- **RBO** (Ruled Based Optimizer): Otimizador baseado em regra;
- **CBO** (Cost Based Optimizer): Otimizador baseado em custo, que passou a ser o padrão na versão 9i.

A partir da versão 10g do Oracle, o otimizador baseado em regra (RBO) deixou de ser utilizado e o otimizador baseado em custo (CBO) passou a ser o único existente.

Mesmo com o Oracle 10g utilizando apenas o CBO, vale a pena conhecermos os dois e é o que veremos a seguir.

Otimizador baseado em regra (RBO)

O RBO utiliza uma série de regras rígidas para determinar um plano de execução para cada SQL. Se você conhecer as regras você pode construir uma consulta SQL para acessar os dados da maneira desejada.

Só pra exemplificar, quando você criava um índice na sua tabela e monta uma SQL para acessá-la, filtrando as colunas do índice criado, sendo o otimizador RBO, com certeza ele utilizaria esse índice para acesso. É como se obedecesse uma ordem sua. O problema é que o simples fato de existir um índice que corresponde ao seu filtro, não quer dizer que ele seja seletivo e se não for, causa efeito contrário. Talvez seja mais eficiente acessar toda a tabela (o famoso `TABLE ACCESS FULL`) do que acessar um índice a procura de um ponteiro.

O RBO deixou de ser aperfeiçoado na versão 9i do Oracle Database, quando o CBO, que veremos a seguir, passou a ser o otimizador recomendado.

No Oracle 10g foi descontinuado.

Otimizador baseado em custo (CBO)

Introduzido no Oracle 7, o CBO tenta achar o plano de execução que possui o menor custo para acessar os dados tanto para uma quantidade de trabalho específica como para um tempo inicial de resposta mais rápido. Os Custos de diferentes planos são calculados e a opção que apresentar o menor custo de execução é escolhida. São coletadas estatísticas referentes às tabelas do banco de dados e estas são usadas para determinar um plano de execução ótimo.

A fonte de informações do otimizador CBO são as estatísticas coletadas no banco de dados e essa é uma das razões que levam o Oracle 10g a adotar a coleta de estatísticas automática, uma vez que só trabalha com o otimizador CBO.

IMPORTANTE: Um banco de dados configurado para trabalhar com CBO que não tiver uma política de atualização periódica de estatísticas, provavelmente será vítima de desempenho ruim.

Seletividade

A seletividade é a primeira e mais importante medida do Otimizador Baseado em Custo. Ela representa uma fração de linhas de um conjunto resultante de uma tabela ou o resultado de um *join* ou um agrupamento.

O CBO utiliza estatísticas para determinar a seletividade de um determinado predicado (cláusula *WHERE* ou *HAVING*). A seletividade é diretamente ligada ao predicado da consulta, como por exemplo

```
WHERE PRO_IN_CODIGO = 1245
```

Ou uma combinação de predicados, como:

```
WHERE PRO_IN_CODIGO = 1245  
AND PRO_ST_CESTOQUE='S'
```

O propósito do predicado de uma consulta é limitar o escopo dela a um certo número de linhas em que estamos interessados. Portanto, a seletividade de um predicado indica quantas linhas de um conjunto vão ser filtradas por uma determinada condição.

A seletividade varia numa faixa de valores de 0.0 até 1.0 onde a seletividade de 0 indica que nenhuma linha será selecionada e 1 que todas as linhas serão selecionadas. A seletividade é igual ao número de valores distintos que uma coluna possui (1/NVD onde NVD significa o Número de Valores Distintos).

Variáveis de ligação (Bind Variables)

As variáveis de ligação (*bind*) permitem que uma instrução SQL seja preparada uma única vez pelo banco de dados e executada inúmeras vezes, mesmo com valores diferentes para estas variáveis. Esta economia na fase de preparação a cada execução representa um ganho de eficiência (tempo e recursos) na aplicação e no servidor de banco de dados.

Além disso, variáveis de ligação facilitam a validação de tipo de dados dos valores de entrada fornecidos dinamicamente e evitam os riscos de vulnerabilidade de segurança e integridade existentes quando se constrói uma instrução SQL por concatenação de *strings*. Assim, este recurso trás também robustez e segurança à execução de SQL nas aplicações.

Por exemplo, imagine que você precise buscar de dentro de uma aplicação Delphi a descrição de um determinado produto a partir de um código informado. Uma função (em Delphi) como a abaixo pode resolver

isso:

```
function DescricaoProduto(pCodProduto: integer): string;
begin
  with Query1 do
  begin
    SQL.Close;
    SQL.Clear;
    SQL.Add('SELECT PRO_ST_NOME');
    SQL.Add('FROM PRODUTO');
    SQL.Add('WHERE PRO_IN_CODIGO = ' + pCodProduto);
    SQL.Open;

    Result := FieldByName('PRO_ST_NOME').AsString;
  end;
end;
```

O problema é que toda vez a declaração `SELECT` dessa função for executada, o Oracle vai fazer uma análise da instrução para definir o plano de execução (o chamado `PARSE`). Isso acontece porque o código do produto, recebido como argumento, é concatenado à SQL, ou seja, para o Oracle essa mesma instrução, quando o código for 1 é diferente de quando o código for 2.

Outro impacto negativo dessa instrução refere-se ao uso de memória. O servidor Oracle possui uma área de memória compartilhada que aloca as instruções SQLs executadas mais recentemente. Quando uma instrução é recebida pelo servidor, antes de fazer o seu `PARSE`, ele verifica se essa instrução existe nessa área. Se existir, ele utiliza o plano que já foi definido (e que fica alocado com a instrução). Muito bem e daí? Daí que se o Oracle acha que a instrução acima com o código 1 é diferente da mesma instrução com o código 2, ele vai consumir dois espaços de memória. Agora imagine um sistema com mil usuários conectados e todas as instruções SQLs escritas dessa forma!

A SQL do exemplo acima, ficaria alocado na memória da servidor Oracle da seguinte forma:

Quando for código 1

```
SELECT PRO_ST_NOME
FROM PRODUTO
WHERE PRO_IN_CODIGO = 1
```

Quando for código 2

```
SELECT PRO_ST_NOME
FROM PRODUTO
WHERE PRO_IN_CODIGO = 2
```

Literalmente, as duas instruções acima são diferentes.

Para que o Oracle reaproveite as instruções, eficientemente, é **IMPORTANTÍSSIMO** que os aplicativos façam uso das chamadas *BIND VARIABLES* ao executar SQLs.

Abaixo, segue a mesma função Delphi do exemplo anterior, utilizando esse recurso:

```
function DescricaoProduto(pCodProduto: integer): string;
begin
  with Query1 do
  begin
    SQL.Close;
```

```

SQL.Clear;
SQL.Add('SELECT PRO_ST_NOME');
SQL.Add('FROM PRODUTO');
SQL.Add('WHERE PRO_IN_CODIGO = :PRO_IN_CODIGO');

ParamByName('PRO_IN_CODIGO').AsInteger := pCodProduto;
SQL.Open;

Result := FieldByName('PRO_ST_NOME').AsString;
end;
end;

```

A *BIND VARIABLE* é indicada pelos dois pontos dentro da declaração SQL (:PRO_IN_CODIGO). O nome da *variable* é livre, mas observe que no nosso caso foi utilizado o nome da coluna que está sendo filtrada. **Esse é apenas um padrão.**

Bem, esta instrução na memória do servidor Oracle ficará semelhante com o demonstrado abaixo:

```

SELECT PRO_ST_NOME
FROM PRODUTO
WHERE PRO_IN_CODIGO = :1

```

Este “:1” não representa o código de produto 1 e sim uma *BIND VARIABLE*. Essa variável assumirá como valor qualquer código de produto informado na aplicação, ou seja, sempre será a mesma instrução, não importando a quantidade de vezes que ela for executada e, conseqüentemente, o mesmo plano de execução (já armazenado da primeira vez).

Portanto, há grande importância e vantagens no uso de SQL que usam variáveis de ligação (*bind*) em aplicações que interagem com bancos de dados, especialmente quando envolvem valores dinâmicos e parâmetros fornecidos pelo usuário, de forma que este recurso deve ser utilizado sempre, tratando-se de boa prática de programação.

SQL Dinâmico

Tudo que falamos sobre *BIND VARIABLE*, pode induzir o leitor a achar que o recurso de SQL Dinâmico da PL/SQL funciona de forma semelhante. **ENGANO!**

Os SQLs Dinâmicos do PL/SQL SEMPRE são submetidos ao `PARSE` e SEMPRE ocupam lugar exclusivo na memória.

Quando desenvolver rotinas PL/SQL, procure escrever SQLs nativos (estáticos).

Por exemplo, na função `pck_cliente.GetMediaCompraMensal` poderia ser reescrita da seguinte forma:

```

-- Função que retorna a media de compra (geral,
-- por ramo de atividade, cidade ou ramo + cidade)
FUNCTION GetMediaCompraMensal
    (pRamo   cliente.cli_st_ramoatividade%TYPE
    ,pCidade cliente.cli_st_cidade%TYPE)
RETURN NUMBER IS
    vlResult   cliente.cli_vl_mediacomprasmensal%TYPE := 0;
BEGIN

    SELECT AVG(c.cli_vl_mediacomprasmensal)
    INTO vlResult

```

```
FROM cliente c
WHERE c.cli_st_amoatividade =
      decode(pRamo,NULL,c.cli_st_amoatividade,pRamo)
AND c.cli_st_cidade =
      decode(pCidade,NULL,c.cli_st_cidade,pCidade);

RETURN(vlResult);
EXCEPTION
WHEN no_data_found THEN
  RETURN(0);
WHEN OTHERS THEN
  raise_application_error
    (-20100, 'Não foi possível recuperar média de compra para: '||
      chr(13)||'Ramo: '||pRamo||
      chr(13)||'Cidade: '||pCidade||
      chr(13)||
      chr(13)||'Erro: '||sqlerrm);

END GetMediaCompraMensal;
```

IMPORTANTE: no exemplo acima, aplicamos uma função do Oracle nas condições da consulta. Esse tipo de implementação deve ser avaliado caso a caso, pois pode afetar a performance (negativamente). Via de regra, os problemas acontecem quando a função é aplicada na coluna e não no valor de restrição.

O uso de índices

Para tirar vantagem dos índices, escreva seu SQL de uma maneira que o Oracle faça uso dele. O otimizador do Oracle não usará o acesso através de um índice simplesmente porque ele existe em uma coluna, caso o otimizador seja por custo.

Lembre-se também que um índice NÃO SELETIVO pode ser ignorado pelo otimizador CBO.

Tenha certeza de ter criado todos os índices necessários nas tabelas, mas tome cuidado com o excesso de índices, pois eles podem degradar a performance de DMLs na tabela. Então como escolher que colunas indexar?

- Use índices em colunas que são frequentemente usados na cláusula `WHERE` de consultas da aplicação ou de consultas usadas por usuários finais.
- Indexe as colunas que são frequentemente usadas para juntar *join* as tabelas nas diferentes consultas. Prefira fazer *join* pelas chaves primárias e chaves estrangeiras. Use índices apenas em colunas que possuem uma baixa porcentagem de linhas com valores iguais (**isso está diretamente ligado a seletividade**).
- Não use índices em colunas que são usadas apenas com funções e operadores NÃO EXATOS (<, <=, >, >=, <>, !=) na cláusula `WHERE`, pois a partir do momento em que você aplica uma função à coluna ou utiliza um operador que permite um *range* de valores, o índice é ignorado pois vai apontar para *n* possíveis entradas.
- Não indexe colunas que são frequentemente modificadas ou quando a eficiência ganha através da criação de um índice não valha a pena devido à perda de performance em operações de `INSERT`, `UPDATE` e `DELETE`. Com a criação do índice, estas operações perderão em performance devido à necessidade de manter o índice correto.
- Índices únicos (`UNIQUE`) são melhores que os não únicos devido a melhor seletividade. Use índices únicos em chaves primárias e índices não únicos em chaves estrangeiras (`FOREIGN KEY`) e colunas muito usadas nas cláusulas `WHERE`;

➤ **AVALIE SEMPRE** a possibilidade de criar um índice para cada `FOREIGN KEY` da tabela. Isso pode resultar em grandes ganhos de desempenho em instruções onde ocorre o relacionamento entre as tabelas.

Colunas indexadas no `ORDER BY`

O otimizador do Oracle irá utilizar uma varredura por índice se a cláusula `ORDER BY` possuir uma coluna indexada. Uma consulta usará o índice criado para uma coluna, mesmo que a coluna não seja especificada na cláusula `WHERE`. A consulta obterá o `ROWID` para cada linha do índice e acessará a tabela usando o `ROWID`.

EXPLAIN PLAN

Se familiarize com a ferramenta `EXPLAIN PLAN` e use-a para otimizar seu SQL. O `EXPLAIN PLAN` irá te ajudar a descobrir, através do plano de execução da consulta, os meios de acesso que o Oracle está utilizando para acessar as tabelas do banco de dados.

Uma vez que identificamos um SQL com uma performance ruim, podemos usar o comando `EXPLAIN PLAN FOR` para gerar um plano de execução para este SQL.

Para usar esse comando é necessário criar a tabela `PLAN_TABLE`, através do *script* `utlxplan.sql`, que normalmente fica em "Oracle_home/rdbms/admin"

Vejamos um exemplo no SQL*Plus de como usar o `EXPLAIN PLAN`:

```
EXPLAIN PLAN FOR
SELECT n.*,
       i.*
FROM   nota_fiscal_venda n,
       item_nota_fiscal_venda i
WHERE  n.nfv_in_numero = i.nfv_in_numero;
```

Desta forma estamos populando a tabela `PLAN_TABLE` com o plano de execução do SQL. A consulta não é executada. Apenas o plano de execução é gerado.

Mas como visualizamos o plano?

Antes de executar o `EXPLAIN PLAN`, certifique-se de truncar a tabela `PLAN_TABLE` ou de usar o comando "SET STATEMENT_ID=" nas execuções do comando `EXPLAIN PLAN FOR`, pois de outra forma o plano de execução de diversas consultas será guardado na `PLAN_TABLE` tornando sua interpretação extremamente complicada.

Então, vamos recriar nosso plano de execução definindo um `STATEMENT_ID`:

```
EXPLAIN PLAN SET STATEMENT_ID = 'XXX' FOR
SELECT n.*,
       i.*
FROM   nota_fiscal_venda n,
       item_nota_fiscal_venda i
WHERE  n.nfv_in_numero = i.nfv_in_numero;
```

Agora você pode consultar a `PLAN_TABLE` para ver como a consulta será executada. A consulta a seguir pode ser usada para extrair a parte importante do plano de execução da tabela `PLAN_TABLE`.

```
SELECT LPAD(' ', 2*(LEVEL-1)) ||
       OPERATION || ' ' ||
       OPTIONS || ' ' ||
       OBJECT_NAME || ' ' ||
```

```

        DECODE (ID, 0, 'COST = ' || POSITION) "QUERY PLAN"
FROM PLAN_TABLE
START WITH ID = 0 AND STATEMENT_ID = 'XXX'
CONNECT BY PRIOR ID = PARENT_ID AND STATEMENT_ID = 'XXX';

```

O plano de execução da consulta ficará como a seguinte amostra:

```

QUERY PLAN
-----
SELECT STATEMENT      COST = 11
  MERGE JOIN
    SORT JOIN
      TABLE ACCESS FULL  NOTA_FISCAL_VENDA
    SORT JOIN
      TABLE ACCESS FULL  ITEM_NOTA_FISCAL_VENDA

```

IMPORTANTE: Para que o custo da declaração (COST) seja informado, é necessário que estatísticas dos objetos acessados pela declaração estejam devidamente coletados. Essas estatísticas podem ser coletadas através do pacote `dbms_stats`. O procedimento `gather_schema_stats`, por exemplo, faz a coleta de estatísticas de todos os objetos do *schema*. NO exemplo abaixo é coletada as estatísticas do *schema* JSILVA:

```
SQL> execute dbms_stats.gather_schema_stats(ownname => 'JSILVA');
```

Existem boas ferramentas de programação que possuem o EXPLAN, como: Toad, Oracle Sql Developer, PLSQL Developer, etc..

O AUTOTRACE do SQL*Plus

O AUTOTRACE é um recurso do SQL*Plus que permite gerar, automaticamente, um relatório baseado no plano de execução usado pelo otimizador assim como também as estatísticas referentes à execução daquele SQL. O Relatório é gerado após a execução com sucesso de comandos DML (SELECT, DELETE, UPDATE e INSERT). Ele é usado para monitorar e otimizar a performance dessas declarações.

O AUTOTRACE pode ser utilizado de cinco maneiras:

1. SET AUTOTRACE OFF: Nenhum relatório de AUTOTRACE é gerado. Esta é a opção padrão (Default).
2. SET AUTOTRACE ON EXPLAIN: O relatório de AUTOTRACE mostra apenas o Plano de execução.
3. SET AUTOTRACE ON STATISTICS: O relatório de AUTOTRACE mostra apenas as estatísticas referentes a execução do SQL.
4. SET AUTOTRACE ON: O Relatório de AUTOTRACE inclui tanto o Plano de execução como as estatísticas referentes à execução do SQL. Também requer executar o script `plustrce.sql` e receber alguns privilégios de DBA.
5. SET AUTOTRACE TRACEONLY: Funciona como o SET AUTOTRACE ON, mas suprime a impressão do resultado da declaração se ela possuir. Também requer executar o script `plustrce.sql` e receber alguns privilégios de DBA.

IMPORTANTE: Para utilizar as opções que exibem estatísticas é necessário:

- Executar o script `plustrce.sql`, que fica em `ORACLE_HOME\sqlplus\admin` com o usuário `SYS` (como `SYSDBA`);
- Ainda com o usuário `SYS`, atribuir o *role* `PLUSTRACE` para o usuário que fará uso do recurso, caso não seja o próprio `DBA`.

Exemplo:

1. Ative o `SET AUTOTRACE TRACEONLY` para exibir o plano de execução e as estatísticas, sem as linhas de retorno:

```
SQL> SET AUTOTRACE TRACEONLY;
```

2. Agora execute a seguinte consulta:

```
SELECT n.nfv_in_numero,
       i.pro_in_codigo
FROM nota_fiscal_venda n,
     item_nota_fiscal_venda i
WHERE n.nfv_in_numero = i.nfv_in_numero;
```

3. O resultado deve ser parecido com:

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=60 Bytes=780)
1      0      NESTED LOOPS (Cost=4 Card=60 Bytes=780)
2      1      TABLE ACCESS (FULL) OF 'ITEM_NOTA_FISCAL_VENDA' (Cost=4
Card=60 Bytes=480)

3      1      INDEX (UNIQUE SCAN) OF 'PK_NOTA_FISCAL_VENDA' (UNIQUE)
```

Statistics

```
-----
0      recursive calls
0      db block gets
24     consistent gets
0      physical reads
0      redo size
1467   bytes sent via SQL*Net to client
532    bytes received via SQL*Net from client
5      SQL*Net roundtrips to/from client
0      sorts (memory)
0      sorts (disk)
60     rows processed
```

Lendo a saída de um AUTOTRACE

No exemplo acima, onde usamos o `AUTOTRACE TRACEONLY`, temos o plano de execução e as estatísticas, mas como interpretamos isso?

O plano de execução:

- `Optimizer` indica o otimizador Oracle usado: `CHOOSE` ou `RULE` (Custo ou Regra);
- `Cost` indica o custo em cada ponto do plano, ou seja, o custo da `SELECT STATEMENT` é o

custo total da declaração;

- `Card` indica a cardinalidade em cada ponto do plano. Entenda como cardinalidade o número de relacionamentos para recuperar os dados;
- `Bytes` indica o número de `bytes` envolvidos em cada ponto do plano;
- `Nested`, como o nome já diz, é algo aninhado e no nosso exemplo temos `NESTED LOOP`, ou seja, um `LOOP` aninhado que foi gerado devido ao relacionamento de *master/detail* entre nota fiscal e item nota fiscal;
- `TABLE ACCESS (FULL)` indica o objeto em que foi realizado um acesso completo, ou seja, no nosso caso todas as linhas da tabela de item nota fiscal foram verificadas. Isso não é bom em tabelas com grande e médio volumes de dados. O ideal é um acesso parcial através de um índice, como aconteceu com a tabela de notas;
- `INDEX SCAN` indica o índice utilizado para acessar um conjunto de dados, no nosso caso, na tabela de nota fiscal foi usada a chave primária.

As estatísticas:

- `recursive calls`: Número de chamadas recursivas dentro da instrução;
- `db block gets`: é o número de vezes que um bloco foi requisitado para o buffer cache;
- `consistent gets`: é o número de vezes que uma leitura consistente foi requisitada para um bloco do buffer cache.
- `physical reads`: é o número total de blocos de dados lidos do disco para o buffer cache.
- `redo size`: É o tamanho (em bytes) do log redo gerado durante essa operação;
- `bytes sent via SQL*Net to client`: Total de bytes enviados ao cliente pelos processos de segundo plano;
- `bytes received via SQL*Net from client`: Total de bytes enviados ao servidor pelo cliente;
- `SQL*Net roundtrips to/from client`: Número total de mensagens enviadas e recebidas pelo cliente;
- `sorts (memory)`: Número de ordenações em memória;
- `sorts (disk)`: Número de ordenações em disco. Isso é muito ruim para o desempenho pois executa I/O;
- `rows processed`: Número de linhas processadas na operação.

INDEX SCAN versus FULL TABLE SCAN

Se você estiver selecionando mais de 15 % das linhas de uma tabela, um `FULL TABLE SCAN` é geralmente mais rápido do que o acesso pelo índice. Quando o acesso por índice causar lentidão ao invés de apresentar um ganho de performance, você pode utilizar algumas técnicas para eliminar o uso do índice:

```
SELECT *  
FROM produto p  
WHERE p.pro_vl_ultimavenda+0 = 10000;
```

Supondo que existisse um índice na coluna `pro_vl_ultimavenda`, ele não seria usado, pois o

índice seria pela coluna `pro_vl_ultimavenda` e não pela coluna mais zero.

Um índice também não é usado se o Oracle tiver que realizar uma conversão implícita de dados. No nosso exemplo, `pro_vl_ultimavenda` é do tipo `NUMBER`, ou seja, se filtrarmos o valor como *string*, o índice não será usado:

```
SELECT *  
FROM produto p  
WHERE p.pro_vl_ultimavenda = '10000';
```

Essa manobra também poderia ser realizada aplicando uma função na coluna filtrada, sem que isso afetasse seu conteúdo, é claro.

CUIDADO: Lembre-se sempre que essa teoria só vale quando estamos selecionando mais do que 15% dos dados e, mesmo assim, deve ser analisada com atenção, pois as exceções também existem. Tenha responsabilidade no uso dessas técnicas.

A cláusula `WHERE` é crucial!

Informe o máximo de restrições possíveis na cláusula `WHERE`, pois isso diminuirá o universo dos dados a serem recuperados, mas tome cuidado com algumas situações onde o uso de restrições pode ser prejudicial, principalmente para a utilização de índices.

As seguintes cláusulas no `WHERE` não farão uso do índice mesmo que ele esteja disponível:

- `A.COL1 > A.COL2`
- `A.COL1 < A.COL2`
- `A.COL1 >= A.COL2`
- `A.COL1 <= A.COL2`
- `COL1 IS NULL`
- `COL1 IS NOT NULL`.

Uma entrada de índice possui ponteiros para valores exatos da tabela e não um *range* de valores, logo o uso de operadores `>`, `<`, `>=` e `<=`, inviabilizam o uso de um índice.

Um índice não guarda o `ROWID` (que é o ponteiro) de colunas que possuem valores nulos. Qualquer consulta em linhas que possuam valores nulos o índice não pode ser utilizado.

- `COL1 NOT IN (value1, value2)`
- `COL1 != expression`
- `COL1 LIKE '%teste'`

Neste caso, o uso do `"%"` no início da *string* acaba por suprimir a parte por onde coluna é indexada e por isso o índice não é usado. Por outro lado, `COL1 LIKE 'teste%'` ou `COL1 LIKE 'teste%teste%'` faz uso do índice resultando em uma busca por faixas limites.

- `UPPER(COL1) = 'TESTE'`

Quaisquer expressões, funções ou cálculos envolvendo colunas indexadas não farão uso do índice se

ele existir. No exemplo acima, o uso da função `UPPER` vai impedir do índice ser usado.

Use o `WHERE` ao invés do `HAVING` para filtrar linhas

Evite o uso da cláusula `HAVING` junto com `GROUP BY` em uma coluna indexada. Neste caso o índice não é utilizado. Além disso, exclua as linhas indesejadas na sua consulta utilizando a cláusula `WHERE` ao invés do `HAVING`. Se a tabela `produto` possuísse um índice na coluna `pro_st_marca`, a seguinte consulta não faria uso dele:

```
SELECT p.pro_st_marca,  
       AVG(p.pro_vl_maiorvenda) pro_vl_mediamaiorvenda  
FROM produto p  
GROUP BY p.pro_st_marca  
HAVING p.pro_st_marca = 'Kaiser';
```

A mesma instrução poderia ser reescrita da seguinte maneira para aproveitar o índice:

```
SELECT p.pro_st_marca,  
       AVG(p.pro_vl_maiorvenda) pro_vl_mediamaiorvenda  
FROM produto p  
WHERE p.pro_st_marca = 'Kaiser'  
GROUP BY p.pro_st_marca;
```

Especifique as colunas principais do índice na cláusula `WHERE`

Em um índice composto a consulta apenas utilizará o índice se as principais colunas do índice estiverem especificada na cláusula `WHERE`.

Se você possui um índice na tabela de item nota fiscal cujas colunas são `NFV_IN_NUMERO` e `INFV_IN_NUMERO`, o uso de `NFV_IN_NUMERO` na cláusula `WHERE` pode levar ao uso do índice. O mesmo não acontece se apenas `INFV_IN_NUMERO` for utilizada. Obviamente que a utilização das duas colunas (de preferência na ordem de criação do índice) levará ao uso do índice.

Evite a cláusula `OR`

Se um SQL envolve `OR` na cláusula `WHERE`, ele também pode ser reescrito substituindo o `OR` pelo `UNION`. Você deve verificar cuidadosamente os planos de execução de cada SQL para decidir qual o mais adequado e com melhor desempenho para a aplicação. O uso de `OR` não é recomendado em aplicações transacionais. Seu uso é muito difundido em aplicações de *Datawarehouse*.

Cuidado com “Produto Cartesiano”

Produto cartesiano é a combinação de dois conjuntos, de forma que resulte em um terceiro conjunto constituído por todos os elementos do primeiro combinados com todos os elementos do segundo.

Isso é o que acontece quando você declara em sua SQL duas tabelas ou mais e não implementa o *join* correto entre elas. Como no exemplo abaixo:

```
SELECT c.cli_in_codigo  
      ,c.cli_st_nome  
      ,nfv.nfv_in_numero  
      ,nfv.nfv_dt_emissao  
      ,nfv.nfv_st_condicaopagamento  
      ,nfv.nfv_vl_total  
FROM cliente c  
      ,nota_fiscal_venda nfv;
```

No nosso caso, as tabelas possuem poucos registros, mas imagine uma situação onde existem 10.000 clientes cadastrados e sejam emitidas 50.000 notas por dia?! Teríamos 500.000.000 registros processados no banco de dados e retornados para nossa aplicação. Provavelmente essa consulta travaria o banco de dados.

Por isso, muito cuidado para não produzir um produto cartesiano indesejado. Existem situações em que o produto cartesiano é produzido de propósito, mas são raros ... Muito raros!

SQLs complexas

Comandos SQLs muito complexos podem sobrecarregar o otimizador; As vezes a melhor solução pode ser escrever vários SQLs simples com uma boa performance do que um único SQL complexo. Por exemplo: se uma junção envolve mais de 8 tabelas com uma grande quantidade de dados seria melhor quebrar o SQL em dois ou três SQLs menores, cada um envolvendo no máximo 4 junções de tabelas, e guardar os resultados em tabelas temporárias criadas previamente ou criar um bloco ou função para realizar toda a operação, passo a passo.

O SQL não é uma linguagem procedural. Usar um código SQL para executar diferentes coisas geralmente resulta em baixa performance para cada uma das tarefas. Se você deseja que seu SQL faça diferentes tarefas, então escreva vários SQLs, ao invés de escrever apenas um para fazer as diferentes tarefas dependendo dos parâmetros passados para o script.

Quando sua declaração SQL estiver se parecendo mais com um programa do que com uma instrução de comandos, analise a possibilidade de quebrá-la em pedaços.

Quando usar MINUS, IN e EXISTS

Em vários casos, mais de um SQL pode lhe trazer o resultado desejado. Cada um deles pode ter um plano de execução diferente e assim se comportar de forma diferente.

MINUS

O operador MINUS, por exemplo, pode ser muito mais rápido do que usar WHERE NOT IN ou WHERE NOT EXISTS. Vamos dizer que tenhamos um índice na coluna cli_st_uf e outro índice na coluna cli_st_ramoatividade. Desconsiderando a disponibilidade de índices, a consulta a seguir vai requerer um FULL TABLE SCAN devido ao uso do predicado NOT IN:

```
SELECT cli_in_codigo
FROM cliente
WHERE cli_st_uf IN ('SP', 'RJ')
AND cli_st_ramoatividade NOT IN ('SUPERMERCADO', 'MERCADO');
```

Entretanto se a mesma query for escrita da seguinte forma vai resultar em uma varredura por índice (INDEX SCAN):

```
SELECT cli_in_codigo
FROM cliente
WHERE cli_st_uf IN ('SP', 'RJ')
MINUS
SELECT cli_in_codigo
FROM cliente
WHERE cli_st_ramoatividade IN ('SUPERMERCADO', 'MERCADO');
```

EXISTS

A função EXISTS procura pela presença de uma única linha que satisfaz o critério de pesquisa ao contrário do comando IN que procura por todas as ocorrências. Por exemplo, vamos supor que nossa

tabela de produtos tivesse 1000 linhas e nossa tabela de itens de nota fiscal também tivesse 1000 linhas.

Executando a consulta a seguir, todas as linhas da tabela de itens seriam lidas para cada linha da tabela de produto. O Resultado final vai ser de 1.000.000 de linhas lidas:

```
SELECT p.pro_in_codigo
FROM produto p
WHERE p.pro_in_codigo IN (SELECT i.pro_in_codigo
                        FROM item_nota_fiscal_venda i
                        );
```

Se alterarmos a declaração para usar o EXISTS, no máximo, uma linha será lida para cada linha correspondente da tabela produto, reduzindo desta forma a sobrecarga de processamento da consulta:

```
SELECT p.pro_in_codigo
FROM produto p
WHERE EXISTS (SELECT 1
             FROM item_nota_fiscal_venda i
             WHERE i.pro_in_codigo = p.pro_in_codigo
             );
```

Evite o SORT

SORT é como referenciamos operações de ordenação realizadas pelo Oracle. No ponto de vista de desempenho, SORT nunca é bom. Para resumir, o Oracle executa um SORT quando precisa *arrumar* os dados antes de enviá-los para o solicitante. Ele recupera os dados e executa o SORT para organizá-los da maneira que o solicitante pediu. Por exemplo, quando é necessário devolver o resultado ordenado, agrupado ou distinto, haverá um SORT, ou seja, um ORDER BY, um GROUP BY ou um DISTINCT vai gerar SORT.

Quando é necessário realizar essa operação, o Oracle tenta fazê-lo em memória, mas se a área reservada para isso não for suficiente para a quantidade de dados recuperada, o Oracle vai utilizar o TABLESPACE TEMPORÁRIA em disco, ou seja, vai gerar um I/O nada desejável, pois I/O é o que há de pior para desempenho de banco de dados.

EXISTS ao invés de DISTINCT

Sempre que puder, Use o EXISTS ao invés do DISTINCT se você desejar que o resultado contenha apenas valores distintos ao fazer a junção de tabelas.

Por exemplo:

```
SELECT DISTINCT r.rep_in_codigo, r.rep_St_nome
FROM representante r,
     nota_fiscal_venda n
WHERE r.rep_in_codigo = n.rep_in_codigo;
```

A declaração abaixo é uma alternativa melhor:

```
SELECT r.rep_in_codigo, r.rep_St_nome
FROM representante r
WHERE EXISTS (SELECT n.rep_in_codigo
             FROM nota_fiscal_venda n
             WHERE r.rep_in_codigo = n.rep_in_codigo);
```

UNION e UNION ALL

É sempre melhor escrever SQLs diferentes para tarefas diferentes, mas se você tem que usar apenas um SQL, você pode fazer ele parecer menos complexo usando o operador `UNION`. Toda via, o `UNION` tem um problema de desempenho. O `UNION` executa, implicitamente um `DISTINCT`, ou seja, faz um `SORT` para eliminar linhas repetidas.

Mas é possível executar união de declarações SQL evitando o `SORT` do `UNION`. Você pode usar o `UNION ALL` que retorna todas as linhas, não se preocupando com a distinção entre elas. Isso quer dizer que ele não faz um `SORT`.

Se você tem certeza que os resultados das `SELECTs` unidas não geraram linhas repetidas, utilize o `UNION ALL`.

Cuidados ao utilizar `VIEWS`

Joins em `VIEWS` complexas

Joins em `VIEWS` complexas não são recomendados e devem ser evitados quando possível, especialmente em *joins* entre duas ou mais `VIEWS` complexas. Quando trabalhamos com *joins* entre `VIEWS` complexas, as mesmas têm que ser instanciadas primeiramente em memória e depois a consulta é efetuada em cima dos dados resultantes das `VIEWS`. Deu pra imaginar o custo?

Reciclagem de `VIEWS`

Esteja atento ao fato de escrever uma `VIEW` com um propósito e depois utilizá-la para outro onde ela seria mal empregada. Uma consulta à `VIEW` requer que todas as tabelas por ela referenciadas sejam acessadas para que os dados sejam retornados. Antes de usar uma `VIEW`, verifique se todas as tabelas desta `VIEW` precisam ser realmente acessadas para retornar os dados que você precisa. Senão, não utilize a `VIEW`. Ao invés disso use as tabelas base ou crie uma nova `VIEW` (se necessário) para sua necessidade em específico.

Database Link

As vezes sua consulta está bem elaborada, o banco possui índices seletivos e tudo mais que tenda para a boa performance, mas a sua consulta demora para dar retorno.

Isso pode ser causado por causa de *Database Link*.

Se a sua consulta acessa um objeto de outro banco de dados (diretamente ou por sinônimo), você pode ter um problema de performance, pois existe o tempo rede entre um banco de dados e outro (além do tráfego de rede do banco de dados base com a sua aplicação).

Como se não bastasse o fato de existir um tráfego de rede extra, a performance de Database Link não é das melhores.

Os Database Links são recomendados apenas para carga de dados em processos agendados (jobs noturnos, por exemplo).

Aplicativos versus Banco de Dados

As aplicações devem tentar acessar os dados apenas uma vez. Isto reduz o tráfego de informações na rede além da carga no banco de dados. Considere as seguintes técnicas:

Utilize o comando `CASE` para combinar múltiplas varreduras:

Isso é possível movendo a condição `WHERE` de cada varredura em um comando `CASE`, que filtra os dados de cada agregação. Eliminando $n-1$ varreduras, pode representar um grande ganho de performance. O exemplo a seguir que saber o numero de representantes cuja maior venda é menor do que 20000, entre 20001 e 40000, e maior que 40000 todo mês. Isto pode ser feito através de 3 consultas:

```
SELECT COUNT (*)
FROM representante r
WHERE r.rep_vl_maiorvenda <= 20000;

SELECT COUNT (*)
FROM representante r
WHERE r.rep_vl_maiorvenda BETWEEN 20001 AND 40000;

SELECT COUNT (*)
FROM representante r
WHERE r.rep_vl_maiorvenda > 40000;
```

Neste caso executamos três comandos para conseguir as informações e conseguimos com sucesso, entretanto, é mais eficiente rodar toda consulta como um único SQL. Cada número é calculado como uma coluna. O `count` usa um filtro com o comando `CASE` para contabilizar apenas as linhas onde a condição é válida, como mostrado abaixo:

```
SELECT COUNT (CASE WHEN r.rep_vl_maiorvenda <= 20000
                  THEN 1 ELSE NULL END) menor_20000,
       COUNT (CASE WHEN r.rep_vl_maiorvenda BETWEEN 20001 AND 40000
                  THEN 1 ELSE NULL END) entre_20000_40000,
       COUNT (CASE WHEN r.rep_vl_maiorvenda > 40000
                  THEN 1 ELSE NULL END) maior_40000
FROM representante r;
```

Teremos as mesmas informações com um único acesso ao banco e sem prejudicar o plano de execução!

Utilize blocos PL/SQL para executar operações SQL repetidas em LOOPS de sua aplicação

Imagine que você acaba de incluir uma nota fiscal com 200 itens e para cada item, você precisa validar alguma coisa no cadastro de produto? Você terá que executar 200 vezes a mesma `SELECT` mudando apenas o produto, ou seja, duzentas vezes esse comando será enviado pela rede ao servidor e o resultado será devolvido. Agora imagine que você emita 400 notas por dia! Que tráfego de rede, não?!

Existem maneiras de evitar esse problema!

Muitas linguagens possuem recursos para que você forneça um *array* como parâmetro para o bloco SQL e então processe as instruções de cada linha do *array* antes de retornar à aplicação. No nosso exemplo, o *array* conteria os itens da nota e as instruções seriam as `SELECTs` de verificação.

No Delphi, por exemplo, existe um conjunto de componentes específicos para acesso à banco de dados Oracle (não gratuito) que disponibiliza um tipo chamado `TPLSQLTable`. Você pode definir uma variável desse tipo, alimentá-lo com os itens que desejar e terá um *array* em Delphi. O seu bloco PL/SQL só precisa ter uma coleção do tipo tabela por índice e receber o *array* do Delphi como parâmetro, como abaixo:

DECLARE


```
-- Definir um vetor para conter itens
TYPE TCodigoProduto IS TABLE OF produto.pro_in_codigo%TYPE INDEX BY
binary_integer;

-- Declaração de variáveis
cCodigoProduto TCodigoProduto;
BEGIN
-- Coleção PL/SQL recebendo como parâmetro array do Delphi com itens da nota
-- (pCodigoProduto é o array em Delphi e cCodigoProduto o array em PL/SQL)
cCodigoProduto := :pCodigoProduto;

-- Executa loop na lista de itens
FOR vCount IN cCodigoProduto.FIRST..cCodigoProduto.LAST LOOP

    /* Processamento de cada linha do array montado no Delphi*/

END LOOP;
END;
/
```

Use a cláusula RETURNING em declarações DML:

Quando apropriado, use INSERT, UPDATE ou DELETE com a cláusula RETURNING para selecionar e modificar dados com uma única chamada. Esta técnica aumenta a performance diminuindo o número de chamadas ao banco de dados.

O RETURNING pode te retornar um dos valores que foram alterados. Por exemplo, você inclui uma nova nota fiscal em seu sistema e após a gravação precisa informar o número da nota gravada para o usuário. Ao invés de executar uma SELECT depois do INSERT só para recuperar esse número, você inclui no seu INSERT a cláusula RETURNING para retornar o valor gravado na coluna de número.

Exercícios Propostos

1. Revise todos os procedimentos criados nos pacotes pck_cliente, pck_produto e pck_representant, aplicando as boas prática de desenvolvimento SQL.